Developing Liferay DXP 7.2

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 7.2
by The Liferay Documentation Team

Contents

Contents **i**

Preface **lxix**
Conventions lxix
Publisher Notes lxix

I Developer Tutorials **1**

1 Developer Tutorials **3**

2 Developing a Web Application **5**

3 Setting Up a Development Environment **7**
3.1 Installing a Liferay Dev Studio DXP Bundle 7
3.2 Creating a Liferay Workspace 8

4 Generating the Back-end **11**

5 What is Service Builder? **13**
5.1 Guestbook Application Design 13
5.2 Service Layer 14

6 Generating Model, Service, and Persistence Layers **15**

7 Implementing Service Methods **21**
7.1 Updating Generated Classes 25

8 Building the Web Front-End **27**

9 Creating the Web Project **29**
9.1 What is a Portlet? 31
9.2 What is a Component? 31
9.3 Deploying the Application 32

10 Defining the Component Metadata Properties **37**

11 Creating Portlet Keys **41**

12 Integrating the Back-end **43**

13 Creating an Add Entry Button **45**

14 Generating Portlet URLs **47**

15 Linking to Another Page **49**

16 Forms and Action URLs **51**
16.1 Action URLs 51
16.2 Forms 51

17 Implementing Portlet Actions **53**
17.1 Creating an Add Entry Action 53
17.2 Creating a Delete Entry Action 55

18 Displaying Guestbook Entries **57**
18.1 Rendering the Portlet 57
18.2 Displaying Guestbook Entries 58
18.3 Creating an Actions JSP 60

19 Fitting it All Together **63**
19.1 The Back-End 63
19.2 The Front-End 64
19.3 Deploying and Testing the Application 64
19.4 What's Next? 64

20 Writing an Administrative Portlet **67**

21 Creating the Classes **69**
21.1 Panels and Categories 69

22 Adding Metadata **71**

23 Updating Your Service Layer **75**
23.1 Adding Guestbook Service Methods 75

24 Defining Portlet Actions **77**
24.1 Adding Three Portlet Actions 77

25 Adding Tabs to the Guestbook Portlet **81**

26 Creating a User Interface **83**
26.1 Step 1: Creating the Default View 83
26.2 Step 2: Creating the Actions Button 84
26.3 Step 3: Creating the Edit Guestbook JSP 86

27 Displaying Messages and Errors **89**

28 Creating Language Keys **91**

29 Adding Failure and Success Messages **93**

30 Adding Messages to JSPs **95**

31 Using Resources and Permissions **97**

32 Defining Permissions **99**
32.1 Defining Model Permissions 101
32.2 Defining Portlet Permissions 103

33 Registering Your Permissions in the Database **105**
33.1 Registering Guestbook Resources 105
33.2 Registering Guestbook Entry Resources 106

34 Registering Permissions with the Container **107**

35 Assigning Permissions to Resources **111**
35.1 Creating a Guestbook Portlet Permission Helper 111
35.2 Creating Model Permission Helpers 112

36 Checking for Permission in JSPs **115**
36.1 Checking Permissions in the UI 115
36.2 Testing the Application 117

37 Search and Indexing **119**

38 Enabling Search and Indexing for Guestbooks **121**

39 Understanding Search and Indexing **123**

40 Registering Guestbooks with the Search Framework **125**

41 Indexing Guestbooks **127**
41.1 Implementing `ModelDocumentContributor` 127
41.2 Implementing `ModelIndexerWriterContributor` 128

42 Querying for Guestbook Documents **131**
42.1 Implementing `KeywordQueryContributor` 131

43 Generating Results Summaries **133**

44 Handling Indexing in the Guestbook Service Layer **135**

45 Enabling Search and Indexing for Entries **137**

46 Registering Entries with the Search Framework **139**

47 Indexing Entries **141**
47.1 Implementing `ModelDocumentContributor` . . . 141
47.2 Implementing `ModelIndexerWriterContributor` . . . 142
47.3 Implementing `GuestbookEntryBatchReindexer` . . . 143

48 Querying for Guestbook Entry Documents **145**
48.1 Implementing `KeywordQueryContributor` . . . 145

49 Generating Results Summaries **147**

50 Handling Indexing in the Entry Service Layer **149**

51 Updating Your User Interface For Search **151**

52 Adding a Search Bar to the Guestbook Portlet **153**

53 Creating a Search Results JSP for the Guestbook Portlet **155**

54 Assets: Integrating with Liferay's Framework **161**

55 Enabling Assets at the Service Layer **163**

56 Handling Assets for the Guestbook Service **165**

57 Handling Assets for the GuestbookEntry Service **167**

58 Implementing Asset Renderers **169**

59 Implementing a Guestbook Asset Renderer **171**
59.1 Creating the AssetRenderer Class . . . 171
59.2 Creating the GuestbookAssetRendererFactory Class . . . 175

60 Implementing a Guestbook Entry Asset Renderer **179**
60.1 Creating the GuestbookEntryAssetRenderer Class . . . 179
60.2 Creating the GuestbookEntryAssetRendererFactory Class . . . 182

61 Adding Asset Features to Your User Interface **187**

62 Creating JSPs for Displaying Custom Assets in the Asset Publisher **189**

63 Enabling Tags, Categories, and Related Assets for Guestbooks **191**
63.1 Enabling Asset Features . . . 191

64 Enabling Tags, Categories, and Related Assets for Guestbook Entries **195**

65 Using Workflow **201**

66 Supporting Workflow at the Service Layer **203**

67 Setting the Guestbook Status **205**
67.1 Creating the updateStatus Method . . . 206

68 Setting the Entry Workflow Status **209**

69 Retrieving Guestbooks and Entries by Status **211**
69.1 Calling the Persistence Layer . . . 212

70 Handling Workflow **213**

71 Creating a Workflow Handler for Guestbooks **215**

72 Creating a Workflow Handler for Guestbook Entries **219**

73 Displaying Approved Workflow Items **221**

74 Displaying Guestbook Status **223**

75 Displaying Approved Entries **225**

76 Upgrading Code to 7.0 **227**

77 Upgrading Your Development Environment **231**
77.1 Setting Up Liferay Workspace . . . 231
77.2 Creating New Liferay Workspace . . . 231
77.3 Importing Existing Liferay Workspace . . . 232
77.4 Configuring Liferay Workspace Settings . . . 232
77.5 Configure Workspace Product Key . . . 232
77.6 Initializing Server Bundle . . . 232
77.7 Migrate .cfg Files to .config Files . . . 232

78 Migrating Plugins SDK Projects to Liferay Workspace **233**
78.1 Importing Existing Plugins SDK Projects . . . 233
78.2 Migrating Existing Plugins to Workspace . . . 233

79 Upgrading Build Dependencies **235**
79.1 Updating the Repository URL . . . 235
79.2 Updating the Workspace Plugin Version . . . 236
79.3 Removing Your Project's Build Dependency Versions . . . 236

80 Fixing Upgrade Problems **237**
80.1 Auto-Correcting Upgrade Problems . . . 237
80.2 Finding Upgrade Problems . . . 237
80.3 Resolving Upgrade Problems . . . 238
80.4 Removing Problem Markers . . . 238

81 Resolving a Project's Dependencies **239**
81.1 Class Moved to a Package in the Classpath . . . 239
81.2 Class Moved to a Module Not in the Classpath . . . 240
81.3 Class Replaced or Removed . . . 241

82 Resolving Breaking Changes **243**

83 Upgrading Service Builder Services **245**

84 Removing Legacy Files **247**

85 Converting a Service Builder Module from Spring DI to OSGi DS **249**

86 Rebuilding Services **251**

87 Upgrading Customization Plugins **253**

88 Upgrading Customization Modules **255**

89 Upgrading Core JSP Hooks **257**

90 Upgrading Portlet JSP Hooks **259**

91 Upgrading Service Wrapper Hooks **261**

92 Upgrading Core Language Key Hooks **263**

93 Upgrading Portlet Language Key Hooks **265**

94 Upgrading Model Listener Hooks **267**

95 Upgrading Event Action Hooks **269**

96 Upgrading Servlet Filter Hooks **271**

97 Upgrading Portal Property Hooks **273**

98 Upgrading Struts Action Hooks **275**
98.1 Converting Your Old Wrapper to `MVCCommands` 275
98.2 Mapping Your `MVCCommand` URLs 276

99 Upgrading a Theme to 7.2 **277**

100 Upgrading Your Theme from Liferay Portal 6.2 to 7.2 **279**

101 Setting up the Development Environment **281**

102 Installing the Liferay Theme Generator to Import a 6.2 Theme **283**

103 Importing the Theme into the Liferay JS Theme Toolkit **285**

104 Running the Upgrade Task for 6.2 Themes **287**

105 Updating 6.2 CSS Code **289**

106 Updating 6.2 CSS Rules and Imports **291**

107 Updating the Responsiveness **295**

108 Updating 6.2 Theme Templates **297**

109 Updating 6.2 Portal Normal Theme Template **299**

110 Updating 6.2 Navigation Theme Template **301**

111 Updating 6.2 Init Custom Theme Template **303**

112 Updating the Resources Importer **305**

113 Updating 6.2 Liferay Plugin Package Properties **307**

114 Updating 6.2 Web Content **309**

115 Updating the 6.2 Sitemap **313**

116 Applying Clay Design Patterns **315**

117 Upgrading Your Theme from Liferay Portal 7.0 to 7.2 **317**

118 Running the Upgrade Task for 7.0 Themes **319**

119 Updating 7.0 CSS Code **321**

120 Updating 7.0 CSS File Names for Clay **323**

121 Updating 7.0 Class Variables **325**

122 Updating 7.0 Imports **327**

123 Updating 7.0 Theme Templates to 7.2 **329**

124 Updating 7.0 Theme Templates **331**

125 Using the Bootstrap 3 Lexicon CSS Compatibility Layer **333**

126 Upgrading 7.1 Themes to 7.2 **335**

127 Upgrading a Layout Template to 7.2 **337**

128 Upgrading 6.2 Layout Templates to 7.2 **339**

129 Upgrading 7.0 and 7.1 Layout Templates to 7.2 **341**

130 Upgrading Frameworks and Features **343**

131 Upgrading JNDI Data Source Usage **345**

132 Upgrading Service Builder Service Invocation **347**

133 Upgrading Service Builder **349**

133.1 Step 1: Adapt the Code to 7.0's API 349

133.2Step 2: Resolve Dependencies . . . 350
133.3Step 3: Build the Services . . . 350

134Migrating Off of Velocity Templates **351**

135Upgrading Portlets **353**

136Upgrading a GenericPortlet **355**

137Upgrading a Liferay MVC Portlet **357**

138Upgrading a Liferay JSF Portlet **359**

139Upgrading a Servlet-based Portlet **363**

140Upgrading a Spring Portlet MVC Portlet **365**

141Upgrading a Struts 1 Portlet **367**

142Upgrading Web Plugins **369**

143Upgrading Ext Plugins **371**

144Creating a Theme **373**

145Setting up the Theme **375**

146Customizing the Lunar Resort's Header and Logo **377**

147Customizing the Navigation **381**

148Defining the Lunar Resort's Footer and Footer Navigation **387**

149Adding a Color Scheme Variant for the Lunar Resort Theme **391**

II Customization **397**

150Liferay Customization **399**

151Fundamentals **401**

152Configuring Dependencies **403**

153Finding Artifacts **405**
153.1Finding Core Artifact Attributes . . . 405
153.2Finding Liferay App and Independent Artifacts . . . 407
153.3App Manager . . . 407
153.4Reference Docs . . . 408
153.5Maven Central . . . 410
153.6Related Topics . . . 411

154 Specifying Dependencies **413**
154.1 Related Topics . . . 414

155 Resolving Third Party Library Package Dependencies **415**
155.1 Library Package Resolution Workflow . . . 416
155.2 Embedding Libraries in a Project . . . 417
155.3 Embedding Libraries Using Gradle . . . 417
155.4 Embedding a Library Using Maven . . . 417
155.5 Related Topics . . . 418

156 Understanding Excluded JARs **419**
156.1 Related Topics . . . 420

157 Using the Felix Gogo Shell **421**

158 Importing Packages **423**
158.1 Automatic Package Import Generation . . . 423
158.2 Manually Adding Package Imports . . . 425
158.3 Related Topics . . . 426

159 Exporting Packages **427**
159.1 Related Topics . . . 428

160 Semantic Versioning **429**
160.1 Baselining Your Project . . . 429
160.2 Managing Artifact and Dependency Versions . . . 430
160.3 Related Topics . . . 431

161 Deploying WARs (WAB Generator) **433**
161.1 WAR versus WAB Structure . . . 434
161.2 Deploying a WAR . . . 435
161.3 Saving a Copy of the WAB . . . 435
161.4 Related Topics . . . 435

162 Architecture **437**
162.1 Core . . . 437
162.2 Services . . . 439
162.3 UI . . . 440

163 Liferay Portal Classloader Hierarchy **443**
163.1 Web Application Classloading Perspective . . . 445
163.2 Other Classloading Perspectives . . . 445
163.3 Related Topics . . . 445

164 Liferay DXP Startup Phases **447**
164.1 Portal Context Initialization Phase . . . 447
164.2 Main Servlet Initialization Phase . . . 448
164.3 Acting on Events . . . 448
164.4 ModuleServiceLifecycle Events . . . 448

164.5Portal Startup Events . . . 448
164.6Related Topics . . . 449

165The Benefits of Modularity **451**
165.1Modularity Benefits for Software . . . 452
165.2Distinct Functionality . . . 452
165.3Encapsulation . . . 452
165.4Dependencies . . . 453
165.5Reusability . . . 453
165.6Example: Designing a Modular Application . . . 453

166OSGi and Modularity **455**
166.1Modules . . . 455
166.2API . . . 456
166.3Provider . . . 457
166.4Consumer . . . 458
166.5A Typical Liferay Application . . . 460
166.6Related Topics . . . 460

167Module Lifecycle **461**
167.1Related Topics . . . 463

168UI Architecture **465**
168.1Content . . . 465
168.2Applications . . . 465
168.3Themes . . . 466
168.4Product Navigation Sidebars and Panels . . . 469

169Theme Components **471**
169.1Theme Templates and Utilities . . . 471
169.2CSS Frameworks and Extensions . . . 473
169.3Theme Customizations and Extensions . . . 473
169.4Portlet Customizations and Extensions . . . 473
169.5Related Topics . . . 474

170Understanding the Page Structure **475**
170.1Portlets or Fragments . . . 475
170.2Layout Templates, Page Templates, and Site Templates . . . 476
170.3Product Navigation Sidebars and Panels . . . 478
170.4Related Topics . . . 480

171Bundle Classloading Flow **481**

172Finding Extension Points **483**
172.1Locate the Related Module and Component . . . 483
172.2Finding Extension Points in a Component . . . 484

173Troubleshooting Customizations **487**

174 Why doesn't the package I use from the fragment host resolve? 489

175 Why Aren't JSP overrides I Made Using Fragments Showing? 491
175.1 Related Topics . . . 491

176 Using OSGi Services from EXT Plugins 493
176.1 Related Topics . . . 493

177 Contributing to Liferay Portal 495
177.1 Building Liferay Portal from source . . . 495
177.2 Tooling . . . 495
177.3 Additional Resources . . . 496

178 Model Listeners 497
178.1 Creating a Model Listener Class . . . 498
178.2 Register the Model Listener Service . . . 498
178.3 Listening For Persistence Events . . . 498
178.4 Related Topics . . . 499

179 Customizing JSPs 501
179.1 Using Liferay DXP's API to Override a JSP . . . 501
179.2 Overriding a JSP Without Using Liferay DXP's API . . . 501

180 Customizing JSPs with Dynamic Includes 503

181 JSP Overrides Using Portlet Filters 507

182 JSP Overrides Using OSGi Fragments 511
182.1 Declaring a Fragment Host . . . 511
182.2 Provide the Overridden JSP . . . 512
182.3 Using Fragment Host Internal Packages . . . 513
182.4 Related Topics . . . 514

183 JSP Overrides Using Custom JSP Bag 515
183.1 Providing a Custom JSP . . . 515
183.2 Implement a Custom JSP Bag . . . 516
183.3 Extend a JSP . . . 518
183.4 Site Scoped JSP Customization . . . 518
183.5 Related Topics . . . 518

184 Overriding Inline Content Using JSPs 519
184.1 Example: Overriding the fieldset Taglib Tag . . . 520
184.2 Related Topics . . . 523

185 Customizing Widgets 525
185.1 Implementing the TemplateHandler Interface . . . 526
185.2 Defining Display Template Definitions . . . 526
185.3 Defining Permissions . . . 526
185.4 Exposing the Widget Template Selection . . . 526
185.5 Recommendations for Using Widget Templates . . . 527

186 Implementing Widget Templates **529**

187 Dynamic Includes **537**

188 WYSIWYG Editor Dynamic Includes **539**
188.1 Related Topics . . . 540

189 Top Head JSP Dynamic Includes **541**
189.1 Related Topics . . . 542

190 Top JS Dynamic Include **543**
190.1 Related Topics . . . 544

191 Bottom JSP Dynamic Includes **545**
191.1 Related Topics . . . 546

192 Waiting on Lifecycle Events **547**
192.1 Taking action from a component . . . 547
192.2 Taking action from a non-component class . . . 548
192.3 Related Topics . . . 549

193 Liferay Forms **551**
193.1 Liferay Forms Extension Points . . . 551

194 Form Storage Adapters **553**
194.1 Storage Adapter Methods . . . 553
194.2 The CRUD Methods . . . 554
194.3 Validating Form Entries . . . 555
194.4 Enabling the Storage Adapter . . . 556

195 Creating a Form Storage Adapter **557**
195.1 Storage Adapter CRUD Operations . . . 557
195.2 Create . . . 557
195.3 Read . . . 558
195.4 Update . . . 559
195.5 Delete . . . 559
195.6 Beyond CRUD: Validation . . . 560

196 Overriding Language Keys **561**

197 Overriding Global Language Keys **563**
197.1 Determine the language keys to override . . . 563
197.2 Override the keys in a new language properties file . . . 564
197.3 Create a Resource Bundle service component . . . 564
197.4 Related Topics . . . 566

198 Overriding a Module's Language Keys **567**
198.1 Find the module and its metadata and language keys . . . 567
198.2 Write custom language key values . . . 568
198.3 Prioritize Your Module's Resource Bundle . . . 569

198.4Related Topics . . . 570

199Overriding Liferay Services (Service Wrappers) 571
199.1Related Topics . . . 572

200Overriding lpkg Files 573

201Overriding Liferay MVC Commands 575

202Adding Logic to MVC Commands 577
202.1Step 1: Implement the interface . . . 577
202.2Step 2: Publish as a component . . . 578
202.3Step 3: Refer to the original implementation . . . 578
202.4Step 4: Add the logic . . . 579

203Overriding MVCRenderCommands 581
203.1Adding Logic to an Existing MVC Render Command . . . 582
203.2Redirecting to a New JSP . . . 582
203.3Related Topics . . . 584

204Overriding MVCActionCommands 585
204.1Related Topics . . . 586

205Overriding MVCResourceCommands 587
205.1Related Topics . . . 588

206Overriding OSGi Services 589

207Examining an OSGi Service to Override 591
207.1Gathering Information on a Service . . . 591
207.2Step 1: Copy the Service Interface Name . . . 592
207.3Step 2: Copy the Existing Service Name . . . 592
207.4Step 3: Gather Reference Configuration Details (if reconfiguration is needed) . . . 593
207.5Related Topics . . . 594

208Creating a Custom OSGi Service 595
208.1Related Topics . . . 596

209Reconfiguring Components to Use Your OSGi Service 597
209.1Reconfiguring the Service Reference . . . 598
209.2Related Topics . . . 599

210Portlet Filters 601
210.1Sample Portlet . . . 602
210.2Render filter 1 hides parts of user email addresses . . . 602
210.3RenderFilter 2 Logs Statistics . . . 604
210.4Related Topics . . . 605

211Product Navigation 607
211.1Product Menu . . . 608

211.2Control Menu . . . 608
211.3Simulation Menu . . . 609
211.4User Personal Menu . . . 610

212Customizing the Product Menu **613**
212.1PanelCategory Interface . . . 613
212.2BasePanelCategory . . . 614
212.3BaseJSPPanelCategory . . . 614
212.4PanelApp Interface . . . 614

213Adding Custom Panel Categories **617**
213.1Creating the OSGi Module . . . 617
213.2Implementing Liferay's Frameworks . . . 617

214Adding Custom Panel Apps **621**

215Customizing the Control Menu **623**
215.1ProductNavigationControlMenuEntry Interface . . . 623
215.2BaseProductNavigationControlMenuEntry . . . 624
215.3BaseJSPProductNavigationControlMenuEntry . . . 624

216Creating Control Menu Entries **625**
216.1Creating the OSGi Module . . . 625
216.2Implementing Liferay's Frameworks . . . 625

217Defining Icons and Tooltips **629**
217.1Control Menu Entry Icons . . . 629
217.2Control Menu Entry Tooltips . . . 630

218Extending the Simulation Menu **631**

219Customizing the User Personal Bar and Menu **635**
219.1Displaying the Personal Menu . . . 636

220Using a Custom Portlet in Place of the User Personal Bar **637**
220.1Related Topics . . . 638

221Customizing the Personal Menu **639**
221.1Adding an Entry to the Personal Menu . . . 639
221.2Adding a Portlet Entry to the Personal Menu . . . 641
221.3Related Topics . . . 642

222Customizing Workflow **643**

223Creating SLA Calendars **645**
223.1Dependencies . . . 646
223.2Implementation Steps . . . 646

224Customizing Core Functionality with Ext **649**

225 Extending Core Classes Using Spring with Ext Plugins **651**

226 Overriding Core Classes with Ext Plugins **653**

227 Adding to the web.xml with Ext Plugins **655**

228 Modifying the web.xml with Ext Plugins **657**

III Application Development Platform **659**

229 Application Development **661**
229.1 Getting Started with Liferay Development . . . 662
229.2 Create Your Object Model and Database in One Shot . . . 662
229.3 Create a REST Interface . . . 663
229.4 Create a Web Client . . . 663
229.5 Use Liferay's Frameworks . . . 663
229.6 Next Steps . . . 663

230 Developing Web Front-Ends **665**
230.1 Using Popular Frameworks . . . 665
230.2 Getting Started . . . 666

231 Developing an Angular Application **669**
231.1 Related Topics . . . 673

232 Developing a React Application **675**
232.1 Related Topics . . . 678

233 Developing a Vue Application **679**
233.1 Related Topics . . . 683

234 Liferay MVC Portlet **685**
234.1 MVC Layers and Modularity . . . 685
234.2 Liferay MVC Command Classes . . . 686
234.3 Liferay MVC Portlet Component . . . 686
234.4 A Simpler MVC Portlet . . . 687

235 Creating an MVC Portlet **689**
235.1 Related Topics . . . 691

236 Writing MVC Portlet Controller Code **693**
236.1 Action Methods . . . 693
236.2 Render Logic . . . 694
236.3 Setting and Retrieving Request and Response Parameters and Attributes . . . 695
236.4 Related Topics . . . 696

237 Configuring the View Layer **697**
237.1 Using the init.jsp . . . 697
237.2 Using Render URLs . . . 698

237.3Using Action URLs . . . 699
237.4Related Topics . . . 699

238MVC Action Command **701**
238.1Related Topics . . . 703

239MVC Render Command **705**
239.1Related Topics . . . 706

240MVC Resource Command **707**
240.1Related Topics . . . 709

241PortletMVC4Spring **711**

242Developing a Portlet Using PortletMVC4Spring **715**
242.1Related Topics . . . 719

243Migrating to PortletMVC4Spring **721**
243.1Related Topics . . . 722

244JSF Portlet **723**

245Developing a JSF Portlet Application **725**

246Bean Portlet **729**
246.1Portlet Configuration Annotations . . . 729
246.2Dependency Injection . . . 730
246.3Portlet Phase Methods . . . 731

247Creating a Bean Portlet **733**
247.1Related Topics . . . 735

248Service Builder **737**
248.1Customization via Implementation Classes . . . 738
248.2Hibernate Configurations . . . 738
248.3Caching . . . 738
248.4Dynamic Query and Custom SQL Query . . . 738

249Creating a Service Builder Project **741**
249.1Related Topics . . . 742

250Creating the service.xml File **743**

251Defining Global Service Information **745**
251.1Dependency Injector . . . 745
251.2Package Path . . . 746
251.3Multiversion concurrency control (MVCC) . . . 746
251.4Namespace Options . . . 747
251.5Author . . . 747

252 Defining Service Entities **749**

253 Defining the Columns (Attributes) for Each Service Entity **751**
253.1 Create Entity Columns . . . 751
253.2 Support Multi-tenancy . . . 752
253.3 Workflow Fields . . . 752
253.4 Audit Entities . . . 752

254 Defining Relationships Between Service Entities **753**

255 Defining Ordering of Service Entity Instances **755**

256 Defining Service Entity Finder Methods **757**
256.1 Creating Finders . . . 757

257 Running Service Builder **759**
257.1 Gradle . . . 759
257.2 Maven . . . 759

258 Understanding the Code Generated by Service Builder **761**

259 Iterative Development **767**
259.1 Related Topics . . . 767

260 Customizing Model Entities With Model Hints **769**
260.1 Model Hint Types . . . 771
260.2 Default Hints . . . 771
260.3 Hint Collections . . . 772

261 Configuring service.properties **775**
261.1 Related Topics . . . 775

262 Connecting Service Builder to an External Database **777**

263 Connecting the Data Source Using a DataSourceProvider **779**
263.1 Related Topics . . . 781

264 Connecting the Data Source Using Spring Beans **783**
264.1 Specify Your Database and a Data Source Name in Your `service.xml` . . . 784
264.2 Create the Database Manually . . . 784
264.3 Define the Data Source . . . 784
264.4 Connect Your Service Builder Module to the Data Source Via a Spring Bean . . . 784
264.5 Run Service Builder . . . 786
264.6 Related Topics . . . 786

265 Migrating a Service Builder Module from Spring DI to OSGi DS **787**
265.1 Step 1: Prepare Your Project for DS . . . 787
265.2 Step 2: Update Your Spring Bean Classes . . . 788
265.3 Step 3: Resolve Any Circular Dependencies . . . 789
265.4 Related Topics . . . 790

266 Business Logic with Service Builder **791**

267 Implementing an Add Method **793**
267.1 Step 1: Declare an add method with parameters for creating the entity . . . 794
267.2 Step 2: Validate the parameters . . . 795
267.3 Step 3: Generate a primary key . . . 795
267.4 Step 4: Create an entity instance . . . 795
267.5 Step 5: Populate the entity attributes . . . 796
267.6 Step 6: Persist the entity . . . 796
267.7 Step 7: Return the entity . . . 796

268 Implementing Update and Delete Methods **797**
268.1 Implementing an Update Method . . . 797
268.2 Step 1: Declare an Update Method with Parameters for Updating the Entity . . . 798
268.3 Step 2: Validate the Parameters . . . 798
268.4 Step 3: Retrieve the Entity Instance . . . 799
268.5 Step 4: Update the Entity Attributes . . . 799
268.6 Step 5: Persist and Return the Updated Entity Instance . . . 799
268.7 Step 6: Run Service Builder . . . 799
268.8 Implementing a Delete Method . . . 799
268.9 Related Topics . . . 800

269 Implementing Methods to Get and Count Entities **801**
269.1 Getter Methods . . . 801
269.2 Counter Methods . . . 802
269.3 Service Method Prefixes and Transactional Aspects . . . 803
269.4 Related Topics . . . 804

270 Implementing Any Other Business Logic **805**

271 Integrating with Liferay's Frameworks **807**
271.1 Related Topics . . . 808

272 Invoking Local Services **809**
272.1 Step 1: Reference the Local Service Component . . . 809
272.2 Step 2: Call the Component's Methods . . . 810
272.3 Related Topics . . . 811

273 Invoking Services from Spring Service Builder Code **813**
273.1 Referencing a Spring Bean that is in the Application Context . . . 814
273.2 Referencing OSGi Services . . . 814
273.3 Related Topics . . . 815

274 Advanced Queries **817**

275 Custom SQL **819**
275.1 Specify Your Custom SQL . . . 819
275.2 Related Topics . . . 820

276 Defining a Custom Finder Method **821**

277 Dynamic Query **823**
277.1 Example Finder Method: findByGuestbookNameEntryName . . . 823
277.2 Using a Hibernate Session . . . 825
277.3 Creating Dynamic Queries . . . 826
277.4 Restriction Criteria . . . 826
277.5 Projection Criteria . . . 827
277.6 Order Criteria . . . 827
277.7 Executing the Dynamic Query . . . 828

278 Accessing Your Custom Finder Method from the Service Layer **829**
278.1 Related Topics . . . 830

279 Actionable Dynamic Queries **831**
279.1 Related Topics . . . 833

280 REST Builder **835**

281 Generating APIs with REST Builder **837**
281.1 Related Topics . . . 838

282 Troubleshooting Application Development Issues **839**
282.1 Modules . . . 839
282.2 Services and Components . . . 840

283 Adjusting Module Logging **841**
283.1 Related Topics . . . 842

284 Identifying Liferay Artifact Versions for Dependencies **843**
284.1 Related Topics . . . 843

285 Resolving Bundle-SymbolicName Syntax Issues **845**
285.1 Related Topics . . . 845

286 Calling Non-OSGi Code that Uses OSGi Services **847**
286.1 Related Topics . . . 847

287 Connecting to JNDI Data Sources **849**

288 Detecting Unresolved OSGi Components **851**
288.1 Declarative Services Components . . . 851
288.2 Declarative Services Unsatisfied Component Scanner . . . 852
288.3 ds:unsatisfied Command . . . 852
288.4 Service Builder Components . . . 853
288.5 Unavailable Component Scanner . . . 853
288.6 dm na Command . . . 854
288.7 ServiceProxyFactory . . . 854
288.8 Related Topics . . . 854

289 Disabling Cache for Table Mapper Tables **855**
289.1 Why would I want to disable cache on a table mapper? . . . 855
289.2 Disabling a Table Mapper Cache . . . 856

290 Implementing Logging **857**
290.1 Related Topics . . . 858

291 Declaring Optional Import Package Requirements **859**
291.1 Related Topics . . . 860

292 Resolving Bundle Requirements **861**
292.1 Related Topics . . . 862

293 Resolving ClassNotFoundException and NoClassDefFoundError in OSGi Bundles **863**
293.1 Case 1: The Missing Class Belongs to an OSGi Module . . . 863
293.2 Case 2: The Missing Class Doesn't Belong to an OSGi Module . . . 864
293.3 Case 3: The Missing Class Belongs to a Global Library . . . 864
293.4 Case 4: The Missing Class Belongs to a Java Runtime Package . . . 865
293.5 Related Topics . . . 865

294 System Check **867**
294.1 Related Topics . . . 867

295 Troubleshooting Front-End Development Issues **869**
295.1 CSS . . . 869
295.2 Modules . . . 869
295.3 Portlets . . . 870

IV Liferay Frameworks **871**

296 Application Security **873**

297 Defining Application Permissions **875**

298 Defining Resources and Permissions **877**
298.1 Defining Portlet Resource Permissions . . . 877
298.2 Defining Model Resource Permissions . . . 879
298.3 Enabling Your Permissions Configuration . . . 880

299 Registering Permissions **881**
299.1 Registering Permissions Resources in the Database . . . 881
299.2 Registering Entities to the Permissions Service . . . 882

300 Associating Permissions with Resources **885**

301 Checking Permissions **887**
301.1 Add Permission Checks to Your Service Calls . . . 887
301.2 Create Permission Helper Classes in Your Web Module . . . 888
301.3 Add Permission Checks to Your Web Application . . . 890

302 Using JSR Roles in a Portlet **893**
302.1 JSR Portlet Security . . . 893
302.2 Mapping Portlet Roles to Portal Roles . . . 894
302.3 Related Topics . . . 895

303 Authentication Pipelines **897**

304 Auto Login **899**
304.1 Creating an Auto Login Component . . . 899
304.2 Related Topics . . . 900

305 Password-Based Authentication Pipelines **901**
305.1 Anatomy of an Authenticator . . . 901
305.2 Creating an Authenticator . . . 902
305.3 Related Topics . . . 907

306 Writing a Custom Login Portlet **909**
306.1 Authenticating to Liferay DXP . . . 909
306.2 Related Topics . . . 911

307 Service Access Policies **913**
307.1 How Service Access Policies Work . . . 913
307.2 API Overview . . . 915
307.3 Service Access Policy Example . . . 916

308 Frameworks **919**

309 Asset Framework **921**
309.1 Persistence Operations for Assets . . . 921
309.2 Rendering an Asset . . . 922
309.3 Asset Features . . . 922
309.4 Tags and Categories . . . 923
309.5 Relating Assets . . . 923
309.6 Implementing Asset Priority . . . 924

310 Adding, Updating, and Deleting Assets **925**
310.1 Preparing Your Project for the Asset Framework . . . 925
310.2 Adding and Updating Assets . . . 925
310.3 Deleting Assets . . . 927

311 Creating an Asset Renderer **929**

312 Configuring JSP Templates for an Asset Renderer **935**

313 Creating a Factory for the Asset Renderer **939**

314 Implementing Asset Categorization and Tagging **945**
314.1 Adding Tags and Categories . . . 945
314.2 Displaying Tags and Categories . . . 945

315 Relating Assets **947**
315.1 Relating Assets in the Service Layer . . . 947
315.2 Relating Assets in the UI . . . 948
315.3 Showing Related Assets . . . 949

316 Implementing Asset Priority **951**
316.1 Add the Priority Field to Your JSP . . . 951
316.2 Using the Priority Value in Your Service Layer . . . 951

317 Back-end Frameworks **953**
317.1 Portlet Providers . . . 953
317.2 Portlet Provider Classes . . . 953
317.3 Data Scopes . . . 954
317.4 Accessing the Site Scope Across Apps . . . 954
317.5 Message Bus . . . 955

318 Creating Portlet Providers **957**
318.1 Related Topics . . . 958

319 Retrieving Portlets **959**
319.1 Fetching a Portlet ID . . . 959
319.2 Fetching a Portlet URL . . . 960
319.3 Related Topics . . . 961

320 Enabling and Accessing Data Scopes **963**
320.1 Enabling Scoping . . . 963
320.2 Accessing Your App's Scope . . . 964
320.3 Accessing the Site Scope . . . 964
320.4 Related Topics . . . 965

321 Using the Message Bus **967**
321.1 Messaging Destinations . . . 967
321.2 Destination Configuration . . . 967
321.3 Message Listeners . . . 968
321.4 Sending Messages . . . 969

322 Creating a Destination **971**
322.1 Related Topics . . . 975

323 Message Bus Event Listeners **977**
323.1 Listening for Destinations . . . 977
323.2 Listening for Message Listeners . . . 977
323.3 Related Topics . . . 978

324 Registering Message Listeners **979**
324.1 Automatic Registration as a Component . . . 979
324.2 Registering via a MessageBus Reference . . . 980
324.3 Registering Directly to the Destination . . . 980
324.4 Related Topics . . . 981

325 Creating a Message **983**
325.1 Related Topics . . . 983

326 Sending a Message **985**
326.1 Directly with MessageBus . . . 985
326.2 Asynchronously with SingleDestinationMessageSender . . . 986
326.3 Synchronously with SynchronousMessageSender . . . 987
326.4 Related Topics . . . 988

327 Sending Messages Across a Cluster **989**
327.1 Related Topics . . . 990

328 Cache Configuration **991**
328.1 Cache Types . . . 991
328.2 Cache Configuration . . . 992
328.3 Initial Global Cache Configuration . . . 992
328.4 Module Cache Configuration . . . 992
328.5 Portlet WAR Cache Configuration . . . 993
328.6 Cache Names and Registration . . . 993
328.7 EntityCache Names . . . 993
328.8 FinderCache Names . . . 994

329 Overriding Cache **995**
329.1 Related Topics . . . 997

330 Caching Data **999**
330.1 Step 1: Determine Cache Pool Requirements . . . 999
330.2 Step 2: Implement a Cache Key . . . 999
330.3 Step 3: Implement Cache Logic . . . 1001
330.4 Step 4: Configure the Cache . . . 1002
330.5 Related Topics . . . 1003

331 Collaboration **1005**

332 Item Selector **1007**

333 Adaptive Media **1009**

334 Social API **1011**

335 Documents and Media API **1013**

336 Item Selector **1015**

337 Understanding the Item Selector API's Components **1017**

338 Getting an Item Selector **1019**

339 Understanding Custom Selection Views **1021**
339.1 The Selection View's Class . . . 1021

340 Selecting Entities with an Item Selector **1023**
340.1 Get an Item Selector . . . 1023
340.2 Using the Item Selector Dialog . . . 1024
340.3 Related Topics . . . 1027

341 Creating Custom Criterion and Return Types **1029**
341.1 Related Topics . . . 1030

342 Creating Custom Item Selector Views **1031**
342.1 Configuring Your Selection View's OSGi Module . . . 1031
342.2 Implementing Your Selection View's Class . . . 1032
342.3 Writing Your View Markup . . . 1034
342.4 Related Topics . . . 1037

343 Documents and Media API **1039**

344 Getting Started with the Documents and Media API **1041**

345 Key Interfaces **1043**

346 Getting a Service Reference **1045**

347 Specifying Repositories **1047**

348 Specifying Folders **1049**

349 Creating Files, Folders, and Shortcuts **1051**

350 Files **1053**

351 Folders **1055**
351.1 Folders and External Repositories . . . 1055

352 File Shortcuts **1057**

353 Creating Files **1059**
353.1 Related Topics . . . 1060

354 Creating Folders **1061**
354.1 Related Topics . . . 1062

355 Creating File Shortcuts **1063**
355.1 Related Topics . . . 1064

356 Deleting Entities **1065**

357 Files **1067**

358 File Versions **1069**
358.1 Identifying File Versions . . . 1069

359 File Shortcuts **1071**

360 Folders **1073**

361 Recycle Bin **1075**

362 Deleting Files **1077**
362.1 Related Topics 1078

363 Deleting File Versions **1079**
363.1 Related Topics 1080

364 Deleting File Shortcuts **1081**
364.1 Related Topics 1081

365 Deleting Folders **1083**
365.1 Related Topics 1084

366 Moving Entities to the Recycle Bin **1085**
366.1 Related Topics 1085

367 Updating Entities **1087**

368 Files **1089**

369 Folders **1091**

370 File Shortcuts **1093**

371 Updating Files **1095**
371.1 Related Topics 1096

372 Updating Folders **1097**
372.1 Related Topics 1098

373 Updating File Shortcuts **1099**
373.1 Related Topics 1100

374 File Checkout and Checkin **1101**

375 File Checkout **1103**
375.1 Fine-tuning Checkout 1103

376 File Checkin **1105**

377 Canceling a Checkout **1107**

378 Checking Out Files **1109**
378.1 Related Topics 1109

379 Checking In Files **1111**

379.1Related Topics 1112

380Canceling a Checkout **1113**
380.1Related Topics 1113

381Copying and Moving Entities **1115**

382Copying Folders **1117**

383Moving Folders and Files **1119**

384Copying Folders **1121**
384.1Related Topics 1122

385Moving Folders and Files **1123**
385.1Related Topics 1124

386Getting Entities **1125**

387Files **1127**

388Folders **1129**

389Multiple Entity Types **1131**

390Getting Files **1133**
390.1Related Topics 1134

391Getting Folders **1135**
391.1Related Topics 1136

392Getting Multiple Entity Types **1137**
392.1Related Topics 1138

393Adaptive Media **1139**

394The Adaptive Media Taglib **1141**

395Adaptive Media's Finder API **1143**
395.1Calling the API 1143
395.2Adaptive Media API Constants 1144
395.3Approximate Attributes 1144

396Image Scaling in Adaptive Media **1145**

397Displaying Adapted Images in Your App **1147**
397.1Related Topics 1148

398Finding Adapted Images **1151**
398.1Getting Adapted Images for File Versions 1151
398.2Getting the Adapted Images for a Specific Image Resolution 1152

398.3Getting Adapted Images in a Specific Order 1152
398.4Using Approximate Attributes 1153
398.5Using the Adaptive Media Stream 1153
398.6Related Topics 1154

399Creating an Image Scaler **1155**
399.1Related Topics 1157

400Social API **1159**

401Social Bookmarks **1161**

402Ratings **1163**
402.1Rating Type Selection 1163
402.2Rating Value Transformation 1164

403Applying Social Bookmarks **1165**
403.1Related Topics 1166

404Creating Social Bookmarks **1167**
404.1Implementing the SocialBookmark Interface 1167
404.2Creating Your JSP 1168
404.3Related Topics 1169

405Adding Comments to Your App **1171**
405.1Related Topics 1172

406Rating Assets **1173**
406.1Related Topics 1174

407Implementing Rating Type Selection **1175**
407.1Related Topics 1175

408Customizing Rating Value Transformation **1177**
408.1Related Topics 1178

409Flagging Inappropriate Asset Content **1179**
409.1Related Topics 1180

410Configurable Applications **1181**
410.1Using a Configuration Interface 1181
410.2Reading Configuration Values 1183
410.3Further Customization 1183

411Creating A Configuration Interface **1185**

412Categorizing the Configuration **1187**
412.1Specifying a Configuration Category 1187
412.2Creating New Sections and Categories 1188

413 Scoping Configurations **1191**
413.1 Step 1: Setting the Configuration Scope . . . 1191
413.2 Step 2: Enabling the Configuration for Scoped Retrieval . . . 1191

414 Reading Scoped Configuration Values **1193**
414.1 Using the Configuration Provider . . . 1193
414.2 Accessing the Portlet Instance Configuration Through the `PortletDisplay` . . . 1194

415 Reading Unscoped Configuration Values from an MVC Portlet **1197**
415.1 Accessing the Configuration Object in the Portlet Class . . . 1197
415.2 Accessing the Configuration from a JSP . . . 1198
415.3 Accessing the Configuration from the Portlet Class . . . 1199

416 Reading Unscoped Configuration Values from a Component **1201**

417 Customizing the Configuration User Interface **1203**
417.1 Providing Custom Configuration Forms . . . 1203
417.2 Creating a Completely Custom Configuration UI . . . 1204
417.3 Excluding a Configuration UI . . . 1206
417.4 Using `generateUI` . . . 1206
417.5 Using the Configuration Visibility SPI . . . 1206

418 Configuration Form Renderer **1209**
418.1 Creating a `DisplayContext` . . . 1209
418.2 Implementing a `ConfigurationFormRenderer` . . . 1210
418.3 Writing the JSP Markup . . . 1213

419 Using DDM Form Annotations in Configuration Forms **1215**
419.1 Step 1: Declare the Dependencies . . . 1215
419.2 Step 2: Write the Configuration Form . . . 1216
419.3 Step 3: Write the Form Declaration . . . 1218

420 Upgrading a Legacy App **1219**

421 Dynamically Populating Select List Fields in the Configuration UI **1221**

422 Content Publication Management **1225**
422.1 Export/Import . . . 1225
422.2 Staging . . . 1226

423 Export/Import **1227**
423.1 Staged Models . . . 1227
423.2 Data Handlers . . . 1228
423.3 Provide Entity Specific Local Services . . . 1228
423.4 Export/Import Event Listeners . . . 1228
423.5 Export/Import Processes . . . 1230

424 Developing Staged Models **1233**
424.1 Staged Model Interfaces . . . 1233
424.2 Staged Model Attributes . . . 1234

424.3Adapting Your Business Logic to Build Staged Models 1234

425Generating Staged Models Using Service Builder 1237

426Creating Staged Models Manually 1239

427Developing Data Handlers 1243
427.1Understanding the `PortletDataHandler` Interface 1244
427.2Understanding the `StagedModelDataHandler` Interface 1245

428Creating Portlet Data Handlers 1247

429Creating Staged Model Data Handlers 1253

430Providing Entity-Specific Local Services for Export/Import 1257
430.1Understanding the `StagedModelRepository` Interface 1257
430.2Using a Staged Model Repository 1258

431Implementing the Staged Model Repository Framework 1261

432Using the Staged Model Repository Framework 1263

433Using the Export/Import Lifecycle Listener Framework 1265

434Initiating New Export/Import Processes 1269

435Staging 1271
435.1Controlling Staging's UI Settings 1271
435.2Filtering Staging-Specific Processes and States 1272

436Dependency Injection 1275

437CDI Dependency Injection 1277
437.1Related Topics 1279

438OSGi CDI Integration 1281
438.1Use Case: Registering a CDI bean as an OSGi service 1281
438.2Use Case: Using an OSGi service in a bean 1282

439Publishing CDI Beans as OSGi Services 1285

440Using OSGi Services in a Bean 1287

441Declarative Services 1289

442Service Trackers for OSGi Services 1291

443Using a Service Tracker 1293
443.1Creating a New Service Tracker Where You Need It 1293
443.2Create a Class That Extends ServiceTracker 1294
443.3Creating a Service Tracker that Tracks Service Events Using a Callback Handler . . 1295

444 Friendly URLs **1297**

445 Friendly URLs **1299**
445.1 Related Topics . . . 1302

446 Front-End Development **1303**
446.1 Lexicon and Clay . . . 1303
446.2 Templates . . . 1304
446.3 Themes . . . 1304
446.4 Front-End Extensions . . . 1304

447 Themes **1305**

448 Theme Workflow **1307**

449 Developing Themes **1309**

450 Using Developer Mode with Themes **1311**
450.1 Enabling Developer Mode Manually . . . 1311
450.2 Setting Developer Mode in Dev Studio DXP . . . 1311
450.3 Configuring FreeMarker System Settings . . . 1312
450.4 JavaScript Fast Loading . . . 1313
450.5 Related Topics . . . 1313

451 Building Your Theme's Files **1315**
451.1 Related Topics . . . 1315

452 Deploying and Applying Themes **1317**
452.1 Related Topics . . . 1318

453 Updating Your Theme's App Server **1319**
453.1 Related Topics . . . 1320

454 Automatically Deploying Theme Changes **1321**
454.1 Related Topics . . . 1321

455 Creating a Thumbnail Preview for Your Theme **1323**
455.1 Related Topics . . . 1325

456 Creating Color Schemes for Your Theme **1327**
456.1 Related Topics . . . 1329

457 Making Configurable Theme Settings **1331**
457.1 Related Topics . . . 1333

458 Using Font Awesome and Glyph Icons in Your Theme **1335**
458.1 Disabling Enabling Global Font Awesome and Glyphicons in Portal . . . 1335
458.2 Including Font Awesome and Glyphicons in Your Theme . . . 1335
458.3 Related Topics . . . 1336

459 Extending Themes **1337**

460 Installing a Themelet in Your Theme **1339**
460.1 Related Topics . . . 1340

461 Injecting Additional Context Variables and Functionality into Your Theme Templates **1341**
461.1 Related Topics . . . 1342

462 Packaging Independent UI Resources for Your Site **1343**
462.1 Related Topics . . . 1344

463 Changing Your Base Theme **1345**
463.1 Related Topics . . . 1346

464 Copying an Existing Theme's Files **1347**
464.1 Related Topics . . . 1348

465 Listing Your Theme's Extensions **1349**
465.1 Related Topics . . . 1349

466 Overwriting and Extending Liferay Theme Tasks **1351**
466.1 Related Topics . . . 1353

467 Clay CSS and Themes **1355**

468 Customizing Atlas and Clay Base Themes in Liferay DXP **1357**
468.1 Related Topics . . . 1358

469 Integrating Third Party Themes with Clay **1359**
469.1 Related Topics . . . 1360

470 Using Clay Icons in a Theme **1361**
470.1 Related Topics . . . 1361

471 Using Clay Mixins in Your Theme **1363**
471.1 Related Topics . . . 1363

472 Theming Portlets **1365**
472.1 Portlet Decorators . . . 1366

473 Embedding Portlets in Themes **1371**

474 Embedding Portlets in Themes by Entity Type and Action **1373**
474.1 Related Topics . . . 1375

475 Embedding a Portlet by Portlet Name **1377**
475.1 Related Topics . . . 1377

476 Setting Default Preferences for an Embedded Portlet **1379**
476.1 Related Topics . . . 1380

477 Importing Resources with a Theme **1381**
477.1 Organizing Your Resources . . . 1381

478 Creating a Sitemap for the Resources Importer **1383**

479 Defining Layout Templates and Pages in a Sitemap **1387**
479.1 Related Topics . . . 1389

480 Defining Portlets in a Sitemap **1391**
480.1 Related Topics . . . 1393

481 Retrieving Portlet IDs with the Gogo Shell **1395**
481.1 Related Topics . . . 1395

482 Preparing and Organizing Web Content for the Resources Importer **1397**
482.1 Related Topics . . . 1399

483 Defining Assets for the Resources Importer **1401**
483.1 Related Topics . . . 1402

484 Specifying Where to Import Your Theme's Resources **1403**
484.1 Related Topics . . . 1404

485 Archiving Site Resources **1405**
485.1 Related Topics . . . 1405

486 Troubleshooting Themes **1407**

487 Layout Templates **1409**

488 Creating Custom Layout Template Thumbnail Previews **1411**
488.1 Related topics . . . 1412

489 Including Layout Templates with a Theme **1413**
489.1 Related topics . . . 1414

490 Creating and Bundling JavaScript Widgets with JavaScript Tooling **1415**

491 Configuring System Settings and Instance Settings for Your JavaScript Widget **1417**
491.1 Related Topics . . . 1418

492 Localizing Your Widget **1419**
492.1 Related Topics . . . 1420

493 Using Translation Features in Your Widget **1421**
493.1 Related Topics . . . 1421

494 Configuring Portlet Properties for Your Widget **1423**
494.1 Related Topics . . . 1423

495 JavaScript Module Loaders **1425**

496 Loading AMD Modules in Liferay **1427**
496.1 Related Topics . . . 1428

497 Using External JavaScript Libraries **1429**
497.1 Related Topics . . . 1430

498 Loading Modules with AUI Script **1431**

499 Loading AlloyUI Modules with AUI Script **1433**
499.1 Related Topics . . . 1434

500 Loading ES2015 and Metal.js Modules with AUI Script **1435**
500.1 Related Topics . . . 1436

501 Loading AUI, ES2015, and Metal.js Modules Together with AUI Script **1437**
501.1 Related Topics . . . 1437

502 Loading Bundled npm Modules in Your Portlets **1439**
502.1 Related Topics . . . 1440

503 The Info Framework **1441**
503.1 Using the Info Framework . . . 1441
503.2 List Providers . . . 1442
503.3 Item Renderers . . . 1442

504 Creating an Information List Provider **1443**
504.1 Next steps . . . 1445

505 Custom rendering of information with `InfoItemRenderer` **1447**

506 Using Providers with Custom Applications **1449**
506.1 Leveraging renderers from a custom application . . . 1449

507 Liferay Forms **1451**

508 Form Serialization with the DDM IO API **1453**

509 Serializing Forms **1455**
509.1 Calling the Serializer . . . 1456

510 Localization **1457**

511 Localizing Your Application **1459**
511.1 Related Topics . . . 1461

512 Using Liferay's Localization Settings **1463**
512.1 Localizing User Names . . . 1464
512.2 Identifying User Initials . . . 1466
512.3 Right to Left or Left to Right? . . . 1466
512.4 Related Topics . . . 1467

513 Creating a Language Module **1469**
513.1 Related Topics . . . 1470

514 Using a Language Module **1471**
514.1 Using a Language Module from a Module . . . 1471
514.2 Using a Language Module from a Traditional Plugin . . . 1472
514.3 Related Topics . . . 1473

515 Automatically Generating Translations **1475**
515.1 Configuring the Language Builder Plugin . . . 1475
515.2 Running Language Builder . . . 1476
515.3 Translating Language Keys Automatically . . . 1477
515.4 Related Topics . . . 1478

516 Portlets **1479**
516.1 Related Topics . . . 1481

517 Using JavaScript in Your Portlets **1483**

518 Using ES2015 Modules in your Portlet **1485**
518.1 Related Topics . . . 1486

519 Using npm in Your Portlets **1487**

520 Formatting Your npm Modules for AMD **1489**
520.1 Related Topics . . . 1490

521 Migrating a liferay-npm-bundler Project from 1.x to 2.x **1491**

522 Migrating a Plain JavaScript, Billboard JS, JQuery, Metal JS, React, or Vue JS Project to Use Bundler 2.x **1493**
522.1 Related Topics . . . 1494

523 Migrating an Angular Project to Use Bundler 2.x **1495**
523.1 Related Topics . . . 1496

524 Migrating Your Project to Use liferay-npm-bundler's New Mode **1497**
524.1 Related Topics . . . 1498

525 Creating Custom Loaders for the liferay-npm-bundler **1499**
525.1 Related Topics . . . 1501

526 Using the NPMResolver API in Your Portlets **1503**

527 Referencing an npm Module's Package to Improve Code Maintenance **1505**
527.1 Related Topics . . . 1506

528 Obtaining an OSGi bundle's Dependency npm Package Descriptors **1507**
528.1 Related Topics . . . 1508

529 Automatic Single Page Applications **1509**
529.1 The Benefits of SPAs . . . 1509
529.2 What is SennaJS? . . . 1510

530 Configuring SPA System Settings **1511**
530.1 Related Topics . . . 1511

531 Disabling SPA **1513**
531.1 Disabling SPA across an Instance . . . 1513
531.2 Disabling SPA for a Portlet . . . 1513
531.3 Disabling SPA for an Individual Element . . . 1514
531.4 Related Topics . . . 1514

532 Specifying How Resources Are Loaded During Navigation **1515**
532.1 Related Topics . . . 1515

533 Detaching Global Listeners **1517**
533.1 Related Topics . . . 1518

534 Applying Clay Styles to your App **1519**

535 Applying Clay Patterns to Navigation **1521**
535.1 Related topics . . . 1523

536 Implementing the Management Toolbar **1525**

537 Implementing the View Types **1527**

538 Implementing the Card View **1529**
538.1 Related Topics . . . 1530

539 Implementing the List View **1531**
539.1 Related Topics . . . 1532

540 Implementing the Table View **1533**
540.1 Related Topics . . . 1534

541 Updating the Search Iterator **1535**
541.1 Related Topics . . . 1535

542 Filtering and Sorting Items with the Management Toolbar **1537**
542.1 Related Topics . . . 1538

543 Configuring Your Application's Title and Back Link **1539**
543.1 Related Topics . . . 1540

544 Applying the Add Button Pattern **1541**
544.1 Related Topics . . . 1542

545 Configuring Your Admin App's Actions Menu **1543**
545.1 Related Topics . . . 1546

546 Setting Empty Results Messages **1547**
546.1 Related Topics . . . 1549

547 Search **1551**
547.1 Basic Search Concepts . . . 1551
547.2 Mapping Definitions . . . 1551
547.3 Liferay Search Infrastructure . . . 1552

548 Aggregations **1553**
548.1 Using Aggregations . . . 1554
548.2 External References . . . 1554
548.3 Search Engine Connector Support . . . 1554
548.4 New/Related APIs . . . 1554

549 Creating Aggregations **1557**
549.1 Instantiate and Construct the Aggregation . . . 1557
549.2 Build the Search Query . . . 1557
549.3 Execute the Search Query . . . 1558
549.4 Process the response . . . 1558

550 Statistical Aggregations **1559**
550.1 `StatsRequest` . . . 1559
550.2 `StatsResponse` . . . 1560
550.3 Using the Legacy `Stats` Object . . . 1561
550.4 External References . . . 1561
550.5 Search Engine Connector Support . . . 1561
550.6 New/Related APIs . . . 1561
550.7 Deprecated APIs . . . 1562

551 Model Entity Indexing Framework **1563**
551.1 Search and Indexing Overview . . . 1563
551.2 Mapping the Composite Search and Indexing Framework to `Indexer`/`BaseIndexer` Code 1563
551.3 Permissions-Aware Searching and Indexing . . . 1564
551.4 Annotating Service Methods to Trigger Indexing . . . 1564
551.5 Search and Localization: a Cheat Sheet . . . 1565

552 Indexing Model Entities **1567**
552.1 Contributing Model Entity Fields to the Index . . . 1567
552.2 Configure Re-Indexing and Batch Indexing Behavior . . . 1568
552.3 Contribute Fields to Every Document . . . 1570

553 Searching the Index for Model Entities **1571**
553.1 Adding your Model Entity's Terms to the Query . . . 1571
553.2 Contributing Query Clauses to Every Search . . . 1572
553.3 Pre-Filtering . . . 1572

554 Returning Results **1575**
554.1 Creating a Results Summary . . . 1575
554.2 Controlling the Visibility of Model Entities . . . 1576

555 Search Service Registration **1577**

556 Search Queries and Filters **1579**
556.1 Queries and Filters in Liferay's Search API . . . 1579
556.2 Supported Query Types . . . 1580
556.3 Full Text Queries . . . 1580
556.4 Term Queries . . . 1580
556.5 Compound Queries . . . 1581
556.6 Joining Queries . . . 1581
556.7 Geo Queries . . . 1581
556.8 Specialized Queries . . . 1581
556.9 Other Queries . . . 1582
556.10 Using Queries . . . 1582
556.11 Search Queries in Liferay's Code . . . 1582
556.12 External References . . . 1582
556.13 Search Engine Connector Support . . . 1583
556.14 New/Related APIs . . . 1583

557 Building Search Queries and Filters **1585**
557.1 Queries . . . 1585
557.2 Declare Gradle Dependencies . . . 1585
557.3 Reference the Search Services . . . 1585
557.4 Build the Search Query . . . 1586
557.5 Build the Search Request . . . 1586
557.6 Execute the Search Request . . . 1587
557.7 Process the Search Response . . . 1587
557.8 Search Insights: Request and Response Strings . . . 1588
557.9 Queries Example . . . 1589
557.10 Filters . . . 1590
557.11 Legacy Filters in Legacy Search Calls . . . 1591
557.12 Discovering Indexed Fields . . . 1591

558 Segmentation and Personalization **1593**
558.1 Managing segments . . . 1593
558.2 Extending Segment Criteria . . . 1594
558.3 `RequestContextContributor` . . . 1594
558.4 `SegmentsCriteriaContributor` . . . 1594

559 Segment Management **1597**
559.1 Defining a Segment . . . 1597
559.2 Manual Segment Member Assignments . . . 1597
559.3 Retrieving Segments . . . 1598
559.4 Retrieving segment members . . . 1598

560 Creating a Request Context Contributor **1599**

561 Creating a Segment Criteria Contributor **1603**
561.1 Creating the Entity Model . . . 1603
561.2 Creating the `ODataRetriever` . . . 1604

561.3Creating the SegmentsCriteriaContributor 1606

562ServiceContext **1609**
562.1Service Context Fields 1609
562.2Creating and Populating a Service Context 1611
562.3Creating and Populating a Service Context in JavaScript 1611
562.4Accessing Service Context Data 1612
562.5Related Topics 1615

563Injecting Service Components into Integration Tests **1617**
563.1Related Topics 1618

564Upgrade Processes **1619**

565Creating Upgrade Processes for Modules **1623**
565.1Related Topics 1627

566Upgrade Processes for Former Service Builder Plugins **1629**
566.1Related Topics 1632

567Upgrading Data Schemas in Development **1633**
567.1Related Topics 1634

568Managing User-Associated Data Stored by Custom Applications **1635**

569Adding the UAD Framework to a Service Builder Application **1637**
569.1Update the Service Module 1637
569.2Include Dependencies 1637
569.3Choose the Fields to Anonymize 1637
569.4Update the UAD Module 1638
569.5Include Dependencies 1638
569.6Provide Your App's Name to the UI 1638

570Enhancing the Data Erasure UI **1639**
570.1Filtering and Searching in the Data Erasure UI 1639
570.2Hierarchy Display 1640
570.3UAD Hierarchy Declaration 1640
570.4Add methods to UADDisplay 1641

571Filtering and Searching UAD-Marked Entities **1645**
571.1Filtering 1645
571.2Search 1645

572Web Experience Management **1649**

573Page Fragments **1651**
573.1Developing Page Fragments 1651
573.2Making a Fragment Configurable 1651
573.3Fragments CLI 1652
573.4Contributed Collections 1652

573.5Fragment Specific Tags . . . 1652
573.6Recommendations and Best Practices . . . 1653
573.7CSS . . . 1653
573.8JavaScript . . . 1653

574Developing Fragments 1655
574.1Creating a Section . . . 1655
574.2Creating a Component . . . 1657

575Making a Fragment Configurable 1659

576Managing Fragments and Collections 1663
576.1Collections Management Menu . . . 1663
576.2Fragment Management Menu . . . 1663
576.3Propagating Fragment Changes Automatically . . . 1664

577Developing A Fragment Using Desktop Tools 1667
577.1Collection Format . . . 1667
577.2Fragment CLI . . . 1668
577.3Creating Collections . . . 1668
577.4Creating Fragments . . . 1669
577.5Importing and Exporting Fragments . . . 1669

578Creating a Contributed Fragment Collection 1671
578.1Create a Module . . . 1671
578.2Create the Java Class . . . 1671
578.3Create the Resources . . . 1672
578.4Configuring the Metadata . . . 1673
578.5Providing Thumbnail Images . . . 1673
578.6Providing Language Keys . . . 1673
578.7Deploy the Contributed Fragment Collection . . . 1674

579Including Default Resources in Fragments 1675

580Supporting Custom Content Types in Content and Display Pages 1677

581Mapping a Content Type to a Page 1679

582Specifying the Fields of a Custom Content Type 1681

583Providing Friendly URLs for a Custom Content Type 1685

584Integrating Display Pages into Content Creation 1691
584.1Display Page Taglib Example . . . 1691

585Screen Navigation Framework 1693
585.1Using the Framework for Your Application . . . 1693
585.2Adding Custom Screens to Liferay Applications . . . 1695

586Using Screen Navigation for Your Application 1697

586.1Adding Screens to Your Application's Back-end . . . 1697
586.2Adding Screens to Your Application's Front-end . . . 1700

587Extending Categories Administration **1701**

588Developing a Fragment Renderer **1703**
588.1Implementing the FragmentRenderer Interface . . . 1703
588.2Leveraging the FragmentRendererContext . . . 1704
588.3Rendering JSPs . . . 1704
588.4Choosing When to Display a Component . . . 1705
588.5Translating the Collection Language Key . . . 1706

589Creating a Fragment Renderer **1707**

590Web Services **1711**

591Headless REST APIs **1713**
591.1OpenAPI . . . 1713
591.2API Vocabulary . . . 1713

592Get Started: Find the API **1715**
592.1Related Topics . . . 1716

593How To Invoke a Service **1717**
593.1Related Topics . . . 1719

594Making Authenticated Requests **1721**
594.1Basic Authentication . . . 1721
594.2OAuth 2.0 Authorization . . . 1723
594.3Obtaining the OAuth 2.0 Token . . . 1723
594.4Invoking the Service with an OAuth 2.0 Token . . . 1723
594.5Using Cookie Authentication or Making Requests from the UI . . . 1723
594.6Making Unauthenticated Requests . . . 1724
594.7Cross-Origin Resource Sharing (CORS) . . . 1724
594.8Related Topics . . . 1724

595Working with Collections of Data **1727**
595.1Pagination . . . 1727

596Getting Collections **1729**
596.1Related Topics . . . 1730

597Pagination **1731**
597.1Related Topics . . . 1732

598Navigating from a Collection to its Elements **1733**
598.1Related Topics . . . 1734

599API Formats and Content Negotiation **1735**
599.1Language Negotiation . . . 1737

599.2 Creating Content with Different Languages . . . 1738
599.3 Related Topics . . . 1738

600 OpenAPI Profiles **1739**
600.1 Headless Delivery . . . 1739
600.2 Headless Administration . . . 1740
600.3 Related Topics . . . 1740

601 Filter, Sort, and Search **1741**
601.1 Filter . . . 1741
601.2 Comparison Operators . . . 1741
601.3 Logical Operators . . . 1741
601.4 Grouping Operators . . . 1742
601.5 String Functions . . . 1742
601.6 Lambda Operators . . . 1742
601.7 Operator combinations and OData syntax . . . 1742
601.8 Escaping in Queries . . . 1743
601.9 Filtering in Structured Content Fields (ContentField) . . . 1743
601.10 Search . . . 1743
601.11 Sorting . . . 1744
601.12 Flatten . . . 1744
601.13 Related Topics . . . 1745

602 Restrict Properties **1747**
602.1 Related Topics . . . 1748

603 Multipart Requests **1749**
603.1 Related Topics . . . 1750

604 How to get siteId **1751**
604.1 Using siteId or siteKey . . . 1751
604.2 Obtain siteId . . . 1751

605 Filterable properties **1753**
605.1 Headless Delivery API . . . 1753
605.2 BlogPosting . . . 1753
605.3 BlogPostingImage . . . 1753
605.4 Comment . . . 1753
605.5 ContentStructure . . . 1754
605.6 Document . . . 1754
605.7 DocumentFolder . . . 1754
605.8 KnowledgeBaseArticle . . . 1754
605.9 MessageBoardMessage . . . 1754
605.10 MessageBoardSection . . . 1755
605.11 StructuredContent . . . 1755
605.12 StructuredContentFolder . . . 1755
605.13 WikiNode . . . 1755
605.14 WikiPage . . . 1756
605.15 Headless Admin User API . . . 1756

605.16 Organization . . . 1756
605.17 User . . . 1756
605.18 Headless Admin Taxonomy API . . . 1756
605.19 Category . . . 1756
605.20 Keyword . . . 1757
605.21 Vocabulary . . . 1757

606 Using REST APIs **1759**

607 JAX-RS **1761**
607.1 Authenticating to JAX-RS Web Services . . . 1762
607.2 During Development: Basic Auth . . . 1762
607.3 Using OAuth 2.0 to Invoke a JAX-RS Web Service . . . 1762
607.4 JAX-RS and Service Access Policies . . . 1764
607.5 Public JAX-RS Services . . . 1764
607.6 Using JAX-RS with CORS . . . 1765
607.7 Related Topics . . . 1766

608 JAX-WS **1767**
608.1 Configuring Endpoints and Extenders with the Control Panel . . . 1767
608.2 Configuring Endpoints and Extenders Programmatically . . . 1769
608.3 Publishing JAX-WS Web Services . . . 1771

609 GraphQL APIs **1773**

610 Get Started: Discover the API **1775**
610.1 Unique endpoint and versioning . . . 1775

611 Get Started: Invoke a Service **1777**
611.1 GraphQL Clients . . . 1779

612 Making Authenticated Requests **1781**
612.1 Basic Authentication . . . 1781
612.2 OAuth 2.0 Authorization . . . 1782
612.3 Obtaining the OAuth 2.0 Token . . . 1782
612.4 Invoking the Service with an OAuth 2.0 Token . . . 1782
612.5 Using Cookie Authentication or doing a request from the portal . . . 1783
612.6 Making Unauthenticated Requests . . . 1783
612.7 Related Topics . . . 1783

613 Working with Collections of Data **1785**
613.1 Pagination . . . 1785

614 Mutations **1787**

615 Fragments and Node Patterns **1789**
615.1 Node pattern . . . 1789

616 Language Negotiation **1791**
616.1 Creating Content with Different Languages . . . 1792

617 Filter, Sort, and Search **1793**
617.1 Filter . . . 1793
617.2 Comparison Operators . . . 1793
617.3 Logical Operators . . . 1793
617.4 Grouping Operators . . . 1794
617.5 String Functions . . . 1794
617.6 Lambda Operators . . . 1794
617.7 Escaping in Queries . . . 1794
617.8 Filtering in Structured Content Fields (ContentField) . . . 1795
617.9 Search . . . 1795
617.10 Sorting . . . 1796
617.11 Flatten . . . 1797

618 Multipart Requests **1799**

619 Using GraphQL APIs **1803**

620 REST Builder **1805**
620.1 Why we should use REST Builder . . . 1805

621 How to install REST Builder **1807**

622 REST Builder & OpenAPI **1809**
622.1 OpenAPI profile . . . 1809
622.2 Generation . . . 1811
622.3 Examples . . . 1811
622.4 GET Collection . . . 1811
622.5 DELETE . . . 1812
622.6 POST . . . 1812
622.7 PUT . . . 1812
622.8 Summary . . . 1813

623 Developing an API with REST Builder **1815**
623.1 Development Cycle . . . 1816
623.2 Wrapping Up . . . 1817

624 Managing Collections in REST Builder **1819**
624.1 Pagination . . . 1819
624.2 Filtering, sorting and searching . . . 1819
624.3 Add an EntityModel . . . 1820
624.4 Inject Your EntityModel . . . 1820
624.5 Call search utilities . . . 1821
624.6 Using Your filter, search, and sort . . . 1821

625 REST Builder Scaffolding **1823**
625.1 Context fields . . . 1823
625.2 Automatic transactions . . . 1823
625.3 Test generation . . . 1824
625.4 Client generation . . . 1824

625.5Common utilities . . . 1824

626Support for oneOf, anyOf and allOf 1825
626.1allOf . . . 1825
626.2oneOf . . . 1826
626.3anyOf . . . 1826

627REST Builder Liferay Conventions 1829
627.1YAML & OpenAPI restrictions . . . 1829
627.2Conventions . . . 1829

628The Workflow Framework 1831
628.1Supporting Workflow in the Database . . . 1831
628.2Setting the Status Fields . . . 1831
628.3Sending the Entity to the Workflow Framework . . . 1833
628.4Allowing the Workflow Framework to Handle the Entity . . . 1833
628.5Supporting Workflow in the Service Layer . . . 1833
628.6Database Cleanup: Delete the Workflow Instance Links . . . 1834
628.7Updating the User Interface . . . 1834

629Liferay's Workflow Framework 1835
629.1Creating a Workflow Handler . . . 1835
629.2Updating the Service Layer . . . 1836
629.3Workflow Status and the View Layer . . . 1838

630WYSIWYG Editors 1839

631Adding a WYSIWYG Editor to a Portlet 1841
631.1Related Topics . . . 1842

632Modifying an Editor's Configuration 1843
632.1Related Topics . . . 1846

633AlloyEditor 1847

634Adding Buttons to AlloyEditor's Toolbars 1849

635Creating the OSGi Module and Configuring the EditorConfigContributor Class 1851
635.1Related Topics . . . 1852

636Adding a Button to the Add Toolbar 1853
636.1Related Topics . . . 1855

637Adding a Button to a Styles Toolbar 1857
637.1Related Topics . . . 1862

638Embedding Content in the AlloyEditor 1863
638.1Related Topics . . . 1865

639Adding New Behavior to an Editor 1867

639.1Related Topics 1869

V Reference 1871

640Developer Reference 1873

641Classes Moved From portal-service.jar 1875

642Export/Import and Staging 1929

643Decision to Implement Staging 1931

644Liferay Archive (LAR) File 1933
644.1LAR File Anatomy 1933
644.2LAR Manifest 1934
644.3LAR Folders 1935

645Front-End Reference 1937

646Liferay DXP FreeMarker Macros 1939
646.1Reference Examples 1940

647Front-End Taglibs 1943

648Liferay Theme Objects Available in JSPs 1945

649Liferay Portlet Objects Available in JSPs 1947

650Using the Liferay UI Taglib 1949

651Liferay UI Icons 1951
651.1Related Topics 1953

652Liferay UI Icon Lists 1955
652.1Related Topics 1956

653Liferay UI Icon Menus 1957
653.1Related Topics 1958

654Liferay UI Tabs 1959
654.1Related Topics 1960

655Liferay UI Icon Help 1961
655.1Related Topics 1962

656Using Liferay Front-end Taglibs in Your Portlet 1965

657Liferay Front-end Add Menu 1967
657.1Related Topics 1968

658 Liferay Front-end Cards **1969**
658.1 Horizontal Card . . . 1969
658.2 Icon Vertical Card . . . 1969
658.3 Vertical Card . . . 1970
658.4 HTML Vertical Card . . . 1971
658.5 User Vertical Card . . . 1973
658.6 Related Topics . . . 1974

659 Liferay Front-end Info Bar **1977**
659.1 Related Topics . . . 1979

660 Liferay Front-end Management Bar **1981**

661 Including Actions in the Management Bar **1985**
661.1 Related Topics . . . 1986

662 Disabling All or Portions of the Management Bar **1987**
662.1 Related Topics . . . 1987

663 Using the Liferay Util Taglib **1989**

664 Using Liferay Util Body Bottom **1991**
664.1 Related Topics . . . 1991

665 Using Liferay Util Body Top **1993**
665.1 Related Topics . . . 1993

666 Using Liferay Util Buffer **1995**
666.1 Related Topics . . . 1995

667 Using Liferay Util Dynamic Include **1997**
667.1 Related Topics . . . 1997

668 Using Liferay Util Get URL **1999**
668.1 Related Topics . . . 1999

669 Using Liferay Util HTML Bottom **2001**
669.1 Related Topics . . . 2001

670 Using Liferay Util HTML Top **2003**
670.1 Related Topics . . . 2003

671 Using Liferay Util Include **2005**
671.1 Related Topics . . . 2005

672 Using Liferay Util Param **2007**
672.1 Related Topics . . . 2007

673 Using Liferay Util Whitespace Remover **2009**
673.1 Related Topics . . . 2009

674 Using the Clay Taglib in Your portlets **2011**

675 Clay Alerts **2013**
675.1 Embedded Alerts . . . 2013
675.2 Stripe Alerts . . . 2014
675.3 Related Topics . . . 2015

676 Clay Badges **2017**
676.1 Badge Types . . . 2017
676.2 Related Topics . . . 2020

677 Clay Buttons **2021**
677.1 Types . . . 2021
677.2 Variations . . . 2023
677.3 Related Topics . . . 2024

678 Clay Cards **2025**
678.1 Image Cards . . . 2025
678.2 File Cards . . . 2027
678.3 User Cards . . . 2029
678.4 Horizontal Cards . . . 2030
678.5 Related Topics . . . 2030

679 Clay Dropdown Menus and Action Menus **2035**
679.1 Dropdown Menus . . . 2035
679.2 Actions Menus . . . 2037
679.3 Related Topics . . . 2041

680 Clay Form Elements **2043**
680.1 Checkbox . . . 2043
680.2 Radio . . . 2043
680.3 Selector . . . 2044
680.4 Related Topics . . . 2046

681 Clay Icons **2047**
681.1 Related Topics . . . 2047

682 Clay Labels and Links **2049**
682.1 Labels . . . 2049
682.2 Color-coded Labels . . . 2049
682.3 Removable Labels . . . 2051
682.4 Labels with Links . . . 2051
682.5 Links . . . 2051
682.6 Related Topics . . . 2052

683 Clay Management Toolbar **2053**
683.1 Using a Display Context to Configure the Management Toolbar . . . 2053
683.2 Checkbox and Actions . . . 2054
683.3 Filtering and Sorting Search Results . . . 2055

683.4Search Form . . . 2057
683.5Info Panel . . . 2057
683.6View Types . . . 2058
683.7Creation Menu . . . 2060
683.8Related Topics . . . 2061

684Clay Navigation Bars **2063**
684.1Related Topics . . . 2064

685Clay Progress Bars **2065**
685.1Related Topics . . . 2066

686Clay Stickers **2067**
686.1Related Topics . . . 2068

687Using the Chart Taglib in Your Portlets **2069**

688Bar Charts **2071**
688.1Related Topics . . . 2071

689Line Charts **2073**
689.1Related Topics . . . 2073

690Scatter Charts **2075**
690.1Related Topics . . . 2075

691Spline Charts **2077**
691.1Related Topics . . . 2078

692Step Charts **2081**
692.1Related Topics . . . 2082

693Combination Charts **2085**
693.1Related Topics . . . 2086

694Donut Charts **2087**
694.1Related Topics . . . 2087

695Gauge Charts **2089**
695.1Related Topics . . . 2089

696Pie Charts **2091**
696.1Related Topics . . . 2091

697Geomap Charts **2093**
697.1Related Topics . . . 2096

698Predictive Charts **2097**
698.1Related Topics . . . 2099

699 Refreshing Charts to Reflect Real Time Data **2101**
699.1 Related Topics . . . 2101

700 Using AUI Taglibs **2103**

701 Building Forms with AUI Tags **2105**
701.1 Related Topics . . . 2108

702 liferay-npm-bundler **2109**
702.1 How the Liferay npm Bundler Works Internally . . . 2109

703 Understanding the `.npmbundlerrc`'s Structure **2111**
703.1 The Structure . . . 2111
703.2 Standard Configuration Options . . . 2113
703.3 Package Processing Options . . . 2114
703.4 OSGi Bundle Creation Options . . . 2115

704 How the Default Preset Configures the liferay-npm-bundler **2119**

705 The Structure of OSGi Bundles Containing npm Packages **2121**
705.1 Inline JavaScript packages . . . 2122

706 How the Liferay npm Bundler Publishes npm Packages **2123**
706.1 Package De-duplication . . . 2124
706.2 Isolated Package Dependencies . . . 2125
706.3 De-duplication through Importing . . . 2127
706.4 Strategies When Importing Packages . . . 2129

707 Understanding How liferay-npm-bundler Formats JavaScript Modules for AMD **2131**

708 Understanding How Liferay AMD Loader Configuration is Exported **2135**

709 What Changed Between Liferay npm Bundler 1.x and 2.x **2139**
709.1 Automatically Formatting Modules for AMD . . . 2139
709.2 Isolating Project Dependencies . . . 2139
709.3 Improved Peer Dependency Support . . . 2139
709.4 Manually De-duplicating Through Importing . . . 2140

710 Understanding liferay-npm-bundler's Loaders **2141**

711 Default liferay-npm-bundler Loaders **2143**

712 Liferay JavaScript APIs **2145**

713 Accessing ThemeDisplay Information **2147**

714 Working with URLs in JavaScript **2151**
714.1 Portlet URL Methods . . . 2151
714.2 Liferay Util PortletURL . . . 2152
714.3 Liferay AuthToken . . . 2152

714.4 Liferay CurrentURL . . . 2152
714.5 Liferay CurrentURLEncoded . . . 2153

715 Liferay DXP JavaScript Utilities 2155
715.1 Retrieve Browser Information . . . 2155
715.2 Format XML . . . 2156
715.3 Format Storage Size . . . 2157
715.4 Store and Retrieve Session Form data . . . 2157

716 Invoking Liferay Services 2159
716.1 Invoking Web Services via JavaScript . . . 2159
716.2 Batching Requests . . . 2160
716.3 Nesting Requests . . . 2161
716.4 Filtering Results . . . 2162
716.5 Inner Parameters . . . 2163

717 Handling AJAX Requests with `Liferay.Util.fetch` 2165

718 Working with Addresses 2167

719 FreeMarker Taglib Macros 2169

720 Setting up Your npm Environment 2173

721 Sitemap Page Configuration Options 2175

722 CKEditor Plugin Reference Guide 2177

723 Fully Qualified Portlet IDs 2181

724 Available SPA Lifecycle Events 2185

725 Theme Anatomy Reference Guide 2187
725.1 Theme Files . . . 2188
725.2 _clay_custom.scss . . . 2188
725.3 _clay_variables.scss . . . 2188
725.4 _custom.scss . . . 2188
725.5 _liferay_variables_custom.scss . . . 2188
725.6 init_custom.ftl . . . 2188
725.7 navigation.ftl . . . 2189
725.8 portal_normal.ftl . . . 2189
725.9 portal_pop_up.ftl . . . 2189
725.10 portlet.ftl . . . 2189
725.11 liferay-theme.json . . . 2189
725.12 package.json . . . 2189
725.13 main.js . . . 2189
725.14 liferay-look-and-feel.xml . . . 2189
725.15 liferay-plugin-package.properties . . . 2190

726 Freemarker Variable Reference Guide 2191

727 Gradle Plugins **2197**

728 App Javadoc Builder Gradle Plugin **2199**
728.1 Usage . . . 2199
728.2 Project Extension . . . 2199
728.3 Tasks . . . 2200

729 Baseline Gradle Plugin **2201**
729.1 Usage . . . 2201
729.2 Project Extension . . . 2202
729.3 Tasks . . . 2202
729.4 BaselineTask . . . 2202
729.5 Helper Tasks . . . 2203
729.6 Additional Configuration . . . 2203
729.7 Baseline Dependency . . . 2204
729.8 System Properties . . . 2204

730 Change Log Builder Gradle Plugin **2205**
730.1 Usage . . . 2205
730.2 Tasks . . . 2206
730.3 BuildChangeLogTask . . . 2206

731 CSS Builder Gradle Plugin **2207**
731.1 Usage . . . 2207
731.2 Tasks . . . 2207
731.3 BuildCSSTask . . . 2208
731.4 Additional Configuration . . . 2209
731.5 Liferay CSS Builder Dependency . . . 2209
731.6 Liferay Frontend Common CSS Dependency . . . 2209

732 DB Support Gradle Plugin **2211**
732.1 Usage . . . 2211
732.2 Tasks . . . 2212
732.3 CleanServiceBuilderTask . . . 2212
732.4 Additional Configuration . . . 2212
732.5 JDBC Drivers Dependency . . . 2213
732.6 Liferay DB Support Dependency . . . 2213

733 Dependency Checker Gradle Plugin **2215**
733.1 Usage . . . 2215
733.2 Project Extension . . . 2215
733.3 Additional Configuration . . . 2216
733.4 Project Properties . . . 2216

734 Deployment Helper Gradle Plugin **2217**
734.1 Usage . . . 2217
734.2 Tasks . . . 2217
734.3 BuildDeploymentHelperTask . . . 2218
734.4 Additional Configuration . . . 2218

734.5Liferay Deployment Helper Dependency 2218

735Go Gradle Plugin **2219**
735.1Usage 2219
735.2Project Extension 2219
735.3Tasks 2219
735.4DownloadGoTask 2220
735.5ExecuteGoTask 2220
735.6go*command*{programName} Task 2221

736Gulp Gradle Plugin **2223**
736.1Usage 2223
736.2Tasks 2223
736.3ExecuteGulpTask 2223

737Jasper JSPC Gradle Plugin **2225**
737.1Usage 2225
737.2Tasks 2226
737.3CompileJSPTask 2226
737.4Additional Configuration 2226
737.5JSP Compilation Classpath 2226
737.6Liferay Jasper JSPC Dependency 2227

738Javadoc Formatter Gradle Plugin **2229**
738.1Usage 2229
738.2Tasks 2230
738.3FormatJavadocTask 2230
738.4Additional Configuration 2230
738.5Liferay Javadoc Formatter Dependency 2231
738.6System Properties 2231

739JS Module Config Generator Gradle Plugin **2233**
739.1Usage 2233
739.2Project Extension 2233
739.3Tasks 2234
739.4ConfigJSModulesTask 2234

740JS Transpiler Gradle Plugin **2237**
740.1Usage 2237
740.2JS Transpiler Plugin 2238
740.3JS Transpiler Base Plugin 2238
740.4Tasks 2238
740.5TranspileJSTask 2239

741JSDoc Gradle Plugin **2241**
741.1Usage 2241
741.2JSDoc Plugin 2242
741.3AppJSDoc Plugin 2242
741.4Project Extension 2242

741.5Tasks . . . 2243
741.6JSDocTask . . . 2243

742Lang Builder Gradle Plugin **2245**
742.1Usage . . . 2245
742.2Tasks . . . 2246
742.3BuildLangTask . . . 2246
742.4Additional Configuration . . . 2247
742.5Liferay Lang Builder Dependency . . . 2247

743Maven Plugin Builder Gradle Plugin **2249**
743.1Usage . . . 2249
743.2Tasks . . . 2249
743.3BuildPluginDescriptorTask . . . 2250
743.4Task Methods . . . 2251
743.5WriteMavenSettingsTask . . . 2251
743.6Additional Configuration . . . 2252
743.7Maven Embedder Dependency . . . 2252
743.8System Properties . . . 2252

744Node Gradle Plugin **2253**
744.1Usage . . . 2253
744.2Project Extension . . . 2253
744.3Tasks . . . 2254
744.4DownloadNodeTask . . . 2254
744.5ExecuteNodeTask . . . 2255
744.6ExecuteNodeScriptTask . . . 2256
744.7ExecuteNpmTask . . . 2256
744.8DownloadNodeModuleTask . . . 2257
744.9NpmInstallTask . . . 2257
744.10NpmShrinkwrapTask . . . 2258
744.11PublishNodeModuleTask . . . 2258
744.12npmRun${script} Task . . . 2259

745REST Builder Gradle Plugin **2261**
745.1Usage . . . 2261
745.2Tasks . . . 2261
745.3BuildRESTTask . . . 2262
745.4Additional Configuration . . . 2262
745.5Liferay REST Builder Dependency . . . 2262

746Service Builder Gradle Plugin **2263**
746.1Usage . . . 2263
746.2Tasks . . . 2263
746.3BuildServiceTask . . . 2265
746.4Additional Configuration . . . 2266
746.5Liferay Service Builder Dependency . . . 2266

747Source Formatter Gradle Plugin **2267**

747.1Usage . . . 2267
747.2Tasks . . . 2267
747.3FormatSourceTask . . . 2268
747.4Additional Configuration . . . 2269
747.5Liferay Source Formatter Dependency . . . 2269
747.6System Properties . . . 2269

748Soy Gradle Plugin **2271**
748.1Usage . . . 2271
748.2Soy Plugin . . . 2272
748.3Additional Configuration . . . 2272
748.4Soy Translation Plugin . . . 2272
748.5Tasks . . . 2273
748.6BuildSoyTask . . . 2273
748.7WrapSoyAlloyTemplateTask . . . 2273
748.8ReplaceSoyTranslationTask . . . 2273

749Target Platform Gradle Plugin **2275**
749.1Usage . . . 2275
749.2Target Platform Plugin . . . 2276
749.3Target Platform IDE Plugin . . . 2276
749.4Project Extension . . . 2276
749.5Tasks . . . 2277
749.6ResolveTask . . . 2277
749.7Additional Configuration . . . 2278
749.8Target Platform BOMs Dependency . . . 2278
749.9Target Platform Bundles Dependency . . . 2278
749.10Target Platform Distro Dependency . . . 2278
749.11Target Platform Requirements Dependency . . . 2279

750Theme Builder Gradle Plugin **2281**
750.1Usage . . . 2281
750.2Tasks . . . 2282
750.3BuildThemeTask . . . 2282
750.4Additional Configuration . . . 2283
750.5Liferay Theme Builder Dependency . . . 2283
750.6Parent Theme Dependencies . . . 2283

751TLD Formatter Gradle Plugin **2285**
751.1Usage . . . 2285
751.2Tasks . . . 2285
751.3FormatTLDTask . . . 2286
751.4Additional Configuration . . . 2286
751.5Liferay TLD Formatter Dependency . . . 2286

752TLDDoc Builder Gradle Plugin **2287**
752.1Usage . . . 2287
752.2TLDDoc Builder Plugin . . . 2288

752.3App TLDDoc Builder Plugin . . . 2288
752.4Project Extension . . . 2289
752.5Tasks . . . 2289
752.6TLDDocTask . . . 2289
752.7ValidateSchemaTask . . . 2290
752.8Additional Configuration . . . 2290
752.9Tag Library Documentation Generator Dependency . . . 2290

753 Whip Gradle Plugin 2291
753.1Usage . . . 2291
753.2Project Extension . . . 2292
753.3Task Extension . . . 2292
753.4Additional Configuration . . . 2292
753.5Liferay Whip Dependency . . . 2292

754 WSDD Builder Gradle Plugin 2293
754.1Usage . . . 2293
754.2Tasks . . . 2293
754.3BuildWSDDTask . . . 2294
754.4Additional Configuration . . . 2294
754.5Liferay WSDD Builder Dependency . . . 2294

755 WSDL Builder Gradle Plugin 2297
755.1Usage . . . 2297
755.2Tasks . . . 2297
755.3BuildWSDLTask . . . 2298
755.4Additional Configuration . . . 2299
755.5Apache Axis Dependency . . . 2299

756 XML Formatter Gradle Plugin 2301
756.1Usage . . . 2301
756.2Tasks . . . 2301
756.3FormatXMLTask . . . 2302
756.4Additional Configuration . . . 2302
756.5Liferay XML Formatter Dependency . . . 2302

757 XSD Builder Gradle Plugin 2303
757.1Usage . . . 2303
757.2Tasks . . . 2303
757.3BuildXSDTask . . . 2304
757.4Additional Configuration . . . 2304
757.5Apache XMLBeans Dependency . . . 2304

758 Liferay Faces 2305

759 Liferay Faces Version Scheme 2307
759.1Using The Liferay Faces Archetype Portlet . . . 2307
759.2Liferay Faces Alloy . . . 2307
759.3Liferay Faces Bridge . . . 2307

759.4Liferay Faces Bridge Ext . . . 2308
759.5Liferay Faces Portal . . . 2308
759.6Liferay Faces Util . . . 2308

760Understanding Liferay Faces Bridge 2311
760.1Related Topics . . . 2313

761Understanding Liferay Faces Alloy 2315
761.1Related Topics . . . 2315

762Understanding Liferay Faces Portal 2317
762.1Related Topics . . . 2318

763Maven Plugins 2319

764Bundle Support Plugin 2321
764.1Usage . . . 2321
764.2Goals . . . 2322
764.3clean Goal's Available Parameters . . . 2322
764.4create-token Goal's Available Parameters . . . 2322
764.5deploy Goal's Available Parameters . . . 2323
764.6dist Goal's Available Parameters . . . 2323
764.7init Goal's Available Parameters . . . 2323

765CSS Builder Plugin 2325
765.1Usage . . . 2325
765.2Goals . . . 2325
765.3Available Parameters . . . 2326

766DB Support Plugin 2327
766.1Usage . . . 2327
766.2Goals . . . 2327
766.3Available Parameters . . . 2328

767Deployment Helper Plugin 2329
767.1Usage . . . 2329
767.2Goals . . . 2329
767.3Available Parameters . . . 2329

768Javadoc Formatter Plugin 2331
768.1Usage . . . 2331
768.2Goals . . . 2331
768.3Available Parameters . . . 2332

769Lang Builder Plugin 2333
769.1Usage . . . 2333
769.2Goals . . . 2333
769.3Available Parameters . . . 2333

770REST Builder Plugin 2335

770.1Usage . . . 2335
770.2Goals . . . 2335
770.3Available Parameters . . . 2335

771Service Builder Plugin 2337
771.1Usage . . . 2337
771.2Goals . . . 2337
771.3Available Parameters . . . 2337

772Source Formatter Plugin 2339
772.1Usage . . . 2339
772.2Goals . . . 2339
772.3Available Parameters . . . 2340

773Theme Builder Plugin 2341
773.1Usage . . . 2341
773.2Goals . . . 2341
773.3Available Parameters . . . 2342

774TLD Formatter Plugin 2343
774.1Usage . . . 2343
774.2Goals . . . 2343
774.3Available Parameters . . . 2343

775WSDD Builder Plugin 2345
775.1Usage . . . 2345
775.2Goals . . . 2345
775.3Available Parameters . . . 2345

776XML Formatter Plugin 2347
776.1Usage . . . 2347
776.2Goals . . . 2347
776.3Available Parameters . . . 2347

777PortletMVC4Spring 2349

778PortletMVC4Spring Project Anatomy 2351
778.1Maven Commands for Generating PortletMVC4Spring Projects . . . 2351
778.2SP/JSPX Form Portlet . . . 2351
778.3Thymeleaf Form Portlet . . . 2351
778.4Project Structure . . . 2352

779PortletMVC4Spring Annotations 2353
779.1`@RenderMapping` Annotation Examples . . . 2353
779.2`@ActionMapping` Annotation Examples . . . 2353

780PortletMVC4Spring Configuration Files 2355
780.1web.xml . . . 2355
780.2portlet.xml . . . 2357

780.3liferay-portlet.xml . . . 2358
780.4liferay-display.xml . . . 2359
780.5Portlet Application Context . . . 2359
780.6Portlet Contexts . . . 2360
780.7liferay-plugin-package.properties . . . 2361
780.8Related Topics . . . 2361

781 Project Templates **2363**

782 Activator Template **2365**

783 API Template **2367**

784 Control Menu Entry Template **2369**

785 Form Field Template **2371**

786 Fragment Template **2373**

787 FreeMarker Portlet Template **2375**

788 Layout Template **2377**

789 Modules Ext Template **2379**

790 MVC Portlet Template **2381**

791 Panel App Template **2383**

792 Portlet Configuration Icon **2385**

793 Portlet Provider Template **2387**

794 Portlet Toolbar Contributor Template **2389**

795 REST Template **2391**

796 Service Builder Template **2393**

797 Service Template **2395**

798 Service Wrapper Template **2397**

799 Simulation Panel Entry Template **2399**

800 Social Bookmark Template **2401**

801 Spring MVC Portlet Template **2403**

802 Template Context Contributor Template **2407**

803 Theme Contributor Template **2409**

804 Theme Template **2411**

805 WAR Core Ext **2413**

806 WAR Hook Template **2417**

807 WAR MVC Portlet Template **2419**

808 Sample Projects **2421**

809 Apps **2423**

810 Service Builder Samples **2425**

811 Service Builder Application Demonstrating Actionable Dynamic Query **2427**
811.1 What API(s) and/or code components does this sample highlight? 2428
811.2 How does this sample leverage the API(s) and/or code component? 2428

812 Service Builder Application Using External Database via JDBC **2429**
812.1 What API(s) and/or code components does this sample highlight? 2431
812.2 How does this sample leverage the API(s) and/or code component? 2431
812.3 Configuring the Data Source 2431
812.4 Accessing Data 2431

813 Service Builder Application Using External Database via JNDI **2433**
813.1 What API(s) and/or code components does this sample highlight? 2436
813.2 How does this sample leverage the API(s) and/or code component? 2436
813.3 Additional Information 2436

814 Workflow Samples **2437**

815 Workflow Asset Application **2439**
815.1 What API(s) and/or code components does this sample highlight? 2439
815.2 How does this sample leverage the API(s) and/or code component? 2439

816 Workflow Application **2441**
816.1 What API(s) and/or code components does this sample highlight? 2441
816.2 How does this sample leverage the API(s) and/or code component? 2441

817 Greedy Policy Option Application **2443**
817.1 What API(s) and/or code components does this sample highlight? 2443
817.2 How does this sample leverage the API(s) and/or code component? 2444
817.3 Binding a newly deployed component's service reference to the highest ranking service instance that's available initially 2445
817.4 Deploying a module with a higher ranked service instance for binding to greedy references immediately 2447
817.5 Configuring a component to reference a different service instance dynamically . . 2447
817.6 Where Is This Sample? 2448

818 Kotlin Portlet **2449**

818.1What API(s) and/or code components does this sample highlight? 2449
818.2How does this sample leverage the API(s) and/or code component? 2449
818.3Where Is This Sample? 2450

819 Shared Language Keys **2451**
819.1What API(s) and/or code components does this sample highlight? 2451
819.2How does this sample leverage the API(s) and/or code component? 2452
819.3Where Is This Sample? 2453

820 Simulation Panel App **2455**
820.1What API(s) and/or code components does this sample highlight? 2455
820.2How does this sample leverage the API(s) and/or code component? 2455
820.3Where Is This Sample? 2456

821 Extensions **2457**

822 Control Menu Entry **2459**
822.1What API(s) and/or code components does this sample highlight? 2459
822.2How does this sample leverage the API(s) and/or code component? 2459
822.3Where Is This Sample? 2460

823 Document Action **2461**
823.1What API(s) and/or code components does this sample highlight? 2461
823.2How does this sample leverage the API(s) and/or code component? 2461
823.3Where Is This Sample? 2463

824 Gogo Shell Command **2465**
824.1What API(s) and/or code components does this sample highlight? 2465
824.2How does this sample leverage the API(s) and/or code component? 2465
824.3Where Is This Sample? 2467

825 Index Settings Contributor **2469**
825.1What API(s) and/or code components does this sample highlight? 2469
825.2How does this sample leverage the API(s) and/or code component? 2470
825.3Where Is This Sample? 2471

826 Indexer Post Processor **2473**
826.1What API(s) and/or code components does this sample highlight? 2473
826.2How does this sample leverage the API(s) and/or code component? 2473
826.3Where Is This Sample? 2474

827 Model Listener **2475**
827.1What API(s) and/or code components does this sample highlight? 2475
827.2How does this sample leverage the API(s) and/or code component? 2475
827.3Where Is This Sample? 2477

828 Screen Name Validator **2479**
828.1What API(s) and/or code components does this sample highlight? 2480
828.2How does this sample leverage the API(s) and/or code component? 2480

828.3Where Is This Sample? 2480

829Servlet **2481**
829.1What API(s) and/or code components does this sample highlight? 2482
829.2How does this sample leverage the API(s) and/or code component? 2482
829.3Where Is This Sample? 2482

830Overrides **2483**

831Module JSP Override **2485**
831.1What API(s) and/or code components does this sample highlight? 2486
831.2How does this sample leverage the API(s) and/or code component? 2486
831.3Where Is This Sample? 2486

832Resource Bundle Override **2487**
832.1What API(s) and/or code components does this sample highlight? 2487
832.2How does this sample leverage the API(s) and/or code component? 2487
832.3Where Is This Sample? 2488

833Themes **2489**

834Simple Theme **2491**
834.1What API(s) and/or code components does this sample highlight? 2491
834.2How does this sample leverage the API(s) and/or code component? 2492
834.3Where Is This Sample? 2492

835Template Context Contributor **2493**
835.1What API(s) and/or code components does this sample highlight? 2493
835.2How does this sample leverage the API(s) and/or code component? 2493
835.3Where Is This Sample? 2494

836Theme Contributor **2495**
836.1What API(s) and/or code components does this sample highlight? 2496
836.2How does this sample leverage the API(s) and/or code component? 2496
836.3Where Is This Sample? 2496

837Ext **2497**

838Login Web Ext **2499**
838.1What API(s) and/or code components does this sample highlight? 2500
838.2How does this sample leverage the API(s) and/or code component? 2500
838.3Where Is This Sample? 2501

839Segmentation and Personalization Reference **2503**

840Defining Segmentation Criteria **2505**
840.1User Properties 2506
840.2Organization Properties 2506
840.3Session Properties 2506

841 Tooling **2507**

842 Creating a Project **2509**
842.1 Blade CLI . . . 2509
842.2 Liferay Dev Studio . . . 2510
842.3 Liferay IntelliJ Plugin . . . 2511
842.4 Maven . . . 2512

843 Deploying a Project **2513**
843.1 Blade CLI . . . 2513
843.2 Gradle . . . 2514
843.3 Liferay Dev Studio . . . 2514
843.4 Liferay IntelliJ Plugin . . . 2514
843.5 Maven . . . 2516

844 Blade CLI **2517**

845 Installing Blade CLI **2519**

846 Installing Blade CLI with Proxy Requirements **2521**

847 Managing Your Liferay Server with Blade CLI **2523**

848 Generating Project Samples with Blade CLI **2527**

849 Updating Blade CLI **2529**

850 Converting Plugins SDK Projects with Blade CLI **2531**

851 Extending Blade CLI **2533**

852 Creating Custom Commands for Blade CLI **2535**

853 Creating Custom Project Templates for Blade CLI **2537**

854 Installing New Extensions for Blade CLI **2539**
854.1 Installing a New Extension . . . 2539
854.2 Uninstalling an Extension . . . 2539

855 Creating a Blade Profile **2541**
855.1 Creating a New Profile . . . 2541
855.2 Setting a Profile . . . 2542

856 Common Errors with Blade CLI **2545**
856.1 The blade command is not available in my CLI . . . 2545
856.2 I can't update my Blade CLI version . . . 2546

857 Liferay Dev Studio **2547**

858 Installing Liferay Dev Studio **2549**
858.1 Install the Dev Studio Bundle . . . 2549

858.2Install Dev Studio into Eclipse . . . 2550
858.3Install Dev Studio into Eclipse from a ZIP File . . . 2550

859 Setting Proxy Requirements for Dev Studio 2551
859.1Additional Proxy Settings . . . 2551

860 Installing a Liferay Server in Dev Studio 2553

861 Importing Projects in Dev Studio 2557

862 Using the Gogo Shell in Dev Studio 2559

863 Searching Liferay DXP Source in Dev Studio 2561
863.1Search Class Hierarchy . . . 2561
863.2Search Method Declarations . . . 2562
863.3Search Annotation References . . . 2563

864 Debugging Liferay DXP Source in Dev Studio 2565
864.1Configure Your Target Platform . . . 2565
864.2Configure a Liferay Server and Start It in Debug Mode . . . 2566

865 Updating Liferay Dev Studio 2567

866 Gradle in Dev Studio 2569
866.1Creating Pure Gradle Projects . . . 2570
866.2Importing Pure Gradle Projects . . . 2570
866.3Gradle Tasks and Executions . . . 2570

867 Maven in Dev Studio 2575
867.1Installing Maven Plugins for Dev Studio . . . 2575
867.2Importing Maven Projects . . . 2577
867.3Using the POM Graphic Editor . . . 2577

868 IntelliJ 2581

869 Installing the Liferay IntelliJ Plugin 2583
869.1Installing Via IntelliJ Marketplace . . . 2583
869.2Installing Via Zip File . . . 2583

870 Installing a Server in IntelliJ 2585

871 Updating Liferay IntelliJ Plugin 2587

872 Liferay JS Generator 2589

873 Installing the JS Generator and Generating a Bundle 2591

874 Understanding the JS Portlet Extender Configuration 2593
874.1Manifest Header . . . 2593
874.2Main Entry Point . . . 2593

875 Configuration JSON Available Options **2595**
875.1 JSON Format . . . 2595

876 Adapting Existing Apps to Run on Liferay DXP **2599**

877 Liferay Workspace **2603**
877.1 Workspace Anatomy . . . 2604
877.2 Development Lifecycle . . . 2604
877.3 Creating Projects . . . 2605
877.4 Building Projects . . . 2605
877.5 Deploying Projects . . . 2606
877.6 Testing Projects . . . 2606
877.7 Releasing Projects . . . 2607

878 Installing Liferay Workspace **2609**

879 Creating a Liferay Workspace **2611**
879.1 Blade CLI . . . 2611
879.2 Dev Studio . . . 2612
879.3 IntelliJ . . . 2614
879.4 Maven . . . 2614

880 Importing a Liferay Workspace into an IDE **2617**
880.1 Dev Studio . . . 2617
880.2 IntelliJ . . . 2617

881 Setting Proxy Requirements for Liferay Workspace **2619**
881.1 Gradle . . . 2619
881.2 Maven . . . 2620

882 Adding a Liferay Bundle to Liferay Workspace **2621**

883 Setting Environment Configurations for Liferay Workspace **2623**

884 Building Node.js Themes in Liferay Workspace **2625**

885 Building Gradle/Maven Themes in Liferay Workspace **2627**

886 Managing the Target Platform **2629**
886.1 Dependency Management with BOMs . . . 2629
886.2 Leveraging Target Platform in Dev Studio . . . 2630

887 Setting the Target Platform **2631**

888 Targeting a Platform Outside of Workspace **2633**

889 Targeting a Platform with Maven **2635**

890 Validating Modules Against the Target Platform **2637**
890.1 Resolving Your Modules . . . 2637

890.2Modifying the Target Platform's Capabilities . . . 2639
890.3Depending on Third Party Libraries Not Included in Liferay DXP . . . 2639
890.4Depending on a Customized Distribution of Liferay DXP . . . 2639
890.5Including the Resolver in Your Gradle Build . . . 2640

891Adding a Third Party Library's Capabilities to the Resolver's Capabilities 2641

892Skipping the Resolving Process for a Module 2643

893Depending on a Customized Distribution of Liferay DXP 2645

894Including the Resolver in Your Gradle Build 2647

895How to Resolve Common Output Errors Reported by the Resolve Task 2649
895.1Missing Import Error . . . 2649
895.2Missing Service Reference . . . 2650
895.3Missing Fragment Host . . . 2650

896Validating Modules Outside of Workspace 2653

897Leveraging Docker 2655

898Creating a Liferay DXP Docker Container 2657

899Configuring a Docker Container 2659

900Building a Custom Docker Image 2661

901Updating Liferay Workspace 2663
901.1Gradle . . . 2663
901.2Maven . . . 2664

902Updating Default Plugins Provided by Liferay Workspace 2665

903Maven 2667
903.1Installing Liferay Maven Artifacts . . . 2667
903.2Managing Maven Artifacts in a Repository . . . 2668
903.3Applying Maven Plugins . . . 2668

904Installing Liferay Maven Artifacts 2671

905Creating a Maven Repository 2673

906Configuring Local Maven Settings to Access Repositories 2675

907Deploying Liferay Maven Artifacts to a Repository 2677

908Building an OSGi Module JAR with Maven 2681

909Building a Theme with Maven 2683

910 Compiling Sass Files in a Maven Project **2687**

911 Using Service Builder in a Maven Project **2689**

912 Upgrading Your Maven Build Environment **2691**
912.1 Upgrading to New 7.0 Maven Plugins . . . 2691
912.2 Updating Liferay Maven Artifact Dependencies . . . 2693

913 Theme Generator **2695**
913.1 Sub-generators . . . 2695
913.2 Layouts Sub-generator . . . 2696
913.3 Themelets Sub-generator . . . 2696

914 Installing the Theme Generator and Creating a Theme **2697**

915 Generating Layout Templates with the Theme Generator **2701**

916 Generating Themelets with the Theme Generator **2705**

917 Liferay Upgrade Planner **2707**

918 Using the Upgrade Planner with Proxy Requirements **2711**

919 Web Experience Management Reference **2713**

920 Fragment Specific Tags **2715**
920.1 Making Text Editable . . . 2715
920.2 Making Images Editable . . . 2715
920.3 Creating Editable Links . . . 2716
920.4 Including Widgets Within A Fragment . . . 2716
920.5 Enabling Embedding for Your Widget . . . 2718

921 Fragment Configuration Types **2719**
921.1 Checkbox Configuration . . . 2719
921.2 Color Palette Configuration . . . 2720
921.3 Select Configuration . . . 2721
921.4 Text Configuration . . . 2721

922 Escaping Fragment Configuration Text Values **2723**
922.1 Escaping Values in HTML/FreeMarker . . . 2723
922.2 Escaping Values in JavaScript . . . 2723

923 Asset Display Page Taglib **2725**
923.1 Display Page Attributes . . . 2725

924 Crafting XML Workflow Definitions **2727**
924.1 Existing Workflow Definitions . . . 2727
924.2 Schema . . . 2728
924.3 Metadata . . . 2728

925 Workflow Definition Nodes **2729**
925.1 State Nodes . . . 2729
925.2 Conditions . . . 2730
925.3 Forks and Joins . . . 2731
925.4 Task Nodes . . . 2732

926 Workflow Task Nodes **2733**
926.1 Assignments . . . 2734
926.2 Resource Action Assignments . . . 2735
926.3 Task Timers . . . 2736

927 Workflow Notifications **2739**
927.1 Notification Options . . . 2739

928 Breaking Changes **2743**
928.1 Breaking Changes List . . . 2743
928.2 Removed Support for JSP Templates in Themes . . . 2743
928.3 Lodash Is No Longer Included by Default . . . 2744
928.4 Moved Two Staging Properties to OSGi Configuration . . . 2744
928.5 Remove Link Application URLs to Page Functionality . . . 2745
928.6 Moved TermsOfUseContentProvider out of kernel.util . . . 2746
928.7 Removed HibernateConfigurationConverter and Converter . . . 2746
928.8 Switched to Use JDK Function and Supplier . . . 2747
928.9 Deprecated com.liferay.portal.service.InvokableService Interface . . . 2747
928.10 Dropped Support of ServiceLoaderCondition . . . 2748
928.11 Switched to Use JDK Predicate . . . 2748
928.12 Removed Unsafe Functional Interfaces in Package com.liferay.portal.kernel.util . . . 2749
928.13 Deprecated NTLM in Portal Distribution . . . 2750
928.14 Deprecated OpenID in Portal Distribution . . . 2750
928.15 Deprecated Google SSO in Portal Distribution . . . 2751
928.16 Updated AlloyEditor v2.0 Includes New Major Version of React . . . 2751
928.17 Deprecated dl.tabs.visible property . . . 2752
928.18 Move the User Menu out of the Product Menu . . . 2753
928.19 Removed Hong Kong and Macau from the List of Countries . . . 2753
928.20 Groups Was Upgraded From 3.6.16 to 4.1.1 . . . 2754
928.21 Liferay `AssetEntries_AssetCategories` Is No Longer Used . . . 2754
928.22 Auto Tagging Must Be Reconfigured Manually . . . 2755
928.23 Blogs Image Properties Were Moved to System Settings . . . 2756
928.24 Removed Cache Bootstrap Feature . . . 2756

929 CDI Portlet Predefined Beans **2759**
929.1 Portlet Request Scoped Beans . . . 2759
929.2 Dependent Scoped Beans . . . 2760
929.3 Related Topics . . . 2760

930 Item Selector Criterion and Return Types **2761**
930.1 Item Selector Criterion Classes . . . 2761
930.2 Item Selector Return Type Classes . . . 2762

931 Java APIs **2763**
931.17.0 Java APIs . . . 2763
931.2 Liferay DXP App Java APIs . . . 2763
931.3 Forms and Workflow . . . 2764
931.4 Foundation . . . 2765
931.5 Web Experience . . . 2767
931.6 JavaScript and CSS . . . 2768
931.7 Descriptor Definitions . . . 2768

932 Meaningful Schema Versioning **2769**
932.1 Micro change examples . . . 2770
932.2 Minor change examples . . . 2770
932.3 Major change examples . . . 2770

933 Portlet 3.0 API Opt In **2771**
933.1 Standard Portlet `@PortletApplication` Annotation . . . 2771
933.2 Liferay MVC Portlet `@Component` Annotation . . . 2771
933.3 `portlet.xml` Descriptor . . . 2771

934 Portlet Descriptor to OSGi Service Property Map **2773**
934.1 Portlet Descriptor Mappings . . . 2774
934.2 Liferay Descriptor Mappings . . . 2774
934.3 Liferay Display . . . 2775
934.4 Liferay Portlet . . . 2775

935 Third Party Packages Portal Exports **2779**

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

Developer Tutorials

Liferay's developer tutorials help you learn from scratch. They are the "opinionated" way to work with code on Liferay's platform. Here, you'll learn these things:

- Writing applications on various frameworks, such as Bean Portlet, PortletMVC4Spring, and Liferay MVC Portlet (coming soon)
- Writing front-end applications using Angular, React, and Vue (coming soon)
- Upgrading your code from older versions of Liferay DXP
- Creating back-end and headless services (coming soon)

CHAPTER 2

DEVELOPING A WEB APPLICATION

This tutorial shows you how to build a Liferay application from beginning to end. Starting by designing the back-end using Liferay's object-relational mapper Service Builder, you'll move from there to designing the widget that calls all of those services. In the process, you'll learn about many of Liferay's development frameworks, including

- Permissions
- Search
- Assets
- REST Builder web services
- Workflow
- Staging

When you're finished, the application looks like this:

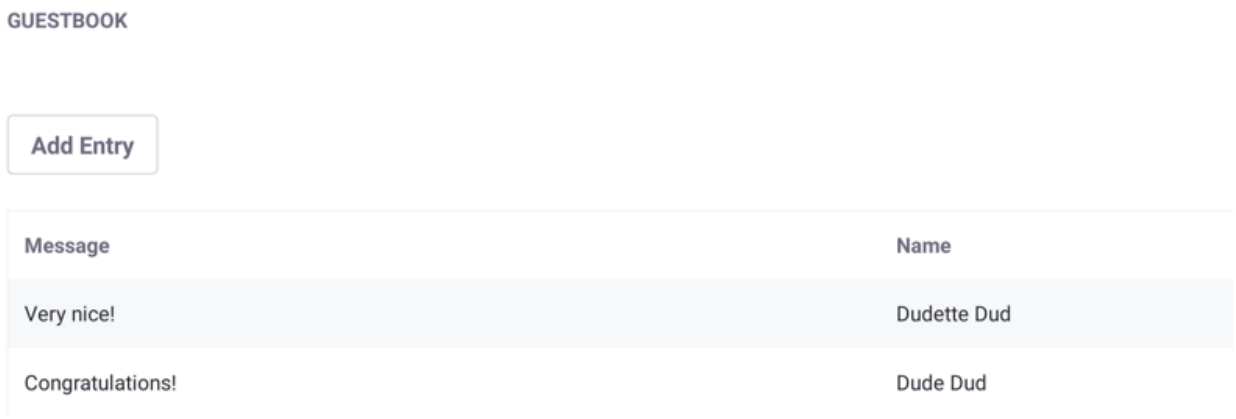

Figure 2.1: It looks humble, but there's a lot of functionality packed in this application.

This tutorial assumes nothing, so it starts 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 have an environment ready, you'll jump right in and start creating your object-relational map. After you've created your back-end, you'll move to the front-end.

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!

CHAPTER 3

SETTING UP A DEVELOPMENT ENVIRONMENT

Liferay's development tools help you get started fast. All you need as a prerequisite is a Java Development Kit version 8 (JDK or OpenJDK is fine). Once that's installed, there are only three steps.

- 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 a Liferay Workspace for developing your first Liferay DXP application.

3.1 Installing a Liferay Dev Studio DXP Bundle

Follow these steps:

1. Download and install Liferay Dev Studio DXP Installing it is easy: run the installer.
2. To run Liferay Dev Studio DXP, run the `DeveloperStudio` executable. On Windows, the installer doesn't create a menu entry, so you should add a shortcut to it to your desktop or task bar.

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*. Windows has path length limitations, so on that OS, you may want to create your workspace the root folder (`C:\`).
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 Eclipse workspace is set up.

3.2 Creating a Liferay Workspace

Now you'll create another kind of workspace: a Liferay Workspace. Liferay Workspace helps manage Liferay projects by providing various build scripts, properties, and configuration for you. 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.

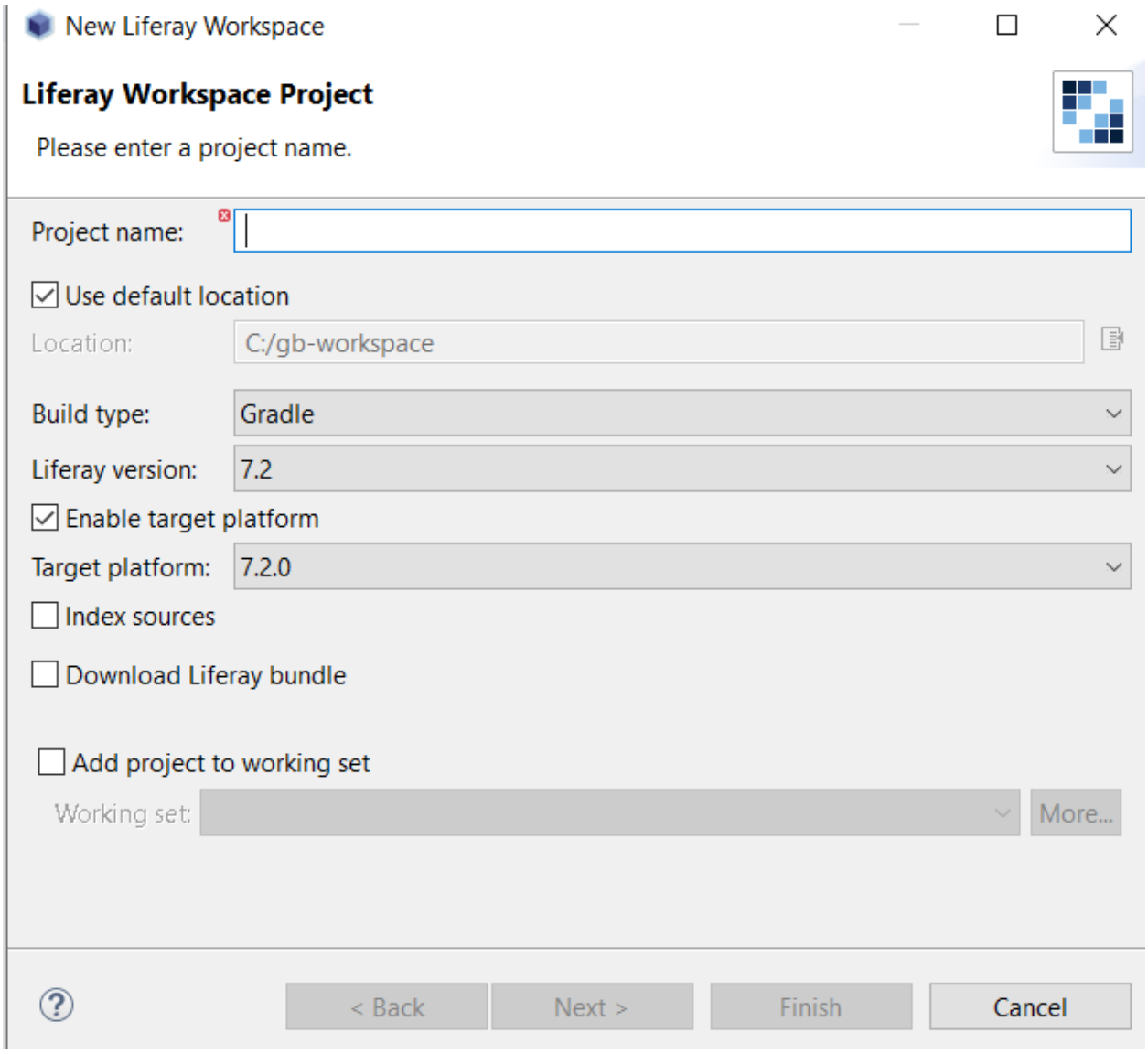

Figure 3.1: By selecting *Liferay Workspace*, you begin the process of creating a new workspace for your Liferay DXP projects.

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,* 7.2 should already be selected.
5. Leave the rest of the defaults.
6. 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`.
7. Click *Finish* to create your Liferay Workspace. This may take a while because Liferay Liferay DXP downloads the Liferay DXP bundle in the background.

Congratulations! Your development environment is ready! Next, you'll get started developing your first Liferay DXP application.

CHAPTER 4

GENERATING THE BACK-END

You can start writing an application in either the front-end or the back-end. If you start with the front-end, you design the screens and forms first, using mock data. If you start with the back-end, you create your data store up front, and then you can create your front-end later. This is what you'll do with the Guestbook application.

A *persistence* layer and a *service* layer make up the bottom layers of your back-end. You'll persist guestbooks and their entries to a database. Your service layer calls your persistence layer, providing a buffer in case you wish to swap out your persistence technology later.

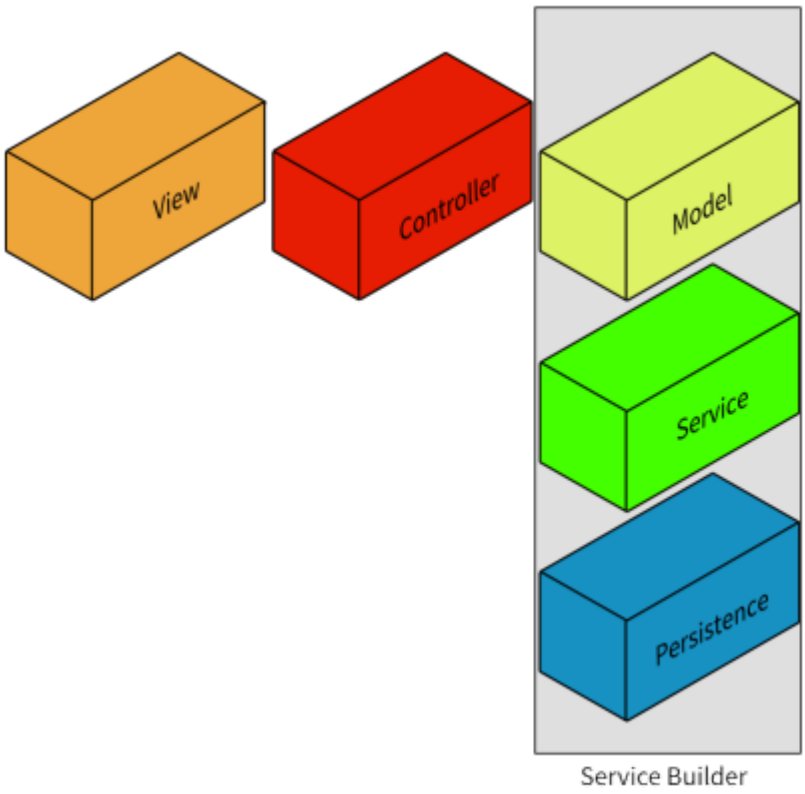

Figure 4.1: Service Builder generates the shaded layers of your application.

Service Builder is Liferay's code generation tool for defining object models and mapping those models to SQL databases. By defining your model in a single XML file, you can generate your object model (the M in MVC), your service layer, and your persistence layer all in one shot. At the same time, you can support every database Liferay DXP supports.

Ready to begin?

Let's Go!

CHAPTER 5

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 your application's Model, Service, and Persistence layers. Then you can add your application's necessary business logic.

5.1 Guestbook Application Design

The Guestbook application handles 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.

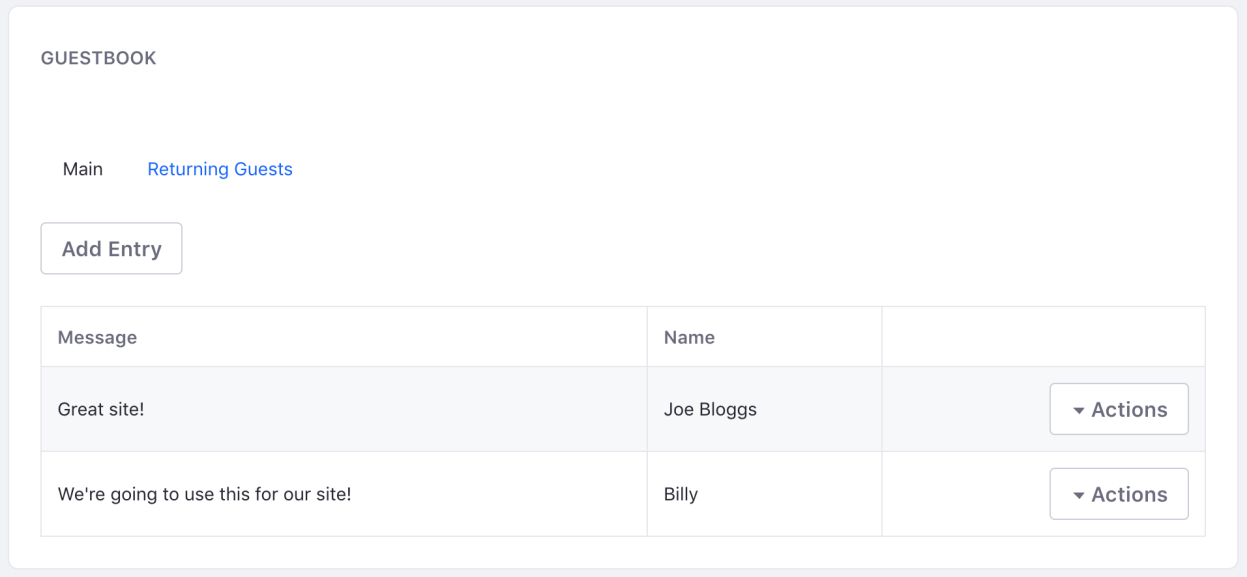

Figure 5.1: When you're done, the Guestbook supports multiple guestbooks and makes use of many Liferay features.

5.2 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, 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.

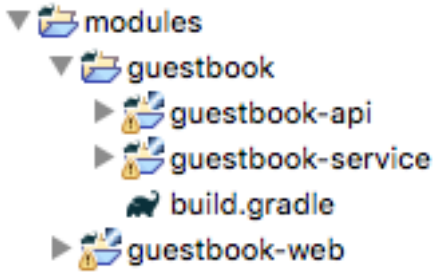

Figure 5.2: Your current project structure.

CHAPTER 6

GENERATING MODEL, SERVICE, AND PERSISTENCE LAYERS

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

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. Remove everything in the file below the `DOCTYPE`. Replace the file's opening contents with the following code:

   ```
   <service-builder dependency-injector="ds" package-path="com.liferay.docs.guestbook" mvcc-enabled="true">
       <author>liferay</author>
       <namespace>GB</namespace>
       <entity name="Guestbook" local-service="true" uuid="true" remote-service="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 and remote services for the entity, define its name, and specify that it should have a universally unique identifier (UUID).

3. The Guestbook requires only two fields: a primary key to identify it uniquely in the database, and a name. Add these fields:

   ```
   <!-- Guestbook fields -->

   <column name="guestbookId" primary="true" type="long" />
   <column name="name" type="String" />
   ```

This defines `guestbookId` as the entity's primary key of the type `long` and the name as a `String`.

4. Next, define the group instance. The `groupId` defines the ID of the Site in Liferay DXP where the entity instance exists. The `companyId` is the primary key of a portal instance.

```
<!-- Group instance -->

<column name="groupId" type="long" />
<column name="companyId" type="long" />
```

5. Next, add audit fields. These fields help you track owners of entity instances, along with those instances' create and modify dates:

```
<!-- Audit fields -->

<column name="userId" type="long" />
<column name="userName" type="String" />
<column name="createDate" type="Date" />
<column name="modifiedDate" type="Date" />
```

6. After this, add fields that support Liferay's workflow system. These provide fields in the database to track your entity's status as it passes through the workflow.

```
<!-- Status fields -->

<column name="status" type="int" />
<column name="statusByUserId" type="long" />
<column name="statusByUserName" type="String" />
<column name="statusDate" type="Date" />
```

7. Before the closing `</entity>` tag, add this finder definition:

```
    <finder name="GroupId" return-type="Collection">
        <finder-column name="groupId" />
    </finder>

</entity>
```

A finder generates a `get` method for retrieving Guestbook entities. The fields used by the finder define the scope of the data retrieved. This finder gets all Guestbooks by their `groupId`. This is how 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 `GuestbookEntry` entity:

1. Add the opening entity tag:

```
<entity name="GuestbookEntry" local-service="true" remote-service="true" uuid="true">
```

As with the `Guestbook` entity, you enable local and remote services, define the entity's name, and specify that it should have a UUID.

2. Add the fields that define the `GuestbookEntry`'s data:

```
<!-- Guestbook Entry fields -->

<column name="entryId" primary="true" type="long" />
<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 a `GuestbookEntry`. 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 foreign key to the `Guestbook` where this entry belongs.

3. Add fields to track the portal instance and group:

```
<!-- Group instance -->

<column name="groupId" type="long" />
<column name="companyId" type="long" />
```

4. Add audit fields:

```
<!-- Audit fields -->

<column name="userId" type="long" />
<column name="userName" type="String" />
<column name="createDate" type="Date" />
<column name="modifiedDate" type="Date" />
```

5. Add status fields to track workflow:

```
<!-- Status fields -->

<column name="status" type="int" />
<column name="statusByUserId" type="long" />
<column name="statusByUserName" type="String" />
<column name="statusDate" type="Date" />
```

6. When querying for `GuestbookEntrys`, you can order them by one or more columns. Since visitors sign `Guestbooks` in order by time, order your `GuestbookEntry` instances by the date they were created:

```
<order>
    <order-column name="createDate" order-by="desc" />
</order>
```

7. Add a finder that retrieves `GuestbooEntrys` by a combination of `groupId` and `guestbookId`. This supports Liferay DXP's multi-tenancy by only returning those entries that belong both to the current Site and the current Guestbook. After defining your finder add the closing entity tag:

```
    <finder name="G_G" return-type="Collection">
        <finder-column name="groupId" />
        <finder-column name="guestbookId" />
    </finder>
</entity>
```

8. Define exception types outside the <entity> tags, just before the closing </service-builder> tag:

```
    <exceptions>
        <exception>GuestbookEntryEmail</exception>
        <exception>GuestbookEntryMessage</exception>
        <exception>GuestbookEntryName</exception>
        <exception>GuestbookName</exception>
    </exceptions>

</service-builder>
```

   These generate exception classes you'll use later in try/catch statements.

9. 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 com-liferay-docs-guestbook → modules → guestbook → 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 Guestbook class that represents your model, Service Builder generates a system of classes that includes a Guestbook interface, a GuestbookBaseImpl abstract class that Service Builder manages, and a GuestbookImpl class that you can customize. With this design, you can customize your model and let Service Builder generate the tedious-to-write code. That's why Service Builder is a code generator for code generator haters.

Next, you'll create the service implementations.

Figure 6.1: The Model, Service, and Persistence Layer comprise a loose coupling design.

CHAPTER 7

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 `GuestbookEntryLocalServiceImpl`, 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 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.List`
    - `java.util.Date`
    - `com.liferay.portal.kernel.util.Validator`

Now you're ready to implement services for entries in `GuestbookEntryLocalServiceImpl`.

1. In the `com.liferay.docs.guestbook.service.impl` package, open `GuestbookEntryLocalServiceImpl`. Add this `addEntry` method:

```
public GuestbookEntry addGuestbookEntry(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();

    GuestbookEntry entry = guestbookEntryPersistence.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);

    guestbookEntryPersistence.update(entry);

    // Calls to other Liferay frameworks go here

    return entry;
}
```

Like the `addGuestbook` method, `addGuestbookEntry` 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 `updateGuestbookEntry` method:

```
public GuestbookEntry updateGuestbookEntry(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);

    GuestbookEntry entry =
        guestbookEntryPersistence.findByPrimaryKey(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);

    guestbookEntryPersistence.update(entry);
```

```
    // Integrate with Liferay frameworks here.

    return entry;
}
```

This method first retrieves the entry and updates its data to reflect what the user submitted, including its date modified.

3. Add two `deleteGuestbookEntry` methods:

```
public GuestbookEntry deleteGuestbookEntry(GuestbookEntry entry)
    throws PortalException {

    guestbookEntryPersistence.remove(entry);

    return entry;
}

public GuestbookEntry deleteGuestbookEntry(long entryId) throws PortalException {

    GuestbookEntry entry =
        guestbookEntryPersistence.findByPrimaryKey(entryId);

    return deleteGuestbookEntry(entry);
}
```

These methods delete entries using the `entryId` or the object. If you supply the `entryId`, the object is retrieved and passed to the method that deletes the object.

4. Add the methods for getting `GuestbookEntry` objects:

```
public List<GuestbookEntry> getGuestbookEntries(long groupId, long guestbookId) {
    return guestbookEntryPersistence.findByG_G(groupId, guestbookId);
}

public List<GuestbookEntry> getGuestbookEntries(long groupId, long guestbookId,
        int start, int end) throws SystemException {

    return guestbookEntryPersistence.findByG_G(groupId, guestbookId, start,
            end);
}

public List<GuestbookEntry> getGuestbookEntries(long groupId, long guestbookId,
        int start, int end, OrderByComparator<GuestbookEntry> obc) {

    return guestbookEntryPersistence.findByG_G(groupId, guestbookId, start,
            end, obc);
}

public GuestbookEntry getGuestbookEntry(long entryId) throws PortalException {
    return guestbookEntryPersistence.findByPrimaryKey(entryId);
}

public int getGuestbookEntriesCount(long groupId, long guestbookId) {
    return guestbookEntryPersistence.countByG_G(groupId, guestbookId);
}
```

These methods, like the getters in `GuestbookLocalServiceImpl`, call the finders you generated with Service Builder. These `getGuestbookEntries*` 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 GuestbookEntryNameException();
    }

    if (!Validator.isEmailAddress(email)) {
        throw new GuestbookEntryEmailException();
    }

    if (Validator.isNull(entry)) {
        throw new GuestbookEntryMessageException();
    }
}
```

   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.List`
   - `java.util.Date`
   - `com.liferay.portal.kernel.util.Validator`

Nice work! These local service methods implement the services that are referenced in the portlet class.

7.1 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 create a web application that uses it.

CHAPTER 8

BUILDING THE WEB FRONT-END

You now have a back-end: you created a `service.xml` file to define your application's data model, and generated the back-end code (the model, service, and persistence layers) with Service Builder. You also added service methods using the appropriate extension points: your entities' `*LocalServiceImpl` classes. Now you can add a front-end to match new back-end, creating a fully functional application.

You'll create two portlets: the Guestbook portlet for users to use to add entries and in a later step, a Guestbook Admin portlet for administrators to add Guestbooks.

Starting with this step, source code is provided in case you get stuck.

Ready to begin?

Let's Go!

CHAPTER 9

CREATING THE WEB PROJECT

Building the Web Front-End Step 1 of 11

Your first step is to create another 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.

Modules can be web modules or OSGi modules. Since you'll be creating a Liferay MVC Portlet, you'll create an OSGI module. The OSGi container in Liferay DXP can run any OSGi module. Each module is packaged as a JAR file that contains a manifest file. The manifest is needed for the container to recognize the module. Technically, a module that contains only a manifest is still valid. Of course, such a module wouldn't be very interesting.

You already created Service Builder modules. Now you'll create your MVC Portlet module. For the purpose of this tutorial, you'll create your modules inside your Liferay Workspace.

1. In Liferay Dev Studio DXP, select *File → 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.
    - The Liferay version is *7.2*.
    - 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`)

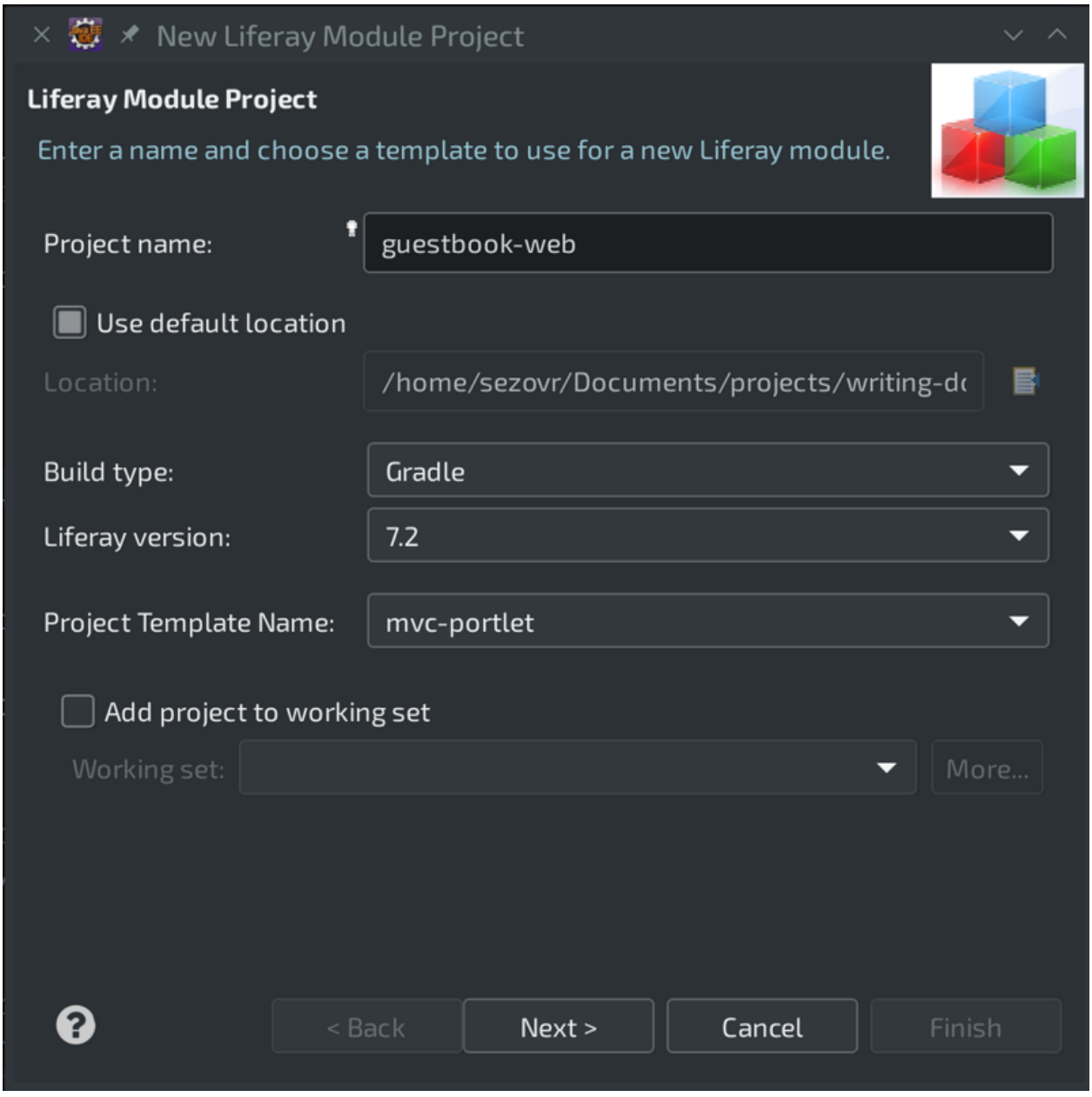

Figure 9.1: Complete the New Module Project wizard.

- Language properties (also in `/src/main/resources`)

Your new module project is a *portlet* application. You'll learn what that is in a moment, but first there's some housekeeping to do.

In larger projects, it is important to have all of your files and modules well organized. Since the `guestbook-web` module really goes with your Service Builder modules, it should be in the `guestbook` folder.

1. In the *Project Explorer*, right-click on `guestbook-web` and select *Move*.
2. In the window that appears, click *Browse*, choose the `guestbook` folder and then click *OK*.

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

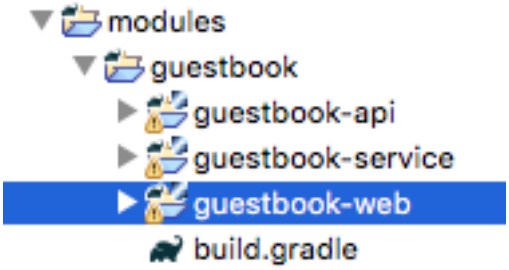

Figure 9.2: After you move it, all of your modules are in the same folder..

Note: Sometimes Eclipse refuses to move your project. If that happens, close Eclipse, use your operating system's file manager to move the `guestbook-web` folder into the `guestbook` folder, and then restart Eclipse.

You're now ready to begin writing your front-end, but first some explanation is in order.

9.1 What is a Portlet?

Web applications can be simple or complex: they might display an article or calculate your taxes. 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. Portlets can do this or share the page with many applications at the same time. Liferay DXP's framework takes this into account at every step.

9.2 What is a Component?

Liferay MVC Portlets are *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

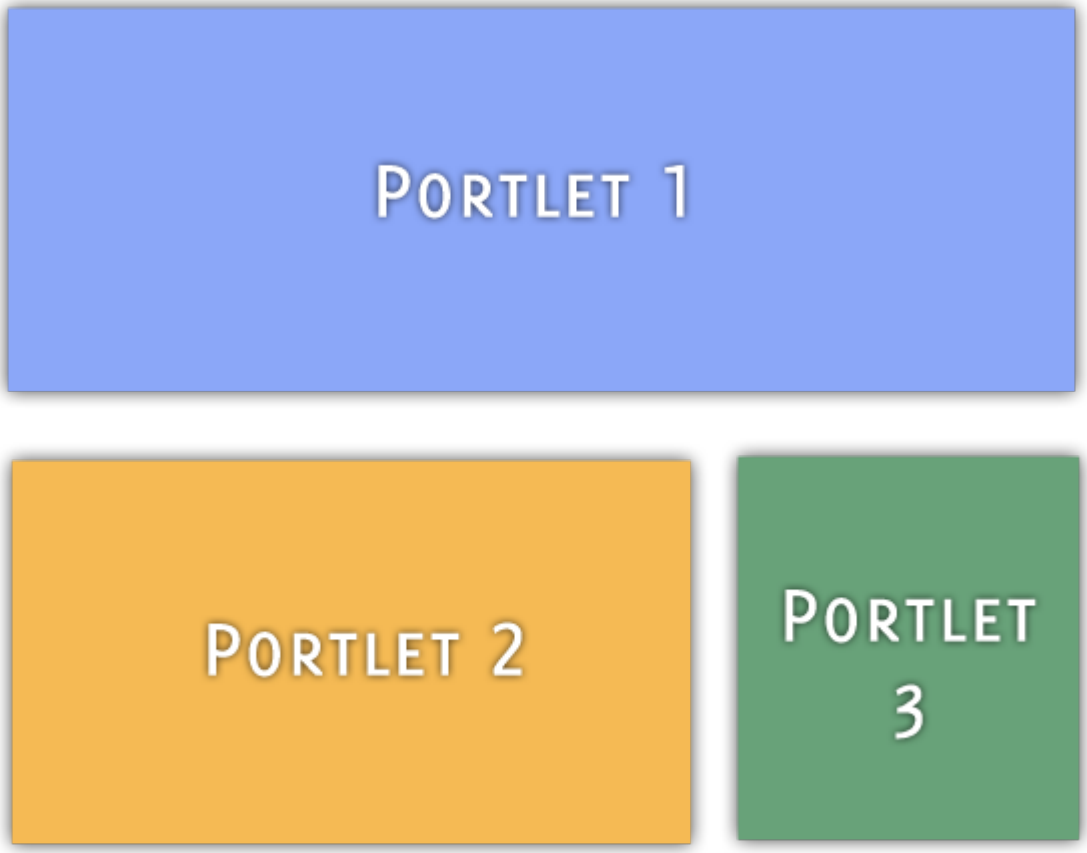

Figure 9.3: Many Liferay applications can run at the same time on the same page.

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.

9.3 Deploying the Application

Even though all you've done is generate it, the `guestbook-web` project is ready to be built and deployed.

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.
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. Click the menu button at the top left and select *Site Builder* → *Pages*.
5. Click the  button at the top right to add a Public Page.
6. Choose *Widget Page* and name it *Guestbook*.
7. Choose the *One Column* layout and click *Save*.
8. Click *Go to Site* on the left, and then navigate to your new Guestbook page.
9. Click *Add* () in the upper right hand corner.
10. Select *Widgets*. In the Applications list, your application appears in the Sample category. Its name is `Guestbook`.

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

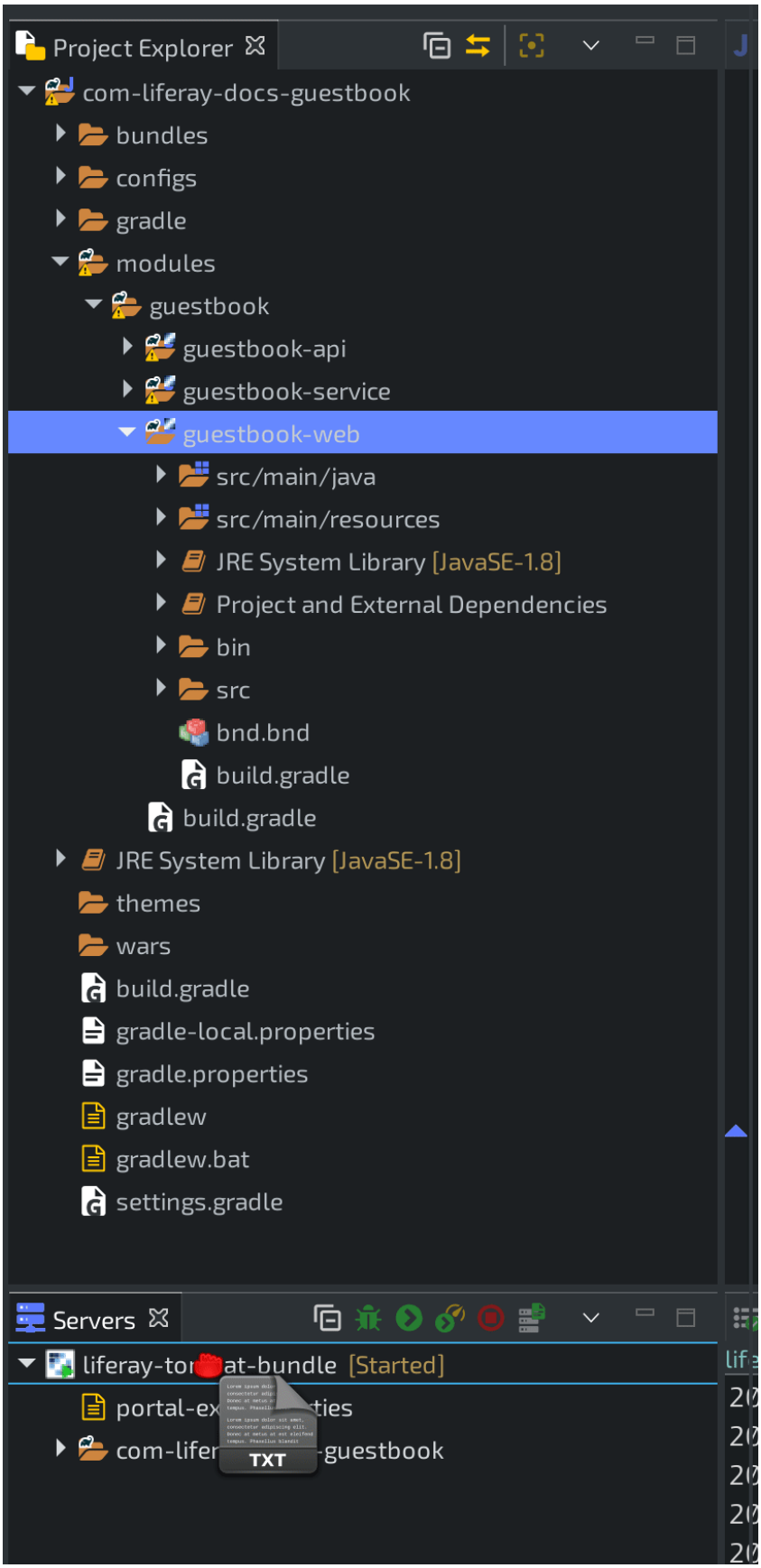

Figure 9.4: Drag and drop the module.

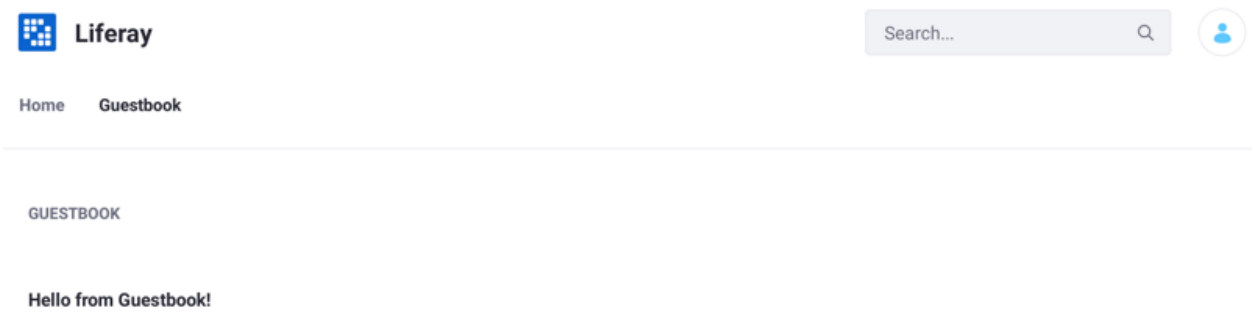

Figure 9.5: This is your new page with the Guestbook application that you created.

CHAPTER 10

DEFINING THE COMPONENT METADATA PROPERTIES

```
<p id="stepTitle">Building the Web Front-End</p><p>Step 2 of 11</p>
```

When users add applications to a page, they pick them 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=/guestbook/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 a folder in `src/main/resources/META-INF/resources` called `/guestbook`. You'll wind up creating two folders there for the two different portlets you'll create: `guestbook` and `guestbook-admin`. For now, just create the `guestbook` folder:

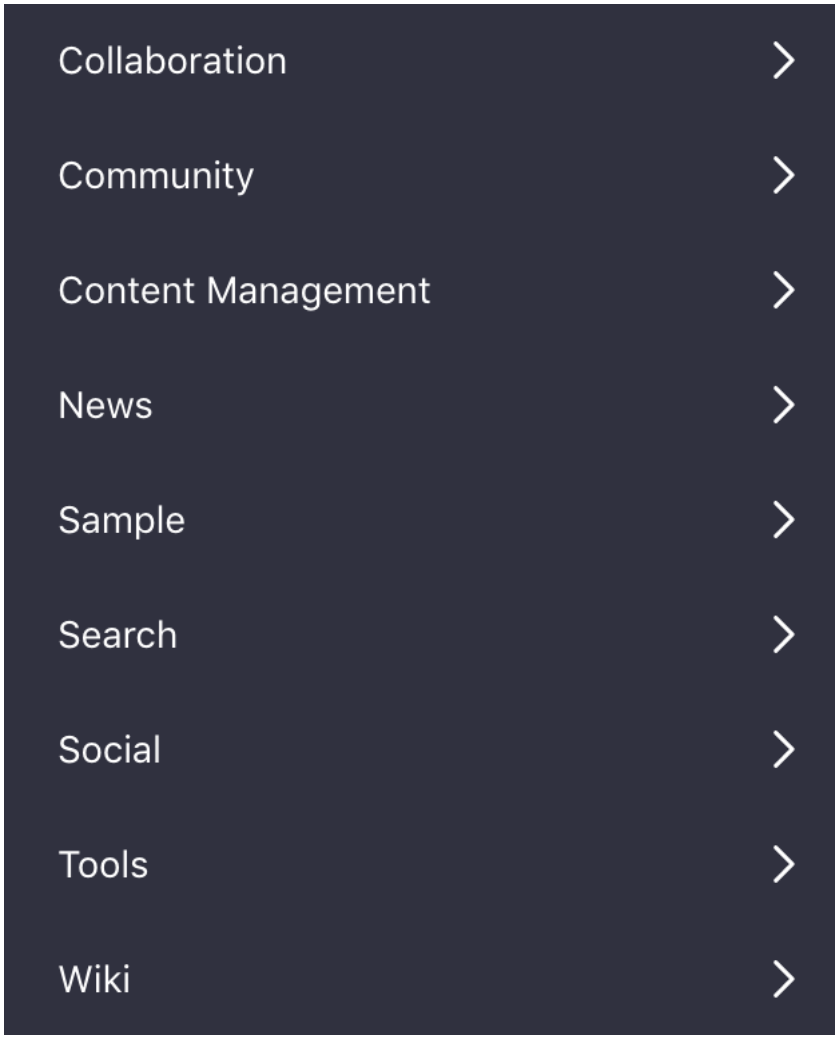

Figure 10.1: Users choose applications from a list of display categories.

1. Open `src/main/resources`, then open `META-INF`. Right-click on the `resources` folder and select *New → Folder*.
2. Name the folder *guestbook* and hit *Enter* (or click OK).
3. Drag `view.jsp` and drop it onto the `guestbook` folder to move it there.
4. Open `view.jsp` and modify the path to `init.jsp` to include it from the parent folder:

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

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 *Widgets*.

5. Open the *Social* category and drag and drop the *Guestbook* widget 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.

CHAPTER 11

Creating Portlet Keys

Building the Web Front-End Step 3 of 11

PortletKeys manage important things like the portlet name or other repeatable, commonly used variables in one place. This way, if you must change the portlet's name, you can do it in one place and then reference it in every class that needs it. Keys are created in a PortletKeys class and then referenced in a component property.

Follow these steps to create your application's PortletKeys:

1. Open the com.liferay.docs.guestbook.constants package.
2. Open GuestbookPortletKeys and make sure there's 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";
   ```

3. Save the file.
4. 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!).
5. Save GuestbookPortlet.

Nice job!

Next, you'll integrate your application with the back-end you generated with Service Builder.

CHAPTER 12

INTEGRATING THE BACK-END

```
<p id="stepTitle">Building the Web Front-End</p><p>Step 4 of 11</p>
```

To use your Service Builder-generated back-end in your front-end, you must tell your front-end project that the back-end exists and where to find it.

For the web module to access the generated services, you must make it aware of the API and service modules:

1. Open `guestbook-web`'s `build.gradle` file and add these dependencies:

   ```
   compileOnly project(":modules:guestbook:guestbook-api")
   compileOnly project(":modules:guestbook:guestbook-service")
   ```

2. Save the file. 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
   private GuestbookEntryLocalService _guestbookEntryLocalService;

   @Reference
   private GuestbookLocalService _guestbookLocalService;
   ```

   Note that it's Liferay's code style to add class variables this way. The `@Reference` annotation causes Liferay's OSGi container to inject references to your generated services so you can use them.

4. Press Ctrl-Shift-O to add the import you need:

   - `org.osgi.service.component.annotations.Reference`

Now you're ready to begin building your front-end.

CHAPTER 13

CREATING AN ADD ENTRY BUTTON

```
<p id="stepTitle">Building the Web Front-End</p><p>Step 5 of 11</p>
```

A guestbook application is pretty simple, right? People come to your site and post their names and brief messages. Others can read these entries and post their own.

When you created your project, it generated a file named `view.jsp`, which you've already moved to `src/main/resources/META-INF/resources/guestbook` 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 Liferay convention, we declare imports and tag library declarations in an `init.jsp` file. The other JSP files in the application import `init.jsp`. This reserves JSP dependency management to 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.

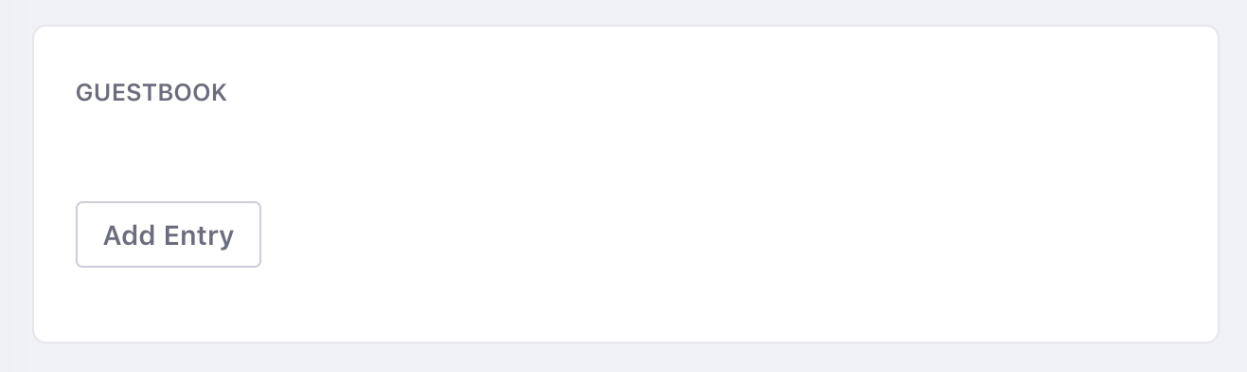

Figure 13.1: Your new button is awesome, but it doesn't work yet.

CHAPTER 14

GENERATING PORTLET URLS

```
<p id="stepTitle">Building the Web Front-End</p><p>Step 6 of 11</p>
```

Since users can place multiple portlets on a single page, 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="/guestbook/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="/guestbook/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.

CHAPTER 15

LINKING TO ANOTHER PAGE

```
<p id="stepTitle">Building the Web Front-End</p><p>Step 7 of 11</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/guestbook` 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. Next, you need a scriptlet that helps determine the function the user accessed. You named this JSP `edit_entry.jsp` because it's used both for adding and editing. Add this scriptlet to add logic for determining which function the user wants:

   ```
   <%

   long entryId = ParamUtil.getLong(renderRequest, "entryId");

   GuestbookEntry entry = null;
   if (entryId > 0) {
     entry = GuestbookEntryLocalServiceUtil.getGuestbookEntry(entryId);
   }

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

   %>
   ```

   If an `entryId` greater than `0` is found in the request, editing a `GuestbookEntry` is assumed. Otherwise, the user is adding.

5. You'll create two URLs: one in the next step to submit the form and one in this step to go back to `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="/guestbook/view.jsp"></portlet:param>
</portlet:renderURL>
```

Next, you must create a new URL for submitting the form. This is a different kind of URL, for it triggers a portlet action.

CHAPTER 16

FORMS AND ACTION URLS

```
<p id="stepTitle">Building the Web Front-End/p><p>Step 8 of 11</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.

16.1 Action URLs

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" />
```

You now have the two required URLs for your form.

16.2 Forms

The form for creating guestbook entries has three fields: one for the name of the person submitting the entry, one for the person's email address, 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:model-context bean="<%= entry %>" model="<%= GuestbookEntry.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>
```

Liferay uses its Alloy UI tag library to create forms.

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.

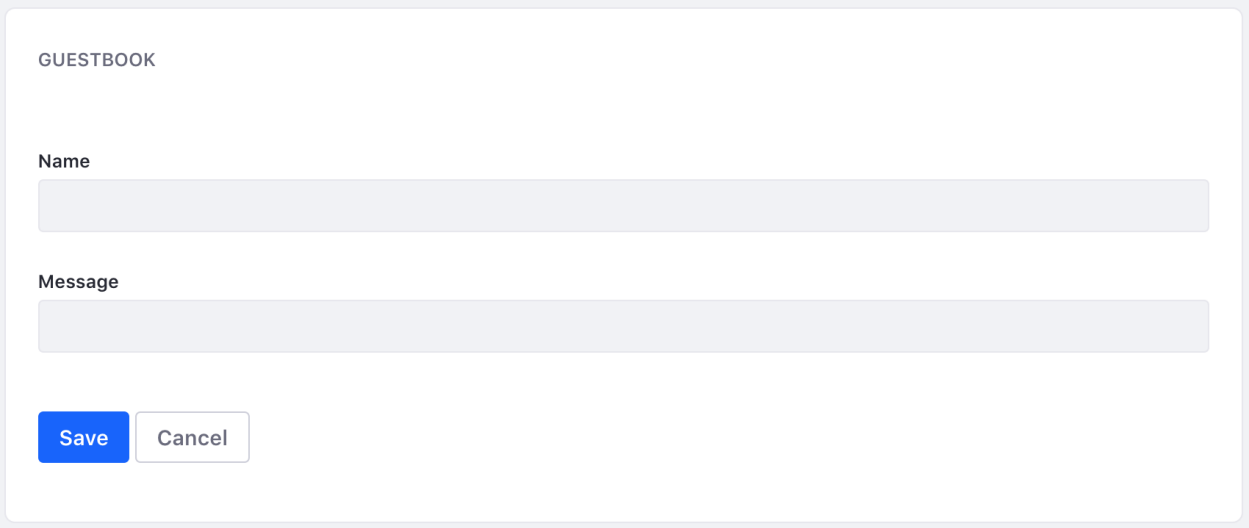

Figure 16.1: This is the Guestbook application's form for adding entries.

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

CHAPTER 17

IMPLEMENTING PORTLET ACTIONS

```
<p id="stepTitle">Building the Web Front-End</p><p>Step 9 of 11</p>
```

When users submit the form, your application stores the form data for display in the guestbook. This is where you call the back-end you generated to store the data for later retrieval in the database.

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 is 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.

17.1 Creating an Add Entry Action

1. Open `GuestbookPortlet`.
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.

The following method implements adding a guestbook entry using the back-end you generated with Service Builder:

```
public void addEntry(ActionRequest request, ActionResponse response)
            throws PortalException {

        ServiceContext serviceContext = ServiceContextFactory.getInstance(
            GuestbookEntry.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 {

            _guestbookEntryLocalService.updateGuestbookEntry(
                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", "/guestbook/edit_entry.jsp");
        }

    }
    else {

        try {
            _guestbookEntryLocalService.addGuestbookEntry(
                serviceContext.getUserId(), guestbookId, userName, email,
                message, serviceContext);

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

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

            PortalUtil.copyRequestParameters(request, response);

            response.setRenderParameter(
                "mvcPath", "/guestbook/edit_entry.jsp");
        }
    }
}
```

1. Replace your existing `addEntry` method with the above method.
2. Press [CTRL]+[SHIFT]+O to organize imports.

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. It also gets a `ServiceContext` so information about the request's current context can be retrieved, such as the ID of the current user. 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. Note the setting of the `mvcPath` render parameter to direct processing to the proper JSP based on what happens.

17.2 Creating a Delete Entry Action

Next, create an action that deletes an entry:

```
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(
            GuestbookEntry.class.getName(), request);

        try {

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

            _guestbookEntryLocalService.deleteGuestbookEntry(entryId);
        }

        catch (Exception e) {
            Logger.getLogger(GuestbookPortlet.class.getName()).log(
                Level.SEVERE, null, e);
        }
}
```

This action accepts an `entryId` from the request and calls the service to delete it.

But there's something missing, isn't there? These methods expect a `guestbookId` to be in the request, so the `GuestbookEntry` can be connected to its `Guestbook`. You'll create that next, as well as a mechanism for viewing guestbook entries.

CHAPTER 18

DISPLAYING GUESTBOOK ENTRIES

```
<p id="stepTitle">Building the Web Front-End</p><p>Step 10 of 11</p>
```

To display guestbook entries, you must do the reverse of what you did to store them: retrieve them the database, loop through them, and present them on the page. To do this, you must override the default MVC Portlet `render` method so you can tell your portlet how to render itself.

18.1 Rendering the Portlet

1. Add the following `render` method to `GuestbookPortlet`:

```
@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 render method checks for guestbooks in the current Site. If there aren't any, it creates one. The guestbookId that it has (either the first one or one that has been selected in functionality you haven't written yet) is set in the request object so that it becomes the current guestbook.

2. Press [CTRL]+[SHIFT]+O to organize imports and then save the file.

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`

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.

18.2 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. Replace the contents of `view.jsp` with this 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="/guestbook/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="<%=GuestbookEntryLocalServiceUtil.getGuestbookEntriesCount()%>">
<liferay-ui:search-container-results
    results="<%=GuestbookEntryLocalServiceUtil.getGuestbookEntries(scopeGroupId.longValue(),
                    guestbookId, searchContainer.getStart(),
                    searchContainer.getEnd())%>" />

<liferay-ui:search-container-row
    className="com.liferay.docs.guestbook.model.GuestbookEntry" modelVar="entry">

    <liferay-ui:search-container-column-text property="message" />

    <liferay-ui:search-container-column-text property="name" />

    <liferay-ui:search-container-column-jsp
        align="right"
        path="/guestbook/entry_actions.jsp" />

</liferay-ui:search-container-row>

<liferay-ui:search-iterator />

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

2. You've used a lot of new objects in this JSP, so you must declare them in `init.jsp`. Replace the contents of `init.jsp` with 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"%>
<%@ taglib uri="http://liferay.com/tld/frontend" prefix="liferay-frontend" %>
<%@ taglib uri="http://liferay.com/tld/security" prefix="liferay-security" %>

<%@ page import="java.util.List" %>
<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>
<%@ page import="com.liferay.portal.kernel.util.HtmlUtil" %>
<%@ page import="com.liferay.portal.kernel.util.WebKeys" %>
<%@ 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.GuestbookEntryLocalServiceUtil" %>
<%@ page import="com.liferay.docs.guestbook.service.GuestbookLocalServiceUtil" %>
<%@ page import="com.liferay.docs.guestbook.model.GuestbookEntry" %>

<liferay-theme:defineObjects />

<portlet:defineObjects />
```

Many of these objects, such as `HtmlUtil`, `ParamUtil`, and `StringPool`, are Liferay helper utilities that enable you with a single line of code do things like extract parameters, escape data, or provide `String`s that otherwise have to be escaped.

Save your work.

18.3 Creating an Actions JSP

Actions can be performed on your entities once they're stored. Users who enter Guestbook entries may wish to edit them or delete them. Now you'll provide that functionality.

1. Right-click on the `src/main/resources/META-INF/resources/guestbook` folder and select *New* → *File*.
2. Name the file `entry_actions.jsp`.
3. Paste the following code into the file:

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

<%
String mvcPath = ParamUtil.getString(request, "mvcPath");

ResultRow row = (ResultRow)request.getAttribute(WebKeys.SEARCH_CONTAINER_RESULT_ROW);

GuestbookEntry entry = (GuestbookEntry)row.getObject();
%>

<liferay-ui:icon-menu>

        <portlet:renderURL var="editURL">
            <portlet:param name="entryId"
                value="<%= String.valueOf(entry.getEntryId()) %>" />
            <portlet:param name="mvcPath" value="/guestbook/edit_entry.jsp" />
        </portlet:renderURL>

        <liferay-ui:icon image="edit" message="Edit"
            url="<%=editURL.toString() %>" />

        <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() %>" />

</liferay-ui:icon-menu>
```

You may have noticed this JSP was included in the Search Container rows in your `view.jsp`. As the Search Container loops through Guestbook entries, this JSP generates an Actions button for each of them containing two functions: a call to your `addEntry` method (which both adds and edits) and a call to your `deleteEntry` method. Both calls supply the current `guestbookId` and `entryId` parameters so the Action method has everything it needs to call the service method that does the work.

Awesome! You've now completed the first iteration of your Guestbook application.

Next you'll review what's been done so far, and you'll deploy and test your application.

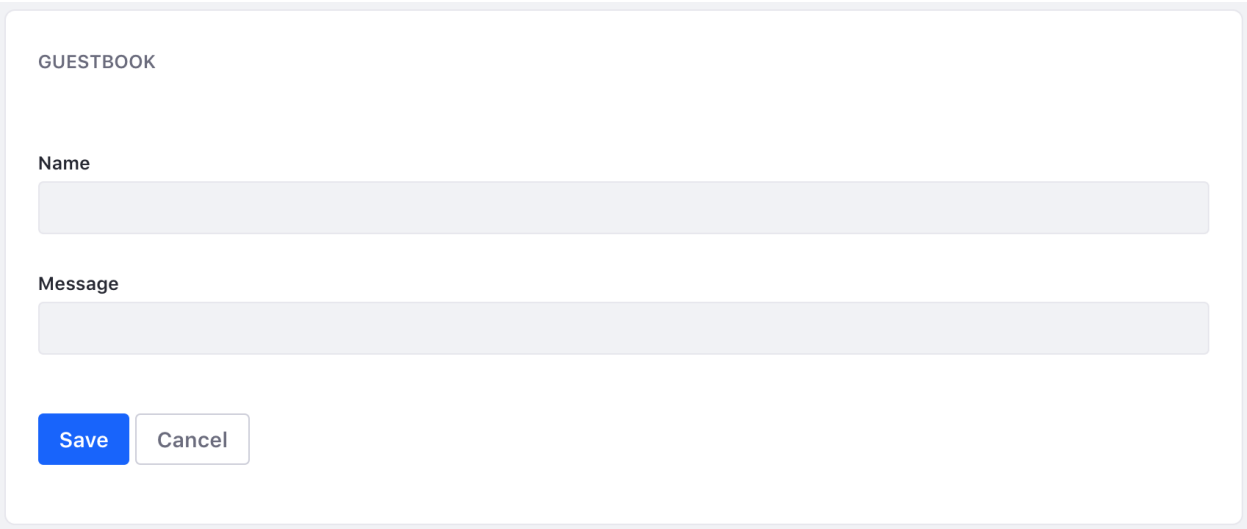

Figure 18.1: You have a form to enter information.

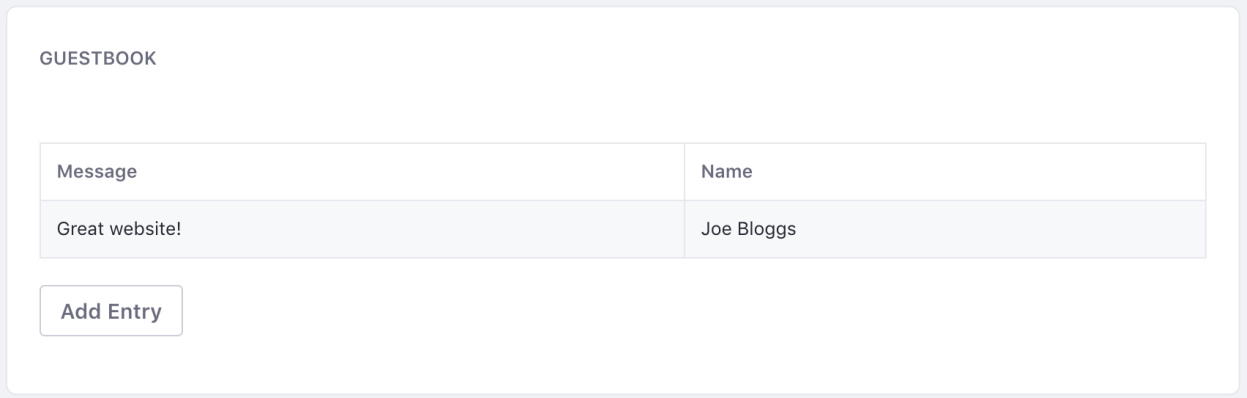

Figure 18.2: Submitted entries are displayed here.

CHAPTER 19

FITTING IT ALL TOGETHER

Building the Web Front-End Step 11 of 11

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.

19.1 The Back-End

The first thing you did was generate back-end services for your application using Liferay's object-relational mapper, Service Builder.

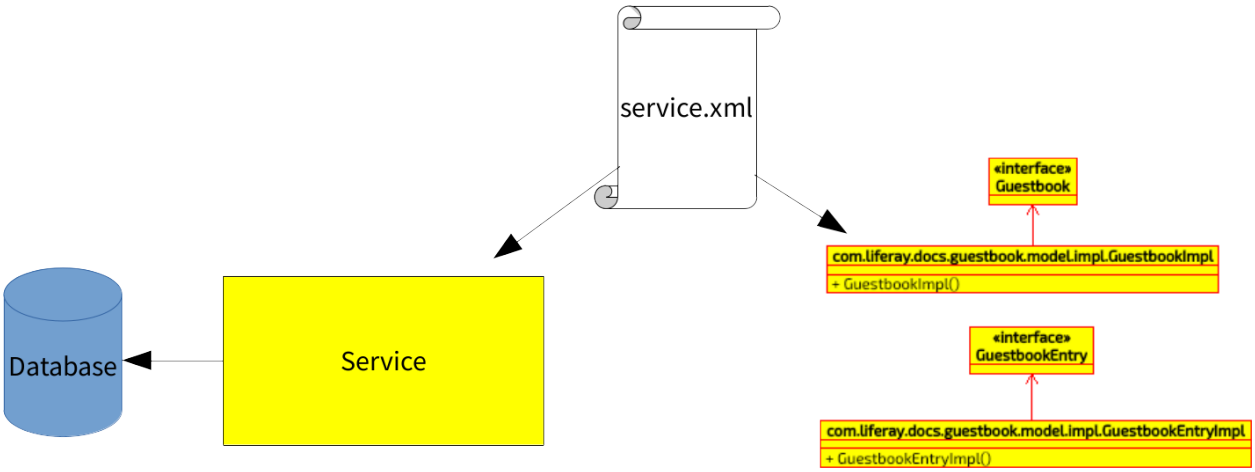

Figure 19.1: Service Builder makes generating your database entities and your Java objects a snap.

Using a single `service.xml` file, you generated your object model, SQL to create your tables, a persistence layer to perform all CRUD operations on your data, and a service layer for your business logic. Then you added business logic to your service layer to expose the persistence layer's functionality according to your business rules.

With all of that ready, it was time to build a front-end client.

19.2 The Front-End

Once you completed your back-end, you decided to develop a web front-end using Liferay's MVC Portlet framework. You generated a Liferay MVC Portlet project and used its Model View Controller development paradigm to implement a user interface for your back-end functionality.

First, you created a default view in `view.jsp`. You created a button there that links to `edit_entry.jsp`, which is used for both adding and editing Guestbook entries (though you haven't implemented editing yet). Here, you created a form to capture information from Users adding Guestbook entries. The form's Action URL directs processing to your controller's portlet action of the same name. This action extracts the data from the form (in the `ActionRequest` object) and passes it to the business logic you created in your service layer.

After the business logic runs, your controller passes processing back to `view.jsp`, where the new Guestbook entry is displayed.

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.

19.3 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. Like you did before, use your administrative account to remove the Guestbook from the page and add it again.
6. Click *Add Entry*.
7. Enter a *Name*, *Message*, and *Email Address*.
8. Click *Submit*.
9. Verify that your entry appears.

19.4 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.

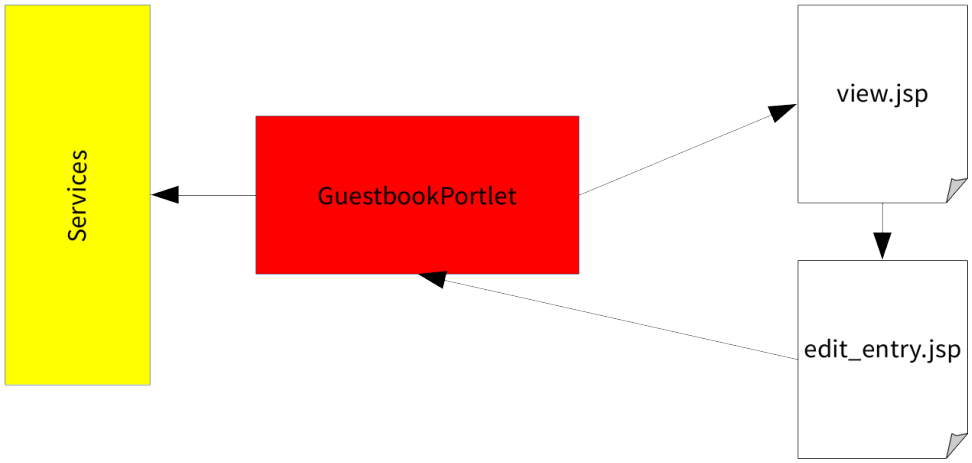

Figure 19.2: The controller directs page flow in an MVC Portlet application.

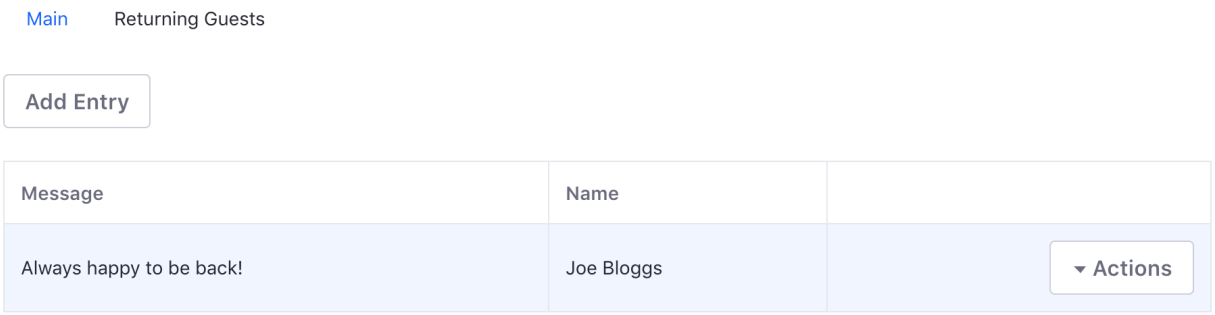

Figure 19.3: Your first guestbook and entry appears. Nice job!

CHAPTER 20

WRITING AN ADMINISTRATIVE PORTLET

The Guestbook 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. But at this point, 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'll 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, while users use the Guestbook portlet to create entries.

In short, this is a simple application with a simple interface:

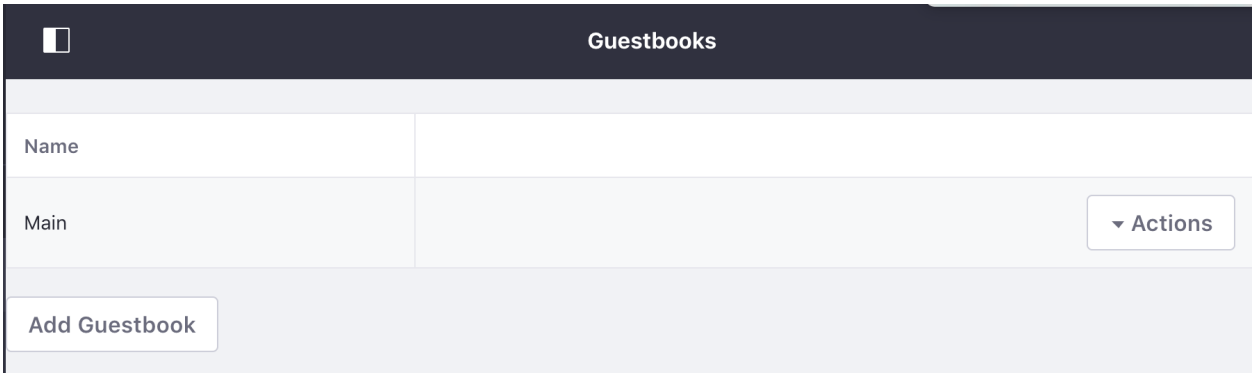

Figure 20.1: The Guestbook Admin portlet lets administrators manage Guestbooks.

If you get stuck, source code for this step is provided.
Are you ready to begin?
Let's Go!

CHAPTER 21

CREATING THE CLASSES

Writing the Guestbook Admin App Step 1 of 6

Because the Guestbook and Guestbook Admin applications should be bundled together, you'll create the new application inside the `guestbook-web` project. 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.

21.1 Panels and Categories

As described in the product menu tutorial, there are three sections of the product menu as illustrated below.

Each section is called a *panel category*. A panel category can hold various menu items called *panel apps*. In the illustration above, the Sites menu is open to reveal its panel apps and categories (yes, you can nest them).

The most natural place for the Guestbook Admin portlet is in the *Content & Data* 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:

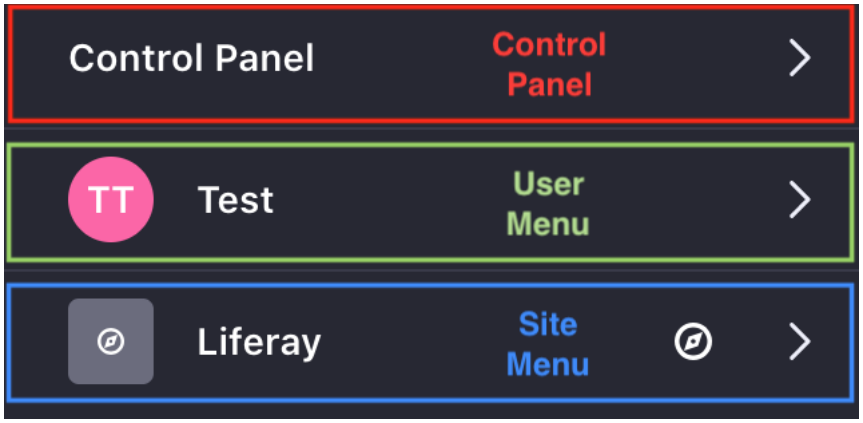

Figure 21.1: The product menu is split into three sections: the Control Panel, the User menu, and the Sites menu.

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 these dependencies:

```
compileOnly group: "com.liferay", name: "com.liferay.application.list.api"
compileOnly group: "com.liferay", name: "com.liferay.petra.string"
```

2. After saving the file, 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.

CHAPTER 22

ADDING METADATA

Writing the Guestbook Admin App Step 2 of 6

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 (it's in the `com.liferay.docs.guestbook.constant` package):

   ```
   public static final String GUESTBOOK_ADMIN =
     "com_liferay_docs_guestbook_portlet_GuestbookAdminPortlet";
   ```

   Save the file.

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=/guestbook_admin/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. The Guestbook Admin portlet goes here because 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 Builder → Content & Data*. 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.

CHAPTER 23

UPDATING YOUR SERVICE LAYER

Writing the Guestbook Admin App Step 3 of 6

Earlier, you wrote an addGuestbook service method in GuestbookLocalServiceImpl and only used it to add a default guestbook. 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.

23.1 Adding Guestbook Service Methods

Remember that when working with Service Builder, you define your service in the *Impl classes. After you add, remove, 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<GuestbookEntry> entries = _guestbookEntryLocalService.getGuestbookEntries(
                        serviceContext.getScopeGroupId(), guestbookId);

        for (GuestbookEntry entry : entries) {
                _guestbookEntryLocalService.deleteGuestbookEntry(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 only deleted the guestbook, the guestbook's orphaned entries would still exist in the database. 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. Add a reference to the `GuestbookEntry` local service, so it can be injected and used by the `deleteGuestbook` method:

```
@Reference
private GuestbookEntryLocalService _guestbookEntryLocalService;
```

By convention, Liferay adds these to the bottom of the class.

4. Use [CTRL]+[SHIFT]+O to update your imports, choosing `com.liferay.portal.kernel.exception.SystemExceptio` and then save `GuestbookLocalServiceImpl.java`.

5. 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`.

Note: If you prefer, you can use Blade CLI to run your Gradle tasks. If you have Blade CLI installed, go to the `guestbook-service` folder on your CLI and enter the command `blade gw buildService`. This runs Service Builder to build your services outside of Eclipse.

Now that you've finished updating the service layer, it's time to work on the Guestbook Admin portlet itself.

CHAPTER 24

DEFINING PORTLET ACTIONS

```
<p id="stepTitle">Writing the Guestbook Admin App</p><p>Step 4 of 6</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 all run 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).

24.1 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", "/guestbook_admin/edit_guestbook.jsp");
```

```
    }
}

@Reference
private 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",
        "/guestbook_admin/edit_guestbook.jsp");
```

Later, you'll use this for field validation and to show error messages to the user. Note that `/guestbook_admin/edit_guestbook.jsp` doesn't exist yet; you'll create it in the next step.

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", "/guestbook_admin/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",
        "/guestbook_admin/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. Import the logging classes from `java.util`. Save the file.

You now have your service methods and portlet action methods in place. Before you implement the Guestbook Admin portlet's user interface, you should update the Guestbook portlet so it can show users all the Guestbooks your administrators add.

CHAPTER 25

ADDING TABS TO THE GUESTBOOK PORTLET

```
<p id="stepTitle">Writing the Guestbook Admin App</p><p>Step 5 of 6</p>
```

Before you finish the Guestbook Admin portlet, you must prepare the Guestbook portlet's UI to display multiple Guestbooks. As administrators add Guestbooks using the Guestbook Admin portlet, users must be able to choose which Guestbook they want to sign. They'll do this using a series of tabs across the top:

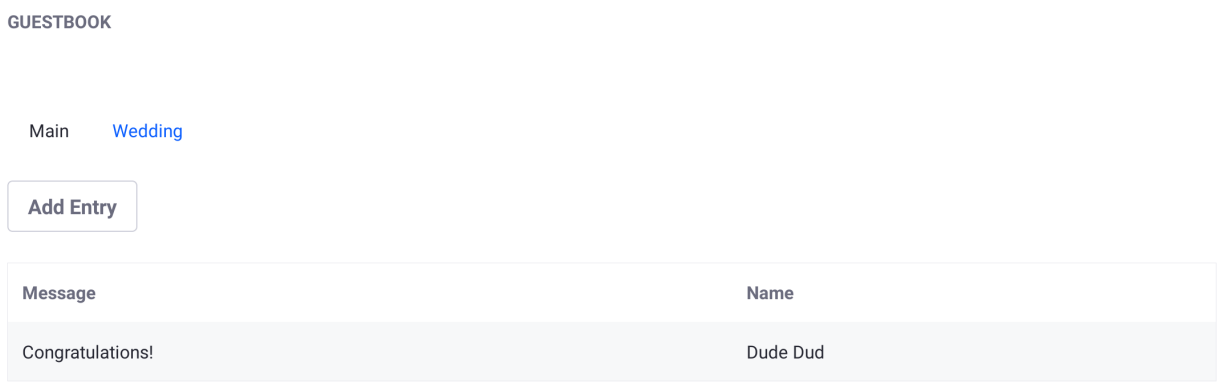

Figure 25.1: Users can click a tab to choose which Guestbook to sign.

This is surprisingly easy to do using Liferay's Alloy UI tag library:

1. Open `view.jsp` from the `src/main/resources/META-INF/resources/guestbook` folder.
2. After the first Java snippet (the one that gets the `guestbookId` out of the request), add this code:

```
<aui:nav cssClass="nav-tabs">

    <%
        List<Guestbook> guestbooks = GuestbookLocalServiceUtil.getGuestbooks(scopeGroupId);

            for (int i = 0; i < guestbooks.size(); i++) {
```

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

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

    %>

    <portlet:renderURL var="viewPageURL">
        <portlet:param name="mvcPath" value="/guestbook/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>
```

3. Save the file.

This code declares the AUI navigation tabs. Then a code scriptlet gets all the Guestbooks in this scope and loops through each one. As it examines them, it checks to see if the one it's examining is the current Guestbook. If so, a CSS style called `active` is applied.

After this, a new URL called `viewPageURL` is created that points to `view.jsp` with a `guestbookId` parameter containing the current Guestbook in the loop. Finally, an `<aui:nav-item>` tag declares the markup for the tab, using the CSS class, the URL containing the parameters to navigate to the new Guestbook, and the name to label it.

The loop continues until all the retrieved Guestbooks have tabs.

Awesome! You've updated the Guestbook portlet so it can display all the Guestbooks administrators add. Now it's time to provide a UI for your Guestbook Admin portlet so they can do just that.

CHAPTER 26

CREATING A USER INTERFACE

```
<p id="stepTitle">Writing the Guestbook Admin App</p><p>Step 6 of 6</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 appears next to an Actions button. The Actions button reveals options for editing the guestbook, configuring its permissions, or deleting it.

26.1 Step 1: Creating the Default View

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. In `src/main/resources/META-INF/resources`, create a folder called `guestbook_admin`, where you'll create your JSPs.
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="/guestbook_admin/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="/guestbook_admin/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 `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 specifies 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 displays 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="/guestbook_admin/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.

26.2 Step 2: Creating the Actions Button

Now create the `guestbook_actions.jsp` file that displays the list of possible actions for each guestbook.

Create a new file called `guestbook_actions.jsp` in your project's `/guestbook_admin` 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="/guestbook_admin/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 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.

26.3 Step 3: Creating the Edit Guestbook JSP

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.

Create a new file called `edit_guestbook.jsp` in your project's `/guestbook_admin` 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="/guestbook_admin/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, you use the `guestbookId` to retrieve the corresponding guestbook via a service call for edit. 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, its 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 is created with the following tag:

```
<aui:form action="<%= editGuestbookURL %>" name="<portlet:namespace />fm">
```

The form is submitted via the `editGuestbookURL`, which calls the Guestbook Admin portlet's `addGuestbook` or `updateGuestbook` action 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:

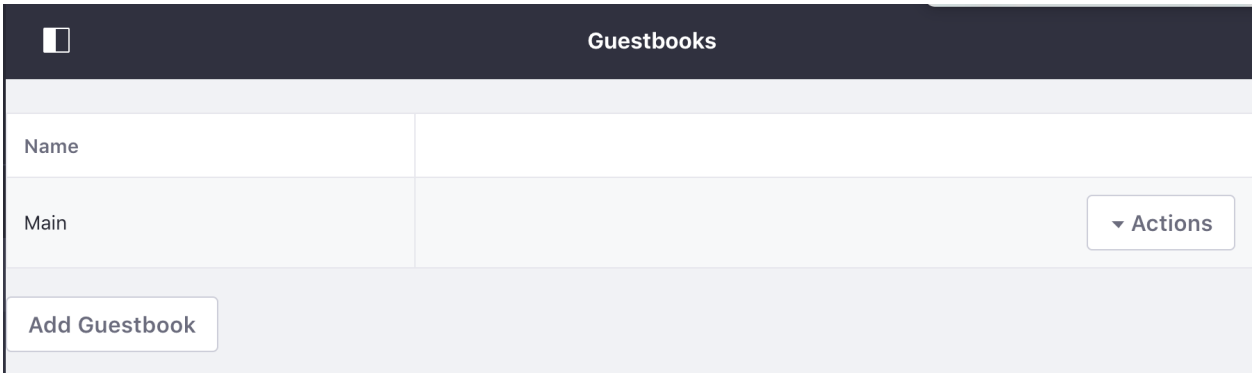

Figure 26.1: The Guestbook Admin portlet can add or edit guestbooks, configure their permissions, or delete them.

Save all your files and wait for redeploy. Test out the Guestbook Admin portlet! Try adding, editing, and deleting guestbooks.

Note: If you get "Guestbook is unavailable" errors, remove the modules from the server, redeploy them, and test again.

Now all the Guestbook application's primary functions work. There are still many missing features, however. For example, if there's ever an error, users never see it: all the code written so far just prints messages in the logs. Next, you'll learn how to display those errors to the user.

CHAPTER 27

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 provide feedback on these operations so users can know they completed. 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?

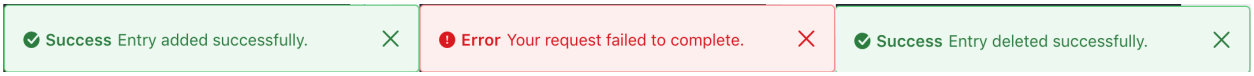

Figure 27.1: You can use Liferay's APIs to display helpful messages.

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.

If you get stuck, source code for this step is provided.
Ready to get started?
Let's Go!

CHAPTER 28

CREATING LANGUAGE KEYS

```
<p id="stepTitle">Displaying Messages and Errors</p><p>Step 1 of 3</p>
```

Modern applications should place messages and form field labels in a language keys files that hold multiple language translations. 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 Generating Translations.

Language keys are stored in the `Language.properties` file included in your `guestbook-web` module. `Language.properties` is the default, but you can create more 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.

CHAPTER 29

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 a Guestbook or Guestbook entry. This 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 `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.

CHAPTER 30

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 `guestbook/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 `guestbook_admin/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" />
   ```

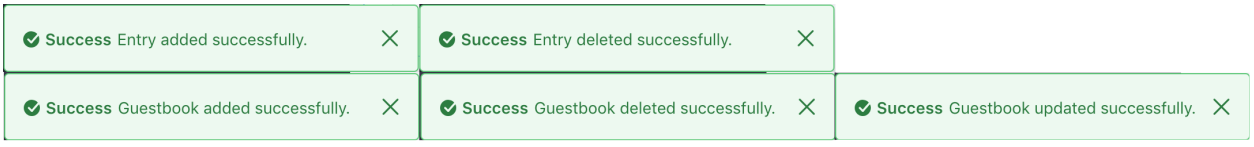

Figure 30.1: Now the message displays the value you specified in `Language.properties`.

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.

Look for the next part soon!

Chapter 31

Using Resources and Permissions

Your application is a great foundation to build on. What comes next? What if users want a Guestbook that's limited to certain trusted people? What if you don't want just any old user to go around editing or deleting people's Guestbook entries? 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

If you get stuck, source code for this step is provided.
Ready to start?
Let's Go!

CHAPTER 32

DEFINING PERMISSIONS

```
<p id="stepTitle">Implementing Permissions</p><p>Step 1 of 5</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 `GuestbookEntry`) 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 `GuestbookEntry`. A user may have permission to view one `GuestbookEntry`, view and edit another `GuestbookEntry`, and may not be able to access another `GuestbookEntry` at all. This is due to a resource permission.

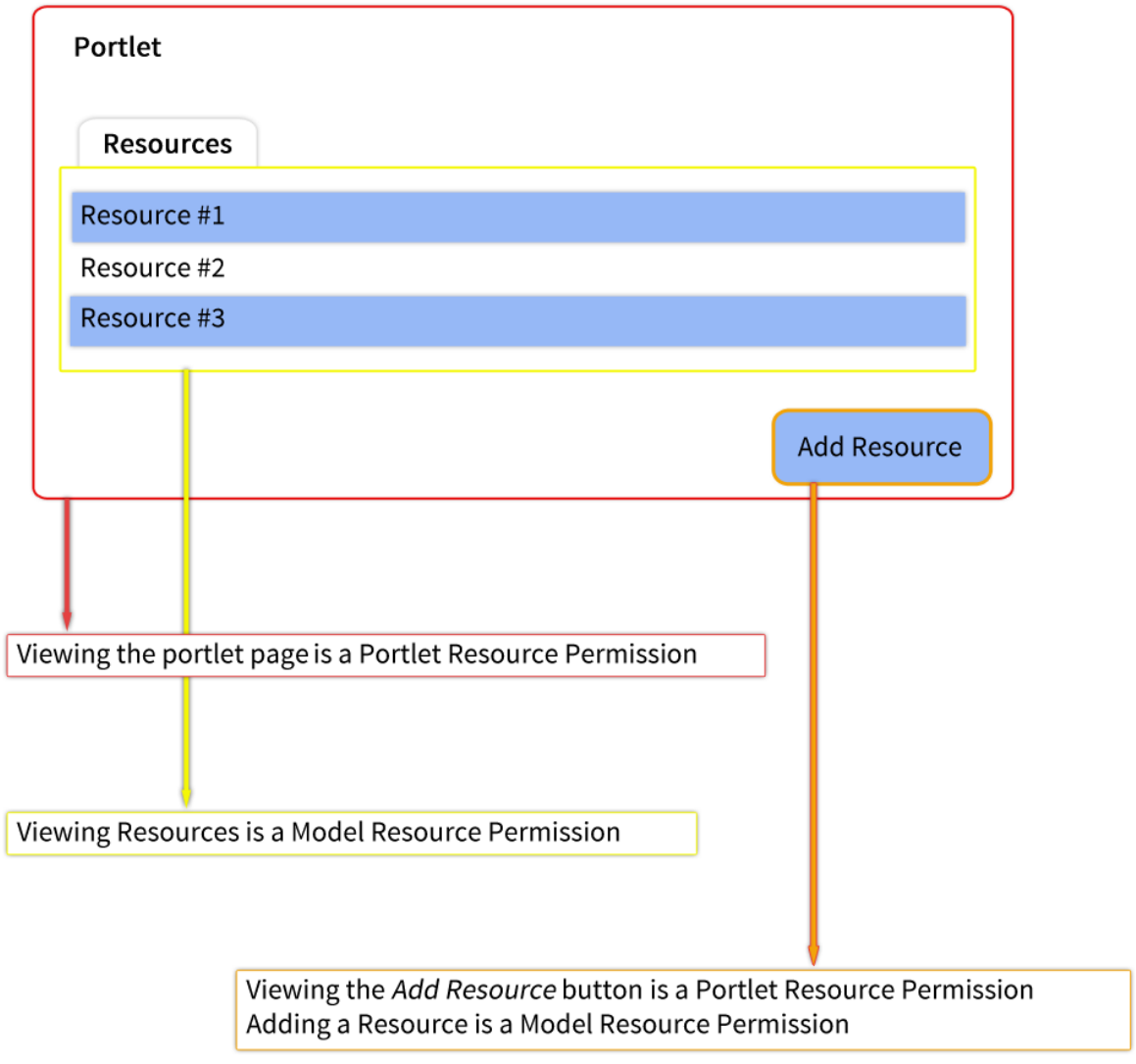

Figure 32.1: Portlet permissions and resource permissions cover different parts of the application.

32.1 Defining Model Permissions

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

1. In the /src/main/resources/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 7.2.0//EN" "http://www.liferay.com/dtd/liferay-resource-action-mapping_7_2_0.dtd">
```

4. Place the following wrapper tags into your default.xml file, below the DOCTYPE declaration:

```
<resource-action-mapping>

</resource-action-mapping>
```

   You'll define your resource and model permissions inside these tags.

5. Next, place the permissions for your com.liferay.docs.guestbook package between the <resource-action-mapping> tags:

```
<model-resource>
    <model-name>com.liferay.docs.guestbook</model-name>
    <portlet-ref>
        <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookPortlet</portlet-name>
        <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookAdminPortlet</portlet-name>
    </portlet-ref>
    <root>true</root>
    <permissions>
        <supports>
            <action-key>ADD_GUESTBOOK</action-key>
            <action-key>ADD_ENTRY</action-key>
            <action-key>VIEW</action-key>
        </supports>
        <site-member-defaults>
            <action-key>ADD_ENTRY</action-key>
        </site-member-defaults>
        <guest-defaults>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
            <action-key>ADD_GUESTBOOK</action-key>
            <action-key>ADD_ENTRY</action-key>
        </guest-unsupported>
    </permissions>
</model-resource>
```

   This defines the baseline configuration for the Guestbook and GuestbookEntry entities. The supported actions are ADD_GUESTBOOK and ADD_ENTRY. Site members can ADD_ENTRY by default, and 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 `GuestbookEntry` model permissions:

```
<model-resource>
    <model-name>com.liferay.docs.guestbook.model.GuestbookEntry</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 `GuestbookEntry` specific actions. By default, a Site member can add or view a Guestbook entry, and a guest can only view a Guestbook 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.

32.2 Defining Portlet Permissions

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 7.2.0//EN" "http://www.liferay.com/dtd/liferay-resource-action-mapping_7_2_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.

CHAPTER 33

REGISTERING YOUR PERMISSIONS IN THE DATABASE

Implementing Permissions Step 2 of 5

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.

Liferay DXP provides a complete API—integrated with Service Builder—for managing resources. 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.

33.1 Registering Guestbook Resources

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 into the base class your implementation class extends. 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.getModelPermissions());
```

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.

33.2 Registering Guestbook Entry Resources

1. Now you'll add resources for the GuestbookEntry entity. Open GuestbookEntryLocalServiceImpl.java from the same package. For addGuestbookEntry, add a line of code that adds resources for this entity, just before the return statement:

```
resourceLocalService.addResources(user.getCompanyId(), groupId, userId,
    GuestbookEntry.class.getName(), entryId, false, true, true);
```

2. Find updateEntry and add its resource action, also just before the return statement:

```
resourceLocalService.updateResources(
      user.getCompanyId(), serviceContext.getScopeGroupId(),
      GuestbookEntry.class.getName(), entryId,
      serviceContext.getModelPermissions());
```

3. For deleteGuestbookEntry, add this code just before the return statement and just after the call to guestbookEntryPersistence:

```
resourceLocalService.deleteResource(
   entry.getCompanyId(), GuestbookEntry.class.getName(),
   ResourceConstants.SCOPE_INDIVIDUAL, entry.getEntryId());
```

4. Hit Ctrl-Shift-O to add imports and save the file.

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 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 later. Now you must register permissions with the permissions system, so it knows how to check for them.

CHAPTER 34

REGISTERING PERMISSIONS WITH THE CONTAINER

```
<p id="stepTitle">Implementing Permissions</p><p>Step 3 of 5</p>
```

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.

1. Open the `build.gradle` file in your `guestbook-service` module.
2. Add the following three dependencies and save the file:

   ```
   compileOnly group: "com.liferay", name: "com.liferay.petra.function"
   compileOnly group: "com.liferay", name: "com.liferay.petra.model.adapter"
   compileOnly group: "com.liferay", name: "com.liferay.petra.reflect"
   ```

3. In your service module, 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 `GuestbookEntry` 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 `GuestbookEntry` entities instead of `Guestbook` entities (the imports have been left off in the snippet below):

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

@Component(immediate = true)
public class GuestbookEntryModelResourcePermissionRegistrar {

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

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

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

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

    @Reference
    private GuestbookEntryLocalService _guestbookEntryLocalService;

    @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;
}
```

3. Hit Ctrl-Shift-O to add the imports, and then save the file.

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:

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

@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;

}
```

3. Type Ctrl-Shift-O to add the imports. Save the file.

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

CHAPTER 35

ASSIGNING PERMISSIONS TO RESOURCES

Implementing Permissions Step 4 of 5

You've now defined your permissions and registered them in both the container and 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 final 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.

35.1 Creating a Guestbook Portlet Permission Helper

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.

35.2 Creating Model Permission Helpers

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 `GuestbookEntry` 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.GuestbookEntry;
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, GuestbookEntry 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.GuestbookEntry)",
            unbind = "-")
    protected void setEntryModelPermission(ModelResourcePermission<GuestbookEntry> modelResourcePermission) {

        _guestbookEntryModelResourcePermission = modelResourcePermission;
    }

    private static ModelResourcePermission<GuestbookEntry>_guestbookEntryModelResourcePermission;

}
```

This class is almost identical to `GuestbookModelPermission`. The only difference is that `GuestbookEntryPermission` is for the `GuestbookEntry` 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. Save the file.

3. Open `guestbook_actions.jsp` from the `guestbook_admin` folder. 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>
```

4. 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.

CHAPTER 36

CHECKING FOR PERMISSION IN JSPS

```
<p id="stepTitle">Implementing Permissions</p><p>Step 5 of 5</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.

36.1 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 check permissions for the guestbook tabs:

1. Open /guestbook/view.jsp and find the scriptlet that gets the guestbookId from the request. Below this are the <aui-nav> tags that generate the tabs. Remove those tags and all the code between them. In its place, add the following code, which is the same thing with the addition of permission checks:

```
<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.

4. Save the file.

Next, you'll add permission checking to `entry_actions.jsp` to determine what options appear for logged in users who can see the actions menu in the portlet. Just like before, you'll wrap each `renderURL` in an `if` statement that checks the permissions against available actions. To do this, follow these steps:

1. Open `src/main/resources/META-INF/resources/guestbook/entry_actions.jsp`.

2. Remove all the code from this file and replace it with what's below:

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

```
<%
String mvcPath = ParamUtil.getString(request, "mvcPath");

ResultRow row = (ResultRow)request.getAttribute(WebKeys.SEARCH_CONTAINER_RESULT_ROW);

GuestbookEntry entry = (GuestbookEntry)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="/guestbook/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="<%= GuestbookEntry.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>
```

3. Save the file.

This code updates each button with a permissions check. If the current user can't perform the given action, the action doesn't appear.

Excellent! You've now implemented all the permissions checks for the Guestbook portlet.

36.2 Testing the Application

Before testing the application, you must reset your database, because guestbook entries you created without resources won't work with permissions.

1. If your server is running, shut it down.
2. Find your Liferay Workspace on your file system (it should be inside your Eclipse workspace). Inside the `bundles/data` folder is a `hypersonic` folder.
3. Remove everything from the `hypersonic` folder.
4. Restart your server.

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).

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 37

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 Guestbook 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 tutorial and in these articles, is implemented for an application.

To enable search, you must index Guestbooks and Guestbook 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 step, you'll asset-enable Guestbooks and Guestbook entries so the Asset Publisher can display them. Enabling search is a prerequisite for this, because the Asset Publisher uses the index to find assets.

If you get stuck, source code for this step is provided.

But assets are for later. Right now it's time to index those Guestbooks. Ready?

Let's Go!

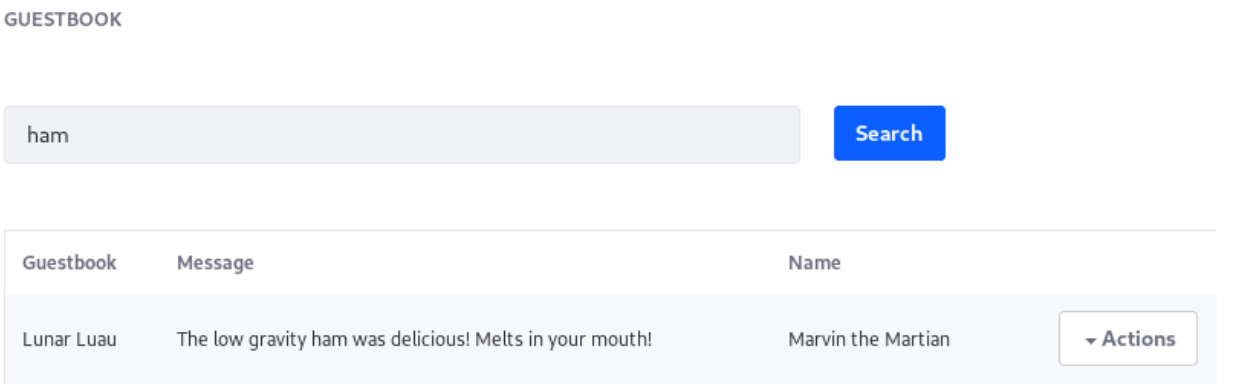

Figure 37.1: Add a search bar so users can search for Guestbook Entries. If a message or name matches the search query, the Entry is displayed in the search results.

CHAPTER 38

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!

CHAPTER 39

UNDERSTANDING SEARCH AND INDEXING

Enabling Search and Indexing for Guestbooks Step 1 of 6

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?

CHAPTER 40

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` with 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"
compileOnly group: "com.liferay", name: "com.liferay.portal.search.api"
```

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). Choose the `com.liferay.portal.kernel.search.Field` class.

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, you'll write the search and indexing logic.

CHAPTER 41

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.

41.1 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.

Use Ctrl-Shift-O to add these imports, and then save the file:

- `com.liferay.portal.kernel.search.Field`
- `com.liferay.portal.kernel.search.Document`

41.2 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) {
        guestbookEntryBatchReindexer.reindex(
    guestbook.getGuestbookId(), guestbook.getCompanyId());
    }
```

```
    @Reference
    protected DynamicQueryBatchIndexingActionableFactory
    dynamicQueryBatchIndexingActionableFactory;

    @Reference
    protected GuestbookEntryBatchReindexer guestbookEntryBatchReindexer;

    @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 Guestbook entries, you'll add the Guestbook title to the `GuestbookEntry` document. Thus, you must provide a way to update the indexed `GuestbookEntry` documents if a Guestbook title is changed. The `modelIndexed` method calls a `reindex` method from an interface that will be created later for `GuestbookEntrys`. For now, ignore the error in the `modelIndexed` method.

Use Ctrl-Shift-O to add this import, and save the file:

- `com.liferay.portal.kernel.search.Document`

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

CHAPTER 42

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.

42.1 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 allows 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.

Don't forget to add imports with Ctrl-Shift-O. Choose `com.liferay.portal.kernel.search.BooleanQuery` and `com.liferay.portal.kernel.search.Field`.

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

CHAPTER 43

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.

Don't forget to Ctrl-Shift-O and import these classes:

- `com.liferay.portal.kernel.search.Field`
- `com.liferay.petra.string.StringPool`
- `com.liferay.portal.kernel.search.Summary`
- `com.liferay.portal.kernel.search.Document`

Save your file.

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.

CHAPTER 44

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. Next, you'll 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. Use Ctrl-Shift-O to 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 45

ENABLING SEARCH AND INDEXING FOR ENTRIES

Now you'll create the classes that control these aspects of the search functionality:

- Registration:
  - `GuestbookEntrySearchRegistrar` registers the search service for the `GuestbookEntry` entity.
- Indexing:
  - `GuestbookEntryModelDocumentContributor` controls which `GuestbookEntry` fields are indexed in the search engine.
  - `GuestbookEntryModelIndexerWriterContributor` configures the re-indexing and batch re-indexing behavior for `GuestbookEntrys`.
  - `GuestbookEntryBatchReindexer`, an interface, and its `GuestbookEntryBatchReindexerImpl`, for re-indexing `GuestbookEntries`when their Guestbook is updated.
- Querying:
  - `GuestbookEntryKeywordQueryContributor` contributes clauses to the ongoing search query.
  - `GuestbookEntryModelPreFilterContributor` controls how search results are filtered before they're returned from the search engine.
- Generating Result Summaries:
  - `GuestbookEntryModelSummaryContributor` constructs the result summary for `GuestbookEntrys`, including specifying which fields to use.

After creating the search classes, modify the service layer to update the search index when a `GuestbookEntry`is persisted:

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

Note: In prior versions of Liferay DXP, search and indexing was accomplished with one `*Indexer` class that extended `BaseIndexer`. This tutorial 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!

CHAPTER 46

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 `GuestbookEntrySearchRegistrar`:

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

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

    @Activate
    protected void activate(BundleContext bundleContext) {

        _serviceRegistration = modelSearchRegistrarHelper.register(
            GuestbookEntry.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.GuestbookEntry)")
    protected ModelIndexerWriterContributor<GuestbookEntry> modelIndexWriterContributor;

    @Reference
    protected ModelSearchRegistrarHelper modelSearchRegistrarHelper;

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

    private ServiceRegistration<?> _serviceRegistration;

}
```

   Target the `GuestbookEntry` 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. Use Ctrl-Shift-O to add imports:

   - `com.liferay.portal.kernel.search.Field`

The entry search and indexing class registration is completed. Next, you'll write the search and indexing logic.

CHAPTER 47

INDEXING ENTRIES

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

To control how `GuestbookEntry`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, `GuestbookEntryBatchReindexer`, with its implementation, `GuestbookEntryBatchReindexerImpl`. These classes contain code to ensure that entries are re-indexed when their Guestbook is updated.

47.1 Implementing `ModelDocumentContributor`

Create `GuestbookEntryModelDocumentContributor` and populate it with this:

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

    @Override
    public void contribute(Document document, GuestbookEntry 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(
    GuestbookEntryModelDocumentContributor.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.

Use Ctrl-Shift-O to add the following imports and save the file:

- `com.liferay.portal.kernel.search.Document`
- `com.liferay.portal.kernel.search.Field`

47.2 Implementing `ModelIndexerWriterContributor`

Create `GuestbookEntryModelIndexerWriterContributor` and populate it with this:

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

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

        batchIndexingActionable.setPerformActionMethod((GuestbookEntry entry) -> {
```

```
            Document document = modelIndexerWriterDocumentHelper.getDocument(
    entry);

            batchIndexingActionable.addDocuments(document);

        });
    }

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

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

    @Reference
    protected DynamicQueryBatchIndexingActionableFactory
    dynamicQueryBatchIndexingActionableFactory;

    @Reference
    protected GuestbookEntryLocalService guestbookEntryLocalService;

}
```

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

Use Ctrl-Shift-O to add an import for `com.liferay.portal.kernel.search.Document` and save the file.

47.3 Implementing `GuestbookEntryBatchReindexer`

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

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

public interface GuestbookEntryBatchReindexer {

    public void reindex(long guestbookId, long companyId);

}
```

Then create the implementation class, `GuestbookEntryBatchReindexerImpl`:

```
@Component(immediate = true, service = GuestbookEntryBatchReindexer.class)
public class GuestbookEntryBatchReindexerImpl implements GuestbookEntryBatchReindexer {

    @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((GuestbookEntry entry) -> {
            Document document = indexerDocumentBuilder.getDocument(entry);

            batchIndexingActionable.addDocuments(document);
        });

        batchIndexingActionable.performActions();

    }

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

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

Use Ctrl-Shift-O to add the following imports, and save the file:

- `com.liferay.portal.kernel.search.Document`
- `com.liferay.portal.kernel.dao.orm.Property`

You should notice that errors in the project go away at this point.

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

CHAPTER 48

QUERYING FOR GUESTBOOK 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 Guestbook entries to the search engine. Next code the behavior necessary for querying the indexed documents.

Implement two classes:

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

48.1 Implementing `KeywordQueryContributor`

Create `GuestbookEntryKeywordQueryContributor` and populate it with this:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.GuestbookEntry",
        service = KeywordQueryContributor.class
)
public class GuestbookEntryKeywordQueryContributor 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 `GuestbookEntry` fields in the index, retrieve the localized value from the search engine.

Use Ctrl-Shift-O to add these imports, and then save the file:

- `com.liferay.portal.kernel.search.BooleanQuery`
- `com.liferay.portal.kernel.search.Field`

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

CHAPTER 49

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 `GuestbookEntryModelSummaryContributor`:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.GuestbookEntry",
        service = ModelSummaryContributor.class
)
public class GuestbookEntryModelSummaryContributor 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.

Use Ctrl-Shift-O to add these imports, and then save the file:

- `com.liferay.portal.kernel.search.Field`
- `com.liferay.petra.string.StringPool`
- `com.liferay.portal.kernel.search.Summary`
- `com.liferay.portal.kernel.search.Document`

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.

CHAPTER 50

HANDLING INDEXING IN THE ENTRY SERVICE LAYER

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

Whenever a Guestbook entry is added, updated, or deleted, the corresponding document should also be updated or deleted. A minor update to each of the `addEntry`, `updateEntry`, and `deleteEntry` service methods for Entries is all it takes.

Follow these steps to update the methods:

1. Open `GuestbookEntryLocalServiceImpl` in the `guestbook-service` module's `com.liferay.docs.guestbook.service.` package, and add the annotation `@Indexable(type = IndexableType.REINDEX)` above the signature for the `addGuestbookEntry` and `updateGuestbookEntry` methods:

   ```
   @Indexable(type = IndexableType.REINDEX)
   public GuestbookEntry addGuestbookEntry(...)

   @Indexable(type = IndexableType.REINDEX)
   public GuestbookEntry updateGuestbookEntry(...)
   ```

   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 `GuestbookEntry`.

2. Add the `@Indexable(type = IndexableType.DELETE)` annotation above the signature for the `deleteEntry` method. The indexable type `IndexableType.DELETE` ensures that the `GuestbookEntry` is deleted from the index:

   ```
   @Indexable(type = IndexableType.DELETE)
   public GuestbookEntry deleteGuestbookEntry(...)
   ```

3. Use Ctrl-Shift-O to 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 `GuestbookEntryLocalServiceImpl`.

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 51

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!

CHAPTER 52

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/guestbook/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="/guestbook/view_search.jsp" />
   </portlet:renderURL>
   ```

   The render URL points to `/guestbook/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 you'll create the `view_search.jsp` file to display the search results.

CHAPTER 53

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.

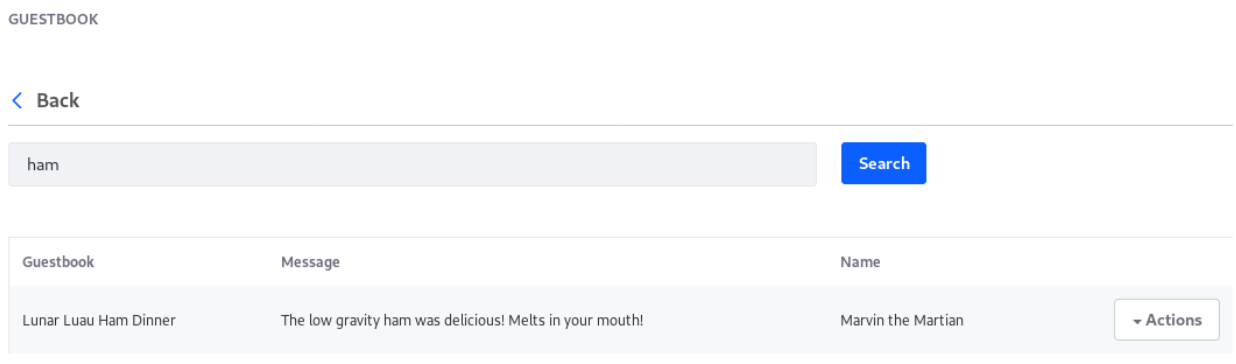

Figure 53.1: The search results should appear in a search container, and the Actions button should appear for each entry. The search bar should also be displayed.

Follow these steps to create the search results JSP:

1. Create a new file called `view_search.jsp` in your `guestbook-web` module's `src/main/resources/META-INF/resources/guestbook` 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="/guestbook/view_search.jsp" />
</portlet:renderURL>

<portlet:renderURL var="viewURL">
    <portlet:param
        name="mvcPath"
        value="/guestbook/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<GuestbookEntry> indexer = IndexerRegistryUtil.getIndexer(GuestbookEntry.class);
```

7. In the same scriptlet, use the indexer and the search context to run a search:

```
Hits hits = indexer.search(searchContext);

List<GuestbookEntry> entries = new ArrayList<GuestbookEntry>();

for (int i = 0; i < hits.getDocs().length; i++) {
        Document doc = hits.doc(i);

        long entryId = GetterUtil
        .getLong(doc.get(Field.ENTRY_CLASS_PK));

        GuestbookEntry entry = null;

        try {
                entry = GuestbookEntryLocalServiceUtil.getGuestbookEntry(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.GuestbookEntry"
        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="/guestbook/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 `GuestbookEntryLocalServiceUtil.getGuestbookEntry` 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.guestbook.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.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.

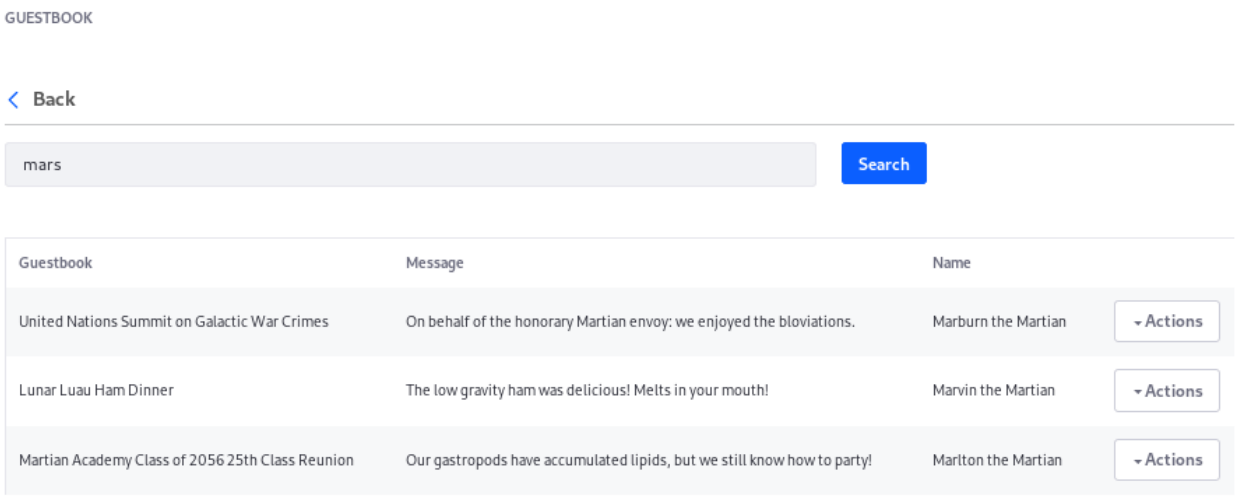

Figure 53.2: The Guestbook Application now supports searching for indexed Guestbook Entries.

As before, remove the `hypersonic` folder from the `data` folder of your Liferay bundle, and remove the modules from your server in Dev Studio DXP. Start your server and redeploy all three modules, add some Guestbook entries, and try searching for them.

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.

Chapter 54

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 these 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.

As always, source code is provided in case you get stuck.

Ready to start?

Let's Go!

CHAPTER 55

ENABLING ASSETS AT THE SERVICE LAYER

Enabling Assets at the Service Layer Step 1 of 3

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 GuestbookEntry:

   ```
   <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 GuestbookEntrys—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 GuestbookEntry 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.

CHAPTER 56

HANDLING ASSETS FOR THE GUESTBOOK SERVICE

```
<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"
compileOnly group: "javax.servlet", name: "javax.servlet-api"
```

3. Save your `build.gradle` file, which refreshes your project.

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.

CHAPTER 57

HANDLING ASSETS FOR THE GUESTBOOKENTRY SERVICE

```
<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 `GuestbookEntry` entities.

1. Open `GuestbookEntryLocalServiceImpl` and add the asset calls to the `addGuestbookEntry` method after the resource calls:

```
AssetEntry assetEntry = assetEntryLocalService.updateEntry(userId,
                groupId, entry.getCreateDate(), entry.getModifiedDate(),
                GuestbookEntry.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 `updateGuestbookEntry` method after the resource calls:

```
AssetEntry assetEntry = assetEntryLocalService.updateEntry(userId,
              serviceContext.getScopeGroupId(),
              entry.getCreateDate(), entry.getModifiedDate(),
              GuestbookEntry.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 `deleteGuestbookEntry` method after the resource calls:

```
AssetEntry assetEntry = assetEntryLocalService.fetchEntry(
                    GuestbookEntry.class.getName(), entry.getEntryId());

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.GuestbookEntry=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 58

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!

CHAPTER 59

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.

59.1 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;
}

// Add the other methods here

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. Add these methods below the comment labeled `Add the other methods here`:

```
@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("mvcPath", "/guestbook/edit_guestbook.jsp");
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("mvcPath", "/guestbook/view.jsp");
      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 these imports:

```
- `java.util.logging.Logger`
- `com.liferay.portal.kernel.exception.SystemException`
- `java.util.logging.Level`
- `com.liferay.petra.string.StringUtil`
```

Next you can create the `AssetRendererFactory` class.

59.2 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);
  }

  // Add the other methods here

  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`.

   Insert the methods below where you see the comment `Add the other methods here`.

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("mvcPath", "/guestbook/edit_guestbook.jsp");
      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) and save the file. Select these imports:

    - `java.util.logging.Logger`
    - `java.util.logging.Level`

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

CHAPTER 60

IMPLEMENTING A GUESTBOOK 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.

60.1 Creating the GuestbookEntryAssetRenderer Class

In the `com.liferay.docs.guestbook.web.internal.asset` package, create a `GuestbookEntryAssetRenderer` class that extends Liferay DXP's `BaseJSPAssetRenderer` class. Replace the contents of your `GuestbookEntryAssetRenderer` 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.GuestbookEntry;
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 GuestbookEntryAssetRenderer extends BaseJSPAssetRenderer<GuestbookEntry> {

    public GuestbookEntryAssetRenderer(GuestbookEntry entry, ModelResourcePermission<GuestbookEntry> modelResourcePermission) {

        _entry = entry;
        _guestbookEntryModelResourcePermission = modelResourcePermission;
    }

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

        return true;
    }

    @Override
    public GuestbookEntry 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 GuestbookEntry.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("mvcPath", "/guestbook/edit_entry.jsp");
    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("mvcPath", "/guestbook/view_entry.jsp");
        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<GuestbookEntry> _guestbookEntryModelResourcePermission;
    private GuestbookEntry _entry;
}
```

This class is similar to the `GuestbookAssetRenderer` class. For the `GuestbookEntryAssetRenderer.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. `GuestbookEntryAssetRenderer.getTitle` returns the entry message. `GuestbookAssetRenderer.getTitle` returns the guestbook name. The other methods of `GuestbookEntryAssetRenderer` are nearly identical to those of `GuestbookAssetRenderer`.

60.2 Creating the GuestbookEntryAssetRendererFactory 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 `GuestbookEntryAssetRendererFactory` 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.GuestbookEntry;
import com.liferay.docs.guestbook.service.GuestbookEntryLocalService;
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 GuestbookEntryAssetRendererFactory extends BaseAssetRendererFactory<GuestbookEntry> {

    public GuestbookEntryAssetRendererFactory() {
        setClassName(CLASS_NAME);
        setLinkable(_LINKABLE);
        setPortletId(GuestbookPortletKeys.GUESTBOOK);
        setSearchable(true);
        setSelectable(true);

    }

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

        GuestbookEntry entry = _guestbookEntryLocalService.getGuestbookEntry(classPK);

        GuestbookEntryAssetRenderer guestbookEntryAssetRenderer = new GuestbookEntryAssetRenderer(entry, _guestbookEntryModelResourcePermission);

        guestbookEntryAssetRenderer.setAssetRendererType(type);
        guestbookEntryAssetRenderer.setServletContext(_servletContext);

        return guestbookEntryAssetRenderer;
    }

    @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 {

        GuestbookEntry entry = _guestbookEntryLocalService.getGuestbookEntry(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", "/guestbook/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 setGuestbookEntryLocalService(GuestbookEntryLocalService guestbookEntryLocalService) {
        _guestbookEntryLocalService = guestbookEntryLocalService;
    }


    private GuestbookEntryLocalService _guestbookEntryLocalService;
    private ServletContext _servletContext;
    private static final boolean _LINKABLE = true;
    public static final String CLASS_NAME = GuestbookEntry.class.getName();
    public static final String TYPE = "entry";

    private ModelResourcePermission<GuestbookEntry>
    _guestbookEntryModelResourcePermission;


}
```

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.

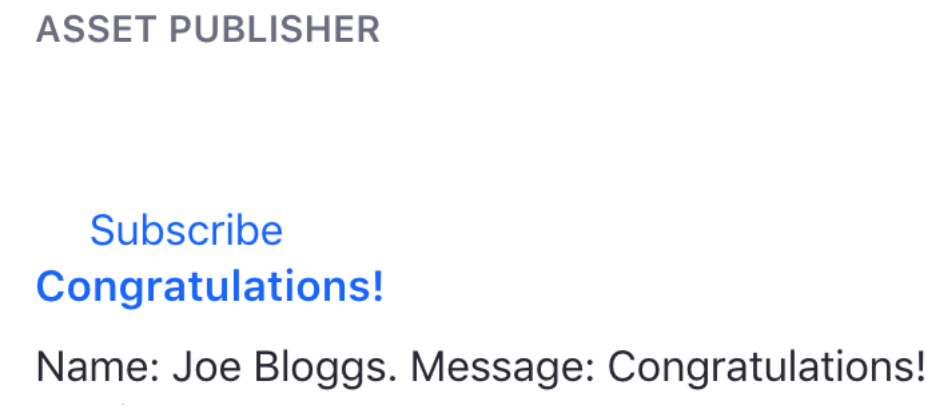

Figure 60.1: After you've implemented and registered your asset renderers for your custom entities, the Asset Publisher can display your entities.

CHAPTER 61

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="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 the steps.

CHAPTER 62

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 `GuestbookEntryAssetRenderer` 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"%>

<%
GuestbookEntry entry = (GuestbookEntry)request.getAttribute("gb_entry");

entry = entry.toEscapedModel();
%>

<dl>
        <dt>Guestbook</dt>
        <dd><%= GuestbookLocalServiceUtil.getGuestbook(entry.getGuestbookId()).getName() %></dd>
        <dt>Name</dt>
        <dd><%= entry.getName() %></dd>
        <dt>Message</dt>
        <dd><%= entry.getMessage() %></dd>
</dl>
```

This JSP shows a combination of fields from the Guestbook and the selected Guestbook 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`:

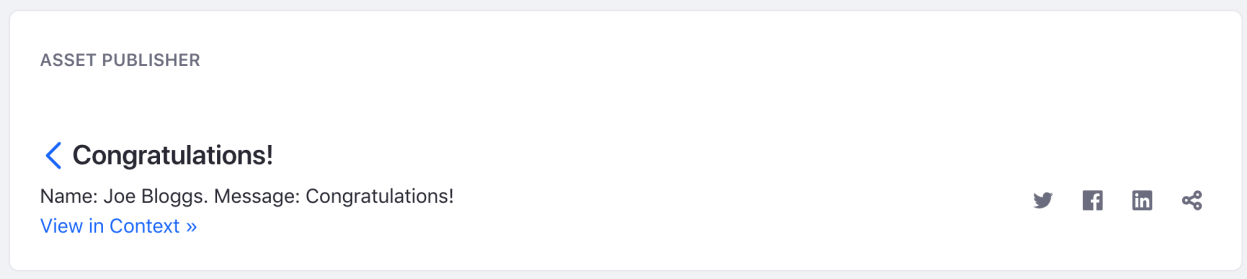

Figure 62.1: When you click the title for a guestbook or guestbook entry in the Asset Publisher, your `full_content.jsp` should be displayed.

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.

CHAPTER 63

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.

63.1 Enabling Asset Features

Follow these steps:

1. In the `guestbook-web` module's `/guestbook_admin/edit_guestbook.jsp`, add the tags `<liferay-asset:asset-categories-error />` and `<liferay-asset:asset-tags-error/>` to the `aui:form` below the closing `</aui:button-row>` tag:

   ```
   <liferay-asset:asset-categories-error />
   <liferay-asset: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 surrounded by a `<c:if>` statement:

   ```
   <c:if test="<%= guestbook ≠ null %>">

       <liferay-ui:panel defaultState="closed" extended="<%= false %>"
         id="guestbookCategorizationPanel" persistState="<%= true %>"
         title="categorization">

       </liferay-ui:panel>

   </c:if>
   ```

The `<liferay-ui:panel>` tag generates a collapsible section. The tags you'll add in the next step don't work if `guestbook` is `null`, so you only display the panel if the current Guestbook is being edited.

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.getGuestbookId() %>" />
    <liferay-asset:asset-tags-selector className="<%= Guestbook.class.getName() %>" classPK="<%= guestbook.getGuestbookId() %>" />
</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-asset: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*. After adding the Guestbook, edit it. 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.

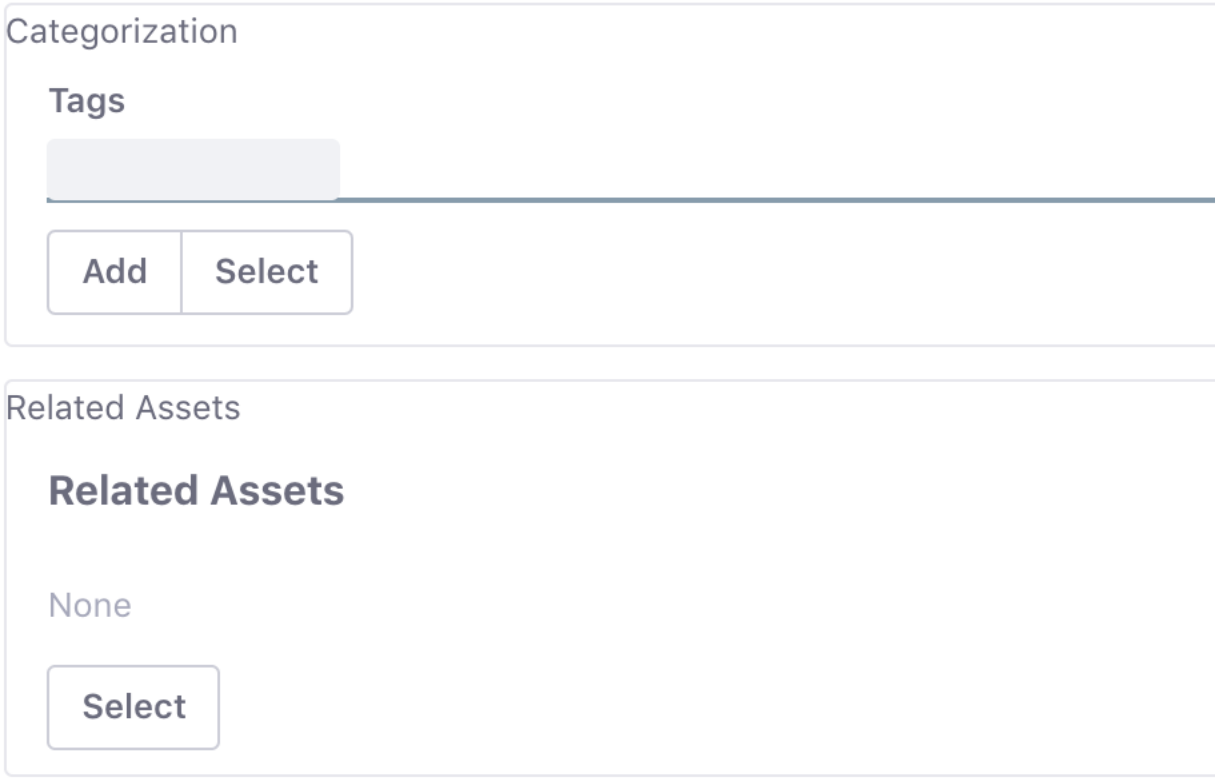

Figure 63.1: Once you've updated your Guestbook Admin portlet's `edit_guestbook.jsp` page, you'll see forms for adding tags and selecting related assets.

CHAPTER 64

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 `guestbook/edit_entry.jsp` file. You'll add two pieces of code: a header for navigation and a panel for tags and categories similar to the one you added to the `edit_guestbook.jsp` file.

1. Add the header after the `addEntry` action URL tag:

```
<liferay-ui:header
    backURL="<%= viewURL.toString() %>"
    title="<%= entry == null ? "Add Entry" : entry.getName() %>"
/>
```

2. Add the asset tags/categories/links in a collapsible panel after the closing `</aui:fieldset>`:

    "'markup

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

<aui:fieldset collapsed="<%= true %>" collapsible="<%= true %>" label="related-assets">

    <liferay-asset:input-asset-links
        className="<%= GuestbookEntry.class.getName() %>"
        classPK="<%= entryId %>"
    />

</aui:fieldset>


Test your JSP by using the Guestbook portlet to add and update Guestbook
entries. Add and remove tags, categories, and related assets.
```

\noindent\hrulefill

 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](/docs/7-2/tutorials/-/knowledge_base/t/making-urls-friendlier)
 in a later step.

\noindent\hrulefill

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

Enabling Comments and Ratings for Guestbook Entries

    Adding Asset Features to Your UI Step 5 of 5

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 `src/main/resources/META-INF/resources/guestbook` folder.

2. Add a Java scriptlet to the file you just created. In this scriptlet, use an
   `entryId` request attribute to get a `GuestbookEntry` 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](/docs/7.2/tutorials/-/knowledge_base/t/creating-jsps-for-displaying-custom-assets-in-the-asset-publisher):

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

   <%
     long entryId = ParamUtil.getLong(renderRequest, "entryId");

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

     GuestbookEntry entry = null;

     entry = GuestbookEntryLocalServiceUtil.getGuestbookEntry(entryId);

     entryId = entry.getEntryId();

     entry = entry.toEscapedModel();

     AssetEntry assetEntry =
     AssetEntryLocalServiceUtil.getEntry(GuestbookEntry.class.getName(),
     entry.getEntryId());
   ```

3. Next, update the breadcrumb entry with the current entry's name:

   ```markup
   String currentURL = PortalUtil.getCurrentURL(request);
   PortalUtil.addPortletBreadcrumbEntry(request, entry.getMessage(),
   currentURL);
   ```

4. End the scriptlet by adding 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:

```markup
    PortalUtil.setPageSubtitle(entry.getMessage(), request);
    PortalUtil.setPageDescription(entry.getMessage(), request);

    List<AssetTag> assetTags =
    AssetTagLocalServiceUtil.getTags(GuestbookEntry.class.getName(),
    entry.getEntryId());
    PortalUtil.setPageKeywords(ListUtil.toString(assetTags, "name"),
    request);
%>
```

5. After the scriptlet, specify the URLs for the page and back link:

```markup
<liferay-portlet:renderURL varImpl="viewEntryURL">
  <portlet:param name="mvcPath"
    value="/guestbook/view_entry.jsp" />
  <portlet:param name="entryId" value="<%=String.valueOf(entryId)%>" />
</liferay-portlet:renderURL>

<liferay-portlet:renderURL varImpl="viewURL">
  <portlet:param name="mvcPath"
    value="/guestbook/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:

```markup
<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 `/asset/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:

```markup
<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:

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

```
    <br />
    ```

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

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

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

    ```markup
    <c:if test="<%= discussion ≠ null %>">
      <h2>
        <strong><liferay-ui:message arguments="<%= discussion.getDiscussionCommentsCount() %>" key='<%= (discussion.getDiscussionCommentsCount() == 1
comment" : "x-comments" %>' /></strong>
    ```

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.

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

  </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()`:

    ```markup
    <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.

\noindent\hrulefill

 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.

```
\noindent\hrulefill

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 `/guestbook/entry_actions.jsp`
and find the following line:

```markup
<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="/guestbook/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!

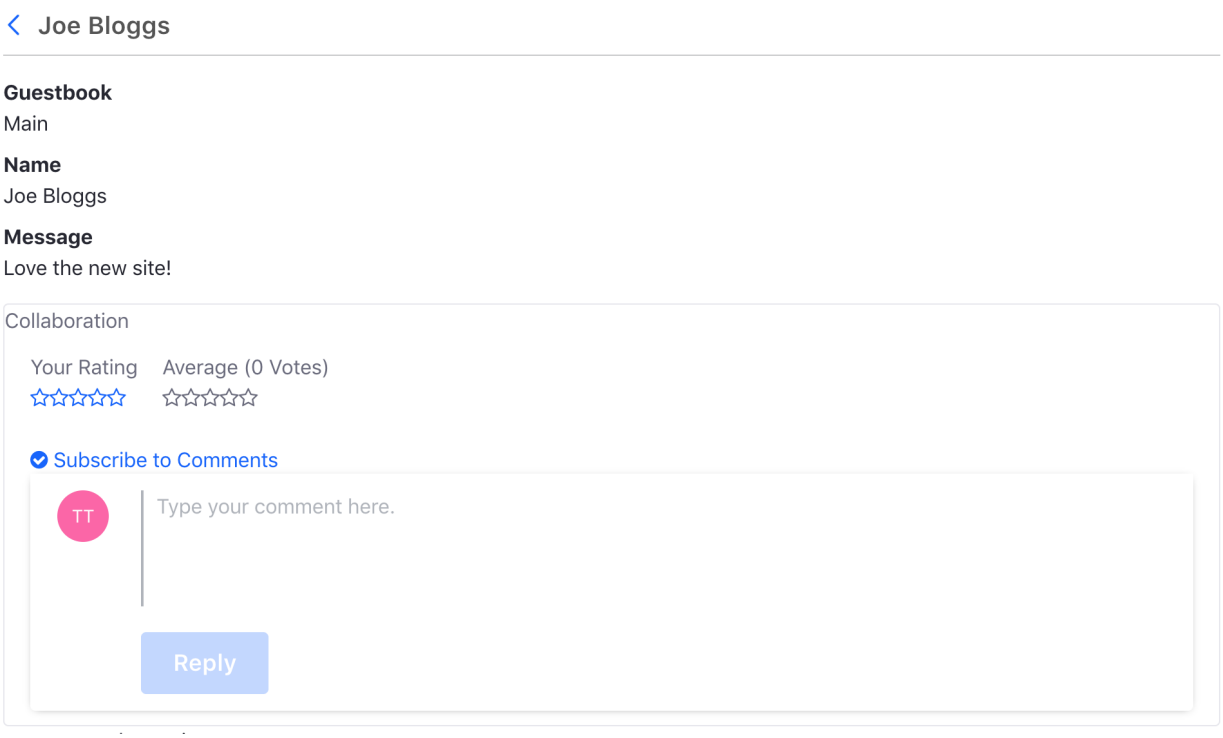

Figure 64.1: Now you can see comments, rating, and the full range of asset features.
```

Your next task is to add Workflow, so you can approve or deny guestbook entries, thereby preventing people from spamming your guestbook.

CHAPTER 65

USING WORKFLOW

The Guestbook application accepts submissions from any logged in user, so there's no telling what people could post. Illegal data, objectionable content, the entire contents of Don Quixote: all of these and more are possibilities. You can make sure user posts don't run afoul of the law or policy by enabling *workflow* in your application.

Workflow is a review process that ensures a submitted entity isn't published before it's reviewed. To prevent posting objectionable content, an initially submitted Guestbook entry should be marked as a *draft* and sent through the workflow framework. It comes back to the application code ready to have any relevant fields updated in the database based on its status. The view layer must filter entities by status to display only reviewed entities.

Note: The exact review process is defined separately from the code that enables workflow. An XML file provides the definition of a workflow in Liferay DXP. If you're a Liferay Digital Enterprise subscriber, you have access to the Workflow Designer, which offers a convenient drag-and-drop user interface for designing workflow definition files. You can read more about this in Liferay DXP's documentation. Liferay DXP comes with a workflow definition called the *Single Approver* definition, but you can write your own workflow definitions according to your organization's requirements.

A few additional definitions are included in Liferay DXP's source code, which you can use to see how workflow definitions are defined. To discover how to access these files, see here.

This tutorial instructs the reader in workflow-enabling the Guestbook App's `Guestbook` and `GuestbookEntry` entities to ensure that only approved content is published after review.

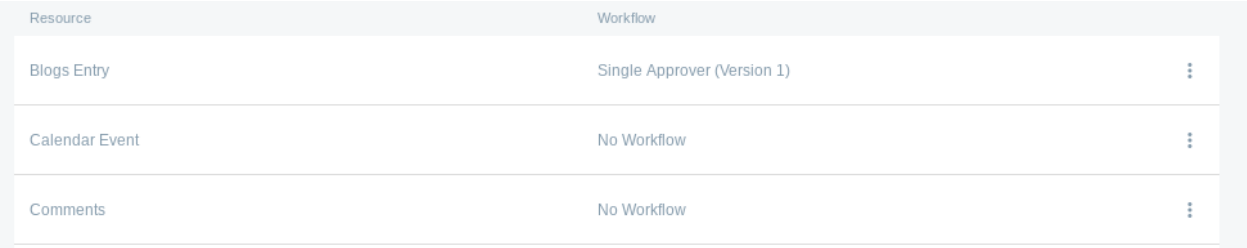

Figure 65.1: Enable workflow in your assets, just like Liferay DXP's own assets.

There are five steps to enabling workflow:

1. Update the service layer to set each entity's status fields.
2. Send the entity to Liferay DXP's workflow framework.
3. Add *getter* methods that account for an entity's workflow status.
4. Handle the entity as it returns from the workflow framework.
5. Update the user interface to account for workflow status.

The first three steps happen in the service layer, so that's a good place to start. Let's Go!

CHAPTER 66

SUPPORTING WORKFLOW AT THE SERVICE LAYER

When you asset enabled the Guestbook Application, you used four database columns in the Guestbook entities that keep track of workflow status (they were added in the beginning; celebrate!). The necessary fields are `status`, `statusByUserName`, `statusByUserId`, and `statusDate`. The columns are defined in the `guestbook-service` module's `service.xml` file.

```
<column name="status" type="int" />
<column name="statusByUserId" type="long" />
<column name="statusByUserName" type="String" />
<column name="statusDate" type="Date" />
```

The `status` field tells you the current status of the entity (it defaults to `0`, which evaluates to *approved*). The other status fields store the date of the last change (`statusDate`) along with the ID and name of the user (`statusByUserId` and `statusByUserName`) who made the update.

Although the status columns are in the Guestbook application's entity tables, you must update the local service implementation's `add` methods to set them, and while you're there, send the entity to the workflow framework. You'll also write a method to update the status fields when the entity returns from the workflow framework, along with getters that take workflow status as a parameter. That sounds like a lot of work, but thanks to Service Builder, you must change only three files: `service.xml`, `GuestbookLocalServiceImpl`, and `GuestbookEntryLocalServiceImpl`.

Let's Go!

CHAPTER 67

SETTING THE GUESTBOOK STATUS

```
<p>Supporting Workflow at the Service Layer<br>Step 1 of 3</p>
```

Before now, you set the status of all added guestbooks to approved in the service layer. Now you'll set it to draft and pass it to the workflow framework.

1. From `guestbook-service`, open `GuestbookLocalServiceImpl` and add the status fields below the existing setter methods in the `addGuestbook` method:

   ```
   guestbook.setStatus(WorkflowConstants.STATUS_DRAFT);
   guestbook.setStatusByUserId(userId);
   guestbook.setStatusByUserName(user.getFullName());
   guestbook.setStatusDate(serviceContext.getModifiedDate(null));
   ```

   This manually populates the status fields and sets the workflow status as a draft in the `GB_GuestbookEntry` database table. At this point they're identical to the similarly named non-status counterparts (like `setUserId` and `setStatusByUserId`), but they'll be updated independently in the `updateStatus` method you write later.

2. Still in the `addGuestbook` method, place the following code right before the `return` statement:

   ```
   WorkflowHandlerRegistryUtil.startWorkflowInstance(guestbook.getCompanyId(),
               guestbook.getGroupId(), guestbook.getUserId(), Guestbook.class.getName(),
               guestbook.getPrimaryKey(), guestbook, serviceContext);
   ```

   The call to `startWorkflowInstance` detects whether workflow is installed and enabled. If it isn't, the added entity is automatically marked as approved. The `startWorkflowInstance` call also calls your `GuestbookWorkflowHandler` class, which you'll create later.

3. Organize imports (*[CTRL]+[SHIFT]+O*), and save your work.

The `startWorkflowInstance` method is where your entity enters the workflow framework, but you're not finished yet. Just like you wouldn't drop your child off at college and then change your number and move to a new address, you're not going to abandon your `Guestbook` entity (yet).

67.1 Creating the updateStatus Method

Exert control over how the status fields are updated in the database.

1. Create an `updateStatus` method in `GuestbookLocalServiceImpl`, immediately following the `deleteGuestbook` method. Here's the first half of it:

```
public Guestbook updateStatus(long userId, long guestbookId, int status,
        ServiceContext serviceContext) throws PortalException,
        SystemException {

    User user = userLocalService.getUser(userId);
    Guestbook guestbook = getGuestbook(guestbookId);

    guestbook.setStatus(status);
    guestbook.setStatusByUserId(userId);
    guestbook.setStatusByUserName(user.getFullName());
    guestbook.setStatusDate(new Date());

    guestbookPersistence.update(guestbook);
```

   If this method is called, it's because your entity is returning from the workflow framework, and it's time to update the status values in the database. Set the status fields, then persist the updated entity to the database.

2. Before saving, finish the method:

```
    if (status == WorkflowConstants.STATUS_APPROVED) {

        assetEntryLocalService.updateVisible(Guestbook.class.getName(),
                guestbookId, true);

    } else {

        assetEntryLocalService.updateVisible(Guestbook.class.getName(),
                guestbookId, false);
    }

    return guestbook;
}
```

   This `if` statement determines the visibility of the asset based on its workflow status. If it's approved, the `assetEntryLocalService.updateVisible` method sets the guestbook in question to `true` so it can be displayed in the Asset Publisher and in the search results. Otherwise (`else`) it sets the visibility to `false` to ensure that unapproved guestbooks aren't displayed to users in the Asset Publisher or the Search portlet.

3. There's one more update to make in the `deleteGuestbook` method. When deleting, you must clean up the workflow system's database tables to avoid leaving orphaned entries when the backing entity is deleted. Before making the method call, open `service.xml` and add the following tag below the existing `<reference>` tags in the `Guestbook` entity:

```
<reference entity="WorkflowInstanceLink" package-path="com.liferay.portal" />
```

4. Back in `GuestbookLocalServiceImpl`, find the `deleteGuestbook` method and put this method call right before the `return` statement:

```
workflowInstanceLinkLocalService.deleteWorkflowInstanceLinks(
    guestbook.getCompanyId(), guestbook.getGroupId(),
    Guestbook.class.getName(), guestbook.getGuestbookId());
```

5. Organize imports (*[CTRL]+[SHIFT]+O*) and save your work. Then run the `buildService` Gradle task. It injects the `WorkflowInstanceLinkLocalService` service into a protected variable in `GuesbookLocalServiceBaseImpl`. Since `GuestbookLocalServiceImpl` extends the base class, you can use it directly.

6. Run *Refresh Gradle Project*.

Now the guestbook entity's service layer populates the status fields in the database, sends the entity into the workflow framework, and cleans up when it's deleted. You'll do the same thing for guestbook entries next.

CHAPTER 68

SETTING THE ENTRY WORKFLOW STATUS

```
<p>Supporting Workflow at the Service Layer<br>Step 2 of 3</p>
```

Now you'll set the status fields, introduce entries to the workflow framework, and add the updateStatus method to GuestbookEntryLocalServiceImpl. It works the same as it did for guestbooks.

1. Add the following lines in the addGuestbookEntry method, immediately after the existing setter methods (e.g., entry.setMessage(message)):

```
entry.setStatus(WorkflowConstants.STATUS_DRAFT);
entry.setStatusByUserId(userId);
entry.setStatusByUserName(user.getFullName());
entry.setStatusDate(serviceContext.getModifiedDate(null));
```

2. Still in the addGuestbookEntry method, place the following code right before the return statement:

```
WorkflowHandlerRegistryUtil.startWorkflowInstance(entry.getCompanyId(),
            entry.getGroupId(), entry.getUserId(), GuestbookEntry.class.getName(),
            entry.getPrimaryKey(), entry, serviceContext);
```

   The startWorkflowInstance call eventually directs the workflow processing to your GuestbookEntryWorkflowHandler class, which you'll create later. That class is responsible for making sure the entity is updated in the database (via an updateStatus method), but it's best practice to make persistence calls in the service layer.

3. Add a corresponding updateStatus method here in GuestbookEntryLocalServiceImpl. Add this method to the bottom of the class:

```
public GuestbookEntry updateStatus(long userId, long guestbookId, long entryId, int status,
       ServiceContext serviceContext) throws PortalException,
       SystemException {

   User user = userLocalService.getUser(userId);
   GuestbookEntry entry = getGuestbookEntry(entryId);

   entry.setStatus(status);
   entry.setStatusByUserId(userId);
```

```
    entry.setStatusByUserName(user.getFullName());
    entry.setStatusDate(new Date());

    guestbookEntryPersistence.update(entry);

    if (status == WorkflowConstants.STATUS_APPROVED) {

        assetEntryLocalService.updateVisible(GuestbookEntry.class.getName(),
                entryId, true);

    } else {

        assetEntryLocalService.updateVisible(GuestbookEntry.class.getName(),
                entryId, false);
    }

    return entry;
}
```

4. As with Guestbooks, you must add a call to `deleteWorkflowInstanceLinks` in the entry's delete method to avoid leaving orphaned database entries in the `workflowinstancelinks` table. First add the following `<reference>` tag to `service.xml`, this time in the `entry` entity section, below the existing reference tags:

```
<reference entity="WorkflowInstanceLink" package-path="com.liferay.portal" />
```

5. Add the following method call to the `deleteGuestbookEntry` method in `GuestbookEntryLocalServiceImpl`, right before the `return` statement:

```
workflowInstanceLinkLocalService.deleteWorkflowInstanceLinks(
    entry.getCompanyId(), entry.getGroupId(),
    GuestbookEntry.class.getName(), entry.getEntryId());
```

6. Organize imports (*[CTRL]+[SHIFT]+O*), save your work, run Service Builder, and refresh the Gradle project.

Now both entities support the status of the entity and can handle it as it enters the workflow framework and as it returns from the workflow framework. There's one more update to make in the local service implementation classes: adding getter methods that take the status as a parameter. Later you'll use these methods in the view layer so you can display only approved guestbooks and entries.

CHAPTER 69

RETRIEVING GUESTBOOKS AND ENTRIES BY STATUS

Supporting Workflow at the Service Layer
Step 3 of 3

The service implementation for both entities now supports adding the status fields to the database tables. There's one more update to make in the service layer, but to understand why, you must think about the view layer. When the Guestbook portlet displays entries, you must make sure it doesn't show entries that haven't been approved. Currently, the entry's view layer shows all guestbooks:

```
List<Guestbook> guestbooks = GuestbookLocalServiceUtil
            .getGuestbooks(scopeGroupId);
```

There's a problem: the getter only takes the `scopeGroupId` as a parameter, so there's no way to get guestbooks by their status.

Likewise, unapproved entries must not be displayed, but the view layer currently gets all entries:

```
<liferay-ui:search-container total="<%=GuestbookEntryLocalServiceUtil.getGuestbookEntriesCount()%>">
<liferay-ui:search-container-results
    results="<%=GuestbookEntryLocalServiceUtil.getGuestbookEntries(scopeGroupId.longValue(),
                    guestbookId, searchContainer.getStart(),
                    searchContainer.getEnd())%>" />
```

The solution is to implement for guestbooks and entries a getter that takes the `status` field as a parameter. Thankfully, Service Builder makes it easy.

Open the `guestbook-service` module's `service.xml` file.

1. For the `GuestbookEntry` entity, remove the following finder:

```
<finder name="G_S" return-type="Collection">
  <finder-column name="groupId" />
  <finder-column name="status" />
</finder>
```

2. Add this finder in its place:

```
<finder name="G_G_S" return-type="Collection">
   <finder-column name="groupId" />
   <finder-column name="guestbookId" />
   <finder-column name="status" />
</finder>
```

Run service builder (double-click `guestbook-service/build/buildService` in the Gradle Tasks pane). Service Builder generates finder methods in the persistence layer that take the specified fields (for example, `status`) as parameters.

69.1 Calling the Persistence Layer

Don't call the persistence layer directly in the application code. Instead expose the new persistence methods in the service layer.

1. Open `GuestbookLocalServiceImpl`, add this getter, and save the file:

```
public List<Guestbook> getGuestbooks(long groupId, int status)
    throws SystemException {

    return guestbookPersistence.findByG_S(
        groupId, WorkflowConstants.STATUS_APPROVED);
}
```

   This getter gets only approved guestbooks. That's why you hard code the workflow constant `STATUS_APPROVED` into the status parameter when calling the persistence method.

2. Now open `GuestbookEntryLocalServiceImpl`, add these two getters, and save the file:

```
public List<GuestbookEntry> getGuestbookEntries(
    long groupId, long guestbookId, int status, int start, int end)
    throws SystemException {

    return guestbookEntryPersistence.findByG_G_S(
        groupId, guestbookId, WorkflowConstants.STATUS_APPROVED);
}

public int getGuestbookEntriesCount(
    long groupId, long guestbookId, int status)
    throws SystemException {

    return guestbookEntryPersistence.countByG_G_S(
        groupId, guestbookId, WorkflowConstants.STATUS_APPROVED);
}
```

   You'll replace the existing methods with these `getGuestbookEntries` and `getGuestbookEntriesCount` methods in the view layer, ensuring that only approved entries are displayed.

3. Save the file, run Service Builder, and refresh the Gradle project.

The work here relates to the UI updates you'll make later. Next, you must implement workflow handlers so that you can call the `updateStatus` service method when the entity returns from the workflow framework.

CHAPTER 70

HANDLING WORKFLOW

The guestbook project's service layer is now updated to handle workflow. It now properly sets the status fields for guestbooks and guestbook entries, gets entities by their statuses, and sends entities to Liferay DXP's workflow framework whenever the `addGuestbook` or `addGuestbookEntry` methods are called. Recall that you still have an uncalled service method, `updateStatus`, for both entities. Now you'll implement workflow handlers, classes that interact with Liferay DXP's workflow framework and your service layer (by calling `updateStatus` on the appropriate entity).

There's a handy abstract class you can extend to make the job easier, called `BaseWorkflowHandler`. You'll do this next for both entities of the guestbook project, starting with guestbooks.

Let's Go!

CHAPTER 71

CREATING A WORKFLOW HANDLER FOR GUESTBOOKS

```
<p>Handling Workflow<br>Step 1 of 2</p>
```

Each workflow enabled entity needs a `WorkflowHandler`.

1. Create a new package in the `guestboook-service` module called `com.liferay.docs.guestbook.workflow`, then create the `GuestbookWorkflowHandler` class in it. Extend `BaseWorkflowHandler` and pass in `Guestbook` as the type parameter:

   ```
   public class GuestbookWorkflowHandler extends BaseWorkflowHandler<Guestbook> {
   ```

2. Make it a Component class:

   ```
   @Component(immediate = true, service = WorkflowHandler.class)
   ```

3. There are three abstract methods to implement: `getClassName`, `getType`, and `updateStatus`. First add `getClassName`:

   ```
   @Override
   public String getClassName() {
       return Guestbook.class.getName();
   }
   ```

   `getClassName` returns the guestbook entity's fully qualified class name (`com.liferay.docs.guestbook.model.Guest`

4. Next, add `getType`:

   ```
   @Override
   public String getType(Locale locale) {
       return _resourceActions.getModelResource(locale, getClassName());
   }
   ```

   `getType` returns the model resource name (`model.resource.com.liferay.docs.guestbook.model.Guestbook`).

5. Finally, add the meat of the workflow handler, which is in the `updateStatus` method:

```
@Override
public Guestbook updateStatus(
        int status, Map<String, Serializable> workflowContext)
    throws PortalException {

    long userId = GetterUtil.getLong(
        (String)workflowContext.get(WorkflowConstants.CONTEXT_USER_ID));
    long resourcePrimKey = GetterUtil.getLong(
        (String)workflowContext.get(
            WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));

    ServiceContext serviceContext = (ServiceContext)workflowContext.get(
        "serviceContext");

    return _guestbookLocalService.updateStatus(
        userId, resourcePrimKey, status, serviceContext);
}
```

   When you crafted the service layer's `updateStatus` method (see the last section for more details), you specified parameters that must be passed to the method. Here you're making sure that those parameters are available to pass to the service call. Get the `userId` and `resourcePrimKey` from `GetterUtil`. Its `getLong` method takes a `String`, which you can get from the `workflowContext` `Map` using `WorkflowConstants` for the context user ID and the context entry class PK.

6. Make sure you inject the `ResourceActions` service into a private variable at the end of the class, using the `@Reference` annotation:

```
@Reference(unbind = "-")
protected void setResourceActions(ResourceActions resourceActions) {

    _resourceActions = resourceActions;
}

private ResourceActions _resourceActions;
```

7. Inject a `GuestbookLocalService` into a private variable using the `@Reference` annotation.

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

        _guestbookLocalService = guestbookLocalService;
    }

    private GuestbookLocalService _guestbookLocalService;

}
```

8. Organize imports (*[CTRL]+[SHIFT]+O*) and save your work.

Now the Guestbook application updates the database with the necessary status information, interacting with Liferay's workflow classes to make sure each entity is properly handled by Liferay DXP. At this point you can enable workflow for the Guestbook inside Liferay DXP and see how it works. Navigate to *Control Panel → Workflow → Process Builder → Configuration*. The Guestbook entity appears among Liferay DXP's native entities. Enable the Single Approver Workflow for Guestbooks;

then go to the Guestbook Admin portlet and add a new Guestbook. A notification appears next to your user name in the product menu. You receive a notification from the workflow that a task is ready for review. Click it, and you're taken to the My Workflow Tasks portlet, where you can complete the review task.

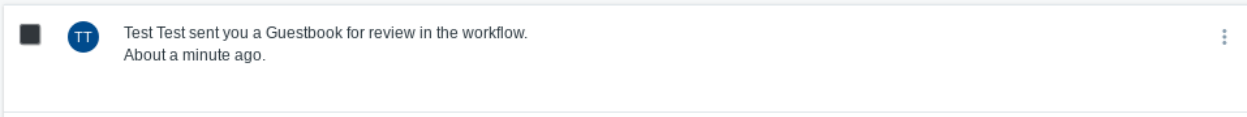

Figure 71.1: Click the workflow notification in the Notifications portlet to review the guestbook submitted to the workflow.

To complete the review, click the actions button ( ) from My Workflow Tasks and select *Assign to Me*. Click the actions button again and select *Approve*.

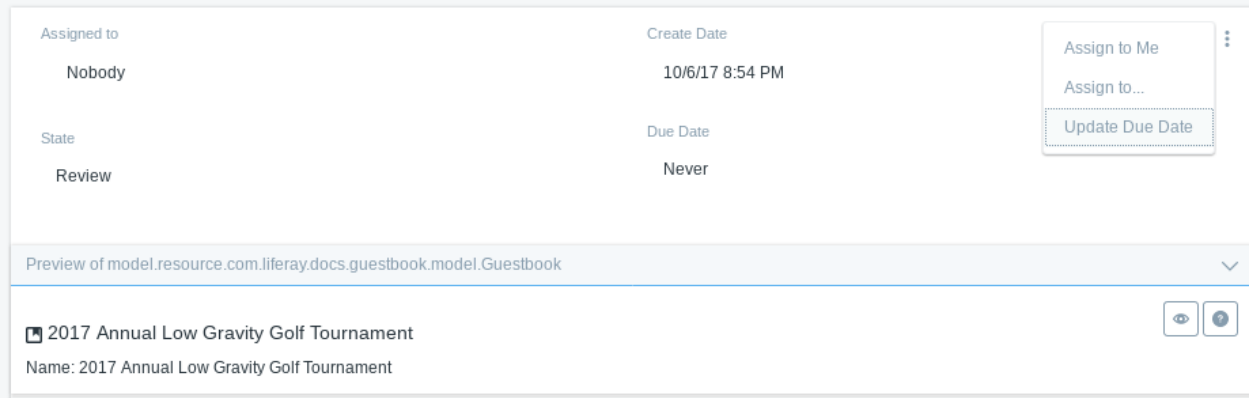

Figure 71.2: Click the workflow notification in the Notifications portlet to review the guestbook submitted to the workflow.

Right now the workflow process for guestbooks is functional, but the UI isn't adapted for it. You'll write the workflow handler for guestbook entries next, and then update the UI to account for each entity's workflow status.

CHAPTER 72

CREATING A WORKFLOW HANDLER FOR GUESTBOOK ENTRIES

Handling Workflow
Step 2 of 2

The Guestbook entry's workflow handler is almost identical to the guestbook's.

1. Create a new class in the `com.liferay.docs.guestbook.workflow` package of the `guestbook-service` module. Name it `GuestbookEntryWorkflowHandler` and extend `BaseWorkflowHandler`. Paste this in as the class body:

```
@Component(immediate = true, service = WorkflowHandler.class)
public class GuestbookEntryWorkflowHandler extends BaseWorkflowHandler<GuestbookEntry> {

    @Override
    public String getClassName() {

        return GuestbookEntry.class.getName();

    }

    @Override
    public String getType(Locale locale) {

        return _resourceActions.getModelResource(locale, getClassName());

    }

    @Override
    public GuestbookEntry updateStatus(
        int status, Map<String, Serializable> workflowContext)
        throws PortalException {

        long userId = GetterUtil.getLong(
            (String) workflowContext.get(WorkflowConstants.CONTEXT_USER_ID));
        long resourcePrimKey = GetterUtil.getLong(
            (String) workflowContext.get(
                WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));

        ServiceContext serviceContext =
            (ServiceContext) workflowContext.get("serviceContext");

        long guestbookId =
```

```
            _guestbookEntryLocalService.getGuestbookEntry(resourcePrimKey).getGuestbookId();

        return _guestbookEntryLocalService.updateStatus(
            userId, guestbookId, resourcePrimKey, status, serviceContext);
    }

    @Reference(unbind = "-")
    protected void setGuestbookEntryLocalService(GuestbookEntryLocalService guestbookEntryLocalService) {

        _guestbookEntryLocalService = guestbookEntryLocalService;
    }

    @Reference(unbind = "-")
    protected void setResourceActions(ResourceActions resourceActions) {

        _resourceActions = resourceActions;
    }

    private GuestbookEntryLocalService _guestbookEntryLocalService;
    private ResourceActions _resourceActions;
}
```

There is nothing unique about this code as compared with the guestbook's workflow handler, except that we need the `gustbookId` for the entry. That's easily obtained by getting the `GuestbookEntry` object with `guestbookEntryLocalService`, then getting its `guestbookId`. See the last article for the rest of the handler's implementation details.

2. Organize imports with *CTRL+SHIFT+O* and save the file.

The back-end of the guestbook project is fully workflow enabled. All that's left is to update the Guestbook Application's UI to handle workflow status.

CHAPTER 73

DISPLAYING APPROVED WORKFLOW ITEMS

There's not much left to do. Both entities in the guestbook project's back-end are workflow enabled, so it's time to update the UI. The Guestbook Admin portlet and the Guestbook portlet each requires its own display strategy.

The Guestbook Admin application is accessed by administrators, so it can display all guestbooks that have been submitted, even if they're not marked as approved. However, adding a *Status* field to the search container makes sense. That way admins can see which guestbooks are already approved, which are drafts, which are pending, etc.

The Guestbook application is meant to be viewed by site members and even guests (unauthenticated users of your site). Here it's smart to display only approved guestbooks and approved entries.

Start by updating the Guestbook Admin UI.

Let's Go!

CHAPTER 74

DISPLAYING GUESTBOOK STATUS

```
<p>Displaying Approved Workflow Items<br>Step 1 of 2</p>
```

The Guestbook Admin application's main view currently has a search container with two columns: the guestbook name and the guestbook actions button.

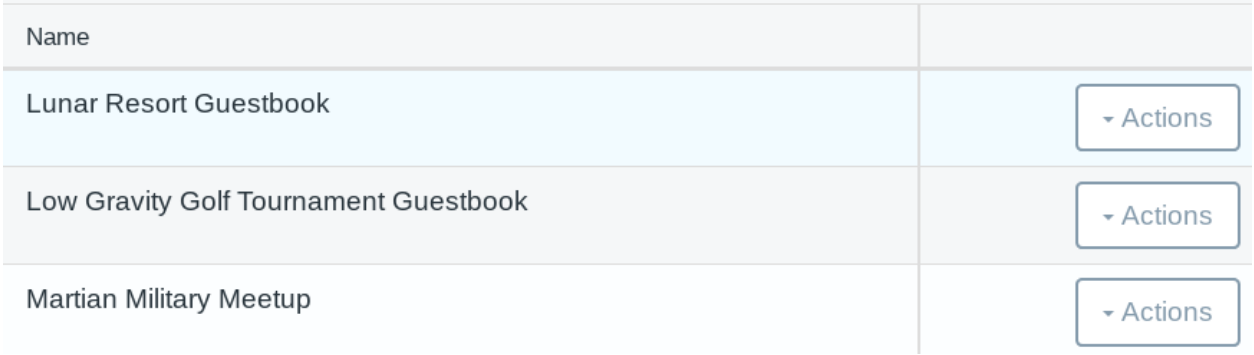

Figure 74.1: The Guestbook Admin's main view currently shows the name of the guestbook and its actions button.

Now you'll add a third column between the two existing ones: *Status*.

1. Open `guestbook-web/src/main/reosurces/META-INF/resources/guestbook_admin/view.jsp`.
2. Find the existing `search-container-column` definitions:

```
<liferay-ui:search-container-column-text property="name" />

<liferay-ui:search-container-column-jsp align="right"
    path="/guestbook_admin/guestbook_actions.jsp" />
```

3. Put the following new column between the existing columns:

```
<liferay-ui:search-container-column-status property="status" />
```

Save the file and wait for the `web` module to redeploy. With the addition of one line in the JSP, the Guestbook Admin application now displays the guestbook's workflow status.

Now you can move on to the Guestbook application's view layer.

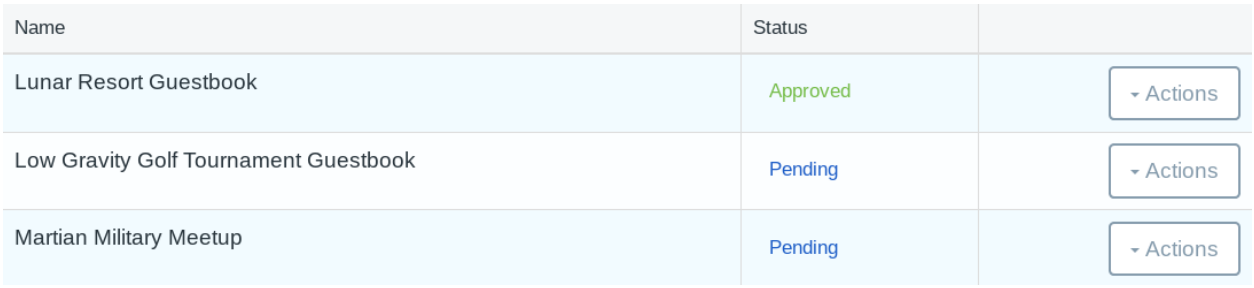

Figure 74.2: The Guestbook Admin's main view, displaying the status of each guestbook.

CHAPTER 75

DISPLAYING APPROVED ENTRIES

```
<p>Displaying Approved Workflow Items<br>Step 2 of 2</p>
```

The Guestbook application needs to be updated so that only guestbooks and entries with a status of *approved* appear in the UI.

Change the getters used to retrieve both entities in the view layer.

1. You need a new import, so first open `guestbook-web/src/main/resources/META-INF/resources/init.jsp` and add this line:

   ```
   <%@ page import="com.liferay.portal.kernel.workflow.WorkflowConstants"%>
   ```

2. Now open `guestbook-web/src/main/resources/META-INF/resources/guestbook/view.jsp`. Find the scriptlet that retrieves guestbooks:

   ```
   <%
       List<Guestbook> guestbooks = GuestbookLocalServiceUtil
                   .getGuestbooks(scopeGroupId);
               for (int i = 0; i < guestbooks.size(); i++) {
                   Guestbook curGuestbook = (Guestbook) guestbooks.get(i);
                   String cssClass = StringPool.BLANK;
                   if (curGuestbook.getGuestbookId() == guestbookId) {
                       cssClass = "active";
                   }
                   if (GuestbookPermission.contains(
                       permissionChecker, curGuestbook.getGuestbookId(), "VIEW")) {

   %>
   ```

   Change it so it calls the getter you added that takes workflow status into account. All you need to do is change this method call

   ```
   List<Guestbook> guestbooks = GuestbookLocalServiceUtil
               .getGuestbooks(scopeGroupId);
   ```

   to

```
List<Guestbook> guestbooks = GuestbookLocalServiceUtil
            .getGuestbooks(scopeGroupId, WorkflowConstants.STATUS_APPROVED);
```

Save the file, and now only approved guestbooks are displayed in the Guestbook application.

3. Next, update the entry's UI in the same `view.jsp`. Find the tags that set the search container's total and its results:

```
<liferay-ui:search-container total="<%=GuestbookEntryLocalServiceUtil.
                getGuestbookEntriesCount()%>">
<liferay-ui:search-container-results results=
                "<%=GuestbookEntryLocalServiceUtil.getGuestbookEntries
                (scopeGroupId.longValue(),
                guestbookId, searchContainer.getStart(),
                searchContainer.getEnd())%>" />
```

Replace the getters to use the ones that take workflow status as a parameter, and pass `WorkflowConstants.STATUS_APPROVED` as the status. Here's what it looks like when you're finished:

```
<liferay-ui:search-container total="<%=GuestbookEntryLocalServiceUtil.
                getGuestbookEntriesCount(scopeGroupId.longValue(),
                guestbookId, WorkflowConstants.STATUS_APPROVED)%>">
<liferay-ui:search-container-results results=
                "<%=GuestbookEntryLocalServiceUtil.getGuestbookEntries(
                scopeGroupId.longValue(), guestbookId,
                WorkflowConstants.STATUS_APPROVED,
                searchContainer.getStart(), searchContainer.getEnd())%>" />
```

Now only approved entries are displayed, and the search container's counter only counts the approved entries. If you update the `getGuestbookEntries` call but not the `getGuestbookEntriesCount` call, the count that's displayed includes approved entries and entries with any other workflow status, and it won't match the total that's displayed at the bottom of the search container.

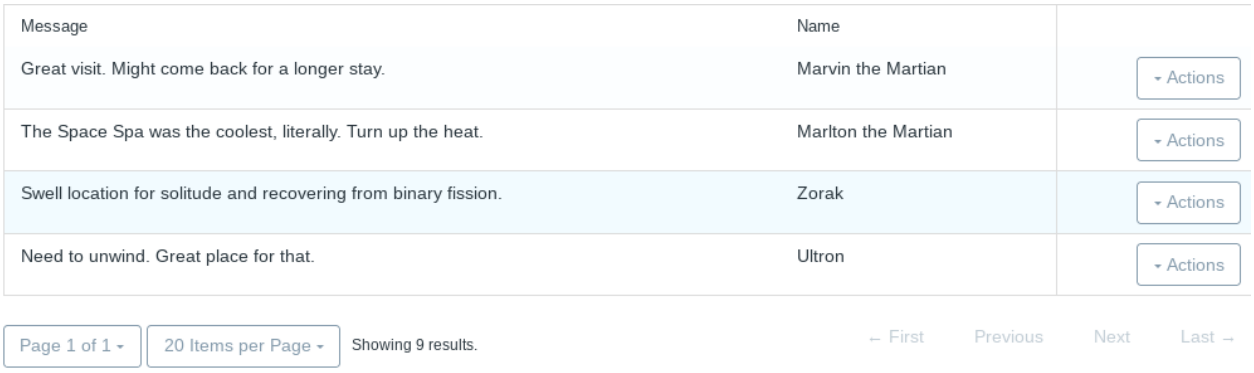

Figure 75.1: If you don't update the counter method to account for workflow status, it displays an incorrect count in the search container.

Now Guestbooks and Guestbook Entries are now fully workflow enabled, to the great relief of the Lunar Resort's site administrators. You've saved them a lot of headaches dealing with inappropriate content, primarily submitted by visitors from Mars. Those Martians really need some lessons in netiquette.

CHAPTER 76

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.

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. You can also use the planner to upgrade your data; this is a separate process that must be done independently from the code upgrade process.

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. {.root}Upgrade Your Development Environment

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

   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.{.summary}

      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.{.summary}
      2. Import Existing Liferay Workspace
         Import an existing Liferay Workspace. If you don't have one, revisit the previous step.{.summary}

   2. Configure Liferay Workspace Settings

      Set the Liferay DXP version in workspace's configuration you intend to upgrade to.{.summary}

      1. Configure Workspace Product Key
         Configure your workspace by setting a product key.{.summay}
      2. Initialize Server Bundle
         Download the Liferay DXP bundle you're upgrading to.

3. Migrate .cfg Files to .config Files
   Convert `.cfg` files to `.config` files.{.summay}

2. Migrate Plugins SDK Projects

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

   1. Import Existing Plugins SDK Projects
      Import your existing Plugins SDK projects.{.summary}
   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.{.summary}

3. Upgrade Build Dependencies

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

   1. Update Repository URL
      Update your repository URL to Liferay's frequently updated CDN repository.{.summary}
   2. Update Workspace Plugin Version
      Update your Workspace plugin version to leverage the latest features of Liferay Workspace.{.summary}
   3. Remove Dependency Versions
      Remove the project's dependency versions since it's leveraging target platform.{.summary}

4. Fix Upgrade Problems

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

   1. Auto-Correct Upgrade Problems
      Auto-correct straightforward upgrade problems.{.summary}
   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.{.summary}
   3. Resolve Upgrade Problems
      Mark upgrade problems as resolved after addressing them.{.summary}
   4. Remove Problem Markers
      After fixing your upgrade problems, remove the problem markers.{.summary}
   5. Resolving a Project's Dependencies
   6. Resolving Breaking Changes

5. Upgrade Service Builder Services

   Upgrade your Liferay Service Builder services.{.summary}

1. Remove Legacy Files
   Remove legacy files that are no longer leveraged by Service Builder.{.summary}
2. Migrate from Spring DI to OSGi Declarative Services
   Leverage OSGi Declarative Services in your Service Builder project.{.summary}
3. Rebuild Services
   Rebuild your project's services to persist your updates.{.summary}

6. Upgrade Customization Plugins

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

   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

7. Upgrade Themes

   Upgrade your themes so they're deployable to 7.0.{.summary}

   1. Upgrade 6.2 Themes to 7.2
   2. Upgrade 7.0 Themes to 7.2
   3. Upgrade 7.1 Themes to 7.2

8. Upgrade Layout Templates

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

9. Upgrade Frameworks & Features

   1. Upgrade JNDI Data Source Usage
      Use Liferay DXP's class loader to access the app server's JNDI API.{.summary}
   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.{.summary}
   3. Upgrade Service Builder
      Adapt your app to account for Service Builder-specific changes.{.summary}

4. Migrate Off of Velocity Templates

   Velocity template usage is deprecated for 7.0. You should convert your template to FreeMarker.{.summary}

10. Upgrade Portlets

    Upgrade your portlets so they're deployable to 7.0.{.summary}

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

11. Upgrade Web Plugins

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

12. 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.{.summary}

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

CHAPTER 77

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.

77.1 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.

77.2 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.

77.3 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.

77.4 Configuring Liferay Workspace Settings

You must configure your workspace with the Liferay DXP version you intend to upgrade to.

77.5 Configure Workspace Product Key

Configure your workspace by setting a product key. This automatically sets the Target Platform version, Docker image name, bundle URL, and other default settings for the Liferay DXP release.

77.6 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.

77.7 Migrate .cfg Files to .config Files

`.config` files are preferred over `.cfg` files because they allow specifying a property value's type, and allow multi-valued properties.

CHAPTER 78

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.

78.1 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!

78.2 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.

CHAPTER 79

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.

79.1 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.

79.2 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.

79.3 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.

CHAPTER 80

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.

80.1 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.

80.2 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.

80.3 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.

80.4 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 Code Upgrade Tool. Next, you'll take a deeper look at resolving project dependency errors.

Let's Go!

CHAPTER 81

RESOLVING A PROJECT'S DEPENDENCIES

Fixing Upgrade Problems Step 1 of 2

You may have compile errors due to missing Liferay 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 project. 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 can be grouped into three categories:

- Class moved to a package in the classpath
- Class moved to a module *not* in the classpath
- Class replaced or removed

Continue on to learn how to resolve each change.

81.1 Class Moved to a Package in the Classpath

This change is common and easy to fix. Consider resolving these classes first.

Since the module is already on your classpath, you need only update the class import. You can do this by using the Liferay Upgrade Planner or by organizing imports in Dev Studio/IntelliJ. The Upgrade Planner reports each moved class for you to address one by one. Organizing imports in Dev Studio/IntelliJ automatically resolves multiple classes at once.

It's typically faster to resolve moved classes using the mentioned IDEs. You can follow similar instructions for both IDEs:

1. Comment out or remove any imports marked as errors.
2. Execute the *Organize Imports* keyboard sequence *Ctrl-Shift-o* (Dev Studio) or *Ctrl-Alt-o* (IntelliJ).

The IDEs automatically generate the new import statements. If there is more than one available import package for a class, a wizard appears that lets you select the correct import.

Great! You've updated your class imports!

81.2 Class Moved to a Module 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.

Before Liferay DXP 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 The Benefits of Modularity. 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.

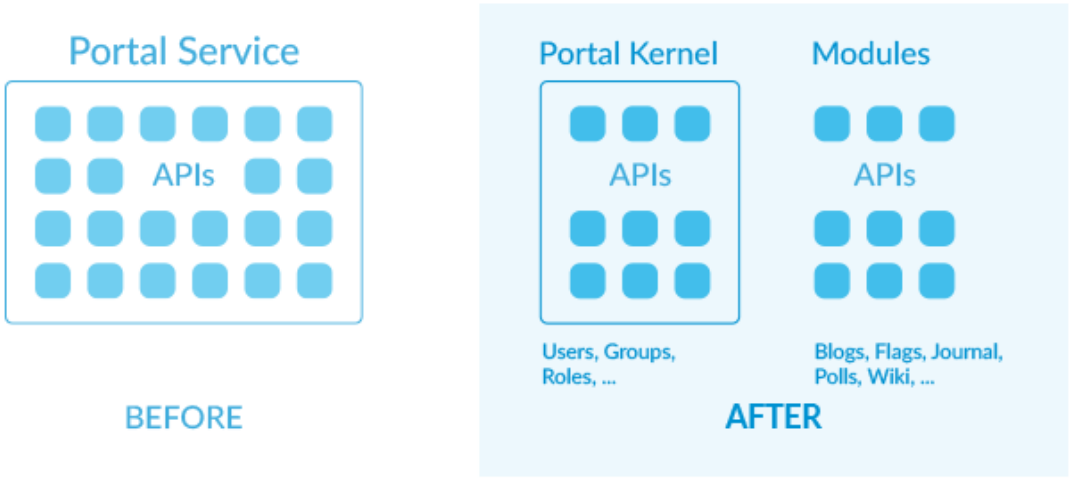

Figure 81.1: Liferay refactored the portal-service JAR for 7.0. Application APIs now exist in their own modules, and the portal-service JAR is now *portal-kernel*.

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 in Liferay DXP 7.1. 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.

For more information on finding and resolving your project dependencies, see Configuring Dependencies.

81.3 Class Replaced or Removed

In cases where the class has been replaced by another class or removed from the product, some investigation is required. The easiest way to resolve this type of issue is to use the Upgrade Planner. It finds removed classes your project is referencing and explains what happened to the class, how to handle the change, and why the change was made. These are listed as breaking changes (among other types of changes). Move on to the next section to learn about Liferay's breaking changes.

CHAPTER 82

Resolving Breaking Changes

Fixing Upgrade Problems Step 2 of 2

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
- Liferay DXP 7.1 Breaking Changes
- 7.0 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 83

Upgrading Service Builder Services

To properly upgrade app's leveraging service builder, you must complete the following steps:

- Remove Legacy Files
- Migrate from Spring DI to OSGi Declarative Services
- Rebuild Services

You'll start by removing legacy files.
Let's Go!

CHAPTER 84

REMOVING LEGACY FILES

Upgrading Service Builder Services Step 1 of 3

The first step in upgrading your Service Builder services is to delete legacy files. These legacy files include

- `portlet-spring.xml`
- `shard-data-source-spring.xml`
- `/src/main/resources/META-INF/` (folder)

When initiating this step, these files/folders are automatically removed from your Service Builder project.

If you're manually upgrading your code, delete the listed files/folders above.

Next, you'll convert your Service Builder Module from Spring DI to OSGi DS.

Chapter 85

Converting a Service Builder Module from Spring DI to OSGi DS

Upgrading Service Builder Services Step 2 of 3

Prior to 7.0, Service Builder modules could only use Spring for dependency injection (DI). Now OSGi Declarative Services (DS) is the default DI mechanism for new Service Builder modules. Although OSGi DS is the default DI mechanism, Spring is still supported. Therefore, this is an optional migration step.

To learn more about the decision to convert your Service Builder modules' DI mechanism and how to complete the conversion process, see the Migrating a Service Builder Module from Spring DI to OSGi DS article.

CHAPTER 86

REBUILDING SERVICES

```
<p id="stepTitle">Upgrading Service Builder Services</p><p>Step 3 of 3</p>
```

To properly upgrade Service Builder projects, you must rebuild all service classes so your changes are persisted across your project. Initiate this step to rebuild your services and finalize your Service Builder upgrade.

Great! Your Service Builder services are upgraded!

CHAPTER 87

UPGRADING CUSTOMIZATION 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
- Portlet JSP Hooks
- Service Wrapper Hooks
- Core Language Key Hooks
- Portlet Language Key Hooks
- Model Listener Hooks
- Event Actions Hooks
- Servlet Filter Hooks
- Portal Properties Hooks
- Struts Action Hooks

Continue on to get started!
Let's Go!

CHAPTER 88

UPGRADING CUSTOMIZATION MODULES

Upgrading Customization Plugins Step 1 of 11

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 Project'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!

CHAPTER 89

UPGRADING CORE JSP HOOKS

Upgrading Customization Plugins Step 2 of 11

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/flagged. For any remaining errors, consult the Resolving a Project'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.

Although you can upgrade your core JSP hook to 7.0, there are better ways to override a core JSP. The two recommended approaches are

- Dynamic includes
- Portlet filters

For more information on recommended ways of customizing JSPs in 7.0, see the Customizing JSPs section.

CHAPTER 90

UPGRADING PORTLET JSP HOOKS

Upgrading Customization Plugins Step 3 of 11

Getting a portlet 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/flagged. For any remaining errors, consult the Resolving a Project'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.

Although you can upgrade your portlet JSP hook to 7.0, there are better ways to override a portlet JSP. The two recommended approaches are

- Dynamic includes
- Portlet filters

For more information on recommended ways of customizing JSPs in 7.0, see the Customizing JSPs section.

CHAPTER 91

Upgrading Service Wrapper Hooks

Upgrading Customization Plugins Step 4 of 11

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/flagged. For any remaining errors, consult the Resolving a Project'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 service wrapper hook is now available in Liferay DXP.

CHAPTER 92

UPGRADING CORE LANGUAGE KEY HOOKS

Upgrading Customization Plugins Step 5 of 11

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` project (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.

CHAPTER 93

UPGRADING PORTLET LANGUAGE KEY HOOKS

Upgrading Customization Plugins Step 6 of 11

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` project (Gradle or Maven). 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.

CHAPTER 94

UPGRADING MODEL LISTENER HOOKS

Upgrading Customization Plugins Step 7 of 11

Developers have been creating model listeners for several Liferay DXP 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/flagged. For any remaining errors, consult the Resolving a Project'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 model listener hook is now available in Liferay DXP.

CHAPTER 95

Upgrading Event Action Hooks

Upgrading Customization Plugins Step 8 of 11

Event action hooks can be upgraded by completing 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/flagged. For any remaining errors, consult the Resolving a Project'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 event action hook is now available in Liferay DXP.

CHAPTER 96

UPGRADING SERVLET FILTER HOOKS

Upgrading Customization Plugins Step 9 of 11

If you have servlet filter hooks ready to be upgraded, this tutorial's for you. The process is 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/flagged. For any remaining errors, consult the Resolving a Project'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 Servlet Filter is running on 7.0!

CHAPTER 97

UPGRADING PORTAL PROPERTY HOOKS

Upgrading Customization Plugins Step 10 of 11

All portal properties in previous Liferay DXP versions that are also used in 7.0 can be overridden using portal property hooks. To upgrade portal property hooks, do this:

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/flagged. For any remaining errors, consult the Resolving a Project'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 are live!

CHAPTER 98

UPGRADING STRUTS ACTION HOOKS

```
<p id="stepTitle">Upgrading Customization Plugins</p><p>Step 11 of 11</p>
```

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 Liferay DXP 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`.

98.1 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.

98.2 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 articles:

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

Now you know how to convert your StrutsActionWrappers to MVCCommands!

CHAPTER 99

Upgrading a Theme to 7.2

In these tutorials, you'll learn how to upgrade your themes from earlier versions of Liferay DXP to 7.0. As you go through this process, you'll learn how to upgrade your theme's metadata, styling, templates, UI, and more using all the best practices and standards. By the end of the tutorial, you'll have a theme that runs on 7.0.

To upgrade your theme, select the tutorial below that corresponds to the current version of your theme:

Let's Go 6.2!

Let's Go 7.0!

Let's Go 7.1!

Chapter 100

Upgrading Your Theme from Liferay Portal 6.2 to 7.2

In this tutorial, you'll upgrade the Lunar Resort theme developed in the Liferay Portal 6.2 Developing a Liferay Theme Learning Path to 7.0 using the Liferay JS Theme Toolkit. The Lunar Resort theme is similar to many Liferay Portal 6.2 themes, as it extends the `_styled` theme, adds configurable settings, and incorporates a responsive design that leverages Font Awesome icons and Bootstrap. The theme ZIP file contains its original source code.

As you upgrade this theme, you'll learn how to update metadata, theme templates, UI, and more using all the best practices and standards. Completing this tutorial prepares you for upgrading your own theme.

Let's Go!

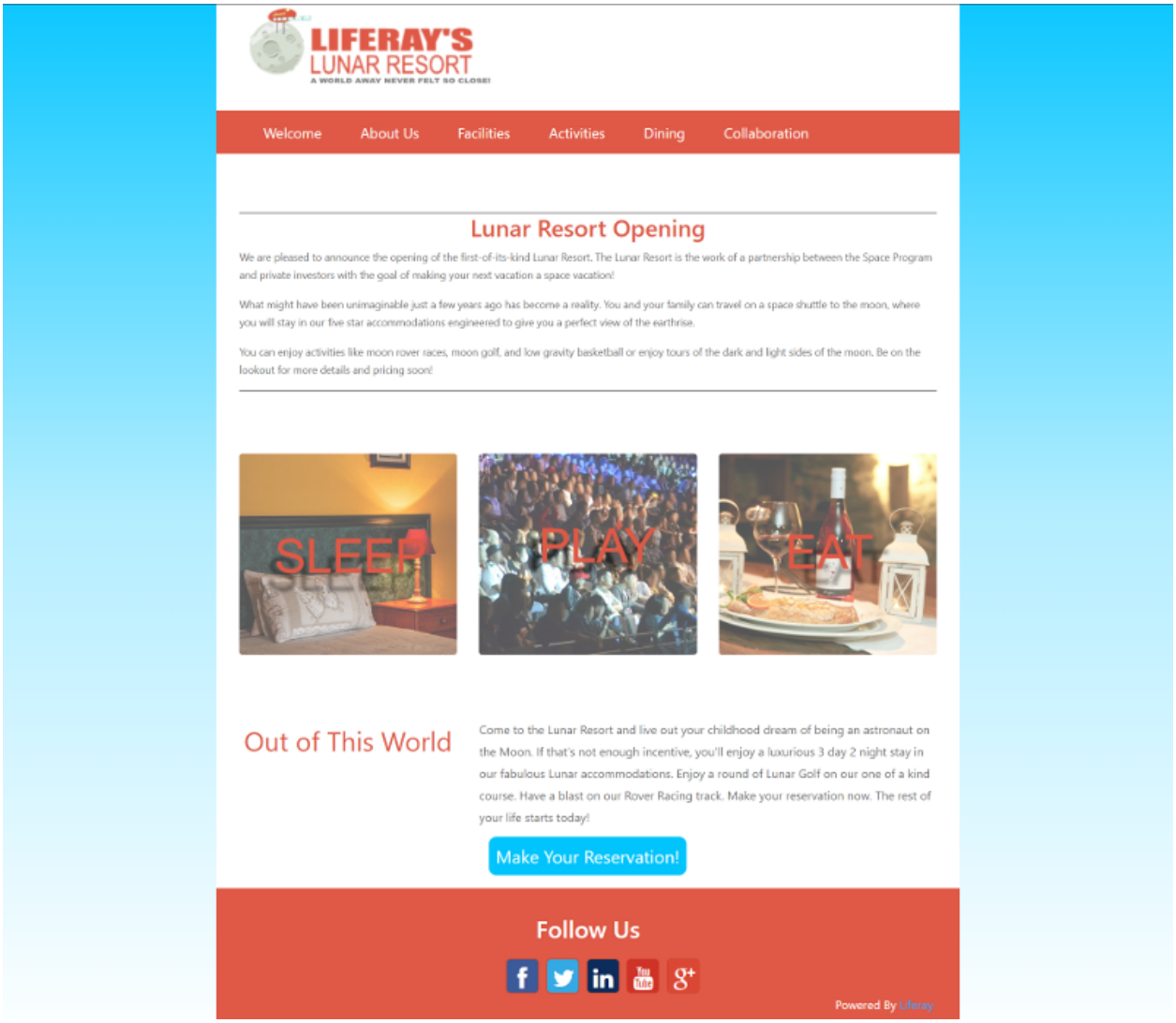

Figure 100.1: The Lunar Resort example theme upgraded in this tutorial uses a clean, minimal design.

CHAPTER 101

SETTING UP THE DEVELOPMENT ENVIRONMENT

In this section, you'll set up your development environment to use the Liferay JS Theme Toolkit. Setting up your development environment involves these steps:

- Install NodeJS with npm.
- Install Yeoman.
- Install the Liferay Theme Generator.
- Import the 6.2 theme to use the Liferay JS Theme Toolkit

Let's Go

CHAPTER 102

INSTALLING THE LIFERAY THEME GENERATOR TO IMPORT A 6.2 THEME

```
<p id="stepTitle">Setting up the Development Environment</p><p>Step 1 of 2</p>
```

Follow these steps:

1. Install NodeJS (along with Node Package Manager(npm)) if it's not already installed. We recommend installing the Long Term Support (LTS) version. Once NodeJS is installed, set up your npm environment.
2. Use npm to install the Yeoman dependency:

   ```
   npm install -g yo
   ```

3. Install the Liferay Theme Generator v8.x.x with the command below:

   ```
   npm install -g generator-liferay-theme@8.x.x
   ```

```
 **Note:** Liferay Theme Generator v8.x.x supports importing 6.2 themes to use
 the Liferay JS Theme Toolkit. Later on, you will install the latest version of
 the Liferay Theme Generator to complete the upgrade process.
```

```
If you're on Windows, follow the instructions in step 4 to install Sass,
otherwise you can skip that step.
```

4. The generator uses node-sass. If you're on Windows, you must also install node-gyp and Python.

Nice job! Your npm environment is set up and the Liferay Theme Generator and dependencies are installed. Next, you can import the Liferay DXP 6.2 theme to use the Liferay JS Theme Toolkit.

Chapter 103

Importing the Theme into the Liferay JS Theme Toolkit

Setting up the Development Environment Step 2 of 2

Now you'll import your theme to use the Liferay JS Theme Toolkit. The Liferay JS Theme Toolkit provides several useful Gulp tasks for automating theme development and maintenance, including upgrading parts of the theme.

Follow these steps to import your theme to use the Liferay JS Theme Toolkit:

1. Run the command below to import your 6.2 theme to use the Liferay JS Theme Toolkit:

   ```
   yo liferay-theme:import
   ```

2. Provide the path (relative or absolute) to the 6.2 theme project's root folder and press the *Enter* key to import the theme.

3. Answer the prompts to configure the location to your 7.0 app server.

Congratulations! Your development environment is set up to use the Liferay JS Theme Toolkit. Next, you'll start upgrading the theme.

CHAPTER 104

RUNNING THE UPGRADE TASK FOR 6.2 THEMES

You can upgrade a Liferay Portal 6.2 theme to 7.0, regardless of the development environment you use. This tutorial uses the Liferay JS Theme Toolkit's Gulp `upgrade` task to automate much of the steps. Because the theme was built on Liferay DXP 6.2, the Gulp `upgrade` task must be run three times to bring it up to 7.0.

The Liferay Theme Generator is available in a few different versions. To update the Liferay DXP 6.2 theme to Liferay DXP 7.0, you must install v8.x.x of the `liferay-theme-tasks` dependency. After the theme is updated to 7.1, you must then install v9.x.x of the `liferay-theme-tasks` dependency to complete the upgrade process.

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
- Updates CSS file names
- Updates theme dependencies
- Suggests specific code updates

Follow these steps to take the theme through the upgrade process:

1. Navigate to the theme's root directory and run the command below to update the theme's `liferay-theme-tasks` dependency to version `8.x.x`:

```
npm install --save-dev liferay-theme-tasks@8.x.x
```

2. Run the command below to initially upgrade it from 6.2 to 7.0.

```
**Note**: The Upgrade task overwrites the theme's files. We recommend that
you backup your files before proceeding with the upgrade process.
```

```bash
gulp upgrade
```

Here's what the 6.2 to 7.0 upgrade task does:

- Updates the theme's Liferay version
- Renames CSS files
- 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.

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

4. You must update the theme's `liferay-theme-tasks` dependency to version `9.x.x` to complete the upgrade process. Install the latest version of the Liferay Theme Generator as well while you're at it, so future uses of the tool will be compatible with the 7.0 theme. Both commands are shown below. Run them separately:

   ```
   npm install --save-dev liferay-theme-tasks@9.x.x

   npm install -g generator-liferay-theme@9.x.x
   ```

5. With the `9.x.x` versions of the `liferay-theme-tasks` and Liferay Theme Generator installed, run the `gulp upgrade` command for the final time to upgrade the 7.1 theme to 7.2:

   Here's what it does:

   - Updates Liferay version references
   - Updates theme dependencies

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 remaining portion of this tutorial covers these manual steps.

Chapter 105

Updating 6.2 CSS Code

7.0's UI improvements require these CSS-related changes:

- Updating rules and imports
- Modifying responsiveness tokens

The theme upgrade process involves conforming to these changes.
Let's Go

Chapter 106

Updating 6.2 CSS Rules and Imports

```
<p id="stepTitle">Updating 6.2 CSS Code<p><p>Step 1 of 2</p>
```

7.0 uses Bootstrap 4.3's CSS rule syntax. Font Awesome icons have been removed from base themes, so you should remove those stale imports if you have them. 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)

Because Liferay Portal 7.0 uses Bootstrap 3, the default box model has been
changed to box-sizing: border-box. So if you were using width or height, and
padding together on an element, you may need to make changes, or those elements
may have unexpected sizes.

File: src/css/_aui_variables.scss
    Line 5: "$white" has been removed
    Line 31: "$white" has been removed
File: src/css/_custom.scss
    Line 201: Padding no longer affects width or height, you may need to change
    your rule (lines 201-227)
    Line 207: Padding no longer affects width or height, you may need to change
    your rule (lines 207-226)
    Line 212: You would change height from "62px" to "82px"
    Line 305: Padding no longer affects width or height, you may need to change
    your rule (lines 305-314)
    Line 308: You would change height from "39px" to "46px"
    Line 409: Padding no longer affects width or height, you may need to change
    your rule (lines 409-418)
```

Follow these steps to update your theme's Bootstrap rules and Font Awesome imports:

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. Update the width and height for all CSS rules that use padding. 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 these variables where they are used in the 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.

3. Font Awesome icons were removed from the theme and Font are now included as a package dependency if you answer yes (y) to include Font Awesome during the upgrade task. If you included the old imports in `_custom.scss`, they must be removed:

```
@import "aui/alloy-font-awesome/scss/mixins-alloy";
@import "aui/alloy-font-awesome/scss/variables";
```

Great! The rules and imports are updated. You can update the responsiveness next.

CHAPTER 107

UPDATING THE RESPONSIVENESS

```
<p id="stepTitle">Updating 6.2 CSS Code</p><p>Step 2 of 2</p>
```

Bootstrap 4 explicit media queries replaced the Bootstrap 2 `respond-to` mixins for CSS responsiveness. Follow these steps to update the theme's responsiveness:

1. Open `_custom.scss`.
2. Replace all `respond-to` mixins with corresponding media queries shown below. Note that some of the dimensions have slightly changed:

   Media Query Replacements

```
 Liferay Portal 6.2 Mixin                          |  &amp;nbsp;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) |

```
The Lunar Resort theme's original and updated syntax is shown below:

Original:

```scss
@include respond-to(phone, tablet) {
    html #wrapper #banner #navigation {
    ...
    }
}
```

Updated:
```

```scss
@include media-breakpoint-down(md) {
    html #wrapper #banner #navigation {
    ...
    }
}
```

The CSS code is updated! Next you'll update the theme's templates.

CHAPTER 108

UPDATING 6.2 THEME TEMPLATES

Liferay DXP 6.2 theme templates and 7.0 theme templates are essentially the same. 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`

The Gulp `upgrade` task 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 logs for the Lunar Resort theme:

```
 Liferay Upgrade (6.2 to 7)

File: portal_normal.ftl
    Warning: <@liferay.dockbar /> is deprecated, replace with
    <@liferay.control_menu /> for new admin controls.
    Warning: not all admin controls will be visible without
    <@liferay.control_menu />
    Warning: ${theme} variable is no longer available in Freemarker
```

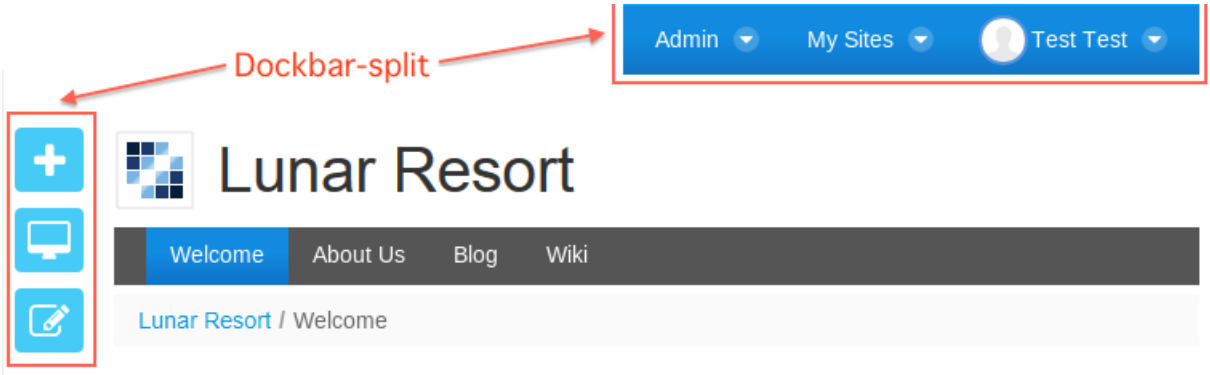

Figure 108.1: The Dockbar was removed and must be replaced with the new Control Menu.

```
    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. In this section you'll learn how to update various theme templates to 7.0.
Let's Go

CHAPTER 109

UPDATING 6.2 PORTAL NORMAL THEME TEMPLATE

Updating 6.2 Theme Templates Step 1 of 3

Follow these steps to update `portal_normal.ftl`:

1. Open `portal_normal.ftl` 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 /> </@>`
`${theme.include(body_bottom_include)}`	`<@liferay_util["include"] page=body_bottom_include />`
`${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. Remove the breadcrumbs and page title code:

```
<nav id="breadcrumbs">
    <@liferay.breadcrumbs />
</nav>
...
<h2 class="page-title">
    <span>${the_title}</span>
</h2>
```

3. 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. Replace the `|| is_signed_in` condition for the `navigation.ftl` theme template include with `&& is_setup_complete`:

```
<#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, and remove the breadcrumbs `<nav>` element from inside it.

`portal_normal.ftl` is updated! Next you can update the navigation template.

CHAPTER 110

UPDATING 6.2 NAVIGATION THEME TEMPLATE

Updating 6.2 Theme Templates Step 2 of 3

Follow these steps to update `navigation.ftl`:

1. Below the `<nav class="${nav_css_class}" id="navigation" role="navigation">` element, add the heading below to improve accessibility for screen readers:

   ```
   <h1 class="hide-accessible">
       <@liferay.language key="navigation" />
   </h1>
   ```

2. Remove the `nav_item_attr_selected` variable declaration at the top, and add the layout declaration shown below instead, to access the layout. Don't forget to remove all uses of `nav_item_attr_selected` throughout the rest of the template:

   ```
   <#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 element below:

   ```
   <@liferay_theme["layout-icon"] layout=nav_item_layout />
   ```

4. Remove the `nav_child_attr_selected` variable from the bottom of the template, including all uses throughout the rest of the template.

The navigation template is updated. You can update `portlet.ftl` next.

CHAPTER 111

UPDATING 6.2 INIT CUSTOM THEME TEMPLATE

```
<p id="stepTitle">Updating 6.2 Theme Templates</p><p>Step 3 of 3</p>
```

The Lunar Resort theme has a couple theme settings defined in `init_custom.ftl`. The syntax has changed slightly in 7.0. Follow these steps to update the theme setting syntax:

1. Replace the `getterUtil.getBoolean(theme_settings` method with `getterUtil.getBoolean(themeDisplay.getTheme`

   Original:

   ```
   <#assign show_breadcrumbs =
   getterUtil.getBoolean(theme_settings["show-breadcrumbs"])/>

   <#assign show_page_title =
   getterUtil.getBoolean(theme_settings["show-page-title"])/>
   ```

   Updated:

   ```
   <#assign show_breadcrumbs =
   getterUtil.getBoolean(themeDisplay.getThemeSetting("show-breadcrumbs"))/>

   <#assign show_page_title =
   getterUtil.getBoolean(themeDisplay.getThemeSetting("show-page-title"))/>
   ```

2. Although the Lunar Resort theme doesn't have any String variables, you would replace the `getterUtil.getString(theme_settings` method with `themeDisplay.getThemeSetting`:

   Original:

   ```
   <#assign string_setting =
   getterUtil.getString(theme_settings["my-string-key"])/>
   ```

   Updated:

   ```
   <#assign string_setting =
   themeDisplay.getThemeSetting("my-string-key")/>
   ```

Awesome! The theme templates are updated. You can always compare theme templates with the updated ones found in the `_unstyled` theme, if you're unsure if something has changed. Refer to the suggested changes that the Gulp `upgrade` task reports for the theme.

CHAPTER 112

UPDATING THE RESOURCES IMPORTER

The Resources Importer is now an OSGi module bundled with Liferay DXP, so you don't have to download the Resources Importer separately. The following components have been updated and are the focus of this section:

- 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 Lunar Resort theme's web content articles to avoid this issue.

Let's Go

CHAPTER 113

UPDATING 6.2 LIFERAY PLUGIN PACKAGE PROPERTIES

```
<p id="stepTitle">Updating 6.2 Resources Importer</p><p>Step 1 of 3</p>
```

Since the Lunar Resort theme was developed in the Plugins SDK, it requires the updates covered in this section. Themes developed outside of the Plugins SDK do not require these changes.

Follow these steps to update the Lunar Resort Theme's `liferay-plugin-package.properties` file:

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 the `liferay-plugin-package.properties` is updated, you can update the theme's web content.

CHAPTER 114

UPDATING 6.2 WEB CONTENT

```
<p id="stepTitle">Updating 6.2 Resources Importer</p><p>Step 2 of 3</p>
```

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 the theme's web content:

1. Create a subfolder called `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 just created (`BASIC_WEB_CONTENT`). The articles and template folder names **must match** for the web content to import properly.

3. XML article structures are now written in JSON. Open the `/resources-importer/journal/structures/` folder and create a new file inside called `BASIC_WEB_CONTENT.json`. The structure name **must match** the folder names 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. Add the JSON structure below to the `BASIC_WEB_CONTENT.json` file. This provides the required metadata to render standard web content articles (i.e. the language, fields, etc.):

```
{
    "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. Create a new FreeMarker template file for basic web content inside the `/resources-importer/journal/templates/BASIC_WEB_CONTENT` folder called `BASIC_WEB_CONTENT.ftl`, and add the method below to retrieve the article's data:

```
${content.getData()}
```

6. Convert the basic web content articles from HTML to XML to conform to the new format. Replace the `.html` file extensions with `.xml`. wrap each basic web content article's content with the XML 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[
                        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 `col-[device-size]-[number]` syntax. `[device-size]` can be `xs`, `sm`, `md`, or `lg`. See Bootstrap's documentation for more information. The original and updated classes for the Lunar Resort's `2 column description.xml` article are shown below:

   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>
```

The web content is updated! Next, you must update the theme's sitemap file.

CHAPTER 115

UPDATING THE 6.2 SITEMAP

```
<p id="stepTitle">Updating 6.2 Resources Importer</p><p>Step 3 of 3</p>
```

In Liferay DXP 6.2, portlet IDs were incremental numbers. In 7.0, they're explicit class names. Update the `sitemap.json` file with the new portlet IDs. Follow these steps to update the sitemap:

1. Replace the portlet IDs with the updated class names. The Portlet ID Quick Reference Guide list 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.

```
 **Note:** you can also retrieve a portlet's ID from the UI. Open the
 portlet's *Options* menu, select *Look and Feel Configuration*.

 ![ You can find the portlet ID in the *Look and Feel Configuration* menu.](./images/upgrading-themes-look-and-feel-menu.png)

 Select the *Advanced Styling* tab. The `Portlet ID` value appears in the
 blue box.

 ![ The portlet ID appears within the blue box in the *Advanced Styling* tab.](./images/upgrading-themes-portlet-id.png)
```

```
The original and updated versions of the Lunar Resort theme's `sitemap.json`
are shown below:

Original:

```json
{
    "name": "Collaboration",
    "title": "Collaboration",
    "friendlyURL": "/collaboration",
    "layoutTemplateId": "2_columns",
    "columns": [
        [
            {
            "portletId": "36"
            }
        ],
        [
            {
```
```

```
            "portletId": "115"
            }
        ]

    ]
}
```

Updated:

```json
{
  "name": "Collaboration",
  "title": "Collaboration",
  "friendlyURL": "/collaboration",
  "layoutTemplateId": "2_columns",
  "columns": [
    [
      {
      "portletId": "com_liferay_wiki_web_portlet_WikiPortlet"
      }
    ],
    [
      {
      "portletId": "com_liferay_blogs_web_portlet_BlogsAgreggatorPortlet"
      }
    ]

  ]
},
```

2. Update references to the web content articles in the `sitemap.json` to use the XML file extensions.

Great! The Resources Importer updates are complete. Next you'll apply Clay markup patterns to the theme's custom UI.

CHAPTER 116

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 the Lunar Resort's form.
Follow these steps:

1. Replace the `control-group` classes with `form-group` classes:
2. Remove the `control-label` classes from the `label` elements:
3. Remove `<div class="controls">` elements.
4. Add the `form-control` class to each `input` element.
5. Add the `btn-primary` class to your submit buttons to emphasize them.

The Lunar Resort's original form and updated form are shown below:
Original form markup:

```
<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>
```

Updated form markup:

```
<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>
```

The Lunar Resort theme is updated for 7.0!

CHAPTER 117

UPGRADING YOUR THEME FROM LIFERAY PORTAL 7.0 TO 7.2

In this tutorial, you'll learn how to use the Liferay JS Theme Toolkit to upgrade a Liferay DXP 7.0 theme to 7.0. As you upgrade this theme, you'll learn how to update metadata, theme templates, UI (including support for Bootstrap 4 and Lexicon 2.0.), and more using all the best practices and standards. Completing this tutorial prepares you for upgrading your own theme.

Theme upgrades involve these steps:

- Updating project metadata
- Updating CSS
- Updating theme templates

Let's Go!

CHAPTER 118

RUNNING THE UPGRADE TASK FOR 7.0 THEMES

You can upgrade a Liferay DXP 7.0 theme to 7.0, regardless of the development environment you use. This tutorial uses the Liferay JS Theme Toolkit's Gulp `upgrade` task to automate much of the steps. The Gulp `upgrade` task must be run twice to bring a Liferay DXP 7.0 theme up to 7.0.

The Liferay Theme Generator is available in a few different versions. To update the Liferay DXP 7.0 theme to Liferay DXP 7.1, you must install v8.x.x of the `liferay-theme-tasks` dependency. After the theme is updated to 7.1, you must then install v9.x.x of the `liferay-theme-tasks` dependency to complete the upgrade process.

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

Follow these steps to take the theme through the upgrade process:

1. Navigate to the theme's root directory and run the command below to update the theme's `liferay-theme-tasks` dependency to version `8.x.x`:

   ```
   npm install --save-dev liferay-theme-tasks@8.x.x
   ```

2. Run the `gulp upgrade` command to upgrade the Liferay DXP 7.0 theme to 7.1.

```
**Note**: The Upgrade task overwrites the theme's files. We recommend that
you backup your files before proceeding with the upgrade process.
```

```
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
```

3. You must update the theme's `liferay-theme-tasks` dependency to version `9.x.x` to complete the upgrade process. Install the latest version of the Liferay Theme Generator as well while you're at it, so future uses of the tool will be compatible with the 7.0 theme. Both commands are shown below. Run them separately:

    ```
    npm install --save-dev liferay-theme-tasks@9.x.x

    npm install -g generator-liferay-theme@9.x.x
    ```

4. With the `9.x.x` versions of the `liferay-theme-tasks` and Liferay Theme Generator installed, run the `gulp upgrade` command for the final time to upgrade the 7.1 theme to 7.2:

    Here's what it does:

    - Updates Liferay version references
    - Updates theme dependencies

```
**Note:** Since Liferay DXP Fix Pack 2 and Liferay Portal 7.2 CE GA2, Font
Awesome is available globally as a system setting, which is enabled by
default. If you're using Font Awesome icons in your theme, answer yes (y)
to the Font Awesome question during the Upgrade task to include Font
Awesome imports in your theme. This ensures that your icons won't break if
a Site Administrator disables the global setting.
```

---

5. Run `gulp init` from your theme's root directory to update the path of your Liferay DXP server to point to your 7.2 Liferay DXP server.

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. This jump-starts the upgrade process, but it doesn't complete it. Manual updates are required. The remaining portion of this tutorial covers these manual steps.

CHAPTER 119

UPDATING 7.0 CSS CODE

7.0's UI improvements requires 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 the theme's CSS files to reflect these changes. Start with updating CSS file names.

Let's Go

Chapter 120

Updating 7.0 CSS File Names for Clay

```
<p id="stepTitle">Updating 7.0 CSS Code</p><p>Step 1 of 3</p>
```

Some of the CSS filenames have changed to reflect the introduction of Clay (previously Lexicon CSS). The file name changes for the Unstyled theme are listed below. Refer to the Theme Reference Guide for a complete list of expected theme CSS files.

Orignal AUI file names:

- `css/`
    - `_aui_custom.scss`
    - `_aui_variables.scss`
    - `aui.scss`

Updated Clay file names:

- `css/`
    - `_clay_custom.scss`
    - `_clay_variables.scss`
    - `clay.scss`

Next, you can update the theme's CSS variables.

CHAPTER 121

UPDATING 7.0 CLASS VARIABLES

```
<p id="stepTitle">Updating 7.0 CSS Code</p><p>Step 2 of 3</p>
```

7.0 uses Bootstrap 4's CSS rule syntax. The new syntax lets developers leverage Bootstrap 4 features and improvements. The Migrating to v4 guide provides complete instructions for updating CSS rules to Bootstrap 4.

Follow these steps to upgrade the theme's CSS variables:

1. Consult the upgrade log produced by the `gulp upgrade` task. It suggests the manual Lexicon updates required for the theme.
2. Make the required changes in the log. 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 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
```

Note: If the `gulp upgrade` task detects any variables in the theme that are removed in Clay from the previous LexiconCSS version, it adds a `_variables_deprecated.scss` file to the theme containing the removed variables, to make sure the theme compiles and to decouple it from future upgrades.

After updating your theme's CSS variables and mixins, you should update the imports next.

CHAPTER 122

UPDATING 7.0 IMPORTS

```
<p id="stepTitle">Updating 7.0 CSS Code</p><p>Step 3 of 3</p>
```

Font Awesome imports and core imports have changed. Follow these steps to update the theme:

1. 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 are now included as a package dependency if you answer yes (y) to include Font Awesome during the upgrade task. If a 7.0 theme was made prior to this move and `_aui_variables.scss` was modified, the Font Awesome imports shown below must be removed from `_clay_variables.scss`:

   ```
   // 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/";
   ```

2. Update the old AUI lexicon paths to use the new 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";`|
```

Great! Your imports are updated, and your CSS upgrade is complete. Next you can upgrade the theme templates.

CHAPTER 123

UPDATING 7.0 THEME TEMPLATES TO 7.2

Liferay DXP 7.0 theme templates and 7.0 theme templates are essentially the same. 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 7.0.

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.

If you haven't converted your Velocity theme templates to FreeMarker, **you must convert your Velocity theme templates to FreeMarker now**.

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 between Liferay DXP 7.0 themes and 7.0 themes 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.

In this section you'll learn how to update various theme templates to 7.0.
Let's Go

CHAPTER 124

UPDATING 7.0 THEME TEMPLATES

Follow these steps to update the theme's templates. Note these changes are only required if the templates are modified in the theme:

1. Open `portal_normal.ftl` and remove the breadcrumbs:

```
<nav id="breadcrumbs">
    <@liferay.breadcrumbs />
</nav>
```

2. Still inside `portal_normal.ftl`, remove `id="main-surface"` from the `body` tag so it looks like the one below. This is not needed for SPA to work properly:

```
<body class="${css_class}">
```

3. Open `navigation.ftl` and remove the `nav_item_attr_selected` variable declaration at the top. Don't forget to remove all uses of the `nav_item_attr_selected` throughout the rest of the template.

4. Also inside `navigation.ftl`, remove the `nav_child_attr_selected` variable from the bottom of the template, including all uses throughout the rest of the template.

5. Open `portlet.ftl` and find the code snippet below:

```
<a
  class="icon-monospaced portlet-icon-back text-default"
  href="${portlet_back_url}"
  title="<@liferay.language key="return-to-full-page" />"
>
```

   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" />"
>
```

6. Still inside `portlet.ftl`, 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 theme templates are updated! If you modified any other FreeMarker theme templates, you can compare them with templates in the `_unstyled` theme. Next you can learn how to use Liferay DXP's compatibility layer to help ease the transition to Bootstrap 4 and Clay CSS.

CHAPTER 125

Using the Bootstrap 3 Lexicon CSS Compatibility Layer

By default, 7.0 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. 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. Inspect your themes UI with the compatibility layer enabled (it's enabled by default), and note any issues.

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, these 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 prevents Liferay DXP's UI from breaking.

Now you know how to use the Bootstrap 3 and Lexicon CSS compatibility layer to provide a smooth transition during your theme upgrade.

CHAPTER 126

UPGRADING 7.1 THEMES TO 7.2

You can upgrade a Liferay Portal 7.1 theme to 7.0, regardless of the development environment you use. This tutorial uses the Liferay JS Theme Toolkit's Gulp `upgrade` task to automate much of the steps. This requires v9.x.x of the Liferay Theme Generator and liferay theme tasks.

Here's what the Upgrade Task does:

- Updates Liferay version references
- Updates theme dependencies

Follow these steps to upgrade the theme:

1. Install the Liferay Theme Generator v9.x.x with the command below:

    ```
    npm install -g generator-liferay-theme@9.x.x
    ```

2. You must update the theme's `liferay-theme-tasks` dependency to version `9.x.x` as well to run the upgrade process:

    ```
    npm install --save-dev liferay-theme-tasks@9.x.x
    ```

3. With the `9.x.x` versions of the `liferay-theme-tasks` and Liferay Theme Generator installed, run the `gulp upgrade` command to upgrade the 7.1 theme to 7.2.

```
**Note**: The Upgrade task overwrites the theme's files. We recommend that
you backup your files before proceeding with the upgrade process.
```

---

4. In 7.1, Font Awesome & Glyphicons were included in the compatibility layer. Since Liferay DXP Fix Pack 2 and Liferay Portal 7.2 CE GA2, Font Awesome is available globally as a system setting, which is enabled by default. If you're using Font Awesome icons in your theme, answer yes (y) to the Font Awesome question during the Upgrade task to include the Font Awesome dependency in your theme. This ensures that your icons won't break if a Site Administrator disables the global setting.

5. Run `gulp init` from the theme's root directory to update the path of the app server to point to the new 7.2 app server.

There you have it! The theme is ready to run on 7.0.

Chapter 127

Upgrading a Layout Template to 7.2

In these tutorials, you'll learn how to upgrade your layout templates from earlier versions of Liferay DXP to 7.0. By the end of the tutorial, you'll have a layout template that runs on 7.0.

Select the tutorial below that corresponds to the current version of your layout template:

Let's Go 6.2!

Let's Go 7.0 and 7.1!

CHAPTER 128

UPGRADING 6.2 LAYOUT TEMPLATES TO 7.2

Upgrading your Liferay DXP 6.2 layout template to 7.0 requires a few updates:

1. Open your layout template's `liferay-plugin-package.properties` file and update the `liferay-versions` property to `7.2.0+`:

   ```
   liferay-versions=7.2.0+
   ```

2. Velocity layout templates are supported, but deprecated as of Liferay DXP 7.1. We recommend that you convert your Velocity layout templates to FreeMarker now. Wrap the `processor.processColumn("column-1", "portlet-column-content portlet-column-content-first")` methods with braces (`{...}`) and change the template's file extension to `.ftl`.

3. Update the Bootstrap `span[number]` classes to use the newer `col-[size]-[number]` classes. See Layout Templates for more information on the updated syntax.

4. Save the changes.

   Below is an example configuration:

   Original:

   ```
   <div class="span4 span6 portlet-column portlet-column-first"
   id="column-1">
           $processor.processColumn("column-1",
           "portlet-column-content portlet-column-content-first")
   </div>
   <div class="span8 span6 portlet-column portlet-column-last"
   id="column-2">
           $processor.processColumn("column-2",
           "portlet-column-content portlet-column-content-last")
   </div>
   </div>
   ```

   Updated:

   ```
   <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>
```

Awesome! Your layout template is upgraded.

CHAPTER 129

UPGRADING 7.0 AND 7.1 LAYOUT TEMPLATES TO 7.2

If you're upgrading your Liferay DXP 7.0 or Liferay DXP 7.1 layout template to 7.0, follow these steps:

1. Open your layout template's `liferay-plugin-package.properties` file and update the `liferay-versions` property to `7.2.0+`:

   ```
   liferay-versions=7.2.0+
   ```

2. Velocity layout templates are supported, but deprecated as of Liferay DXP 7.1. We recommend that you convert your Velocity layout templates to FreeMarker now. Wrap the `processor.processColumn("column-1", "portlet-column-content portlet-column-content-first")` methods with braces (`{...}`) and change the template's file extension to `.ftl`.

3. Save the changes.

   Below is an example configuration:

   Original (`my_layout_template.tpl`):

   ```
   <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>
   ```

   Updated (`my_layout_template.ftl`):

   ```
   <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>
```

Awesome! Your layout template is upgraded.

Chapter 130

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.
Let's Go!

Chapter 131

Upgrading JNDI Data Source Usage

Upgrading Frameworks and Features Step 1 of 4

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.

CHAPTER 132

UPGRADING SERVICE BUILDER SERVICE INVOCATION

Upgrading Frameworks and Features Step 2 of 4

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.

CHAPTER 133

UPGRADING SERVICE BUILDER

```
<p id="stepTitle">Upgrading Frameworks and Features</p><p>Step 3 of 4</p>
```

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.

133.1 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.

133.2 Step 2: Resolve Dependencies

To adapt your app's dependencies, refer to the Resolving a Project's Dependencies tutorial. Once your dependencies are upgraded, rebuild your services!

133.3 Step 3: Build the Services

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!

CHAPTER 134

MIGRATING OFF OF VELOCITY TEMPLATES

Upgrading Frameworks and Features Step 4 of 4

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 135

UPGRADING PORTLETS

All portlet types developed for Liferay Portal 6.x, 7.0, and 7.1 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. This makes resolving a portlet's dependencies 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
- Liferay JSF Portlet
- Servlet-based portlet
- Spring Portlet MVC
- Struts Portlet

Let's get your portlet running on 7.0!

Let's Go!

CHAPTER 136

UPGRADING A GENERICPORTLET

```
<p id="stepTitle">Upgrading Portlets</p><p>Step 1 of 6</p>
```

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. 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

When the portlet WAR file is deployed, 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 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!

CHAPTER 137

UPGRADING A LIFERAY MVC PORTLET

```
<p id="stepTitle">Upgrading Portlets</p><p>Step 2 of 6</p>
```

Liferay's MVC Portlet framework is used extensively in Liferay DXP's portlets and is a popular choice for 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 be upgraded to 7.0 without a hitch.

Upgrading a Liferay MVC 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
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 MVC Portlet on your 7.0 instance. Have fun showing off your upgraded portlet!

CHAPTER 138

UPGRADING A LIFERAY JSF PORTLET

```
<p id="stepTitle">Upgrading Portlets</p><p>Step 3 of 6</p>
```

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 article, you'll upgrade a Liferay DXP JSF portlet's (JSF 2.2) dependencies to 7.0.

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.
3. Compare the generated dependencies with your portlet's dependencies and make any necessary updates. For example, in the sample dependencies listed below, 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>
```

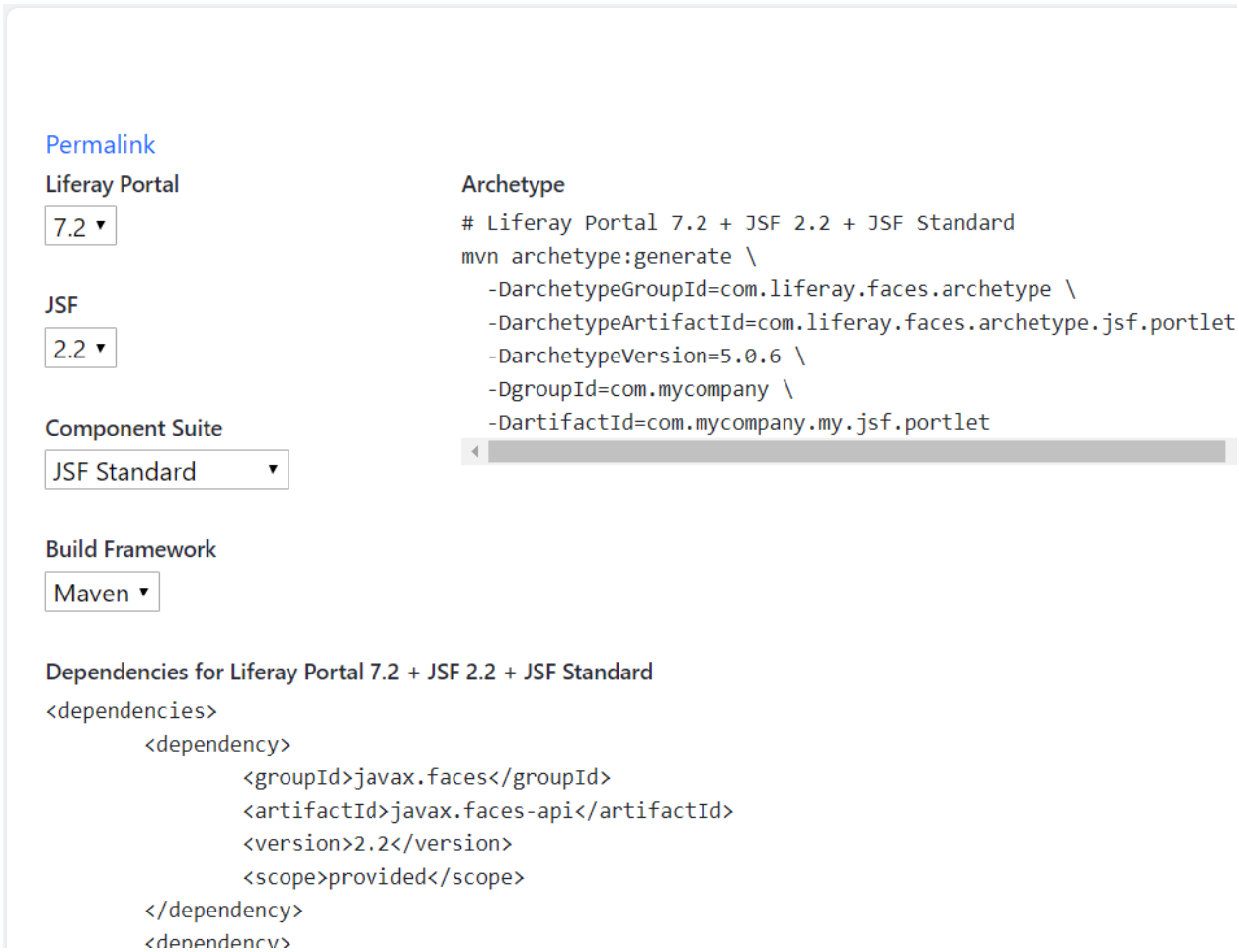

Figure 138.1: The Liferay Faces site gives you options to generate dependencies for many environments.

```
    <version>4.0.0</version>
</dependency>
```

Using the http://liferayfaces.org/ dependency list as a guide, these dependencies would be updated to

```
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.faces</artifactId>
    <version>2.2.19</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>com.liferay.faces</groupId>
    <artifactId>com.liferay.faces.bridge.ext</artifactId>
    <version>5.0.4</version>
</dependency>
<dependency>
    <groupId>com.liferay.faces</groupId>
    <artifactId>com.liferay.faces.bridge.impl</artifactId>
    <version>4.1.3</version>
</dependency>
```

Once your Liferay JSF portlet's dependencies are updated, it's deployable to 7.0! Follow the Deploying a Project article for deployment help.

When the portlet WAR is deployed, 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. 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-7.2-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-7.2-ga1/osgi/war][PortletHotDeployListener:293] 1 portlet for com.liferay.faces.demo.jsf.applica
1.0 is available for use
```

After the portlet deployment is complete, it's available on Liferay DXP.

You've learned how to upgrade and deploy a Liferay JSF portlet. You resolved dependencies and deployed the portlet as you always have. It's just that easy!

CHAPTER 139

UPGRADING A SERVLET-BASED PORTLET

```
<p id="stepTitle">Upgrading Portlets</p><p>Step 4 of 6</p>
```

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.

Follow these steps to upgrade a servlet-based portlet:

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
3. Deploy it

For an example upgrade scenario, consider this:

Some servlet-based portlets relied on Liferay Portal to provide several dependency JAR files. Here's the `portal-dependency-jars` property from a sample portlet's `liferay-plugin-package.properties` file:

```
portal-dependency-jars=\
    dom4j.jar,\
    jabsorb.jar,\
    json-java.jar
```

This property is deprecated in 7.0 because importing and exporting Java packages has replaced wholesale use of JARs. This means modules and WABs can import packages without concerning themselves with JARs. Liferay DXP exports many third party packages for plugins to use. Best practices for using packages that Liferay DXP exports are found here.

Once you've deployed your portlet, the server prints messages that indicate the following portlet status:

- WAR processing
- WAB startup
- Availability to users

The portlet is installed to Liferay's OSGi runtime and is available to users. Congratulations! You've upgraded and deployed your servlet-based portlet to 7.0.

CHAPTER 140

UPGRADING A SPRING PORTLET MVC PORTLET

Upgrading Portlets
Step 5 of 6

Upgraded portlets that use Spring Portlet MVC should be migrated to use PortletMVC4Spring. The main reason is that PortletMVC4Spring is maintained for compatibility with the latest versions of the Spring Framework.

Note: The PortletMVC4Spring project began as Spring Portlet MVC and was part of the Spring Framework. When the project was pruned from version 5.0.x of the Spring Framework under SPR-14129, it became necessary to fork and rename the project. This made it possible to improve and maintain the project for compatibility with the latest versions of the Spring Framework and the Portlet API.

Liferay adopted Spring Portlet MVC in March of 2019 and the project was renamed to Portlet-MVC4Spring.

For more information on PortletMVC4Spring, see its dedicated section of articles. For specific information on migrating a portlet using Spring Portlet MVC to PortletMVC4Spring, see the Migrating to PortletMVC4Spring article.

Once you've migrated your portlet to leverage the PortletMVC4Spring framework, you must also adapt your Liferay-specific APIs and dependencies. To do this, complete the following 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

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 migrated your Spring Portlet MVC portlet to the updated PortletMVC4Spring framework, updated any additional APIs and dependencies, and deployed it to your 7.0 instance. Your portlet's upgrade process is complete!

CHAPTER 141

UPGRADING A STRUTS 1 PORTLET

Upgrading Portlets Step 6 of 6

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!

CHAPTER 142

UPGRADING WEB PLUGINS

Upgrading Web Plugins Step 1 of 1

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!

CHAPTER 143

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 Finding Extension Points for more details.

For more information on Ext projects, how to decide if you need one, and how to manage them, see the Customization with Ext section.

CHAPTER 144

CREATING A THEME

This tutorial takes you step-by-step through the process of creating a theme. You'll create a responsive theme for Liferay's Lunar Resort that demonstrates best practices and uses Liferay DXP's theme tools, extensions, and mechanisms. Several example files are referenced throughout this tutorial. You can download the `lunar-resort-theme.zip` if you want to follow along locally. The Lunar Resort theme's files are also included in the `lunar-resort-theme` folder of the Liferay Docs repo, if you would rather view them there.

This tutorial covers these topics:

- Generating the theme and configuring it to extend the Atlas base theme
- Customizing the Header and logo
- Customizing the Header navigation
- Customizing the Footer and embedding footer navigation
- Creating a color scheme variant

By the end of this tutorial, you'll be able to create the theme below:

Figure 144.1: The finished Lunar Resort Theme uses Liferay DXP's tools to produce a user-friendly UI that is maintainable.

CHAPTER 145

SETTING UP THE THEME

In this section, you'll use the Liferay JS Theme Toolkit's Liferay Theme Generator to generate the theme's files. You'll complete these tasks:

- Install the Liferay Theme Generator and its dependencies
- Generate a theme
- Configure the theme to extend the Atlas base theme.

Atlas provides the look of the Classic theme. It builds on the default Clay Base theme and provides additional styles.

Follow these steps to generate and configure the theme:

1. Install the Theme Generator. Since you're developing a theme for 7.0, install v9.x.x if it's not installed already. Run the command below:

```
npm install -g generator-liferay-theme@9.x.x
```

2. Install the Yeoman and gulp dependencies:

```
npm install -g yo gulp
```

3. Generate the starting theme with the Theme Generator. Enter *Lunar Resort Theme* for the name and *lunar-resort* for the ID, and answer no for the Font Awesome prompt. This theme uses Clay icons instead:

```
yo liferay-theme
```

4. To develop the theme you must copy the default files from the theme's build and modify them. The `/src/css/` folder and `_custom.scss` file are included by default. Run the command below from the theme's root folder to build the files:

```
gulp build
```

```
? What would you like to call your theme? Lunar Resort Theme
? Would you like to use this as the themeId? lunar-resort-theme
? Which version of Liferay is this theme for? 7.2
? Would you like to add Font Awesome to your theme? No
   create package.json
   create .gitignore
   create gulpfile.js
   create src\WEB-INF\liferay-look-and-feel.xml
   create src\WEB-INF\liferay-plugin-package.properties
   create src\css\_custom.scss
```

Figure 145.1: Answer no for the Font Awesome Prompt

5. Create a new `/src/templates/` folder and copy `portal_normal.ftl` from the `build/templates/` folder into it.
6. Configure the theme to extend the Atlas theme. Add a `clay.scss` file to the theme's `/src/css/` folder and add the import shown below:

```
@import "clay/atlas";
```

7. Create an `_imports.scss` file in the `/src/css/` folder and add the imports shown below to it. This includes the default imports and replaces the `clay/base-variables` with the Atlas base variables:

```
@import "bourbon";

@import "mixins";

@import "compat/mixins";

@import "clay/atlas-variables";
```

You've generated the theme, prepared it for development, and configured it to extend the Atlas theme. Continue to the next section to build the Lunar Resort's Header and customize the logo.

CHAPTER 146

CUSTOMIZING THE LUNAR RESORT'S HEADER AND LOGO

The Header contains the navigation and logo for the site. In this section you'll customize the look and feel of the Header and add a custom logo.

Follow these steps:

1. Open `portal_normal.ftl` and replace the `<header>...</header>` element and contents with the updated code snippet below. This updates the structure slightly, making the banner expand the full width of the Header, and adds a new `header_css_class` variable to the `class` attribute. This variable is defined in a later step.

```
<header class="${header_css_class}">
    <div class="container-fluid" id="banner" role="banner">
        <a class="${logo_css_class}" href="${site_default_url}" title="<@liferay.language_format arguments="${site_name}" key="go-
to-x" />">
            <img alt="${logo_description}" height="${site_logo_height}" src="${site_logo}" width="${site_logo_width}" />
            <#if show_site_name>
                ${site_name}
            </#if>
        </a>

        <#if has_navigation>
            <#include "${full_templates_path}/navigation.ftl" />
        </#if>
    </div>
</header>
```

2. Replace the `<div class="container-fluid" id="wrapper">` element with the updated code below to remove some margins and padding:

```
<div class="container-fluid mt-0 pt-0 px-0" id="wrapper">
```

   And move the wrapper down, and place it directly above the `<section id="content">` element:

```
<div class="container-fluid mt-0 pt-0 px-0" id="wrapper">
  <section id="content">
  ...
```

```
  </section>
  <footer...>
  ...
  </footer>
</div>
```

3. The logo's height is retrieved with the `${site_logo_height}` variable. The height of the logo is a bit too large for the Lunar Resort theme, so you must adjust it. Remove the `width` attribute from the logo's image so it defaults to `auto`:

```
<img alt="${logo_description}" height="${site_logo_height}" src="${site_logo}" />
```

4. Create `init_custom.ftl` in your theme's `/src/templates/` folder and assign the logo's `site_logo_height` variable to the value below:

```
<#assign site_logo_height = 56 />
```

5. Assign the new `header_css_class` variable you added in step one to the value below:

```
<
#assign header_css_class =
"navbar navbar-expand-md navbar-dark flex-column flex-md-row bd-navbar"
/>
```

   This applies Bootstrap and Clay utility classes to provide the overall look and feel of the Header. Assigning the classes to a variable keeps `portal_normal` clean and makes the code easy to maintain. If you want to update the classes, you just have to modify the variable (e.g. `header_css_class = header_css_class + " my-new-class"`).

6. Add the code snippet below to update the `logo_css_class` variable to use Bootstrap's `navbar-brand` class:

```
<#assign logo_css_class = logo_css_class + " navbar-brand" />
```

7. Before you upload the theme to see what it looks like so far, you must create a theme thumbnail so you can identify it. To save time, copy the `thumbnail.png` asset from the [lunar-resort-build/assets](./images/](https://github.com/liferay/liferay-docs/tree/master/en/developer/tutorials/code/lunar-resort-theme/lunar-resort-build/assets folder to a new /src](./images/ folder. Note that its dimensions are 480px by 270px. These dimensions are required to display the theme thumbnail properly.

8. The theme isn't complete yet, but you'll deploy what you have so you can replace the default logo with the Lunar Resort logo. Enable Developer Mode before deploying your theme, so the theme's files are not cached for future deployments. Start the server, if it's not already started, and deploy the theme with the command below:

```
gulp deploy
```

9. Before you configure the pages, you must import the Lunar Resort's pages. Open the Control Menu and navigate to *Publishing → Import*. Click the Plus button to create a new import process. Click *Select File* and import the `lunar_resort_pages.lar` from the `lunar-resort-build/assets/` folder. Keep the default settings and click *Import*.

10. Open the Control Menu and navigate to *Site Builder → Pages*. Click the Gear icon next to *Public Pages* to open the configuration menu. Under the *Look and Feel* tab, scroll down and click the *Change Current Theme* button and select the Lunar Resort Theme. Scroll to the Logo heading, click the *Change* button, upload the `lunar-resort-logo.png` asset from the `[lunar-resort-build/assets](./images/](https://github.com/liferay/liferay-docs/tree/master/en/developer/tutorials/code/lunar-resort-theme/lunar-resort-build/assets` folder, and click the *Save* button to apply the theme and logo.

Great! You've customized the Lunar Resort's Header and applied a custom logo. Next, you'll configure and customize the theme's navigation.

CHAPTER 147

CUSTOMIZING THE NAVIGATION

Navigation items (pages) are defined and configured in Liferay DXP. The Navigation template iterates through the existing navigation items (pages) and assigns the template's markup for each of them. Page updates therefore require no updates to the theme directly and can be made by a Site Administrator, thus reducing the maintenance costs.

To customize the navigation, you can either use the default navigation provided in `navigation.ftl` and customize the markup template, or you can embed the navigation portlet in the theme and customize its preferences. Both approaches use the same overall markup. This section takes the former approach and customizes the default configuration in `navigation.ftl`. in the next section, you'll embed the navigation portlet in the Footer and configure its preferences to only display the top level (parent) navigation items.

Follow these steps to configure the Header's navigation:

1. Copy the default `navigation.ftl` file from the `/src/build/templates/` folder into the theme's `/src/templates/` folder. The `build` folder was generated when you built the theme and again when you initially deployed the theme in the last section.

2. By default, the User Personal Bar is hidden from the theme. You can either enable this via System Settings outside the scope of the theme, or you can include it in your theme. In this case, you'll include it in the theme. Open the `navigation.ftl` template you just copied and add this User Personal Bar markup to the top:

   ```
   <div class="mx-1 mx-sm-3 order-md-1 lunar-user">
       <@liferay.user_personal_bar />
   </div>
   ```

   along with some utility classes to position and order the User Personal Bar, this also adds a custom `lunar-user` class, which you'll use later for styling.

3. Modify the default template to use Bootstrap's `navbar` format. Wrap the `<nav>...</nav>` element with the `<div>` shown below:

   ```
   <div class="collapse navbar-collapse" id="lunarNav">
     <nav ... >
     </nav>
   </div>
   ```

4. Open the `portal_normal.ftl` template and find this conditional wrapper:

```
<#if has_navigation>
  <#include "${full_templates_path}/navigation.ftl" />
</#if>
```

   Update the conditional to include the menu toggler for the mobile navigation. This targets the `#lunarNav` wrapper that you added in the previous step:

```
<#if has_navigation>
  <button
    aria-controls="navigation"
    aria-expanded="false"
    class="btn-monospaced ml-auto navbar-toggler"
    data-target="#lunarNav"
    data-toggle="collapse"
    type="button">
    <span class="navbar-toggler-icon"></span>
  </button>
  <#include "${full_templates_path}/navigation.ftl" />
</#if>
```

5. Open `navigation.ftl` and add the `navbar-nav` and `mr-auto` classes to the `<ul>` element at the top:

```
<ul aria-label="<@liferay.language key="site-pages" />" class="navbar-nav mr-auto" role="menubar">
```

6. Open `navigation.ftl` and replace the first `<#assign... />` declaration with the one below. This adds the `nav-item` class to the `nav_item_css_class` variable declaration in `navigation.ftl` and declares a new `nav_item_caret` variable:

```
<#assign
  nav_item_attr_has_popup = ""
  nav_item_css_class = "nav-item"
  nav_item_layout = nav_item.getLayout()
  nav_item_caret = ""
/>
```

7. Replace the `nav_item.isSelected` conditional block with the one shown below. This adds the `selected` class to the existing `nav_item_css_class` classes:

```
<#if nav_item.isSelected()>
  <#assign
    nav_item_attr_has_popup = "aria-haspopup='true'"
    nav_item_css_class = "${nav_item_css_class} selected"
  />
</#if>
```

8. The Lunar Resort contains nested pages (child navigation items). By default, child navigation items are displayed at the block level. Instead, the Administrator wants to display these items in a dropdown list that is only displayed on hover of the parent navigation item. Add this conditional block directly below the `nav_item.isSelected` block you just modified. This adds the `dropdown` class to the parent navigation item and updates the `nav_item_caret` variable to hold Clay caret icon markup to indicate the parent navigation has nested child items:

```
<#if nav_item.hasChildren()>
  <#assign
    nav_item_css_class = "${nav_item_css_class} dropdown"
    nav_item_caret = '<svg class="lexicon-icon">
    <use xlink:href="${images_folder}/lexicon/icons.svg#caret-bottom" />
    </svg>'
  />
</#if>
```

9. Locate the anchor's markup below:

```
<a aria-labelledby="layout_${nav_item.getLayoutId()}"
${nav_item_attr_has_popup}
href="${nav_item.getURL()}"
${nav_item.getTarget()}
role="menuitem"
>
  <span>
    <@liferay_theme["layout-icon"] layout=nav_item_layout />
    ${nav_item.getName()}
  </span>
</a>
```

   Replace it with the updated markup shown below to include the `${nav_item_caret}` variable:

```
<a
  aria-labelledby="layout_${nav_item.getLayoutId()}"
  class="nav-link" ${nav_item_attr_has_popup}
  href="${nav_item.getURL()}"
  ${nav_item.getTarget()}
  role="menuitem"
>
  <span>
    <@liferay_theme["layout-icon"] layout=nav_item_layout />
    ${nav_item.getName()}
  </span>
  ${nav_item_caret}
</a>
```

10. Add the `dropdown-menu` class to the `<ul class="child-menu" role="menu">` element and replace the `nav_child_css_class` variable declarations with the ones below to add the `nav-item` class to them:

```
<#assign
  nav_child_css_class = "nav-item"
/>

<#if nav_item.isSelected()>
  <#assign
    nav_child_css_class = "nav-item selected"
  />
</#if>
```

11. Find the `<a>` element with the `aria-labelledby="layout_${nav_child.getLayoutId()}"` attribute and add the `class="nav-link"` attribute to it:

```markup
<a aria-labelledby="layout_${nav_child.getLayoutId()}" class="nav-link"...></a>
```

12. Add a call to action for the visitors to the Lunar Resort site so they can book their flight. Add the book now button's code below the closing </nav> element. This uses some utility classes for the basic look and feel and ordering, as well as a custom `btn-orange` class that you'll provide styling for later:

```html
<a aria-controls="book-now" class="btn text-white btn-orange order-md-2">
    <p class="book-now-text mb-0">Book Now</p>
</a>
```

13. The Lunar Resort's color scheme is comprised of three colors: orange, white, and blue. Since these colors are used throughout the theme, you'll store them in SASS variables in a separate file. Create a new file called `_colors.scss` inside the theme's `/src/css/` folder and add these variables to it. Note that White is already defined as the global variable `$white` by the Atlas theme.

```
$lunar-resort-orange: #dfa356;
$lunar-resort-blue: #415fa7;
$lunar-resort-link-teal: #00ccFF;
```

14. Now that the main colors are defined, open `/src/css/_custom.scss` and add the code snippet below. This imports the `_colors.scss` file so you can use the variables you just created. It adds some basic styling for the Header and navigation, including a style to highlight the page that is currently active via the `selected` class. It also displays the child menu items at the block level on smaller devices with the `@include media-breakpoint-down` breakpoint:

```
@import 'colors';

body {

  a.btn-orange {
    background-color: $lunar-resort-orange;
    margin-right: 5px;

    &:hover {
      border-color: $white;
    }

    @include media-breakpoint-down(sm){
      width: 100%;
    }
  }

  header {
    background-color: $lunar-resort-blue;

    .lunar-user a {
        color: $lunar-resort-link-teal;
    }

    .user-avatar-link .lexicon-icon {
      color: $lunar-resort-blue;
    }

    li.nav-item {
      & a.nav-link span {
        font-size: 1.5em;
      }
```

```
      &:hover ul.child-menu {
        background-color: $lunar-resort-blue;
        display: block;
        margin-top: -10px;
      }

      &.selected {
        background-color: $white;
        height: 73px;
        & a.nav-link {
          color: $lunar-resort-blue;
          font-weight: bold;
          &:hover {
            color: $lunar-resort-blue;
            font-weight: normal;
            padding-left: 9.619px;
            padding-right: 9.619px;
          }
        }
      }

      @include media-breakpoint-down(sm){
        ul.child-menu {
          display: block;
        }
      }
    }
  }
}
```

15. The Control Menu is displayed on top of everything when the user is signed in, which covers the Header. You must update the `navigation.ftl` template to account for the Control Menu. Liferay DXP provides a unique class that is added to the `body` of the page when each product navigation (which includes the Control Menu) is visible. Use the `has-control-menu` class is added to the body when the Control Menu is visible. Open `_custom.scss` and add this code snippet just above the closing bracket for the `body` to add a top margin to the Header that's equal to the height of the Control Menu:

```
&.has-control-menu {
  header {
    margin-top: 56px;
  }
}
```

16. The Control Menu's height is slightly smaller on mobile devices, so you must account for that responsiveness in your styling. Update the code snippet you just added to match the one below:

```
&.has-control-menu {
  header {
    margin-top: 56px;
    @include media-breakpoint-down(sm){
      margin-top: 48px;
    }
  }
}
```

Great! The Header's navigation is customized. The updated Header and logo should look like the figure below:

Next, you'll define the Footer and embed a navigation portlet to display navigation.

Figure 147.1: The updated Header and navigation are much more user-friendly now.

CHAPTER 148

DEFINING THE LUNAR RESORT'S FOOTER AND FOOTER NAVIGATION

You've configured the Header and its navigation, but at the moment the Footer is a bit bare bones. In this section, you'll update the Footer to include contact information for the Lunar Resort and include navigation with an embedded navigation portlet.

Follow these steps:

1. To keep the Portal Normal template uncluttered, create a separate template to hold the Footer's markup. Create a new file called `footer.ftl` in the theme's `/src/templates/` folder.

2. Copy the Footer markup (shown below) from `portal_normal.ftl` into `footer.ftl`:

```
<footer id="footer" role="contentinfo">
    <p class="powered-by">
        <@liferay.language key="powered-by" /> <a href="http://www.liferay.com" rel="external">Liferay</a>
    </p>
</footer>
```

   And update the <p> element in `footer.ftl` to include the classes shown below:

```
<footer id="footer" role="contentinfo">
    <p class="powered-by text-center text-white py-3 mb-0">
        <@liferay.language key="powered-by" /> <a href="http://www.liferay.com" rel="external">Liferay</a>
    </p>
</footer>
```

3. Add this `@liferay.navigation_menu` macro snippet above the `powered-by` paragraph to embed the navigation portlet. This configuration stores the portlet preferences in a `preferencesMap` variable. The `displayDepth` of `1` specifies that the portlet must only render the top-level parent navigation, and `portletSetupPortletDecoratorId` sets the portlet decorator to `barebone`, which removes the portlet's wrapper and only renders the portlet's content:

```
<nav id="navbarFooter">
    <div class="text-center mx-auto">
        <div class="nav text-uppercase" role="menubar">
            <#assign preferencesMap = {"displayDepth": "1", "portletSetupPortletDecoratorId": "barebone"} />
```

```
            <@liferay.navigation_menu
                default_preferences=freeMarkerPortletPreferences.getPreferences(preferencesMap)
                instance_id="footer_navigation_menu"
            />
        </div>
    </div>
</nav>
```

4. The visitors need some social media links so they can keep tabs on the latest and greatest news from the Lunar Resort. Replace the snippet you just added with the one below. This uses Clay icons and adds a wrapper to prepare for the next step.

```
<div id="navbarContactWrapper" class="row mx-0">
  <nav id="navbarFooter" class="col-12 col-md-6 pt-5">
    <div id="socialMediaWrapper" class="col-12 col-md-4 text-center mx-auto mb-4">
      <h2 class="nav-heading">
          Follow Us
      </h2>
        <div id="socialMediaLinks">
          <ul class="nav flex-row mx-auto">
              <li class="mx-2">
                  <div id="facebook"><a class="text-white"
                  href="http://www.facebook.com/pages/Liferay/45119213107"
                  target="_blank"><span class="hide">Facebook</span>
                  <@clay["icon"] symbol="social-facebook" />
                  </a></div>
              </li>
              <li class="mx-2">
                  <div id="twitter"><a class="text-white"
                  href="http://www.twitter.com/liferay"
                  target="_blank"><span class="hide">Twitter</span>
                  <@clay["icon"] symbol="twitter" />
                  </a></div>
              </li>
              <li class="mx-2">
                  <div id="linked-in"><a class="text-white"
                  href="http://www.linkedin.com/company/83609"
                  target="_blank"><span class="hide">LinkedIn</span>
                  <@clay["icon"] symbol="social-linkedin" />
                  </a></div>
              </li>
              <li class="mx-2">
                  <div id="youtube"><a class="text-white"
                  href="http://www.youtube.com/user/liferayinc"
                  target="_blank"><span class="hide">YouTube</span>
                  <@clay["icon"] symbol="video" />
                  </a></div>
              </li>
          </ul>
        </div>
    </div>
    <div class="text-center mx-auto">
      <div class="nav text-uppercase" role="menubar">
        <#assign preferencesMap = {"displayDepth": "1", "portletSetupPortletDecoratorId": "barebone"} />

        <@liferay.navigation_menu
          default_preferences=freeMarkerPortletPreferences.getPreferences(preferencesMap)
          instance_id="footer_navigation_menu"
        />
      </div>
    </div>
  </nav>
</div>
```

5. Add this snippet below the closing </nav> tag to add the Lunar Resort's contact information. Also, copy the lunar-resort-logo-vertical.png asset from the [lunar-resort-build/assets](./images/](https://github.com/liferay/liferay-docs/tree/master/en/developer/tutorials/code/lun resort-theme/lunar-resort-build/assets folder to the /src](./images/ folder so you can use it in the Footer:

```
<div class="contact-info-container text-center pt-5 pb-2 col-12 col-md-4 mx-auto mb-4">
  <img alt="lunar-resort-logo" height="90" class="mb-2" src="${images_folder}/lunar-resort-logo-vertical.png" />
  <div id="contactTextWrapper" class="row mx-0">
    <p class="col-12 col-md-6">
      123 Mare Nectaris Lane<br>
      Mare Nectaris, Moon Colony 10010<br>
    </p>
    <p class="col-12 col-md-6">
      Tel: 4-919-843-6666<br>
      Fax: 4-919-843-6667<br>
      <a href="mailto:info@lunarresort.com">info@thelunarresort.com</a>
    </p>
  </div>
</div>
```

6. The Administrator doesn't want to display the Footer on every page, so she would like the option to hide it. To do that, create a theme setting to optionally show the Footer. Open your theme's /src/WEB-INF/liferay-look-and-feel.xml file and add this snippet just below the <template-extension>ftl</template-extension> entry. This renders a togglable *Show Footer* option in the *Look and Feel* section for the theme's configuration.

```
<settings>
  <setting configurable="true" key="show-footer" type="checkbox" value="true" />
</settings>
```

7. Now you must define a FreeMarker variable to store the value of the show-footer theme setting so you can check for it in portal_normal.ftl. Open init_custom.ftl and add the variable declaration below to set the show_footer variable to the value (true or false) of the show-footer theme setting:

```
<#assign
  show_footer = getterUtil.getBoolean(themeDisplay.getThemeSetting("show-footer"))
/>
```

8. Open portal_normal.ftl and replace the Footer markup with the code snippet below to include the Footer template when the show-footer theme setting is true:

```
<#if show_footer>
    <#include "${full_templates_path}/footer.ftl" />
</#if>
```

9. Open _custom.scss and add this snippet above the &.has-control-menu styling to style the Footer:

```
#footer {
  background-color: $lunar-resort-blue;
  color: $white;
  ul {
    margin-left: auto;
```

```
    margin-right: auto;

    &.navbar-nav {
      width: 410px;
      .nav-item.hover:after {
        width: auto;
      }

      a {
        color: $white;
        @include media-breakpoint-down(sm) {
          padding-left: 6px;
          padding-right: 6px;
        }
      }
    }
  }

  #socialMediaWrapper ul {
    width: 192px;

    li a {
      font-size: 2rem;
    }
  }

  p.powered-by a, .contact-info-container a {
      color: $lunar-resort-link-teal;
  }

}
```

10. The majority of the Lunar Resort's content is provided with Fragments. Since Fragments are out of the scope of this tutorial, you'll upload the completed fragments. Open the Control Menu and navigate to *Site Builder* → *Page Fragments*, and select the *Import* option from the New dropdown menu. Import the `collections-lunar-resort.zip` asset from the `lunar-resort-build/assets/` folder.

11. Re-deploy the updated theme with the command below:

```
gulp deploy
```

The updated Footer and navigation should look like the figure below:

Figure 148.1: The updated Footer provides everything visitors need to follow and contact the Lunar Resort.

In the next section you'll learn how to create a color scheme for the Lunar Resort.

CHAPTER 149

ADDING A COLOR SCHEME VARIANT FOR THE LUNAR RESORT THEME

In this section, you'll create a color scheme variant for the Lunar Resort Theme to apply during the Lunar Eclipse, when special discounts are available. You'll create a color scheme that reflects the reds and yellows present during a lunar eclipse. Since the majority of the Lunar Resort Site's content is created with page fragments, you must account for the color scheme styles in the page fragments as well. Follow these steps:

1. Open the theme's `WEB-INF/liferay-look-and-feel.xml` file and add these color-scheme entries above the `<portlet-decorator>..</portlet-decorator>` ones:

```
<theme id="my-theme-id" name="My Theme Name">
  <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="Eclipse">
    <css-class>eclipse</css-class>
  </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.
```

---

2. Open `/src/css/_colors.scss` and update the colors to include the two new ones (eclipse yellow and eclipse red) for the color scheme:

```
$lunar-resort-orange: #dfa356;
$lunar-resort-blue: #415fa7;
$lunar-resort-link-teal: #00ccFF;
$lunar-resort-eclipse-yellow: #dfd456;
$lunar-resort-eclipse-red: #a75441;
```

3. Create a /src/css/color_schemes/ folder for the color scheme, and add a eclipse.scss file to it for the Eclipse color scheme. The default color scheme's styles are included in _custom.scss, so you don't need to create anything for them.

4. The color scheme's class is added to the <body> element when the theme's color scheme is applied, so you must prefix the body styles with the eclipse class to target the proper color scheme. Open /src/css/color_schemes/eclipse.scss and add this import and styles to it to use the new colors you defined:

```
@import '../colors';

body.eclipse {

  a.btn-orange {
    background-color: $lunar-resort-eclipse-yellow;
  }

  header {
    background-color: $lunar-resort-eclipse-red;

    .user-avatar-link .lexicon-icon {
      color: $lunar-resort-eclipse-red;
    }

    li.nav-item {

      ul.child-menu {
        background-color: $lunar-resort-eclipse-red;
      }

      &:hover ul.child-menu {
        background-color: $lunar-resort-eclipse-red;
      }

      &.selected {
        & a.nav-link {
          color: $lunar-resort-eclipse-red;
          &:hover {
            color: $lunar-resort-eclipse-red;
          }
        }
      }
    }
  }

  #footer {
    background-color: $lunar-resort-eclipse-red;
  }

}
```

5. Import the eclipse color scheme's CSS file into _custom.scss so it's loaded with the rest of the custom styles:

```
@import "color_schemes/eclipse";
```

6. You must create thumbnails for each color scheme, just like you did the theme. To save time, copy the [lunar-resort-build/assets](./images/color_schemes/](https://github.com/liferay/liferay-docs/tree/master/en/developer/tutorials/code/lunar-resort-theme/lunar-resort-build/assets folder to the theme's /src](./images/ folder. Note that the color scheme folder names match the color scheme CSS class names defined in `liferay-look-and-feel.xml`.

7. Now that the color scheme is created, you must update the page fragments to use the eclipse color scheme class so they have the same look as the color scheme when it's applied to the page. The fragments don't have access to the SASS color variables, so you must use the hexadecimal color codes. To save time, import the updated page fragments from the `lunar-resort-build/assets/` folder. Open the Control Menu and navigate to *Site Builder → Page Fragments*, click the Actions menu next to COLLECTIONS, and select the *Import* option. Import the `collections-lunar-resort-color-scheme.zip` asset from the `lunar-resort-build/assets/` folder. Note that each fragment style that requires a color change is duplicated and prefixed with `body.eclipse`. A couple example configurations are shown below:

```
.fragment_35201 h3.text-center {
  background-color: #dfa356;
  color: #FFF;
}

body.eclipse .fragment_35201 h3.text-center {
  background-color: #dfd456;
}


.fragment_35201 a.btn {
  background-color: #415fa7;
}

body.eclipse .fragment_35201 a.btn {
  background-color: #a75441;
}
```

8. Deploy the theme. Open the Control Menu, navigate to *Site Builder → Pages*, and click the Gear icon next to *Public Pages*. Select the Eclipse color scheme under the *LOOK AND FEEL* tab and save to apply the changes.

   The theme should look like the figure below with the Eclipse color scheme applied:

Great! You've seen how you can quickly change the look and feel of the Lunar Resort with just a simple color scheme. Now you know how to develop a theme to customize the overall look and feel of your site!

See the Themes section for information on developing themes.

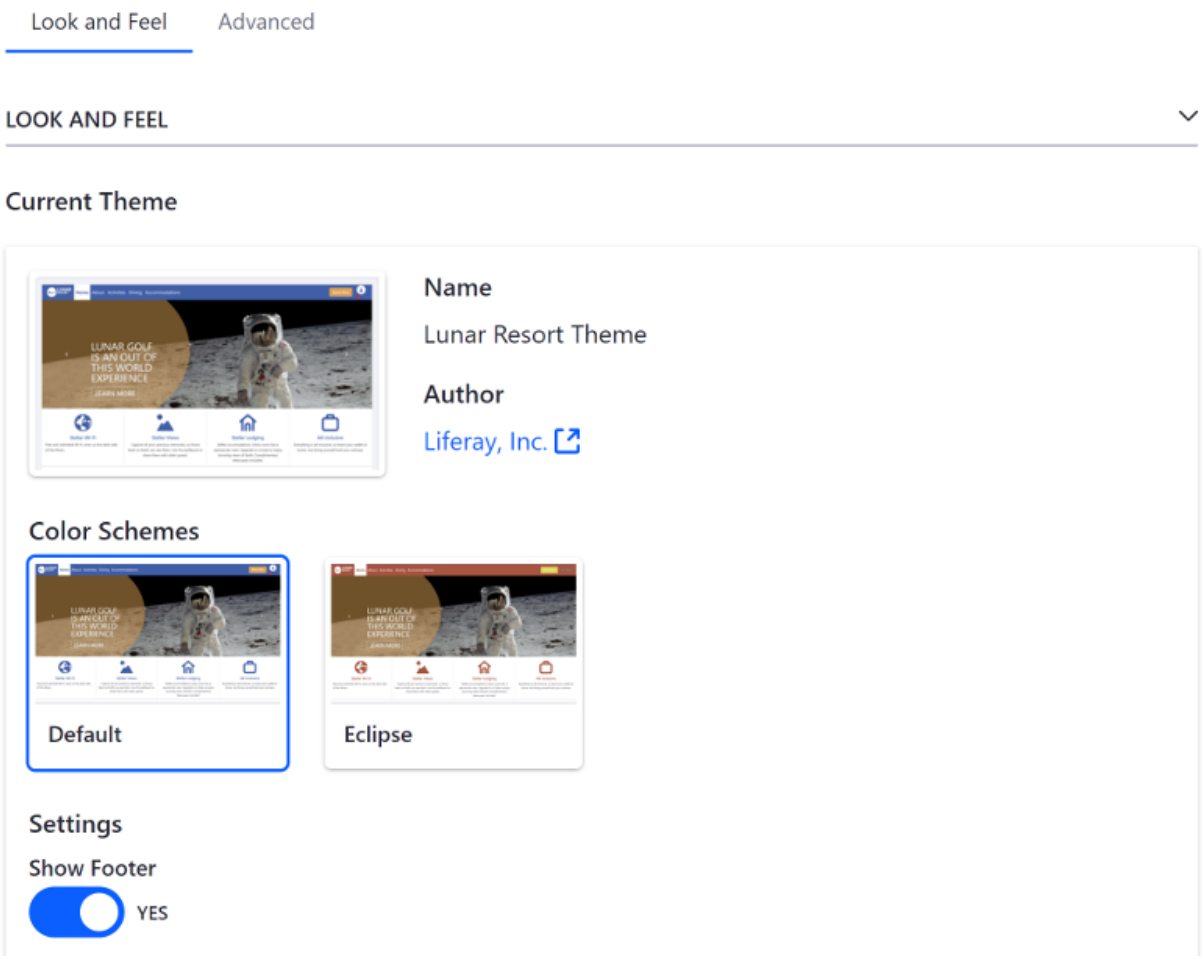

Figure 149.1: Color schemes are a good way to subtly change the look and feel of your site.

Figure 149.2: The finished color scheme gives the Lunar Resort site a fiery glow.

Part II

Customization

Chapter 150

Liferay Customization

Liferay DXP is highly customizable. Its modular architecture contains components you can extend and override dynamically. This section explains Liferay DXP's architecture and customization fundamentals and demonstrates overriding and extending Liferay DXP components and applications using APIs.

- Fundamentals include understanding and configuring dependencies, packaging, and deployment. Here you'll work with module JARs, plugin WARs, components, and Java packages in Liferay DXP.
- Architecture dives deep into how Liferay DXP uses modularity and OSGi to provide the core, application modules, component services, and extension points. Learning the architecture helps you develop better customizations fast, and it empowers you to build extension points into your own applications.
- Built-in customization features, including Widget Templates and Web Experience Management help you customize content and pages faster. All this is done from within the Liferay DXP UI.
- Application customization articles (listed after the Architecture articles) demonstrate modifying Liferay applications via their APIs and extension points.

Start with Fundamentals.

CHAPTER 151

FUNDAMENTALS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

The fundamentals of developing on Liferay DXP and customizing it are perhaps best learned in the context of projects. It's in projects that you configure access to Liferay DXP's API, extend and override Liferay DXP features, and package your software for deployment. Projects are developed as WARs or OSGi JARs, but are all installed to Liferay's OSGi framework as OSGi bundles. These bundles can depend on external Java packages, share Java packages, and be manipulated at run time via Apache Gogo Shell. The fundamentals are explained in the context of projects so that you understand them in a practical sense and can apply them right away. Here are the fundamental topics:

- **WARs Versus OSGi JAR** explains fundamental differences between the WAR and OSGi JAR structures and how they're deployed in Liferay DXP.
- **Configuring Dependencies** demonstrates how to identify and configure Liferay artifacts and third-party artifacts to use their Java packages in your projects.
- **Importing and Exporting Packages** shows how to import the packages your projects need and export packages your projects provide. Liferay's tooling detects package use and specifies package imports automatically.
- **Semantic Versioning** shows how Liferay DXP uses a standard for ascribing meaning to major, minor, and micro versions of modules and Java packages.
- **Deploying WARs (WAB Generator)** explains how Liferay's WAB Generator deploys WAR applications as OSGi Web Application Bundles (WABs).
- **Gogo Shell** enables you to examine components, debug issues, and manage deployments.

Start with understanding how WAR and OSGi JAR project structures are used in development.

CHAPTER 152

CONFIGURING DEPENDENCIES

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Liferay DXP's modular environment lets modules provide and consume capabilities via Java packages. To leverage packages from other modules or traditional libraries in your project, you must configure them as dependencies. Here you'll learn how to find artifacts (modules or libraries) and configure dependencies on them.

- Finding Artifacts explains how to use the Application Manager, Gogo Shell, and Liferay DXP reference documentation to find artifacts deployed on Liferay DXP and available in repositories.
- Specifying Dependencies demonstrates specifying artifacts to Maven and Gradle build frameworks. It shows you how to determine whether Liferay DXP already exports packages from an artifact and how to configure such artifacts as compile-time dependencies.
- Resolving Third-Party Library Package Dependencies provides a workflow for using packages that are only available in traditional library JARs (JARs that aren't OSGi modules). It involves minimizing transitive dependencies so you can resolve dependencies quicker and prevent bloating your project with unnecessary JARs.

Your first step is to find the artifacts you need.

CHAPTER 153

FINDING ARTIFACTS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 or Maven). 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

Here you'll learn how to find artifact attributes to specify artifact dependencies.

153.1 Finding Core Artifact Attributes

Each Liferay artifact is a JAR file whose `META-INF/MANIFEST.MF` file specifies OSGi bundle metadata 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 each core Liferay DXP artifact's group ID and artifact ID and where to find the artifact's manifest, which lists the artifact version:

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
util-bridges.jar	com.liferay.portal	com.liferay.util.bridges	(see JAR's MANIFEST.MF)	fix pack ZIP or Liferay DXP .war
util-java.jar	com.liferay.portal	com.liferay.util.java	(see JAR's MANIFEST.MF)	fix pack ZIP or Liferay DXP .war
util-slf4j.jar	com.liferay.portal	com.liferay.util.slf4j	(see JAR's MANIFEST.MF)	fix pack ZIP or Liferay DXP .war
util-taglibs.jar	com.liferay.portal	com.liferay.util.taglib	(see JAR's MANIFEST.MF)	fix pack ZIP or Liferay DXP .war
com.liferay.* JAR files	com.liferay	(see JAR's MANIFEST.MF)	(see JAR's MANIFEST.MF)	fix pack ZIP, Liferay DXP installation, Liferay DXP dependencies ZIP, or the OSGi ZIP

Next, you'll learn how to find Liferay DXP app and independent module artifact attributes.

153.2 Finding Liferay App and Independent Artifacts

Independent modules and Liferay DXP app modules aren't part of the Liferay DXP core. You must still, however, find their artifact attributes if you depend 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.

153.3 App Manager

The App Manager knows what's deployed on your Liferay instance. Use it to find deployed module attributes.

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 ID, use the Felix Gogo Shell to find it:

1. Navigate to the Gogo Shell portlet in the Control Panel → *Configuration → Gogo Shell*. Enter commands in the Felix Gogo Shell command prompt.
2. Search for the module by its display name (e.g., `Liferay Blogs API`) or a keyword. In the results, note the module's number. You can use it in the next step. For example, Gogo command results in the figure below show the Liferay Blogs API module number.
3. List the module's manifest headers by passing 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:
4. On Maven Central or MVNRepository, search for the module by its artifact ID.
5. Determine the group ID by matching the `Bundle-Vendor` value from step 3 with a group listed that provides the artifact.

Next, Liferay DXP's reference documentation provides Liferay DXP app artifact attributes.

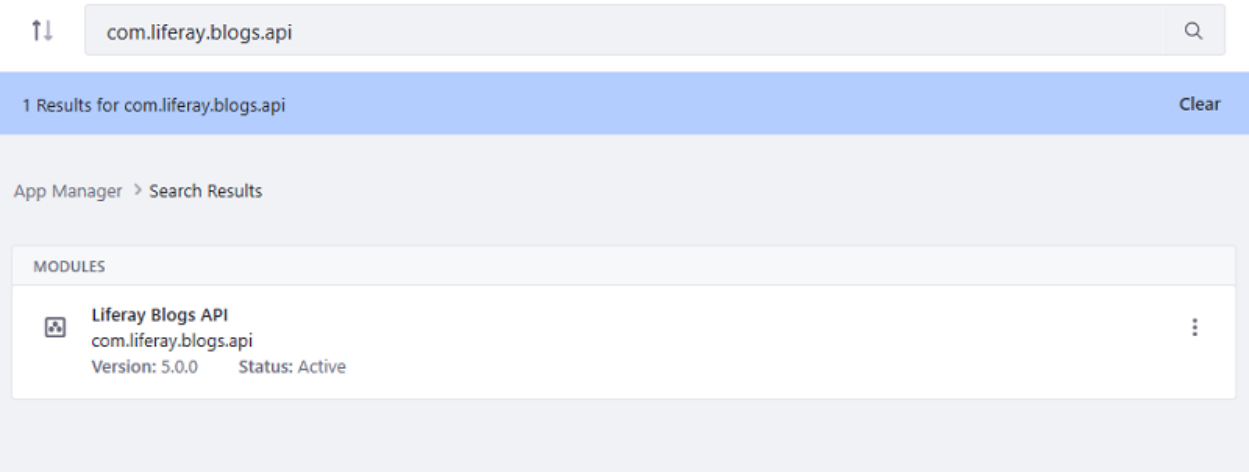

Figure 153.1: You can inspect deployed module artifact IDs and version numbers.

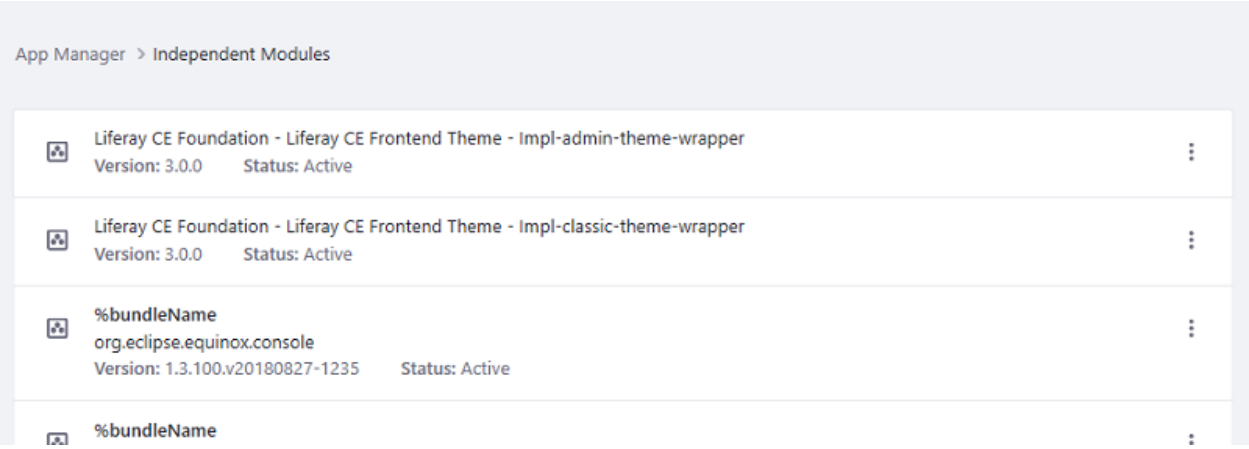

Figure 153.2: The App Manager aggregates Liferay and independent modules.

153.4 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 Liferay DXP Artifact Attributes*.

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](

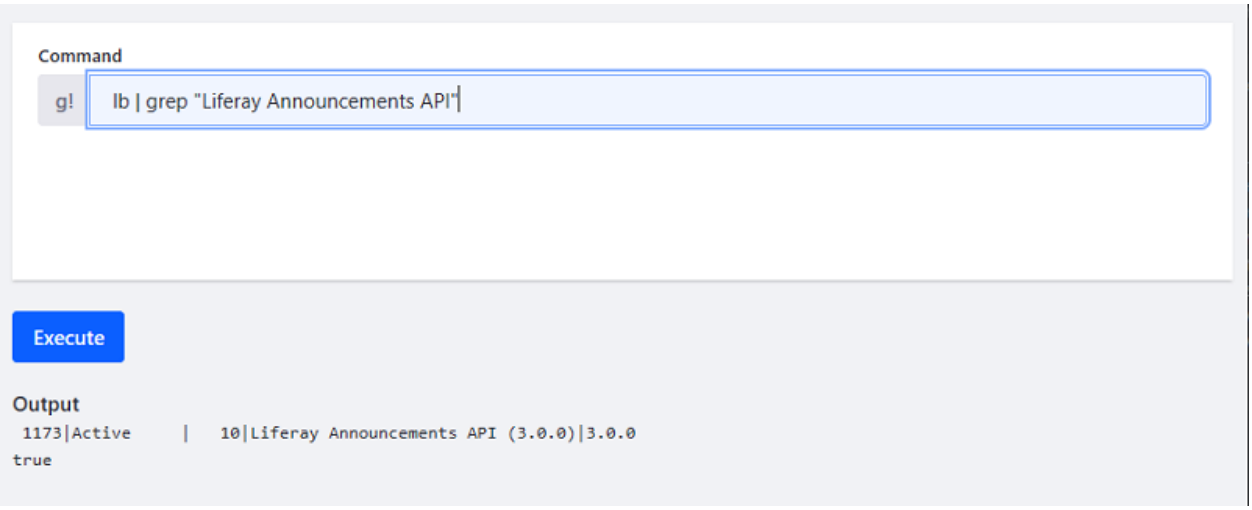

Figure 153.3: Results from this Gogo command show that the module's number is 1173.

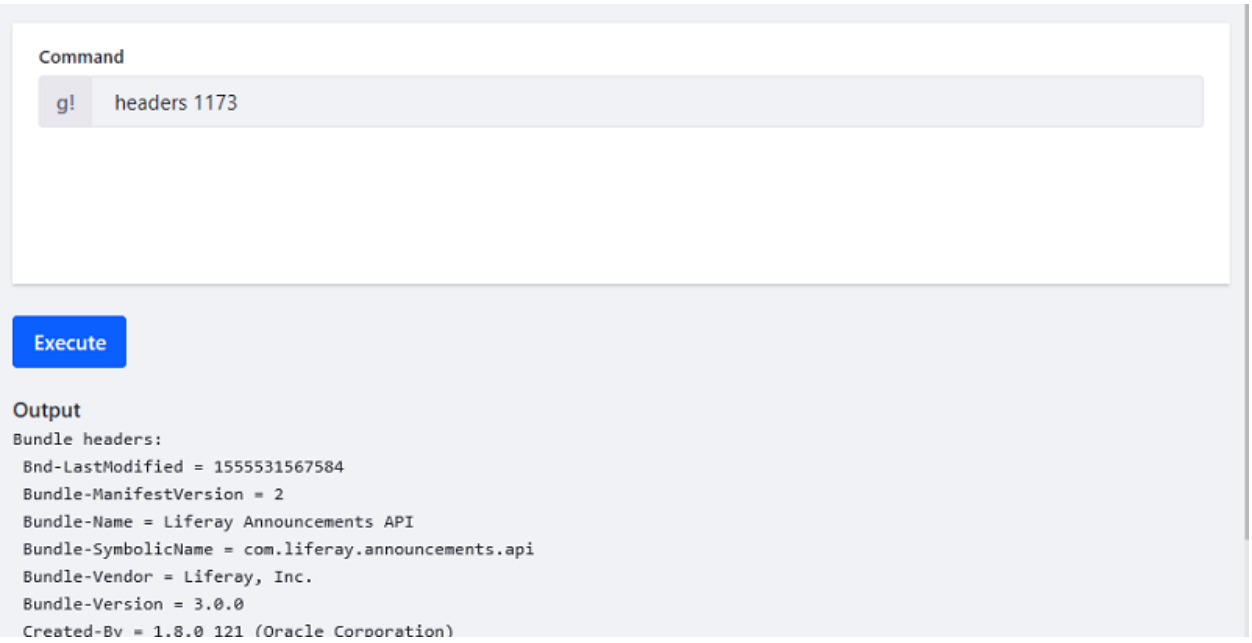

Figure 153.4: Results from running the `headers` command show the module's bundle vendor and bundle version.

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`.

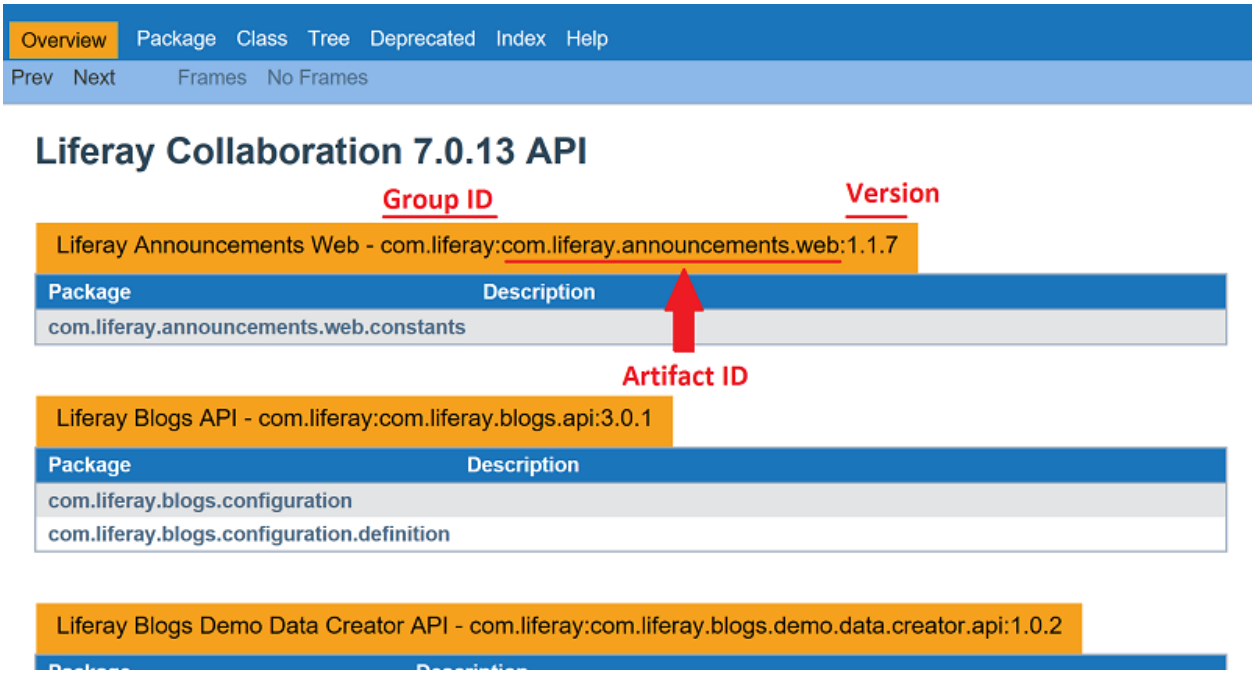

Figure 153.5: Liferay DXP app Javadoc overviews list each app module's display name, followed by its group ID, artifact ID, and version number in a colon-separated string. It's a Gradle artifact syntax.

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

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

153.5 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 know the artifact's attributes, you can configure a dependency on it.

153.6 Related Topics

Specifying Dependencies
- Importing Packages
- Exporting Packages
- Resolving Third Party Library Package Dependencies
- Deploying WARs (WAB Generator)

CHAPTER 154

SPECIFYING DEPENDENCIES

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Compiling your project and deploying it to Liferay DXP requires satisfying its dependencies on external artifacts. After finding the attributes of an artifact, set a dependency for it in your build file. Here's how:

1. Determine whether Liferay DXP provides the Java packages you use from the artifact. These files list the packages Liferay DXP 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.

2. If Liferay DXP exports all the packages you use from the artifact, specify the artifact as a compile-only dependency. This prevents your build framework from bundling the artifact with your project. Here's how to make the dependency compile-only:

    Gradle: Add the `compileOnly` directive to the dependency

    Maven: Add the `<scope>provided</scope>` element to the dependency.

3. Add a dependency entry for the artifact. Here's the artifact terminology for the Gradle and Maven build frameworks:

Artifact Terminology

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

Here is an example dependency on Liferay's Journal API module for Gradle, and Maven:
Gradle (`build.gradle` entry):

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

Maven (`pom.xml` entry):

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

Important: Liferay DXP exports many third-party packages. Deploy your module to check if Liferay DXP or another module in your Liferay instance's OSGi runtime framework provides the package you need. If it's provided already, specify the corresponding dependency 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/7-2/customization/-/knowledge_base/c/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! You know how to specify artifact dependencies. Now that's a skill you can depend on!

154.1 Related Topics

Finding Artifacts
Importing Packages
Exporting Packages
Resolving Third Party Library Package Dependencies
Deploying WARs (WAB Generator)

CHAPTER 155

Resolving Third Party Library Package Dependencies

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Liferay's 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 project needs from non-OSGi third party libraries?

Here is the main workflow for resolving third party Java library packages:

Option 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 downloads (select a build)
- 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 compile-only dependency for it in your project. When you deploy your project, the OSGi framework wires the dependency module to your project's module or web application bundle (WAB). If you don't find an OSGi module based on the Java library, follow Option 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.

Option 2 - Resolve the Java packages privately in your project: Copy *required packages* only from libraries into your project, if you can or embed *libraries* wholesale, if you must. The rest of this article shows you how to do these things.

Note: Features for manipulating library packages are only available to module projects that use bnd and the `com.liferay.plugin` plugin, such as Liferay Workspace modules. WAR projects must embed libraries wholesale into their classpath.

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 and update the version of the `com.liferay.gradle.plugins` artifact to the latest version found in the repository:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins", version: "4.0.4"
    }

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

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.

155.1 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 for module projects that minimizes dependencies and Java package imports:

1. Add the library as a compile-only dependency (e.g., `compileOnly` in Gradle, `<scope>provided</scope>` in Maven).

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 project. 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 project.

155.2 Embedding Libraries in a Project

You can use Gradle or Maven to embed libraries in your project. Below are examples for adding Apache Shiro using both build utilities.

155.3 Embedding Libraries 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.

155.4 Embedding a Library Using Maven

Follow these steps:

1. Open your project's `pom.xml` file and add the library as a dependency in the `provided` scope:

```
<dependency>
  <groupId>org.apache.shiro</groupId>
  <artifactId>shiro-core</artifactId>
  <version>1.1.0</version>
  <scope>provided</scope>
</dependency>
```

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 class path. The `lib:=true` directive adds the embedded JAR to the module's class path 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.

155.5 Related Topics

Importing Packages
Exporting Packages
Creating a Project

CHAPTER 156

UNDERSTANDING EXCLUDED JARS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Portal property `module.framework.web.generator.excluded.paths` declares JARs that are stripped from all Liferay DXP generated WABs. These JARs are excluded from web application bundles (WABs) because Liferay DXP provides them already. All JARs listed for this property are excluded from a WAB, even if the WAB lists the JAR in a `portal-dependency-jars` property in its `liferay-plugin-package.properties` file.

If your WAR 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` module 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 WAR, 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 (the bundled WAR).

156.1 Related Topics

Configuring Dependencies

Deploying WARs (WAB Generator)

CHAPTER 157

USING THE FELIX GOGO SHELL

This document has been updated and ported to Liferay Learn and is no longer maintained here.

The Gogo shell provides a way to interact with Liferay DXP's module framework. You can

- dynamically install/uninstall bundles
- examine package dependencies
- examine extension points
- list service references
- etc.

There are two ways you can access the Gogo shell.

The recommended way to access the Gogo shell for a production environment is through the Control Panel. Accessing it there is the most secure way to use the Gogo shell. You can set permissions in your Liferay DXP instance to only give certain people access to it. The Gogo shell is extremely powerful and should only be given to trusted admins, as you can manipulate the platform's core functionality. You can access the Gogo shell in the Control Panel by navigating to *Configuration → Gogo Shell*.

You can also interact with Liferay DXP's module framework via a local telnet session. This is only recommended when you're developing your Liferay DXP instance. This is not recommended for production environments.

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. You must also have Developer Mode 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

`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 158

IMPORTING PACKAGES

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Plugins often must use Java classes from packages outside of themselves. Another OSGi bundle (a module or an OSGi Web Application Bundle) in the OSGi framework must export a package for your plugin to import it.

When an OSGi bundle (bundle) is set up to import packages, the OSGi framework finds other registered bundles that export the needed packages and wires them to the importing bundle. At run time, the importing bundle gets the class from the wired bundle that exports the class's package.

For this to happen, a bundle's `META-INF/MANIFEST.MF` file must specify the `Import-Package` OSGi manifest header with a comma-separated list of the Java packages it needs. For example, if a bundle 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 bundle uses and add them to the package imports in the bundle'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.

158.1 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 module 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 artifact dependencies. bnd examines your module's class path to determine which packages from those artifacts 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 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 its 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. Bundles that use these APIs must specify requirements on the API contracts. The contract requirement specifies your bundle's relationship with the imported API packages. If the system you're running does *not* provide the exact contract, your bundle 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 OSGi bundle 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.

158.2 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.

158.3 Related Topics

Configuring Dependencies
- Deploying WARs (WAB Generator)
- Project Templates
- Liferay's Maven Archetypes
- Blade CLI
- Liferay Dev Studio DXP

CHAPTER 159

EXPORTING PACKAGES

This document has been updated and ported to Liferay Learn and is no longer maintained here.

An OSGi bundle'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 bundle (bundle) manifest makes all the package's classes available for other bundles 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 JAR (uses bnd)	`[project]/bnd.bnd`
Module JAR (doesn't use bnd)	`[module JAR]/META-INF/MANIFEST.MF`
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 plugin WAR projects, you must add package exports to an `Export-Package` header in the project's `WEB-INF/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 OSGi bundle. 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 bundles that provide a different (compatible) version, or even the same version, of the package. A package from another bundle might provide better "wiring" opportunities with other bundles. 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 OSGi bundles!

159.1 Related Topics

Configuring Dependencies
- Deploying WARs (WAB Generator)
- Project Templates
- Liferay's Maven Archetypes
- Blade CLI
- Liferay Dev Studio DXP
- Semantic Versioning

CHAPTER 160

SEMANTIC VERSIONING

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 Liferay generated module projects) fails that project's build immediately.

The semantic version format looks like this:

```
MAJOR.MINOR.MICRO
```

Certain events force each tier to increment:

- *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 module project's versions is easy.

160.1 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.

160.2 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.

160.3 Related Topics

Importing Packages
Exporting Packages
Configuring Dependencies

CHAPTER 161

DEPLOYING WARS (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. Bean Portlets, PortletMVC4Spring Portlets, and JSF Portlets 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` OSGi 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
- Class loader code

161.1 WAR versus WAB Structure

The WAB folder structure and WAR folder structure differ. Consider the following folder structure of a WAR-style portlet.

WAR

- my-war-portlet
    - src
        * main
            - java
            - webapp
            - WEB-INF
            - classes
            - lib
            - resources
            - views
            - liferay-display.xml
            - liferay-plugin-package.properties
            - liferay-portlet.xml
            - portlet.xml
            - web.xml

When a WAR-style portlet is deployed to Liferay DXP and processed by the WAB Generator, the portlet's folder structure is transformed.

WAB

- my-war-portlet-that-is-now-a-wab
    - META-INF
        * MANIFEST.MF
    - WEB-INF
        * classes
        * lib
        * resources
        * views
        * 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 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.

Note: Adding a `bnd.bnd` file or a build-time plugin (e.g., `bnd-maven-plugin`) to your WAR plugin is pointless, because the generated WAB cannot use them.

161.2 Deploying a WAR

To deploy a WAB based on your WAR plugin, copy your WAR plugin to your Liferay DXP instance's `deploy/` folder in your `[Liferay Home]`.

161.3 Saving a Copy of the WAB

Optionally, save the WAB to a local folder. This gives you the opportunity to inspect the generated WAB. To store generated WABs, add the following portal properties to a `[Liferay Home]/portal-ext.properties` file. Then restart Liferay DXP:

```
module.framework.web.generator.generated.wabs.store=true
module.framework.web.generator.generated.wabs.store.dir=${module.framework.base.dir}/wabs
```

These properties instruct the WAB generator to store generated WABs in your Liferay instance's `osgi/wabs/` folder. The generated WABs have the same structure as the example WAB structure listed above. The Module Framework Web Application Bundles properties section explains more details.

Awesome! You have deployed your WAR plugin as a WAB and you know how to save a copy of the WAB to examine it!

161.4 Related Topics

Creating a Project
    Deploying a Project
    Developing Web Front-Ends

CHAPTER 162

ARCHITECTURE

Liferay DXP architecture comprises these parts:

Core: Bootstraps Liferay DXP and its frameworks. The Core provides a runtime environment for managing services, UI components, and customizations.

Services: Liferay and custom functionality is exposed via Java APIs and web APIs.

UI: The optional web application UI for adding portals, sites, pages, widgets, and content.

You can use the Liferay DXP UI and services together or focus solely on using services via REST web APIs.

The architecture satisfies these requirements:

- Supports using common development technologies
- Leverages development standards
- Facilitates swapping components
- Starts fast and performs well
- Its runtime is easy to configure and inspect

The Core supports UI and service deployments and orchestrates wiring them together.

162.1 Core

Liferay DXP is a web application that runs on your application server. The Core bootstraps the application and Liferay's built-in frameworks.

There are frameworks for these things and more:

- Adaptive Media
- Application Configuration
- Application Security
- Asset Framework
- File Management
- Localization

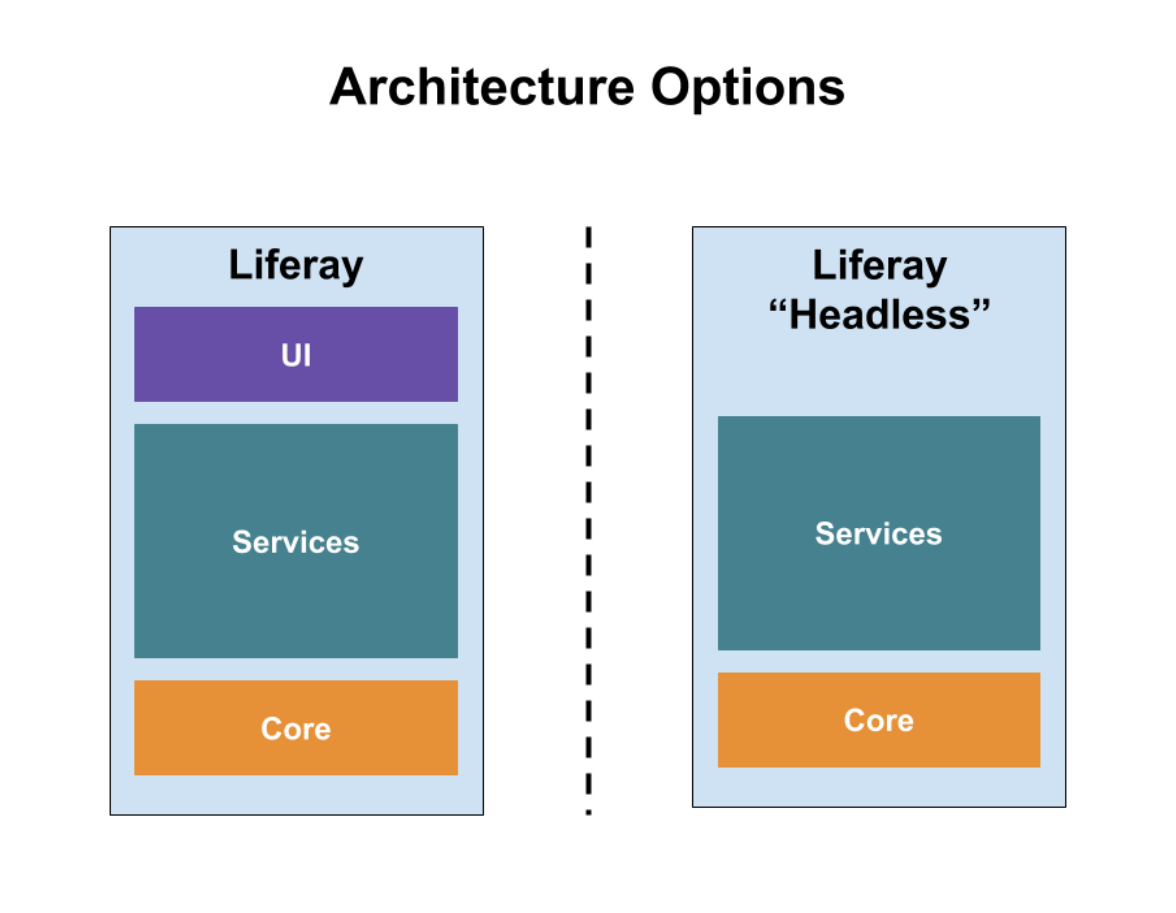

Figure 162.1: Liferay DXP portals and Sites contain content and widgets. Liferay DXP can also be used "headless"—without the UI.

- Search
- Segmentation and Personalization
- Upgrade Processes
- Web Fragments
- Workflow

The Core provides the component runtime environment for the frameworks, services, and UI. Here are some component examples:

- Services
- Service customizations
- Portlets (templates, controllers, and resources)
- JavaScript applications (templates, routers, and resources)
- JSP customization via Portlet Filters
- Theme
- Shared Language Keys
- Navigation components

The following figure shows these component types in the runtime environment.

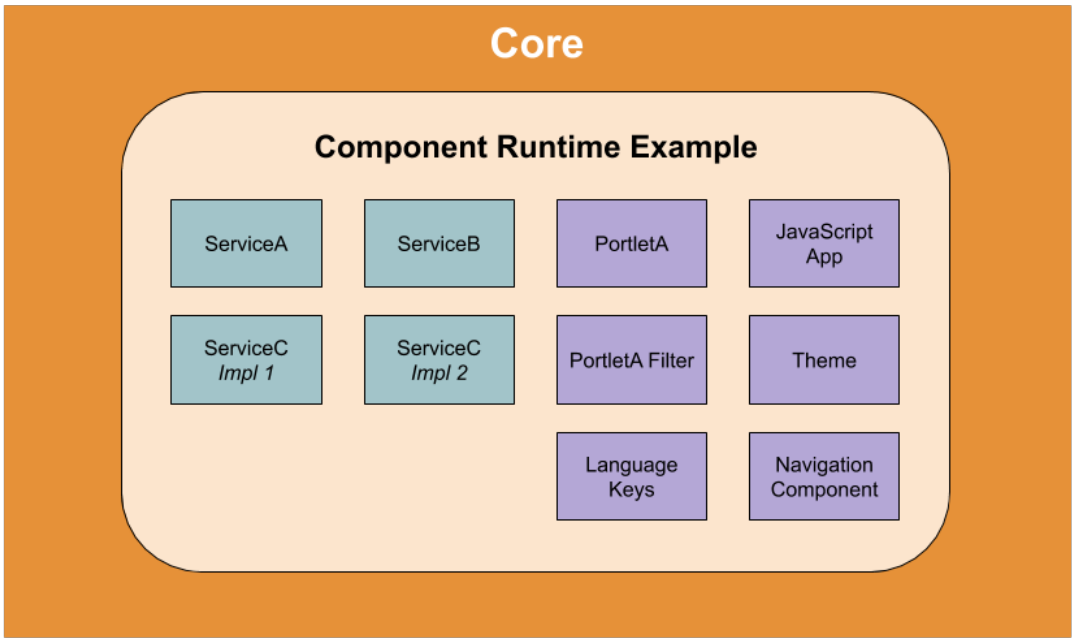

Figure 162.2: The Core provides a runtime environment for components, such as the ones here. New component implementations can extend or replace existing implementations dynamically.

The runtime environment supports adding, replacing, and customizing components on-the-fly. This makes the following scenarios possible:

Replacement: If the `ServiceC Impl 2` component has a higher ranking than existing component `ServiceC Impl 1`, `ServiceC Impl 2` is used in its place.

Customization: The `PortletA Filter` intercepts and modifies requests to and responses from `PortletA`, affecting the content `PortletA` displays.

Component WAR and module JAR projects install as OSGi bundles (modules). Liferay DXP's OSGi framework defines the module lifecycle, enforces dependencies, defines the class loading structure, and provides an API and CLI (Felix Gogo Shell) for managing modules and components. The Core is configured via portal properties files and Server Administration panels.

The service components provide business functionality.

162.2 Services

Business logic is implemented in services deployed to the component runtime environment. Built-in Core services and framework services operate on Liferay models such as Users, Roles, Web

Content, Documents and Media, and more. You can write and deploy custom services to introduce new models and functionality. Service components can access each other in Liferay DXP via dependency injection.

Front-end applications invoke the services to do work. You can deploy Java-based applications that call services directly using the Java APIs, and any web-based (Java and non-Java) application, whether deployed on Liferay DXP or not, can use the web APIs, which include headless REST APIs that conform to the OpenAPI standard and include plain web/REST services. The following figure shows Liferay DXP applications and external clients invoking Liferay services.

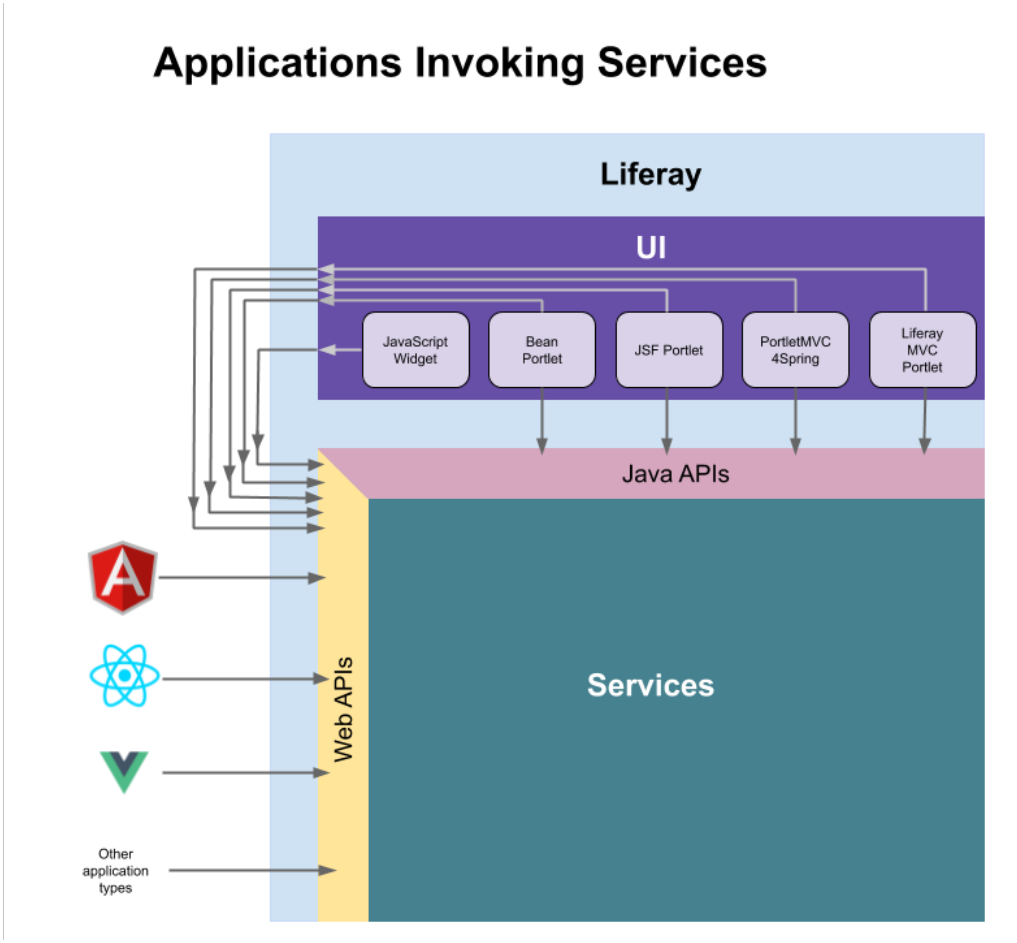

Figure 162.3: Remote and Liferay DXP applications can invoke services via REST web APIs. Liferay DXP Java-based portlets can also invoke services via Java APIs.

Liferay services are built using Service Builder and made REST-ful using REST Builder. The services are easy to override and extend too.

Liferay DXP also provides a web-based UI, which makes content and service functionality available in browsers.

162.3 UI

Liferay DXP's UI helps people do work, collaborate, and enjoy content. The UI consists of

- Liferay DXP application: The web application for managing Portals, Sites, Users, Pages, Widgets, and more.
- Applications: Widgets that provide a user interface for services already deployed.
- Themes: Plugins for styling Sites with a unique look and feel.

The UI concepts article digs deeper into developing and customizing UI components.

As you can see, the Liferay DXP architecture supports developing services, UI components, and customizations. The architecture section covers Core, service, and UI topics. Next, we dive into the Core to describe class loading, modularity, and more. But you can jump ahead to any service or UI architecture topics, if you like. Enjoy exploring the Liferay DXP architecture!

CHAPTER 163

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 article 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 DXP's classloader hierarchy.
Here are the classloader descriptions:

- **Bootstrap**: The JRE's classes (from packages `java.*`) and Java extension classes (from `$JAVA_HOME/lib/ext`). No matter the context, loading all `java.*` classes is delegated to the Bootstrap classloader.
- **System**: Classes configured on the `CLASSPATH` and or passed in via the application server's Java classpath (`-cp` or `-classpath`) parameter.
- **Common**: Classes accessible globally to web applications on the application server.
- **Web Application**: Classes in the application's `WEB-INF/classes` folder and `WEB-INF/lib/*.jar`.
- **Module Framework**: Liferay's OSGi module framework classloader which is used to provide controlled isolation for the module framework bundles.
- **bundle**: Classes from a bundle's packages or from packages other bundles export.
- **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

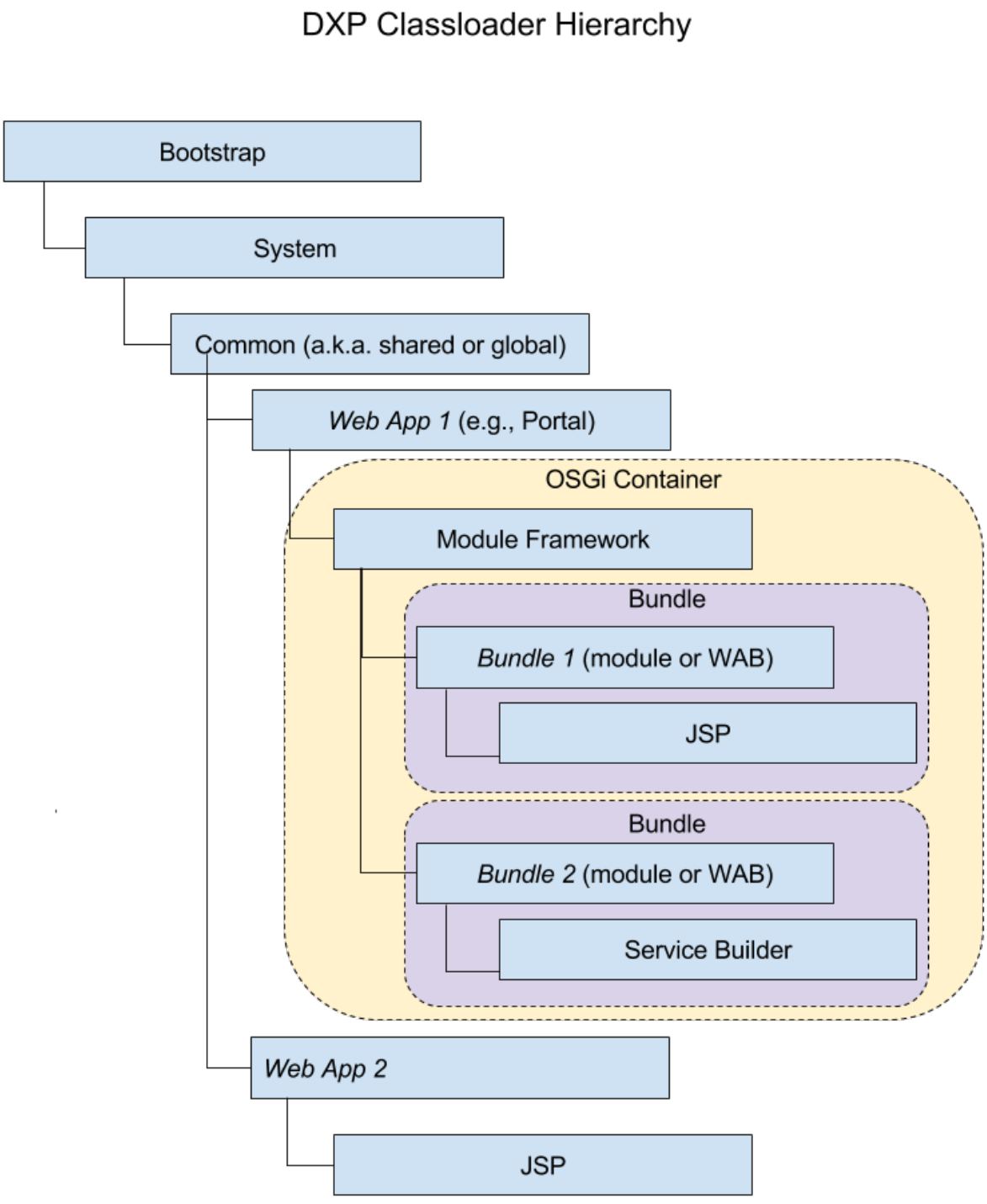

Figure 163.1: 0: Here is Liferay's classloader hierarchy.

- **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.

163.1 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.

163.2 Other Classloading Perspectives

Bundle Classloading Flow 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.

163.3 Related Topics

Bundle Classloading Flow

CHAPTER 164

LIFERAY DXP STARTUP PHASES

Knowing Liferay's startup phases helps you troubleshoot startup failures. By learning the phase triggered events, you can listen for phases and act on them. This article describes the startup phases and identifies how to implement actions for phase events.

Startup consists of these main phases:

1. **Portal Context Initialization Phase:** focuses on low level tasks without a web context.
2. **Main Servlet Initialization Phase:** focuses on the portlet container and the Liferay DXP web application's UI features such as Struts, Themes, and more.

The Portal Context Initialization Phase sets the stage for the Main Servlet Initialization Phase.

164.1 Portal Context Initialization Phase

The Portal Context Initialization phase runs first with these tasks:

1. Set up low level utilities such as logging and those in `PortalUtil` and `InitUtil`.
2. OSGi framework is initialized.
3. Spring Phase 1: INFRASTRUCTURE beans specified by the Spring context files listed in Portal property `spring.infrastructure.configs` are loaded.
4. INFRASTRUCTURE beans are published as OSGi services.
5. OSGi framework starts.

    1. Static bundles are installed and started.
    2. Dynamic bundles are started.

6. OSGi framework starts the runtime.
7. Spring Phase 2: MAIN

1. Load Spring beans specified by the Spring context files listed in Portal property `spring.configs`.
2. A `ModuleServiceLifecycle` event service with a service property `module.service.lifecycle` value `spring.initialized` (i.e., `SPRING_INITIALIZED`) registers.

8. MAIN Spring beans are published as OSGi services.

164.2 Main Servlet Initialization Phase

Here's the phase's activity sequence:

1. The `ModuleServiceLifecycle` event service is updated with the service property `module.service.lifecycle` value `database.initialized` (i.e., `DATABASE_INITIALIZED`).

2. The Global Startup event fires.

3. For each portal instance, the Application Startup events fire.

4. The `ModuleServiceLifecycle` event service is updated with the service property `module.service.lifecycle` value `portal.initialized` (i.e., `PORTAL_INITIALIZED`).

Now that you're acquainted with the startup phases, you can concentrate on the events they fire.

164.3 Acting on Events

The ways to act on events depends on the event type. These subsections describe the event types.

164.4 ModuleServiceLifecycle Events

You can wait for and act on `ModuleServiceLifecycle` event services.

164.5 Portal Startup Events

In your `liferay-portal-ext.properties` file, you can override the following properties and add your own `LifecycleAction` classes to the list of action classes to invoke on the events.

Global Startup Event runs once when Liferay DXP initializes. The `global.startup.events` property defines the event's default actions.

Application Startup Events runs once for each Site instance Liferay DXP initializes. The `application.startup.events` property defines the event's default actions.

164.6 Related Topics

Waiting on Lifecycle Events

OSGi Services and Dependency Injection with Declarative Services

CHAPTER 165

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.

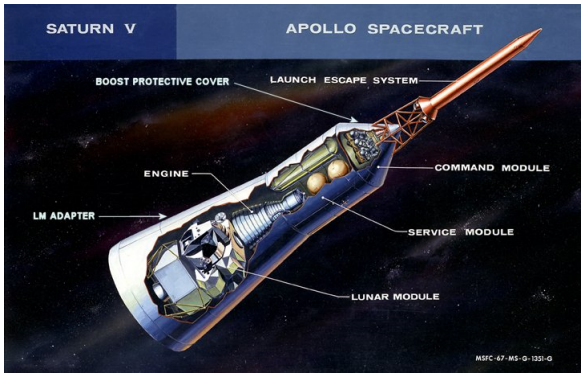

Figure 165.1: The Apollo spacecraft's modules collectively took astronauts to the moon's surface and back to Earth.

The spacecraft and its modules exemplified these modularity characteristics:

- **Distinct functionality**: Each module provides a distinct function (purpose); modules can be combined to provide an entirely new collective function.

  The Apollo spacecraft's modules were grouped together for a distinct collective function: take astronauts from the Earth's atmospheric rim, to the moon's surface, and back to Earth. The previous list identifies each module's distinct function.

- **Dependencies**: Modules can require capabilities other modules satisfy.

  The Apollo modules had these dependencies:

  - 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

165.1 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.

165.2 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.

165.3 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.

165.4 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.

165.5 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.

165.6 Example: Designing a Modular Application

Application design often starts out simple but gets more complex as you determine capabilities the application requires. If a third party library already provides the capability, you can deploy it with your app. You can otherwise implement the capability yourself.

As you design various aspects of your app to support its function, you must decide how those aspects fit into the code base. Putting them in a single monolithic code base often leads to tight coupling, while designating separate modules for each aspect fosters loose coupling. Adopting a modular approach to application design lets you reap the modularity benefits.

For example, you can apply modular design to a speech recognition app. Here are the app's function and required capabilities:

Function: interface with users to translate their speech into text for the computer to understand.
Required capabilities:

- Translates user words to text
- Uses a selected computer voice to speak to users.
- Interacts with users based on a script of instructions that include questions, commands, requests, and confirmations.

You could create modules to provide the required capabilities:

- *Speech to text*: Translates spoken words to text the computer understands.
- *Voice UI*: Interacts with users based on stored questions, commands, and confirmations.
- *Instruction manager*: Stores and provides the application's questions, commands, and confirmations.
- *Computer voice*: Stores and provides computer voices for users to choose from.

The following diagram contrasts a monolithic design for the speech recognition application with a modular design.

Designing the app as a monolith lumps everything together. There are no initial boundaries between the application aspects, whereas the modular design distinguishes the aspects.

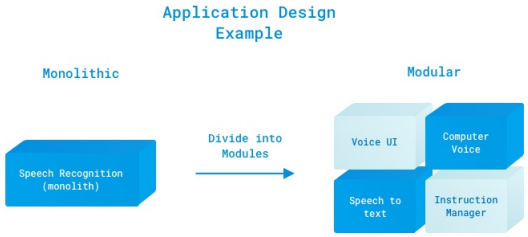

Figure 165.2: The speech recognition application can be implemented in a single monolithic code base or in modules, each focused on a particular function.

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.

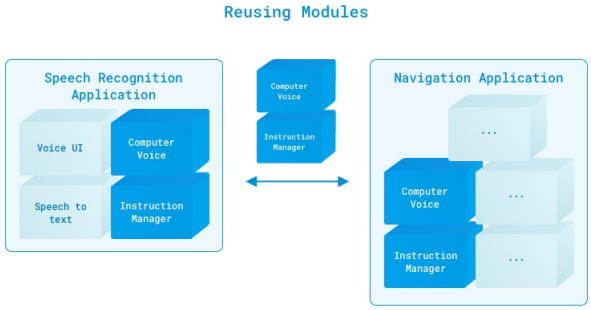

Figure 165.3: The *Instruction manager* and *Computer voice* modules designed for the speech recognition app can be used (or *reused*) by a navigation app.

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.

CHAPTER 166

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.

166.1 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.

166.2 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.

166.3 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.

166.4 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.

166.5 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.

166.6 Related Topics

Liferay Dev Studio
    Liferay Workspace
    Blade CLI
    Maven
    Upgrading Code to 7.2

CHAPTER 167

Module Lifecycle

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 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.

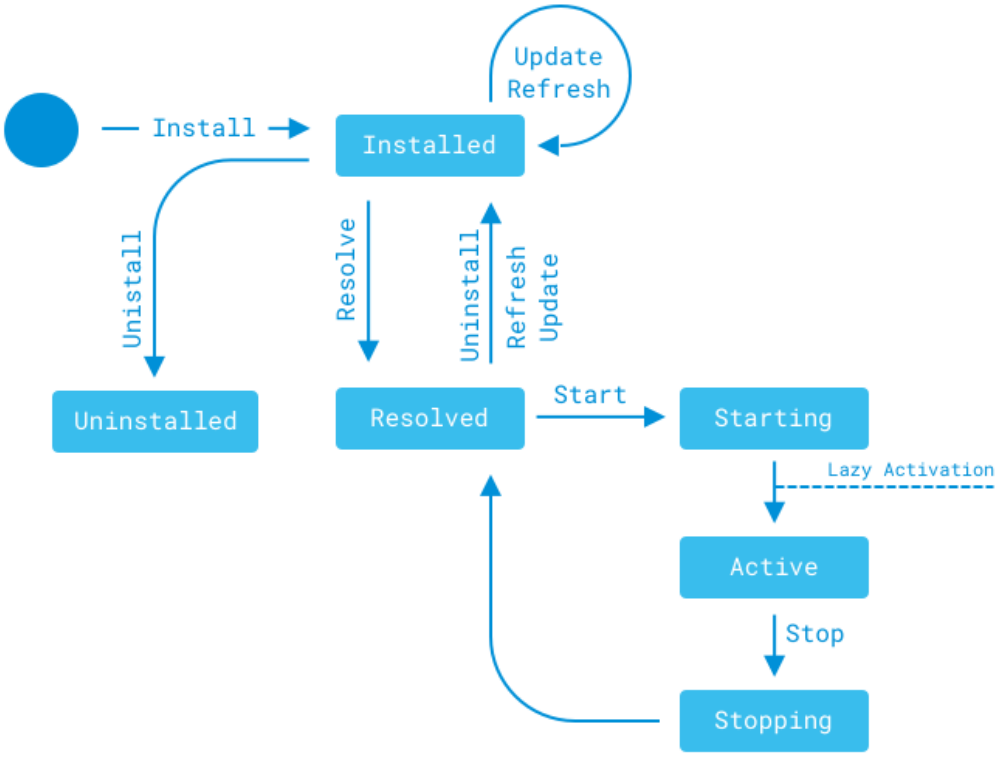

Figure 167.1: This state diagram illustrates the module lifecycle.

```
@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.

167.1 Related Topics

Creating a Project

Upgrading Code to 7.2

CHAPTER 168

UI ARCHITECTURE

Liferay DXP's UI is a portal for adding sites, pages, widgets, and content. It helps people do work, collaborate, and share content.

The UI comprises the following parts:

- Content: images, videos, and text.
- Applications: Widgets and portlets that expose functionality for accomplishing tasks.
- Themes: Plugins that use CSS, FreeMarker templates, HTML, and JavaScript to provide a site's overall look and feel.
- Product navigation sidebars and panels: Use these for administering sites.

168.1 Content

Liferay DXP's built-in applications help you publish images, video, forms, markup text, and more to site pages. Documents and Media stores images, videos, and documents to use throughout your site. The Web Experience Management suite helps you create, maintain, and organize content. Liferay Forms gives you robust form building capability. Message Boards facilitate lively discussions and Blogs let users express themselves with markup text and images. These are just a few of the built-in applications for adding site content.

168.2 Applications

Liferay DXP applications provide content and help users accomplish tasks. They're developed the same way as other web applications, and Liferay DXP can combine multiple applications on one page.

Liferay DXP supports developing JavaScript-based applications using popular front-end frameworks:

- Angular

- React
- Vue

Java-based portlet applications use the latest portlet standards and frameworks, including ones familiar to experienced Liferay portlet developers:

- Liferay MVC Portlet
- PortletMVC4Spring
- JSF Portlet

In the UI, applications are referred to as Widgets and categorized for users to add to pages. Administrative applications are available in the product menu panels.

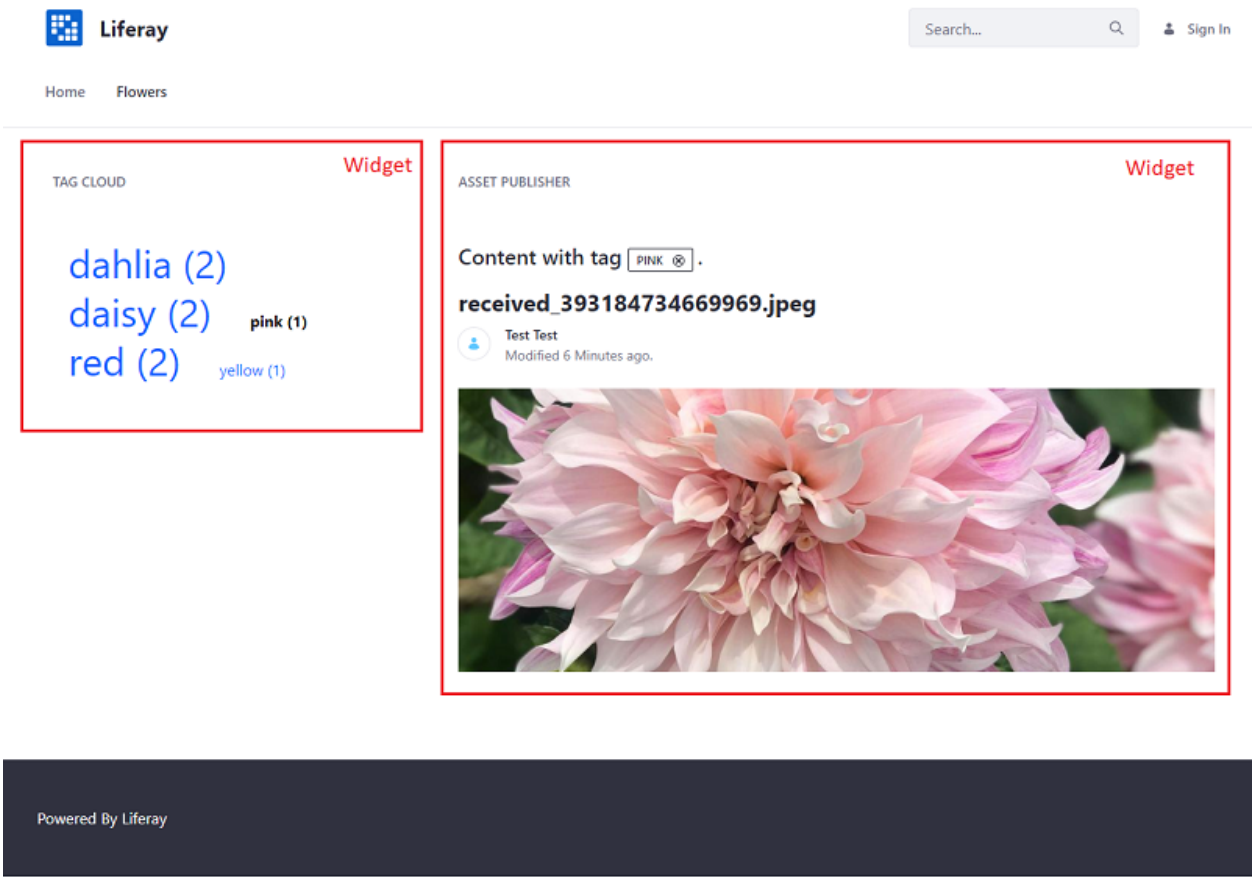

Figure 168.1: Widget pages offer users functionality. Widgets are organized into a page template's rows and columns. This template has two columns: a smaller left column and larger right column. On this page, users select tags in the Tag Cloud widget and the matching tagged images show the Asset Publisher widget.

168.3 Themes

A theme styles a site with a unique look and feel. It's developed as a WAR project that includes CSS, JavaScript, and markup content. You can develop themes using whatever tools you prefer, but

Liferay DXP offers Bootstrap-based components and theme tooling to create and deploy themes in no time.

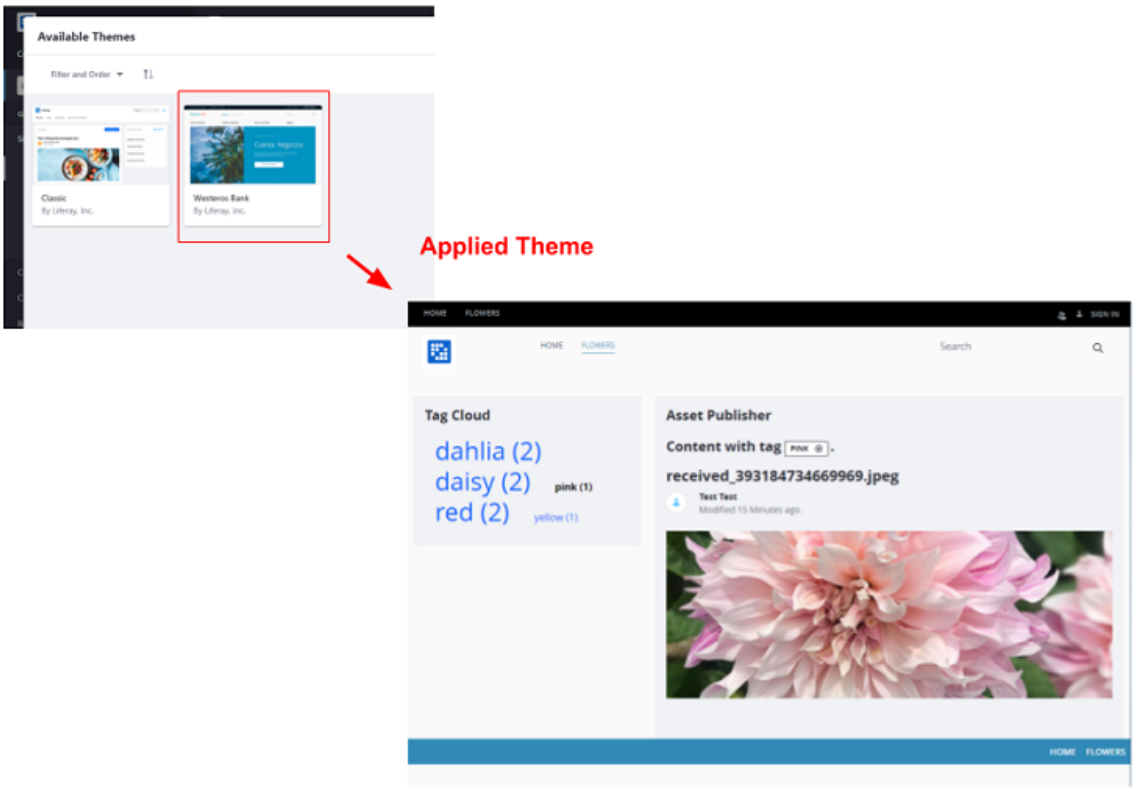

Figure 168.2: You can select an attractive theme and apply it to your site.

Here's a quick demonstration of developing a theme:

1. Create a theme using the Theme Generator. The theme extends the base theme you specified to the Theme Generator—Liferay's Styled theme is the default.
2. Run `gulp build` to generate the base theme files to the `build` folder subfolders:

    - `templates`: FreeMarker templates specify site page markup. `portal_normal.ftl` is the central file; it includes templates that define the page parts (e.g., header, navigation, footer). The `init.ftl` file defines default variables available to the templates.
    - `css`: SCCS files that provide styling.
    - `font`: Font Awesome and Glyphicons fonts.
    - `js`: JavaScript files; `main.js` is the Styled theme's JavaScript.

- `images`: Image files.

3. Override aspects of the base theme by copying relevant files from the `build` subfolders to folders of the same name in your `src` folder. The Theme Anatomy Guide describes all the files. Here's an example of a customized `portal_normal.ftl`:

```
<html class="${root_css_class}">
<head></head>
  <body class="${body_class}">
    <header class="${header_css_class}">
      <a class="${logo_css_class} href="${site_url}"><img src="${site_logo}"/></a>
      <#include "${full_templates_path}/navigation.ftl" />
    </header>
    <section>
      ${portlets}
    </section>
      <#include "${full_templates_path}/footer.ftl" />
  </body>
</html>
```

4. Add custom styling using your theme's `_custom.scss` file (i.e., `src/css/_custom.scss`). Liferay DXP supports Bootstrap, as well as Sass, so you can use Bootstrap utilities in your markup and Sass nesting, variables, and more in your CSS files. This snippet styles the logo:

```
.logo {
  margin-left: 15px;

  img {
    height: auto;
  }

  @include media-breakpoint-down(md) {
    text-align: center;
    width: 100%;
  }
}
```

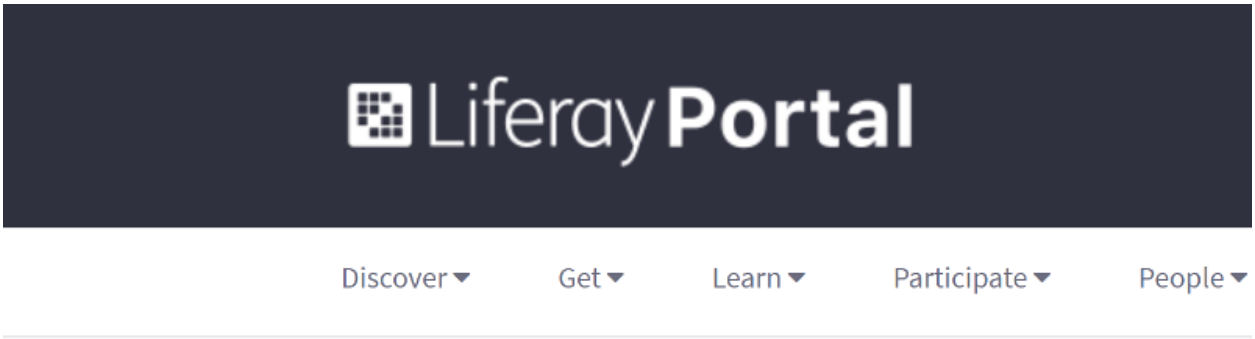

Figure 168.3: You can provide custom styling using the theme's `_custom.sccs` file.

5. Deploy your theme by executing `gulp deploy`.

The theme is available to apply to your site.

For details, Theme Components breaks down a theme's parts, and the Themes section provides theme development details.

168.4 Product Navigation Sidebars and Panels

The product navigation sidebars and panels enable administrators to build sites, add pages, apply themes, and configure the portal. It's also where you can provide administrative functionality for your custom applications. The navigation sidebars and panels are customizable.

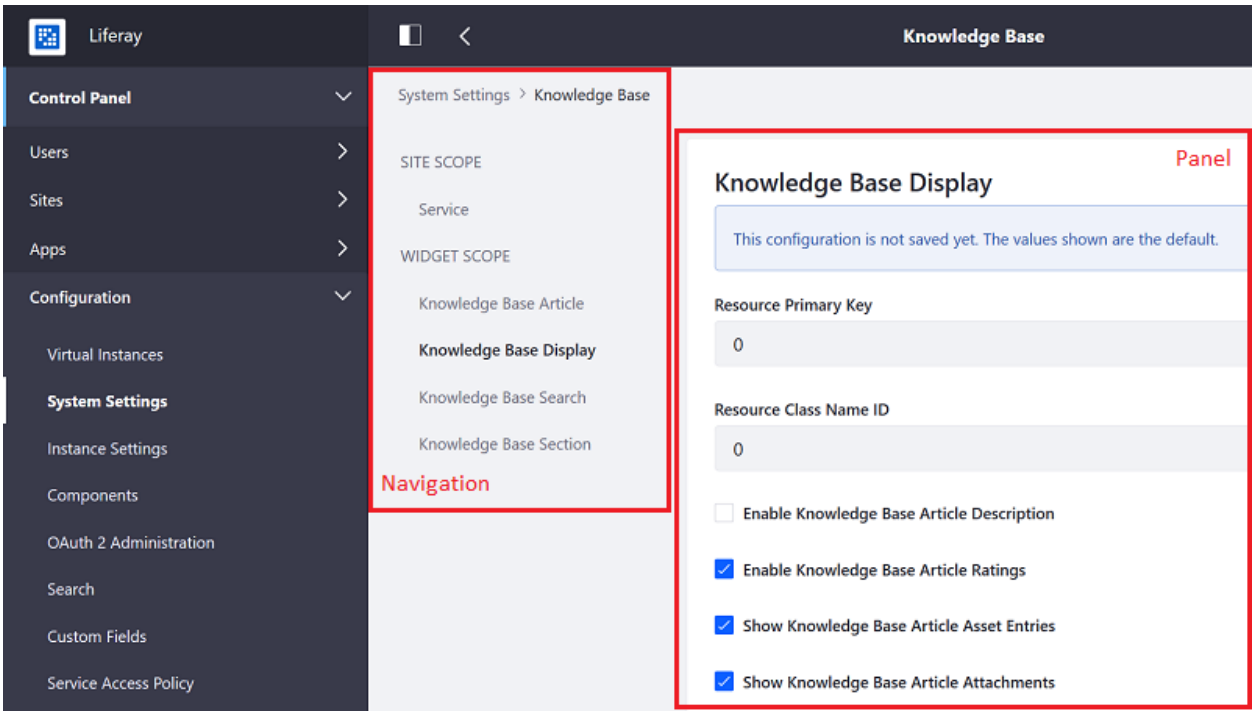

Figure 168.4: Liferay facilitates integrating custom administrative functionality through navigation menus and administrative applications.

As you can see, Liferay DXP's UI is highly flexible and customizable. Here's where to learn more:

- Theme Components: Explains available mechanisms and extensions for customizing and theming pages, content, and applications.
- Understanding the Page Structure: Describes how the page's UI is organized and introduces tools for populating and developing each section.

CHAPTER 169

THEME COMPONENTS

This guide provides an overview of the following theme development and customization topics:

- Theme Templates
- CSS Frameworks and Extensions
- Theme Customizations and Extensions
- Portlet Customizations and Extensions

169.1 Theme Templates and Utilities

The default FreeMarker templates provide helpful utilities and handle key pieces of page layout (page) functionality:

- `portal_normal.ftl`: Similar to a static site's `index.html`, this file is the hub for all the theme templates and provides the overall markup for the page.
- `init.ftl`: Contains variables commonly used throughout the theme templates. Refer to it to look up theme objects. For convenience, the FreeMarker Variable Reference Guide lists the objects. **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`: Controls the theme's portlets. If your theme uses Portlet Decorators, modify this file to create application decorator-specific theme settings.
- `navigation.ftl`: Contains the navigation markup. To customize pages in the navigation, you must use the `liferay.navigation_menu` macro. Then you can leverage widget templates 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>
```

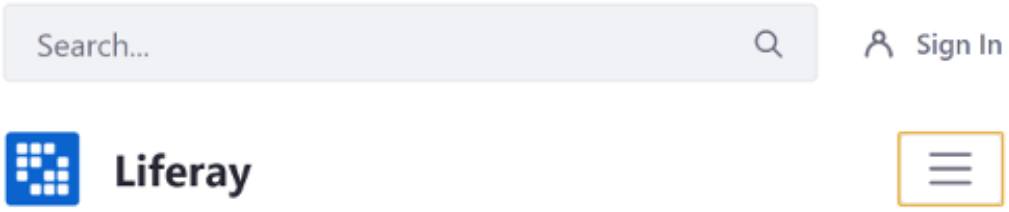

Figure 169.1: The collapsed navbar provides simplified user-friendly navigation for mobile devices.

- `portal_pop_up.ftl`: Controls 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`.

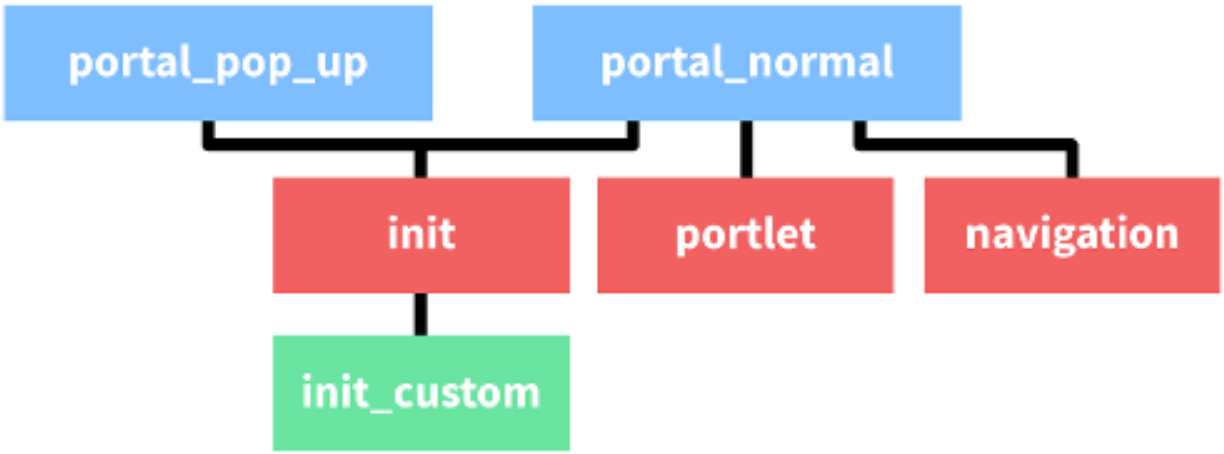

Figure 169.2: Each theme template provides a portion of the page's markup and functionality.

- `FTL_Liferay.ftl`: Provides macros for commonly used portlets and theme resources.
- `taglib-mappings.properties`: Maps the portal taglibs to FreeMarker macros. Taglibs can quickly create common UI components. This properties file is provided separately for each app taglib. For convenience, these FreeMarker macros appear in the FreeMarker Taglib Mappings reference guide. See the Taglib reference for more information on using each taglib in your theme templates.

169.2 CSS Frameworks and Extensions

Themes are integrated with SASS, 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 Customizing Atlas and Clay Base Themes for more information.

169.3 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 Administrators can configure via the Look and Feel menu. See the color scheme tutorial for more information.
- **Configurable Theme Settings:** Administrators can configure theme aspects that change frequently, such as 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 use in FreeMarker templates. This allows non-JSP templating languages in themes, widget templates, and any other templates. 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 containing CSS and JavaScript. They can be shared with other developers to provide common components for themes. See Generating Themelets for more information.

169.4 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 for more information.
- **Widget Templates:** Provides an alternate display style for a portlet. Note that not all portlets support widget templates. See the Widget Templates User Guide for more information.
- **Portlet Decorator:** Customizes the exterior decoration for a portlet. See Portlet Decorators 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.

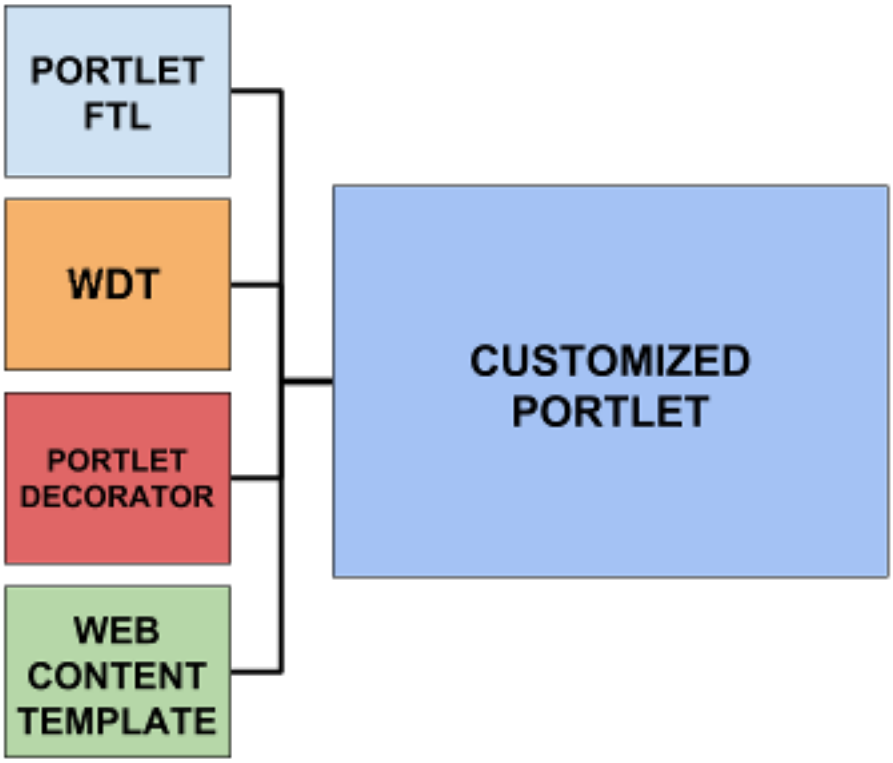

Figure 169.3: There are several extension points for customizing portlets

169.5 Related Topics

- Understanding the Page Structure
- Installing the Theme Generator and Creating a Theme
- Developing Themes

CHAPTER 170

UNDERSTANDING THE PAGE STRUCTURE

Understanding the page'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 in its `portal_normal.ftl` template:

- **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.

170.1 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.

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

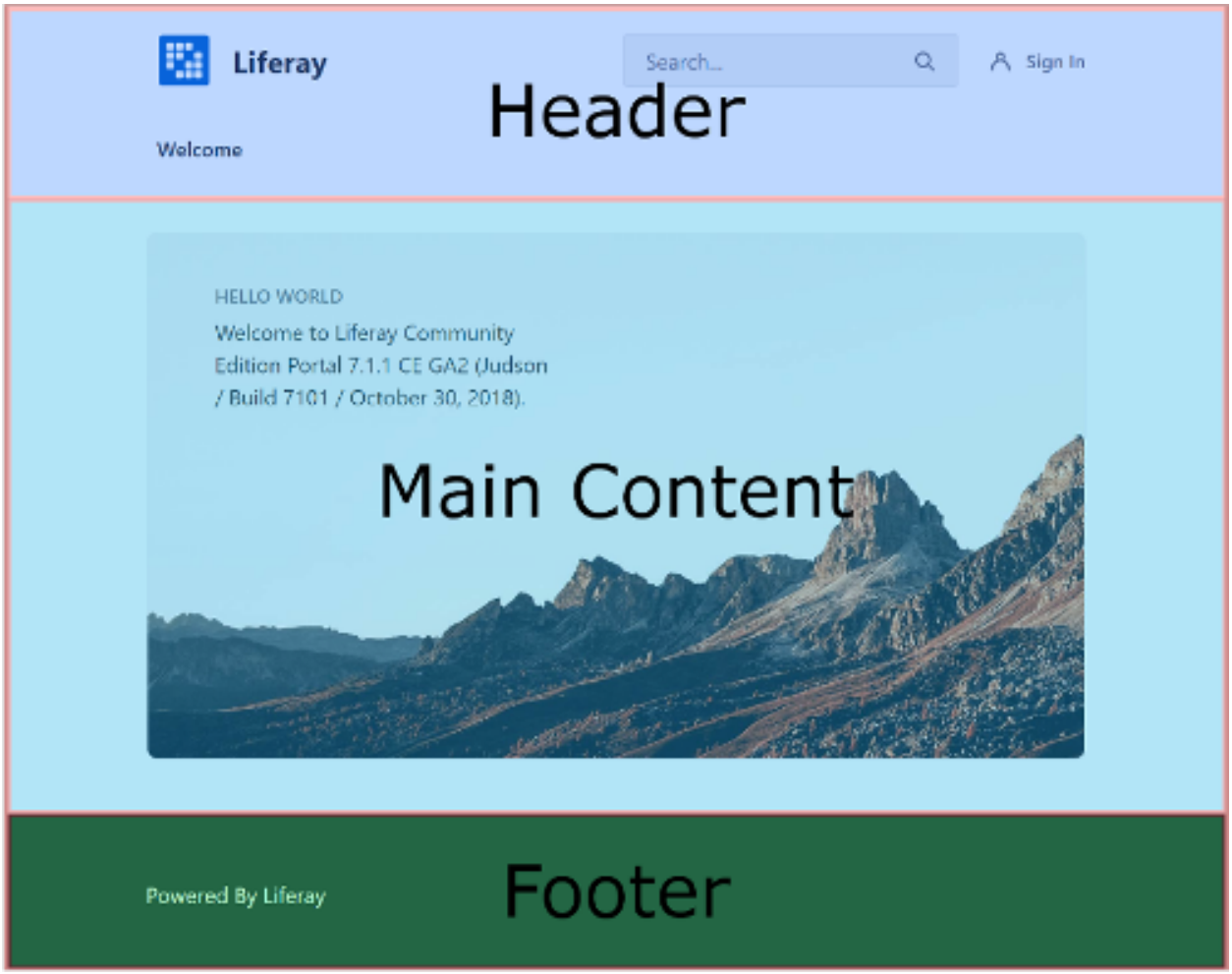

Figure 170.1: The page layout is broken into three key sections.

As shown in the diagram above, you can also add fragments to a page. 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.

170.2 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 with the Liferay Theme Generator's layout sub-generator.

Layout templates can be pre-configured depending on the page type you choose when the page is created. Along with setting the types of portlets to include on the page, the page template may also define the default layout template for the page. Climbing further up the scope chain, you can

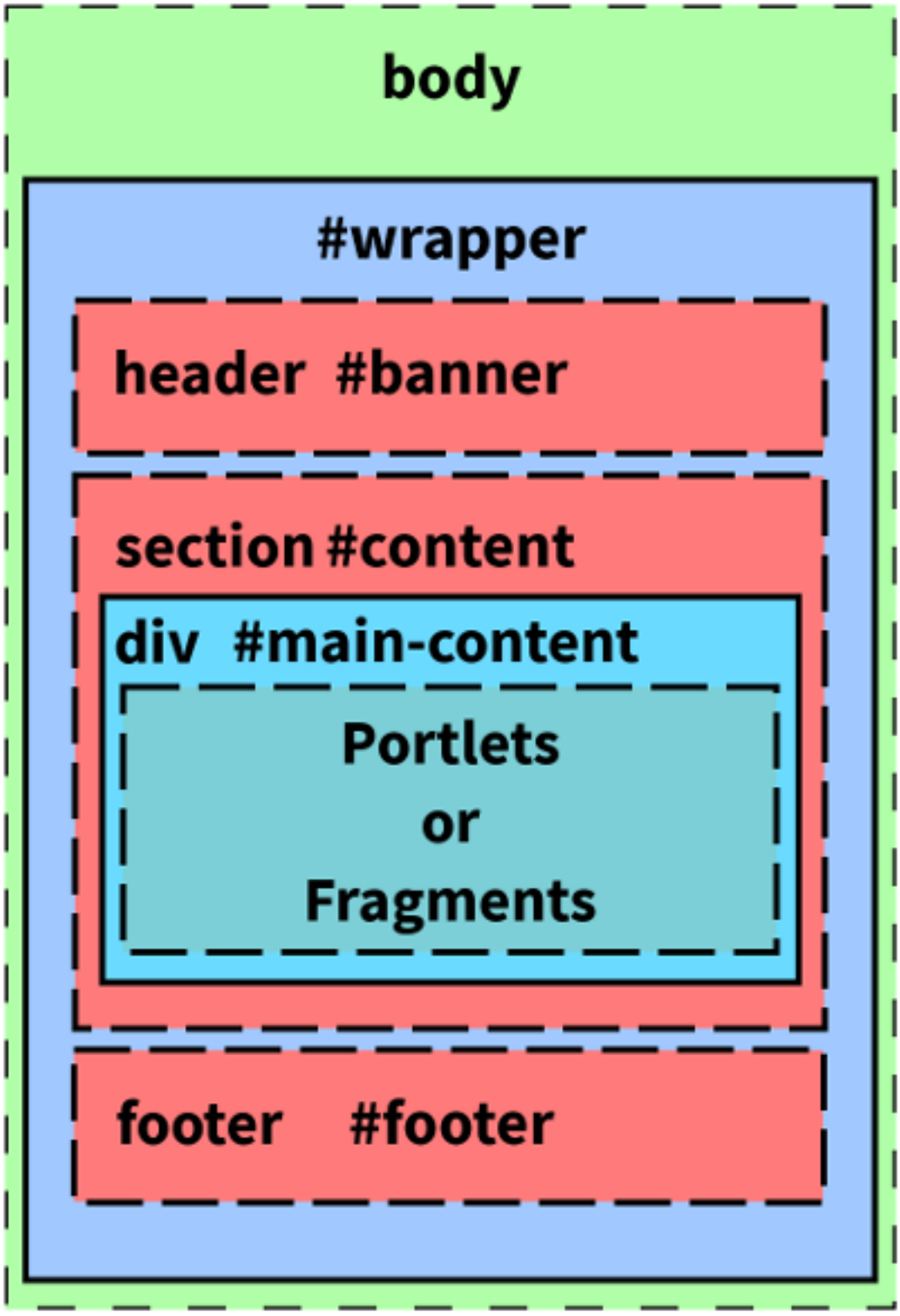

Figure 170.2: Each section of the page has elements and IDs that you can target for styling.

create Site Templates, which can define the pages, page templates, layout templates, and theme(s) to use for site pages.

170.3 Product Navigation Sidebars and Panels

The main page layout also contains a few notable sidebars an administrative user can trigger through the Control Menu. These are listed below:

- **Add Menu:** For adding portlets (widgets) and fragments (if applicable) to the page
- **Control Menu:** Provides the main navigation for accessing the Add Menu, Product Menu, and Simulation Panel
- **Product Menu:** Contains administrative apps, configuration settings, and user account settings, profile, and dashboard page
- **Simulation Panel:** Simulates how the page appears on different devices

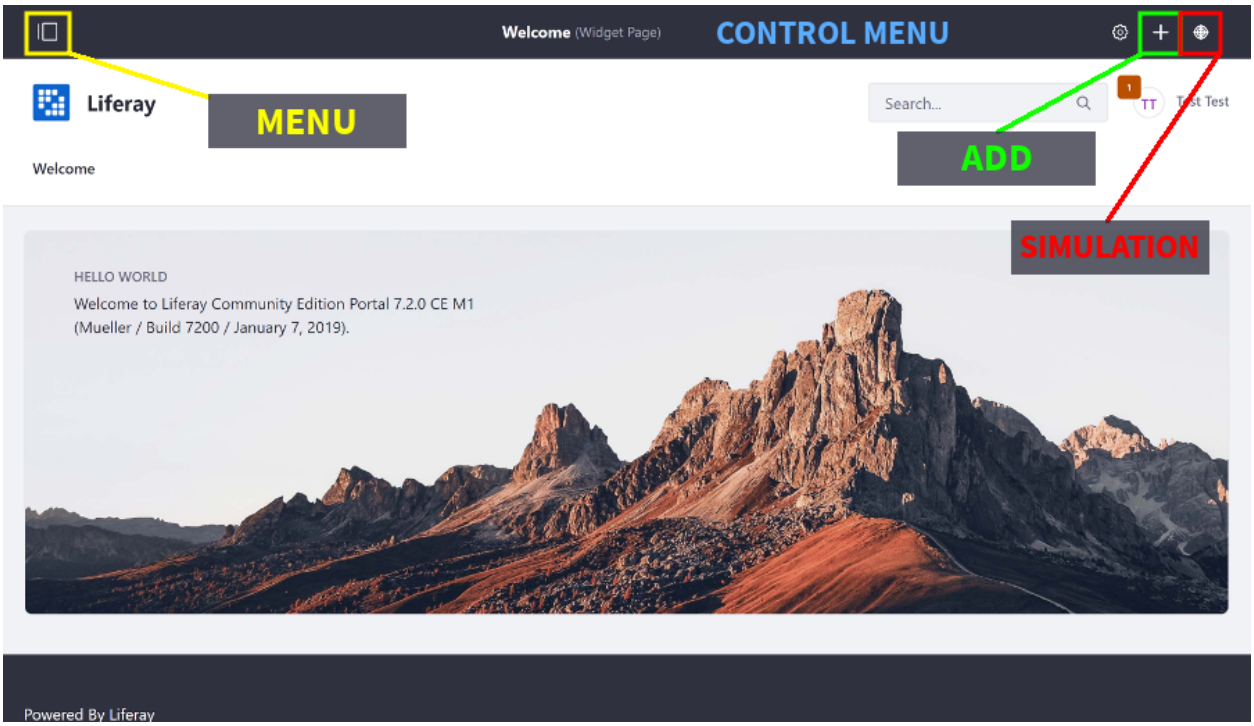

Figure 170.3: Remember to account for the product navigation sidebars and panels when styling your site.

When styling the page, you must keep the navigation menus in mind, especially for absolutely positioned elements, such as a fixed navbar. If the user is logged in and can view the Control Menu, the fixed navbar must have a top margin equal to the Control Menu's height.

See the Product Navigation articles for more information on customizing these menus.

170.4 Related Topics

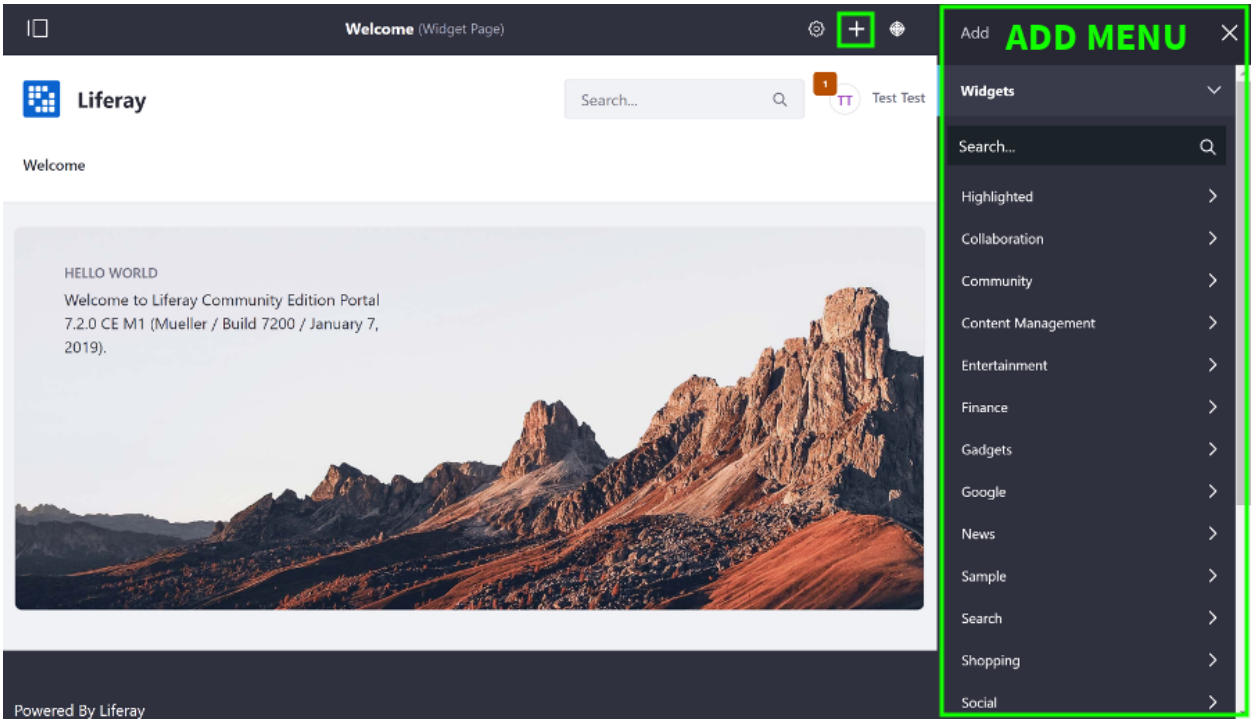

Figure 170.4: The Add Menu pushes the main contents to the left.

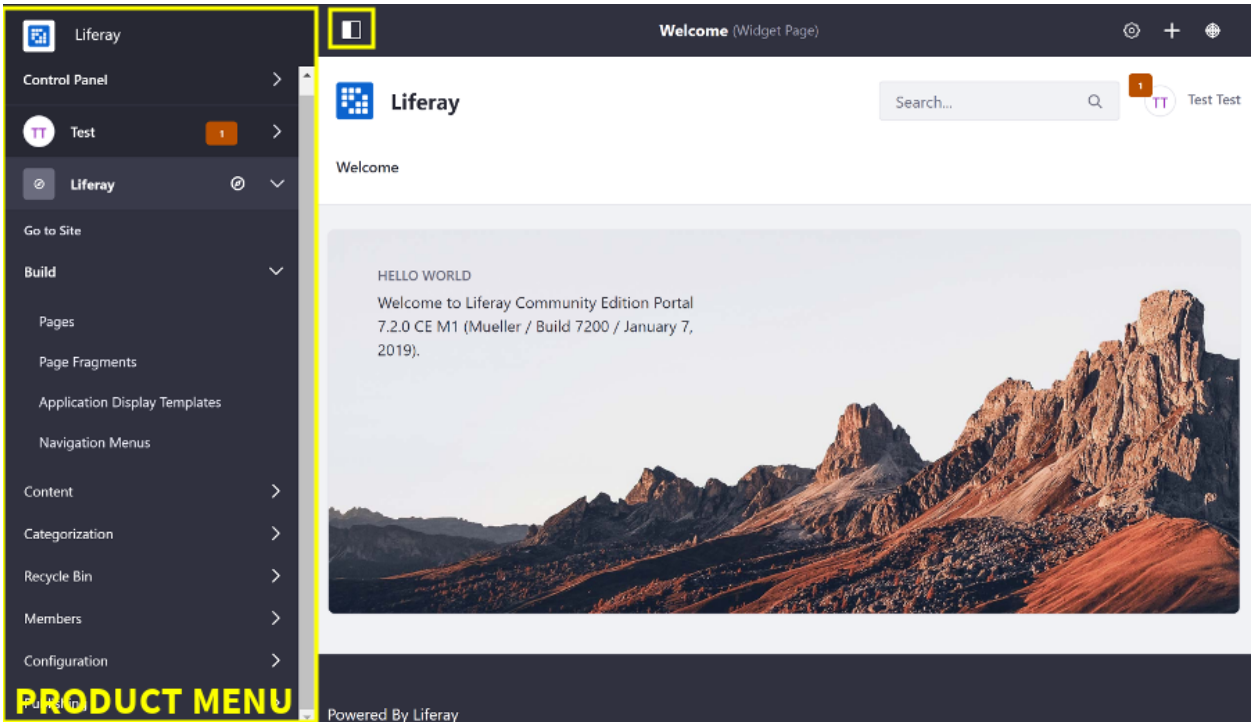

Figure 170.5: The Product Menu pushes the main contents to the right.

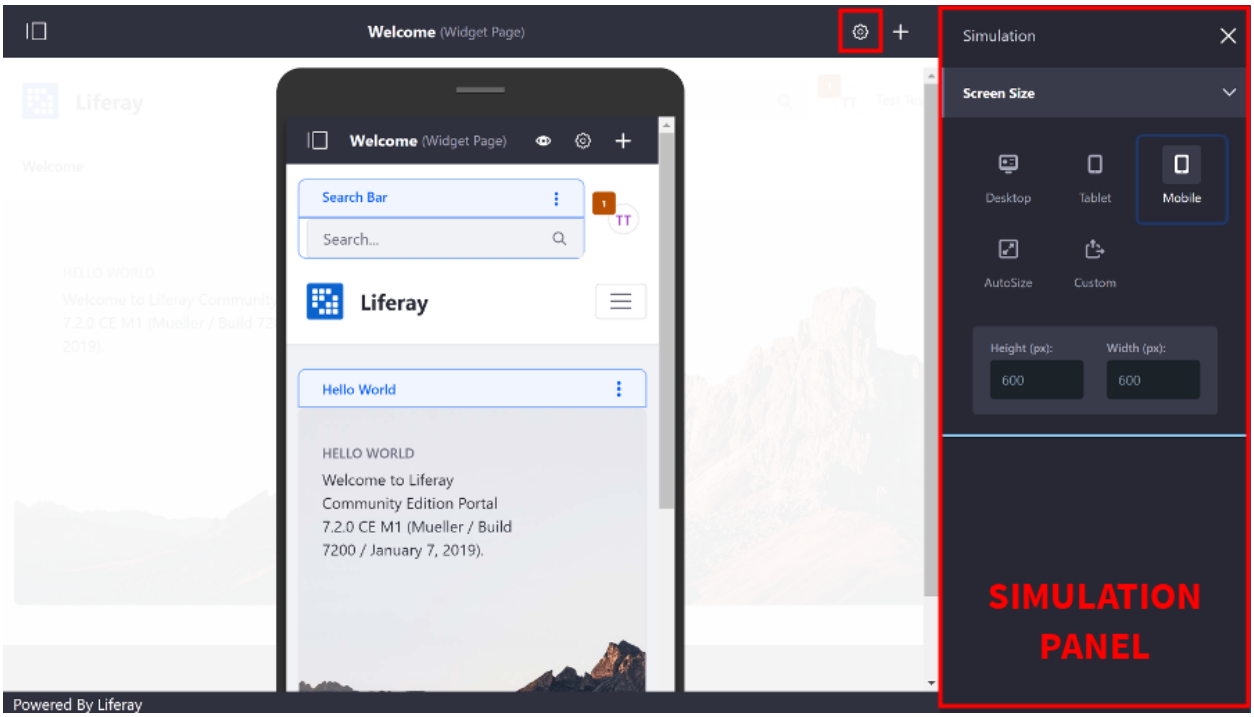

Figure 170.6: The Simulation Panel pushes the main contents to the left.

- Creating Layout Templates with the Layouts Sub-generator
- Bundling Layout Templates with a Theme
- Installing the Liferay Theme Generator and Creating a Theme

CHAPTER 171

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 article walks you through it.

Here is the algorithm for classloading in a bundle:

1. If the class is in a `java.*` package, delegate loading to the parent classloader. Otherwise, continue.
2. If the class is in the OSGi Framework's boot delegation list, delegate loading to the parent classloader. Otherwise, continue.
3. If the class is in one of the packages the bundle imports from a wired exporter, the exporting bundle's classloader loads it. A *wired exporter* is another bundle's classloader that previously loaded the package. If the class isn't found, continue.
4. If the class is imported by one of the bundle's required bundles, the required bundle's classloader loads it.
5. If the class is in the bundle's classpath (manifest header `Bundle-ClassPath`), the bundle's classloader loads it. Otherwise, continue.
6. If the class is in the bundle's fragments classpath, the bundle's classloader loads it.
7. If the class is in a package that's dynamically imported using `DynamicImport-Package` and a wire is established with the exporting bundle, the exporting bundle's classloader loads it. Otherwise, the class isn't found.

Congratulations! Now you know how Liferay DXP finds and loads classes for OSGi bundles.

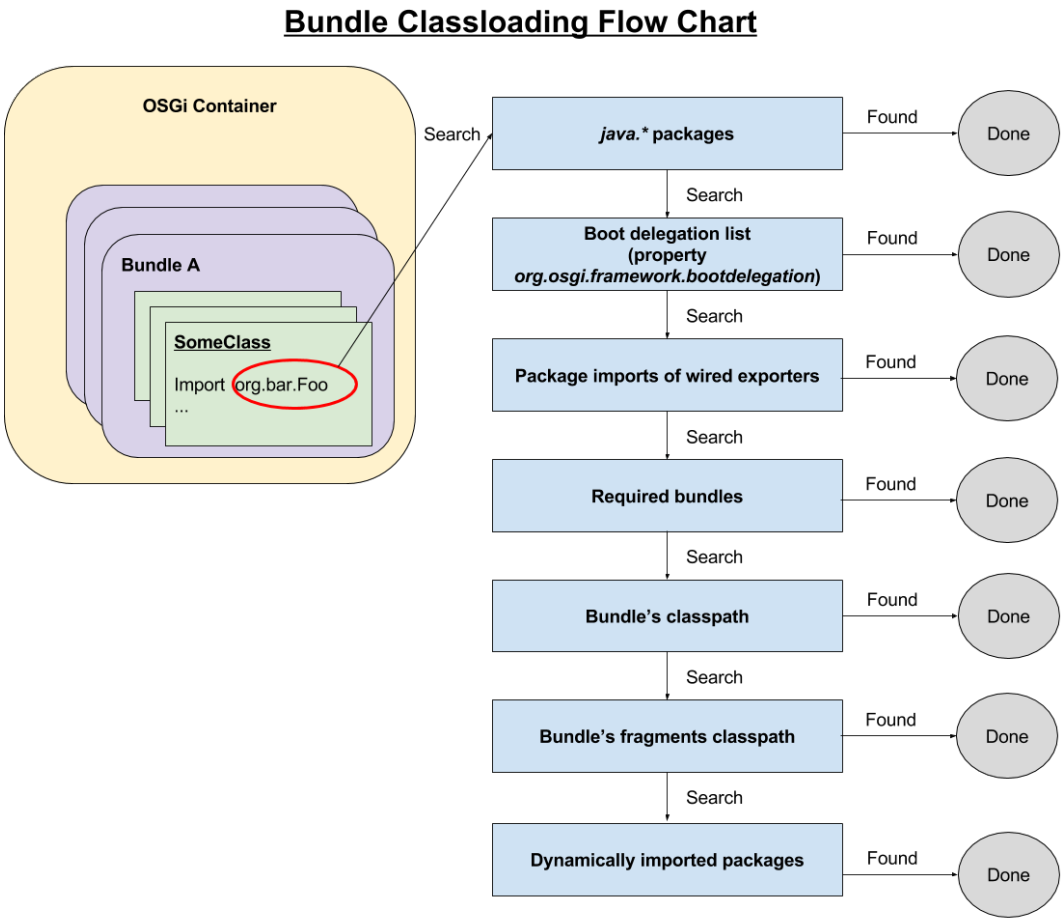

Figure 171.1: This flow chart illustrates classloading in a bundle.

CHAPTER 172

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 article 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.

172.1 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*, *security,* 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 apps 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 independent apps and independent modules.

2. Navigate the apps and modules to find components that might provide your desired extension point. Remember to check for your keywords in element names. The keyword *security* is found in the Liferay CE Portal Security app. Select it.

3. The Security application has several modules to inspect. Select the *Liferay Portal Security LDAP Implementation* module.

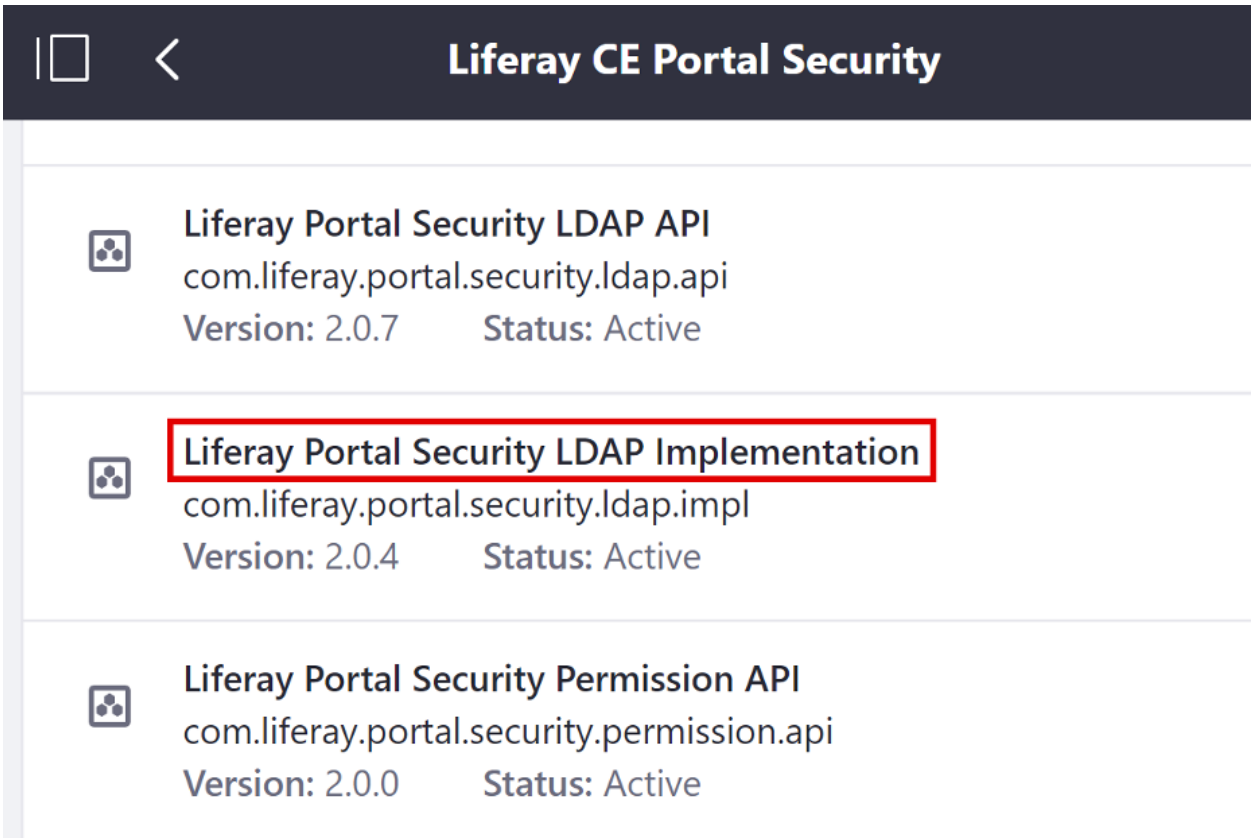

Figure 172.1: The module name can be found using the App Manager.

4. 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.

172.2 Finding Extension Points in a Component

Once you have the component that relates to the functionality you want to extend, you can use the Gogo shell's Service Component Runtime (SCR) commands to inspect it. You can execute SCR

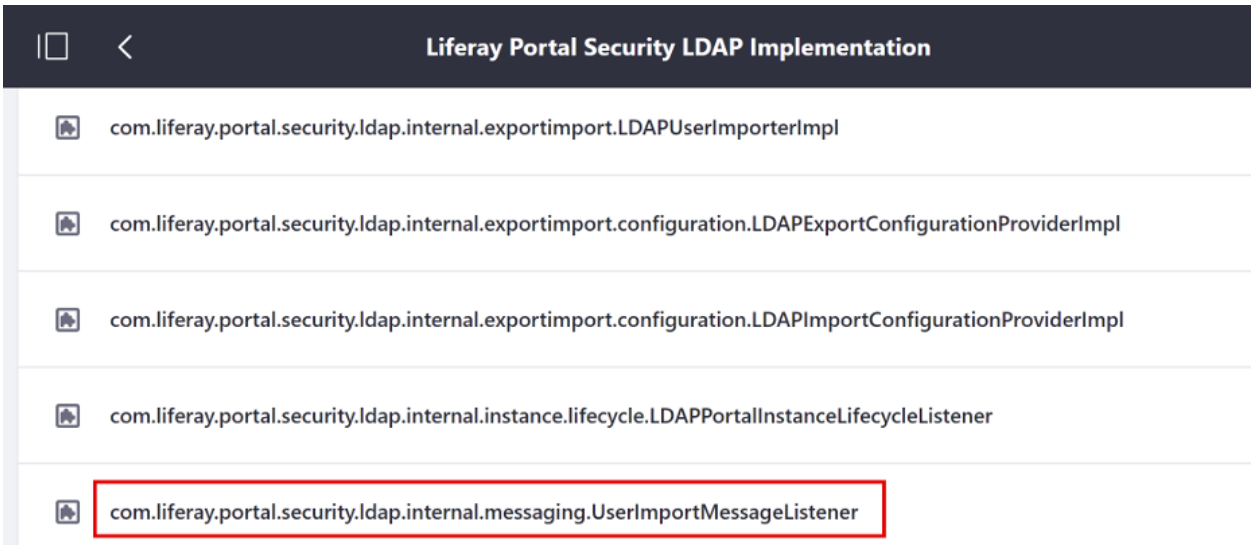

Figure 172.2: The component name can be found using the App Manager.

commands using Liferay Blade CLI or in Gogo shell. This article 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:

```
- _ldapUserImporter:
  com.liferay.portal.security.ldap.exportimport.LDAPUserImporter
  SATISFIED
  1..1
  dynamic+greedy
    target=(*) scope=bundle (1 binding):
    * Bound to [7764] from bundle 1754 (com.liferay.portal.security.ldap.impl:2.0.4)
```

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. In the example, the policy is `dynamic` and the policy option is `greedy`. 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. See Overriding OSGi Services to learn how to override a component's service reference.

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 173

TROUBLESHOOTING CUSTOMIZATIONS

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.

Click a question to view the answer.

Why aren't my fragment's JSP overrides showing?

```
<p><a href="/docs/7-2/customization/-/knowledge_base/c/why-arent-jsp-overrides-i-made-using-fragments-showing">Make sure your <code>Fragment-
Host</code>'s bundle version is compatible with the host's bundle version</a>. </p>
```

Why doesn't the package I use from the fragment host resolve?

```
<p><a href="/docs/7-2/customization/-/knowledge_base/c/why-is-a-package-i-use-from-the-fragment-host-unresolved">Refrain from importing (<code>Import
Package: ...</code>) host packages that the host doesn't export</a>. </p>
```

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>
```

CHAPTER 174

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.*,*
```

CHAPTER 175

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.

175.1 Related Topics

JSP Overrides using Portlet Filters
    Customizing JSPs
    Finding Artifacts

CHAPTER 176

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.

176.1 Related Topics

Detecting Unresolved OSGi Components
    Felix Gogo Shell

CHAPTER 177

CONTRIBUTING TO LIFERAY PORTAL

Liferay Portal is developed by its community consisting of users, enthusiasts, employees, customers, partners, and others. We strongly encourage you to contribute to Liferay's open source projects by implementing new features, enhancing existing features, and fixing bugs. We also welcome your participation in our forums, chat, writing documentation, and translating existing documentation.

Liferay Portal is known for its innovative top quality features. To maintain this reputation, all code changes are reviewed by a core set of project maintainers. We encourage you to join our Slack Chat and introduce yourself to the core maintainer(s) and engage them as you contribute to the areas they maintain.

Developing features and fixes requires cloning the source tree and building Liferay Portal.

177.1 Building Liferay Portal from source

The first step to contributing to Liferay Portal is to clone the `liferay-portal` repo from GitHub and build the platform from source code.

Please follow the instructions for building Liferay Portal from source code.

To better understand the code structure, please also read How the source is organized.

177.2 Tooling

Liferay tooling facilitates creating customizations and debugging code. Consider using these Liferay development tools:

- Blade CLI: a command line interface used to build and manage Liferay Workspaces and Liferay Portal projects. This CLI is intended for Gradle or Maven development.
- Liferay Workspace: a generated Gradle/Maven environment built to hold and manage Liferay Portal projects.
- Liferay Dev Studio: an Eclipse-based IDE supporting development for Liferay Portal.
- Liferay IntelliJ Plugin: a plugin providing support for Liferay Portal development with IntelliJ IDEA.

- Liferay Theme Generator: a generator that creates themes, layouts templates, and themelets for Liferay Portal development.
- Liferay JS Generator: a generator that creates JavaScript portlets with JavaScript tooling.

The Configure an IDE for use with the Liferay Source page, explains how to set up the project in your favorite IDE.

177.3 Additional Resources

Liferay Community Site
    Liferay Community Slack Chat
    Liferay Community Slack Chat Self Invite
    Contributor License Agreement
    General GitHub documentation
    GitHub pull request documentation

CHAPTER 178

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

178.1 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.

178.2 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.

178.3 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.

178.4 Related Topics

- Upgrading Model Listener Hooks
- Service Builder
- Service Builder Persistence

CHAPTER 179

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.

179.1 Using Liferay DXP'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.

179.2 Overriding a JSP Without Using Liferay DXP'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!

CHAPTER 180

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.

Follow these steps:

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!

CHAPTER 181

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.

Follow these steps:

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.

CHAPTER 182

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.

182.1 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.

182.2 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 %>
```

182.3 Using Fragment Host Internal Packages

To use an internal (unexported) host package, the fragment must explicitly exclude the package from its `Import-Package:` manifest header. For example, this `Import-Package` header excludes packages that match `com.liferay.portal.search.web.internal.*`.

```
Import-Package: !com.liferay.portal.search.web.internal.*,*
```

Unless you explicitly exclude the package, bnd adds the package to the `Import-Package:` header. Attempting to start the fragment while requiring an unexported package fails because the package is an unresolved requirement. For this reason, make sure to exclude such packages from your fragment's `Import-Package:` header.

Each fragment has full access to the host packages, including its internal (unexported) packages already.

Now you can easily modify the JSPs of any application in Liferay.

182.4 Related Topics

- JSP Overrides Using Portlet Filters

CHAPTER 183

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.

183.1 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
```

183.2 Implement a Custom JSP Bag

Liferay DXP (specifically the `CustomJspBagRegistryUtil` class) loads JSPs from `CustomJspBag` services. Here are steps for implementing 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.

183.3 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 API based approaches to overriding JSPs mentioned at the beginning of the article.

183.4 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.

183.5 Related Topics

- Upgrading Core JSP Hooks
- JSP Overrides Using Portlet Filters

CHAPTER 184

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!

184.1 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.

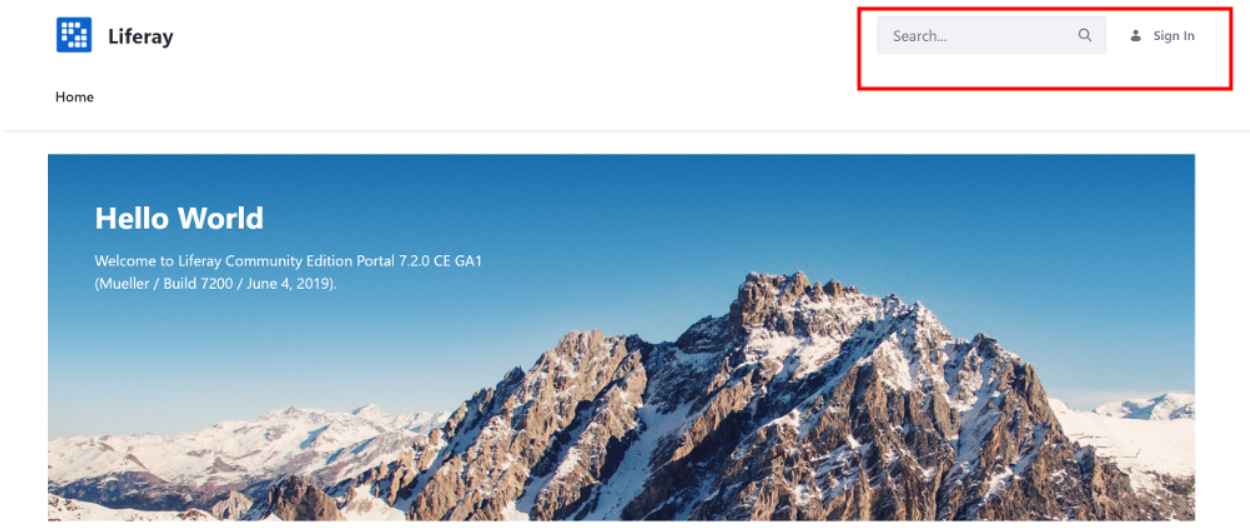

Figure 184.1: Liferay DXP's home page's search and sign in components are in a `fieldset`.

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.

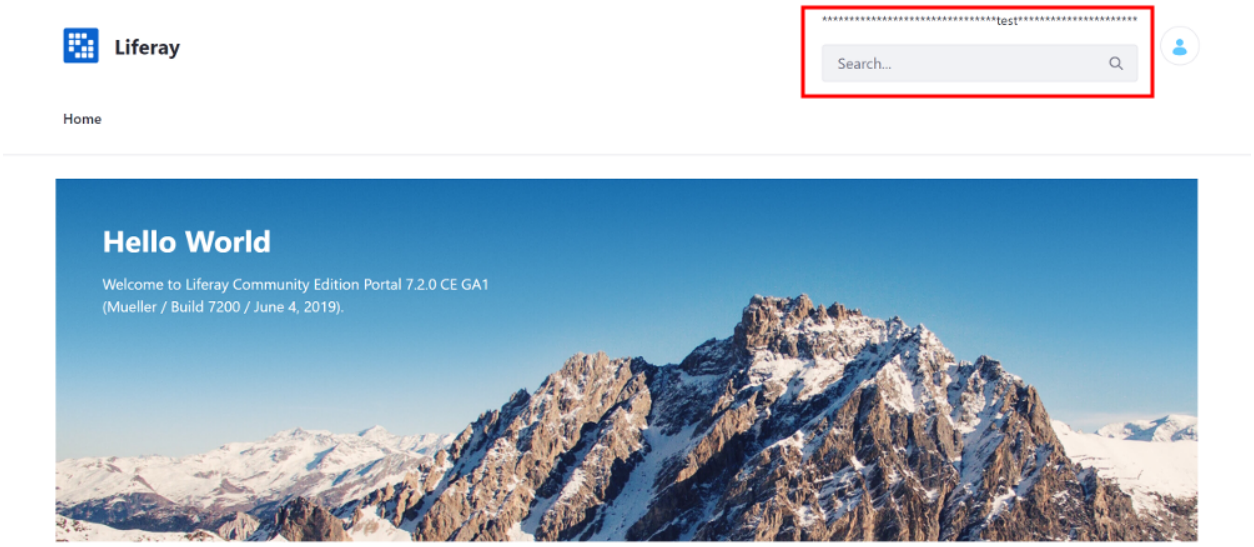

Figure 184.2: Before the `fieldset`'s nested fields, it prints *test* surrounded by asterisks.

You know how to override specific Liferay DXP core inline content using Liferay's `.jsp.readme` files.

184.2 Related Topics

- Customizing JSPs with Dynamic Includes
- JSP Overrides Using Portlet Filters

Chapter 185

Customizing Widgets

It would be nice to apply display changes to specific widget instances without having to create a hook (e.g., HTML-related change) or change a theme (e.g., CSS-related change). Ideally, you should be able to enable authorized users to apply custom display interfaces to widgets.

Be of good cheer! That's precisely what Widget Templates provide. Now you can customize the way widgets appear on a page, removing limitations to the way content is displayed. With Widget Templates, you can define display templates to render asset-centric widgets. Some default widgets already have templating capabilities (e.g., *Web Content* and *Dynamic Data Lists*), in which you can add as many display options (or templates) as you want. You can also add them to your own applications.

Some portlets that already support Widget Templates are

- *Asset Publisher*
- *Blogs*
- *Breadcrumb*
- *Categories Navigation*
- *Language Selector*
- *Media Gallery*
- *Navigation Menu*
- *RSS Publisher*
- *Site Map*
- *Tags Navigation*
- *Wiki*

To leverage the Widget Template API, follow these steps:

- register your portlet to use Widget Templates
- define your display template definitions
- define permissions
- expose the Widget Template functionality to users

The detailed steps are in the Implementing Widget Templates article. Here's a high level overview of what you'll do.

185.1 Implementing the TemplateHandler Interface

To register your portlet to use Widget Templates, you must implement the `TemplateHandler` interface. Read the interface's Javadoc for more information on each method provided by the interface.

Each of the methods in this class have a significant role in defining and implementing Widget Templates for your custom portlet. The list below highlights some of the methods defined specifically for Widget Templates:

`getClassName()`: Defines the type of entry your portlet is rendering.

`getName()`: Declares the name of your Widget Template type (typically, the name of the portlet).

`getResourceName()`: Specifies which resource is using the Widget Template (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.

`getTemplatesConfigPath()`: Defines the configuration file containing the display template definition.

Next, you must define your display template definition(s).

185.2 Defining Display Template Definitions

Once you've registered your portlet to use Widget Templates, you should create the display template definitions. These are used to style the content displayed in the widget.

You must create a `portlet-display-templates.xml` configuration file to define the definitions and point to their styled templated (e.g., FreeMarker). Then you must create the templates. These template definitions are available to apply from a widget's Configuration menu.

Next, you define permissions for your portlet's Widget Templates.

185.3 Defining Permissions

You must define permissions for your Widget Templates; without permissions, anyone in the Site could access and change your widget's display templates. Configuring permissions lets administrative users grant permissions only to the Roles that should create and manage display templates.

This is done by creating a `default.xml` file in your portlet defining the permissions you want to enforce, wiring it up with your portlet, and configuring them for use in Liferay DXP. You can visit this article for step-by-step instructions on how to complete this.

Next, you'll learn how to expose Widget Template selection for users.

185.4 Exposing the Widget Template Selection

To expose the Widget Template option to your users, use the `<liferay-ui:ddm-template-selector>` tag in the JSP file that controls your portlet's configuration. This tag requires the following parameters:

`className`: your entity's class name.

`contextObjects`: accepts a `Map<String, Object>` with any object you want to the template context.

`displayStyle`: your portlet's display style.

`displayStyleGroupId`: your portlet's display style group ID.

`entries`: accepts a list of your entities (e.g., `List<YourEntity>`).

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.

185.5 Recommendations for Using Widget Templates

You can harness a lot of power by leveraging the Widget Template API. Be careful, for with great power, comes great responsibility! Here are 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 Widget Templates can perform. Liferay DXP provides some 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.

Widget 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 their presentation, while using the existing API for complex operations. The best way to make Widget 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 DXP's advanced template editor!

To navigate to the template editor for Widget Templates, go to the Site Admin menu and select *Configuration → Widget Templates* and then click *Add* and select the specific portlet on which you decide to create a custom template.

The template editor provides fields, general variables, and utility variables customized for the portlet you chose. These variable references are on the left-side panel of the template editor. Place your cursor where you want the variable placed and click the desired variable to insert it. You can learn more about the template editor in Styling Widgets with Widget Templates.

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*.

Widget Templates provide power to your portlets by providing infinite ways of editing your portlet to create new interfaces for your users. Be sure to configure your FreeMarker templates appropriately for the most efficient customization process.

Continue on to add support for Widget Templates in your portlet.

CHAPTER 186

IMPLEMENTING WIDGET TEMPLATES

Widget Templates are ways to customize how a widget looks. You can create templates for a widget's display and then choose which template is active.

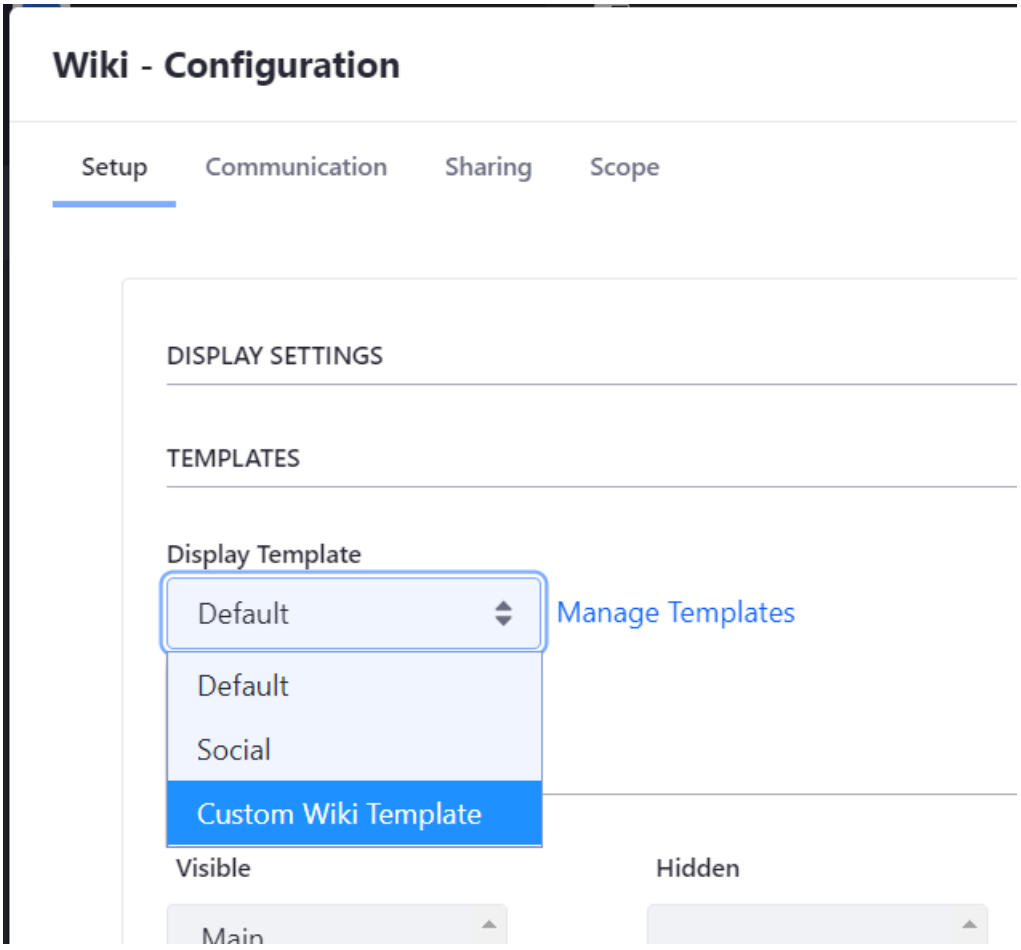

Figure 186.1: By using a custom display template, your portlet's display can be customized.

To add Widget Template support to your portlet, follow the steps below.

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
   )
   ```

   The Site Map widget sets the `@Component` annotation like this:

   ```
   @Component(
       immediate = true,
       property = "javax.portlet.name=" + SiteNavigationSiteMapPortletKeys.SITE_NAVIGATION_SITE_MAP,
       service = TemplateHandler.class
   )
   public class SiteNavigationSiteMapPortletDisplayTemplateHandler
       extends BasePortletDisplayTemplateHandler {
   }
   ```

   You'll continue stepping through the Site map widget's `TemplateHandler` implementation next.

2. Override the base class' `getClassName()`, `getName(...)`, and `getResourceName()` methods:

   ```
   @Override
   public String getClassName() {
       return LayoutSet.class.getName();
   }

   @Override
   public String getName(Locale locale) {
       String portletTitle = _portal.getPortletTitle(
           SiteNavigationSiteMapPortletKeys.SITE_NAVIGATION_SITE_MAP,
           ResourceBundleUtil.getBundle(locale, getClass()));

       return LanguageUtil.format(locale, "x-template", portletTitle, false);
   }

   @Override
   public String getResourceName() {
       return SiteNavigationSiteMapPortletKeys.SITE_NAVIGATION_SITE_MAP;
   }
   ```

   These methods return the template handler's class name, the template handler's name (via resource bundle), and the resource name associated with the Widget Template, respectively.

3. Override the `getTemplateVariableGroups(...)` method to return your widget template's script variable groups. These are used to display hints in the template editor palette.

```
@Override
public Map<String, TemplateVariableGroup> getTemplateVariableGroups(
        long classPK, String language, Locale locale)
    throws Exception {

    Map<String, TemplateVariableGroup> templateVariableGroups =
        super.getTemplateVariableGroups(classPK, language, locale);

    TemplateVariableGroup templateVariableGroup =
        templateVariableGroups.get("fields");

    templateVariableGroup.empty();

    templateVariableGroup.addCollectionVariable(
        "pages", List.class, PortletDisplayTemplateConstants.ENTRIES,
        "page", Layout.class, "curPage", "getName(locale)");
    templateVariableGroup.addVariable(
        "site-map-display-context",
        SiteNavigationSiteMapDisplayContext.class, "siteMapDisplayContext");

    return templateVariableGroups;
}
```

For this example, the *Pages* and *Site Map Display Context* fields are added to the default variables in the template editor palette.

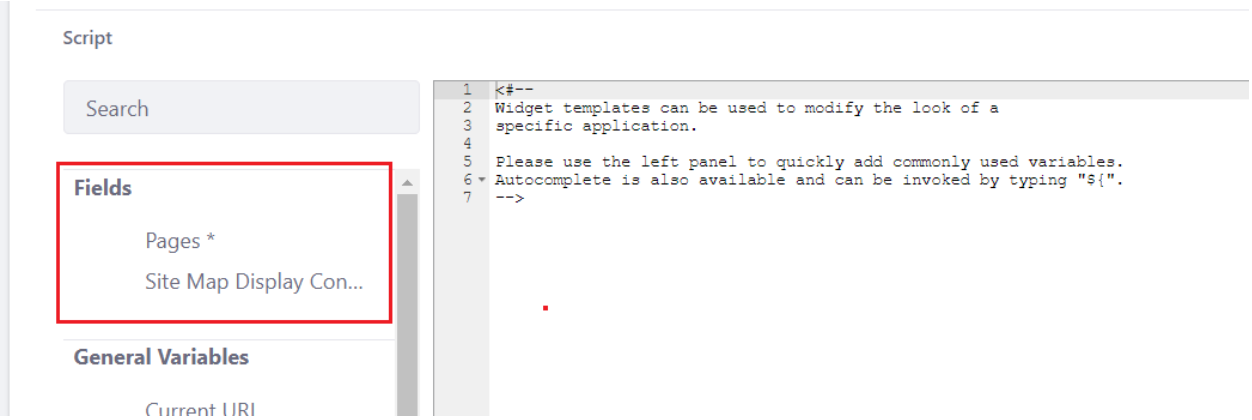

Figure 186.2: You can click a variable to add it to the template editor.

4. Set your display template configuration file path:

```
@Override
protected String getTemplatesConfigPath() {
    return "com/liferay/site/navigation/site/map/web/portlet/template" +
        "/dependencies/portlet-display-templates.xml";
}
```

This method returns the XML file containing the display template definitions available for your portlet. You'll create this file next.

5. Create your `portlet-display-templates.xml` file to define your display template definitions. For example,

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

<root>
    <template>
        <template-key>site-map-multi-column-layout-ftl</template-key>
        <name>portlet-display-template-name-multi-column-layout</name>
        <description>portlet-display-template-description-multi-column-layout-sitemap</description>
        <language>ftl</language>
        <script-file>com/liferay/site/navigation/site/map/web/portlet/template/dependencies/portlet_display_template_multi_column_layout.ftl<
file>
        <cacheable>false</cacheable>
    </template>
</root>
```

This defined template option is read and presented to the user through the widget's Configuration menu. Navigate to the Site Map widget's Configuration menu and you can confirm the *Multi Column Layout* option is available.

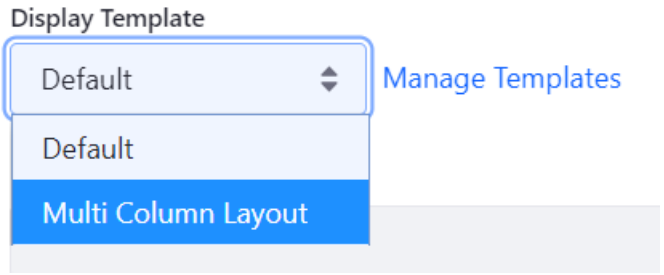

Figure 186.3: You can choose the Widget Template you want to apply from the widget's Configuration menu.

This template is created using FreeMarker. You'll create this template option next.

6. Create your template script file that you specified in the previous step. For the Site Map widget, its Multi Column Layout option is configured in a FreeMarker template like this:

```
<#if entries?has_content>
    <@liferay_aui.row>
        <#list entries as entry>
            <#if layoutPermission.containsWithoutViewableGroup(permissionChecker, entry, "VIEW")>
                <@liferay_aui.col width=25>
                    <div class="results-header">
                        <h3>
                            <a

                            <#assign layoutType = entry.getLayoutType() />

                            <#if layoutType.isBrowsable()>
                                href="${portalUtil.getLayoutURL(entry, themeDisplay)}"
                            </#if>

                            >${entry.getName(locale)}</a>
                        </h3>
                    </div>

                    <@displayPages
                        depth=1
                        pages=entry.getChildren(permissionChecker)
                    />
                </@liferay_aui.col>
            </#if>
```

```
        </#list>
    </@liferay_aui.row>
</#if>

<#macro displayPages
    depth
    pages
>
    <#if pages?has_content && ((depth < displayDepth?number) || (displayDepth?number == 0))>
        <ul class="child-pages">
            <#list pages as page>
                <li>
                    <a

                    <#assign pageType = page.getLayoutType() />

                    <#if pageType.isBrowsable()>
                        href="${portalUtil.getLayoutURL(page, themeDisplay)}"
                    </#if>

                    >${page.getName(locale)}</a>

                    <@displayPages
                        depth=depth + 1
                        pages=page.getChildren(permissionChecker)
                    />
                </li>
            </#list>
        </ul>
    </#if>
</#macro>
```

This template definition enforces page permissions, formats how the pages are displayed (multi column), and provides clickable links for each page.

7. Your widget must define permissions for creating and managing 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 7.2.0//EN" "http://www.liferay.com/dtd/liferay-
resource-action-mapping_7_2_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>
```

8. If your widget hasn't defined Liferay permissions before, 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
```

9. Now expose the Widget Template selector to your users. Include the `<liferay-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 a `<liferay-frontend:fieldset>` in your configuration JSP file like this:

```
<liferay-frontend:fieldset
    collapsible="<%= true %>"
    label="templates"
>
    <div class="display-template">
        <liferay-ddm:template-selector
            classNameId="<%= YourEntity.class.getName() %>"
            displayStyle="<%= displayStyle %>"
            displayStyleGroupId="<%= displayStyleGroupId %>"
            refreshURL="<%= PortalUtil.getCurrentURL(request) %>"
            showEmptyOption="<%= true %>"
        />
    </div>
</liferay-frontend:fieldset>
```

   In this JSP, the `<liferay-ddm:template-selector>` tag specifies the Display Template drop-down menu to be used in the widget's Configuration menu.

10. You must now extend your view code to render your portlet using the selected Widget Template.

    First, initialize the Java variables needed for the Widget Template:

```
<%
String displayStyle = GetterUtil.getString(portletPreferences.getValue("displayStyle", StringPool.BLANK));
long displayStyleGroupId = GetterUtil.getLong(portletPreferences.getValue("displayStyleGroupId", null), scopeGroupId);
%>
```

    Next, you can test if the Widget Template is configured, grab the entities to be rendered, and render them using the Widget Template. 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 renders the default view 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.

    As an example, the Site Map widget implements the `<liferay-ddm:template-renderer>` tag in its `view.jsp` like this:

```
<liferay-ddm:template-renderer
    className="<%= LayoutSet.class.getName() %>"
    contextObjects="<%= contextObjects %>"
    displayStyle="<%= siteNavigationSiteMapPortletInstanceConfiguration.displayStyle() %>"
    displayStyleGroupId="<%= siteNavigationSiteMapDisplayContext.getDisplayStyleGroupId() %>"
    entries="<%= siteNavigationSiteMapDisplayContext.getRootLayouts() %>"
>
    <%= siteNavigationSiteMapDisplayContext.buildSiteMap() %>
</liferay-ddm:template-renderer>
```

This logic builds the site's navigation map when the widget is added to a page.

Awesome! Your portlet now supports Widget Templates! Once your script is uploaded 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 Widget Templates section for more details on using Widget Templates.

CHAPTER 187

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:

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.

CHAPTER 188

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 alloyeditor_bbcode alloyeditor_creole
ckeditor	ckEditor	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.

188.1 Related Topics

- Bottom JSP Dynamic Includes
- Top Head JSP Dynamic Includes
- Top JS Dynamic Include

CHAPTER 189

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");
    }

}
```

Page Source:

```
<head>
  ...
  <link href="http://localhost:8080/o/my-custom-dynamic-include/css/mentions.css" rel="stylesheet" type="text/css">
  ...
</head>
```

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.

189.1 Related Topics

- Bottom JSP Dynamic Includes
- Top JS Dynamic Include
- WYSIWYG Editor Dynamic Includes

CHAPTER 190

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"
    );
    }
}
```

Page Source:

```
<head>
  ...
  <script charset="utf-8" src="/o/my-custom-dynamic-include/my_example_javascript.js" async>...</script>
  ...
</head>
```

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.

190.1 Related Topics

- Bottom JSP Dynamic Includes
- Top Head JSP Dynamic Includes
- WYSIWYG Editor Dynamic Includes

CHAPTER 191

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.

Page Source:

```
<body class="controls-visible has-control-menu closed  yui3-skin-sam guest-site signed-in public-page site lfr-has-simulation-
panel" id="senna_surface1">
```

Now you know how to use the `bottom.jsp` dynamic includes.

191.1 Related Topics

- Top Head JSP Dynamic Includes
- Top JS Dynamic Include
- WYSIWYG Editor Dynamic Includes

CHAPTER 192

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.

192.1 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.

192.2 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.

192.3 Related Topics

Service Trackers
    Liferay DXP Startup Phases

CHAPTER 193

LIFERAY FORMS

The Liferay Forms application is a full-featured form building tool for collecting data. There's lots of built-in functionality. For the pieces you're missing, there are extension points.

This section of articles shows developers how to

1. Store form entry data in an alternative format. The default storage type is JSON.
2. [Coming Soon] Create new form field types.

193.1 Liferay Forms Extension Points

Here's a compilation of the Liferay Forms application's extension points that are ready for your customization:

- Create a Form Storage Adapter by implementing a `StorageAdapter` or by extending the Abstract implementation, `BaseStorageAdapter`.
- Create a Form Field Type by implementing a `DDMFormFieldType`, `DDMFormFieldTypeSettings`, and a `DDMFormFieldTemplateContextContributor`.
- Create custom validation rules for form fields by implementing a DDMFormFieldValueValidator.

CHAPTER 194

FORM STORAGE ADAPTERS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 Dynamic Data Mapping (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 your own implementation of 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: The `StorageAdapter` interface and it's abstract implementation, `BaseStorageAdapter`, are deprecated in 7.0. In the future your code should be migrated to implement the `DDMStorageAdapter` interface. If you need a storage adapter, the current extension of `BaseStorageAdapter` (demonstrated in this documentation), is still the way to create one, but be aware that it will not be available in a future version.

A newly added storage adapter 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.

194.1 Storage Adapter Methods

Before handling the CRUD logic, write a `getStorageType` method.

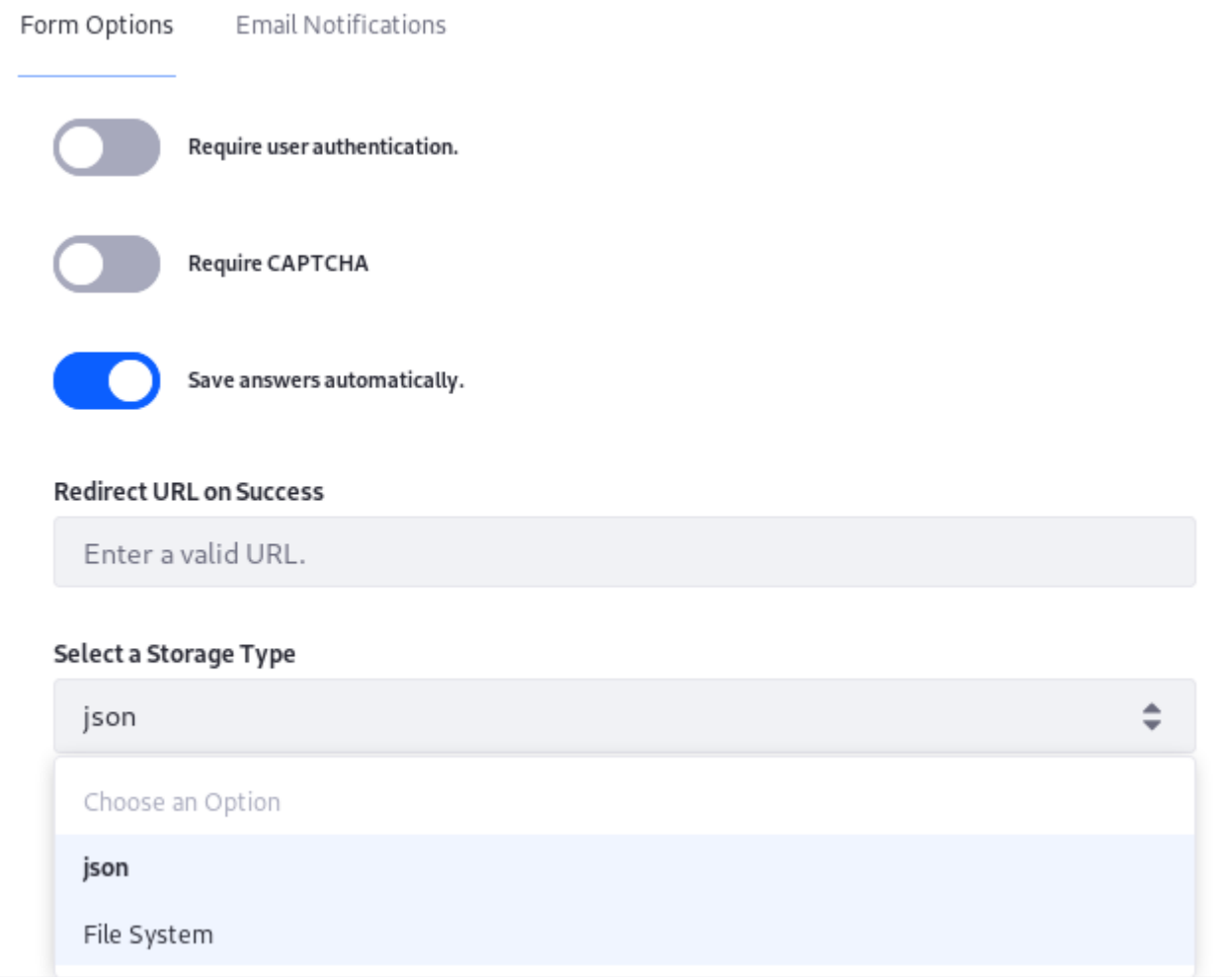

Figure 194.1: Choose a Storage Type for your form records.

`getStorageType` 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.

194.2 The CRUD Methods

`doCreate`: Return a `long` that identifies each form record with a unique file ID. Almost as important is to validate the form values being sent through the storage adapter API. This is as simple as calling `DDMFormValuesValidator.validate(ddmFormValues)`. In addition, you'll interact with at least two other DDM services to get the form the values are associated with, and to make sure they're linked: `DDMStructureVersionLocalService` and `DDMStorageLinkLocalService`. Lastly, the form values in the `DDMFormValues` object must be serialized (converted) into the right storage format. If JSON works for your use case, feel free to use the `DDMFormValuesJSONSerializer` service in the Liferay Forms code, as demonstrated in the following article. Otherwise you'll need to provide your own serialization

service for the form values.

doGetDDMFormValues Return the form values (`DDMFormValues`) for a form. You'll call the `deserialize` method after retrieving them, to take them from the storage format (e.g., JSON) to a proper `DDMFormValues` object. You can use the Liferay Forms `DDMFormValuesJSONDeserializer` if you're retrieving JSON data.

doUpdate A request to update the values comes from a User in the Liferay Forms application, so call the validator again, serialize the values into the proper format, and save them.

doDeleteByClass When a delete request is made on a form record directly, delete the form values in whatever format they're currently being stored in (this is entirely dependent on your own application of the storage adapter). In addition, retrieve and delete the DDM class storage link using `DDMStorageLinkLocalService`.

doDeleteByDDMStrcuture When a delete request is made on an entire form, delete all the form records associated with it. In addition, take the form's `ddmStructureId` and delete all the DDM structure storage links that were created for it.

194.3 Validating Form Entries

Because the Storage Adapter handles User entered data during the `add` and `update` operations, it's important to validate that the entries include only appropriate data. Add a `validate` method to the `StorageAdapter`, calling the Liferay Forms' `DDMFormValuesValidator` method to do the heavy lifting.

```
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 `boolean validateDDMFormValues` attribute from the service context.
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.

All the Java code for the logic discussed here is shown in the next article, Creating Form Storage Adapters.

194.4 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 195

CREATING A FORM STORAGE ADAPTER

This document has been updated and ported to Liferay Learn and is no longer maintained here.

There's only one class to create when implementing a Form Storage Adapter, and it extends the base `StorageAdapter` implementation.

```
@Component(service = StorageAdapter.class)
public class FileSystemStorageAdapter extends BaseStorageAdapter {
```

The only method without a base implementation in the abstract class is `getStorageType`. For file system storage, it can return `"File System"`.

```
@Override
public String getStorageType() {
    return "File System";
}
```

195.1 Storage Adapter CRUD Operations

The CRUD operations must be created to properly handle the Form Records.

195.2 Create

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;
```

These are the utility methods invoked in the create method:

```
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;
```

195.3 Read

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;
```

195.4 Update

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);
}
```

195.5 Delete

Override the `doDeleteByClass` method to delete the `File` representing the form record, using the `classPK`, and to delete the class storage links:

```
@Override
protected void doDeleteByClass(long classPK) throws Exception {
    DDMStorageLink storageLink =
        _ddmStorageLinkLocalService.getClassStorageLink(classPK);

    FileUtil.delete(getFile(storageLink.getStructureId(), classPK));

    _ddmStorageLinkLocalService.deleteClassStorageLink(classPK);
}
```

Provide form record deletion logic to be called when deleting all the records and storage links associated with a form (using its `ddmStructureId`):

```
@Override
protected void doDeleteByDDMStructure(long ddmStructureId)
    throws Exception {

    FileUtil.deltree(getStructureFolder(ddmStructureId));

    _ddmStorageLinkLocalService.deleteStructureStorageLinks(ddmStructureId);
}
```

195.6 Beyond CRUD: Validation

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);
}
```

Deploy your storage adapter and it's ready to use.

Chapter 196

Overriding Language Keys

Core and portlet module `Language*.properties` files implement site internationalization. They're fully customizable, too. This section demonstrates this in the following topics:

- Overriding Liferay's Language Keys
- Overriding a Module's Language Keys

CHAPTER 197

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.

197.1 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`
```

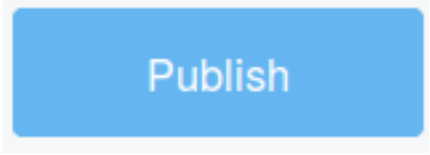

Figure 197.1: Messages displayed in Liferay's user interface can be customized.

Next, you'll learn how to override this key.

197.2 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.

197.3 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.

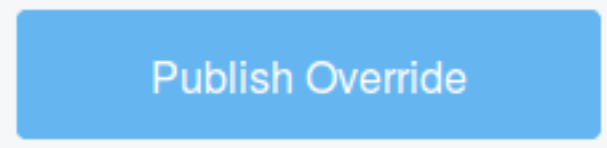

Figure 197.2: This button uses the overridden `publish` key.

That's all there is to overriding Liferay's language keys.

197.4 Related Topics

- Upgrading Core Language Key Hooks
- Overriding a Module's Language Keys

CHAPTER 198

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

198.1 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: `4.0.16`
- 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 4.0.16 module is in `com.liferay.blogs.web-4.0.16.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.

198.2 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.

198.3 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.b
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.

198.4 Related Topics

- Upgrading Core Language Key Hooks
- Overriding Global Language Keys

CHAPTER 199

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, use Blade CLI to create a `servicewrapper` project type with the command below (replace the class and package names with your own):

```
blade create -t service-wrapper -p com.liferay.docs.serviceoverride
-c UserLocalServiceOverride -s
com.liferay.portal.kernel.service.UserLocalServiceWrapper service-override
```

As an example, here's the `UserLocalServiceOverride` class that's generated with the Service Wrapper Template:

```
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 Service Builder.

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!

199.1 Related Topics

- Upgrading Service Wrappers
- Installing Blade CLI
- Creating Projects with Blade CLI

CHAPTER 200

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 .lpkg file. Here are the steps:

1. Shut down Liferay DXP.
2. Create a folder called `override` in the `Liferay Home]/osgi/marketplace` folder.
3. Name your updated .jar the same as the .jar in the original .lpkg, minus the version information. For example, if you're overriding the `com.liferay.amazon.rankings.web-1.0.5.jar` from the `Liferay CE Amazon Rankings.lpkg`, you'd name your .jar `com.liferay.amazon.rankings.web.jar`.
4. Copy this .jar into the `override` folder you created in step one.

This works for applications from Marketplace, but there's also the static .lpkg that contains core Liferay technology and third-party utilities (such as the servlet API, Apache utilities, etc.). 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.

Chapter 201

Overriding Liferay 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

This section demonstrates each MVC command customization option. Since the steps for adding logic are generally the same across MVC command types, start with adding logic.

CHAPTER 202

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

202.1 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.

202.2 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.

202.3 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.

202.4 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.

CHAPTER 203

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.

203.1 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.

203.2 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.

203.3 Related Topics

- Adding Logic to MVC Commands
- Converting StrutsActionWrappers to MVCCommands

CHAPTER 204

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.

204.1 Related Topics

- Adding Logic to MVC Commands
- Overriding MVCRenderCommands
- Converting StrutsActionWrappers to MVCCommands

CHAPTER 205

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.

205.1 Related Topics

- Adding Logic to MVC Commands
- Overriding MVCRenderCommands

CHAPTER 206

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.

CHAPTER 207

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.

207.1 Gathering Information on a Service

1. Since component service references are extension points, start with determining 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.

207.2 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).

207.3 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;
```

207.4 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!

207.5 Related Topics

- OSGi Services and Dependency Injection with Declarative Services
- Gogo Shell

CHAPTER 208

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` implements service interface (from sample module `overriding-service-reference`) `SomeService`, declares itself an OSGi service of the `SomeService` service type, and even delegates work to the existing service. Examine the example code below 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.

208.1 Related Topics

OSGi Services and Dependency Injection with Declarative Services

CHAPTER 209

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 article 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 apply the steps below to configure your own customization.

- `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.

209.1 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.

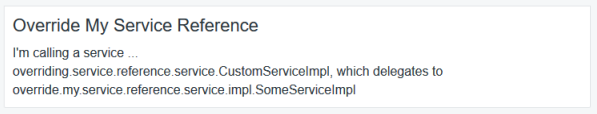

Figure 209.1: Because the example component's service reference is overridden by the configuration file deployment, the portlet indicates it's calling the custom service.

Liferay DXP processed the configuration file and injected the service reference, which in turn bound the custom service to the referencing component!

209.2 Related Topics

- OSGi Services and Dependency Injection with Declarative Services
- Using Felix Gogo Shell

CHAPTER 210

PORTLET FILTERS

Portlet filters intercept requests and responses at the start of the portlet request processing phase. Portlet filters are commonly used for these things:

- Transform content
- Add or modify request and response attributes
- Suspend a portlet phase to get user input
- Audit portlet activity

The `javax.portlet.filter` package defines a portlet filter interface for each phase. Here are the steps for developing a portlet filter:

1. Implement the portlet filter interface for the phase it's intercepting. Here are common interface methods to override:

   `doFilter`: Here's where you take action. This method is invoked at the start of the portlet request processing phase. The request and response parameters provide access to portlet content and attributes. The `FilterChain` parameter can be used to invoke the next filter in the phase.

   `init`: Initialize the filter. The `FilterConfig` parameter can be used to prepare the filter.

   `destroy`: Perform any filter cleanup.

2. Target the desired portlet(s).

3. Choose how to prioritize the filter among other filters in the phase:

   - OSGi Declarative Service Component portlet filters use a service ranking property. High ranking filters execute before lower ones.
   - `<filter-mapping>` element order in a portlet application's `portlet.xml` file.
   - The `ordinal` element value of a filter class annotated with `@PortletLifecycleFilter`. Low ordinal value filters execute before higher ones.

Below is demonstrated applying multiple filters to a portlet's render phase. The filters are OSGi Declarative Service (DS) Components, but filters can also be applied to a portlet using a `portlet.xml` descriptor or a `@PortletLifecycleFilter` annotation. See the Portlet 3.0 Specification for details. The sample code is available here.

210.1 Sample Portlet

The sample portlet `MembersListPortlet` is a Liferay MVC Portlet that lists names and email addresses when users click its *Load Users* button. The information is based on `Person` objects that the portlet class passes to the View template via a request attribute called `MembersListPortlet.MEMBERLIST_ATTRIBUTE`.

```
public void loadUsers(ActionRequest actionRequest, ActionResponse actionResponse) {

    actionRequest.setAttribute(MembersListPortlet.MEMBERLIST_ATTRIBUTE, createStaticUserList());
}
```

Two render filters are applied to the portlet:

1. Render filter 1 hides parts of the user email addresses (e.g., for privacy) by modifying the request object.
2. Render filter 2 logs portlet render phase statistics.

Adding the `MemberList` portlet to a page and clicking the `Load Users` button renders each `Person`'s name and partially hidden email address, thanks to the filter `EncodingPersonEmailsRenderFilter`.

```
Sievert Shayne

Sievert.Sha...@...mple.com

Vida Jonas

Vida.Jo...@...mple.com
...
```

If you set the portlet's log level to `debug`, it prints the render phase statistics.

```
Portlet com_liferay_code_samples_portal_modules_applications_portlets_render_filter_MembersListPortlet rendered in 7791 ms
Portlet com_liferay_code_samples_portal_modules_applications_portlets_render_filter_MembersListPortlet rendered 2 times with an average 356135 ms re
```

The first filter modifies portlet content via the request object.

210.2 Render filter 1 hides parts of user email addresses

`EncodingPersonEmailsRenderFilter` is a `RenderFilter` that hides parts of user email addresses by modifying a request attribute. Here is the class:

```
@Component(
        immediate = true,
        property = {
                "javax.portlet.name=" + MembersListPortlet.MEMBERSLIST_PORTLET_NAME,
                "service.ranking:Integer=1"
        },
        service = PortletFilter.class
)
public class EncodingPersonEmailsRenderFilter implements RenderFilter {

    @Override
    public void doFilter(RenderRequest request, RenderResponse response, FilterChain chain)
            throws IOException, PortletException {
```

```
        //This is executed before the portlet render
        Optional.ofNullable((List<Person>)request.getAttribute(MembersListPortlet.MEMBERLIST_ATTRIBUTE))
                .ifPresent(personList ->
                        request.setAttribute(MembersListPortlet.MEMBERLIST_ATTRIBUTE, ofuscateEmails(personList)));

        // Invoke the rest of the filters in the chain
        //  (it also invokes the Portlet render method if this is the last filter in the chain
        chain.doFilter(request, response);

    }

    private List<Person> ofuscateEmails(List<Person> list) {
        return list.stream()
                .map(this::ofuscatePersonEmail)
                .collect(Collectors.toList());
    }

    private Person ofuscatePersonEmail(Person person) {
        return new Person(person.getName(),
                          person.getEmail().replaceFirst("(.+)(...)@(...)(.*)", "$1...@...$4"));

    }

    @Override
    public void init(FilterConfig filterConfig) throws PortletException {

    }

    @Override
    public void destroy() {

    }
}
```

The `@Component` annotation declares the filter to be an OSGi DS Component. Here are its elements and properties:

`immediate = true` sets the component ready to start upon being installed.

`service = PortletFilter.class` defines the component to be a `PortletFilter` service.

`javax.portlet.name = + MembersListPortlet.MEMBERSLIST_PORTLET_NAME` links the filter to the target portlet. Note, multiple portlets can be listed.

`service.ranking:Integer=1` sets the filter to execute after filters that are ranked higher than 1.

`EncodingPersonEmailsRenderFilter` *implements* the `RenderFilter` interface, overriding the `doFilter`, `init`, and `destroy` methods.

`doFilter` modifies the attribute `MembersListPortlet.MEMBERLIST_ATTRIBUTE`'s list of `Persons` by replacing parts of their email addresses with ellipses (...). It delegates the `ofuscatePersonEmail` method to do the modifications. Then `doFilter` invokes `chain.doFilter(request, response)` to execute the next `RenderFilter` or next portlet processing phase.

Note: Filters can also intercept and block the execution of a portlet phase. In the `doFilter` method, this is usually done by throwing an exception or by not calling the next element in the filter chain.

210.3 RenderFilter 2 Logs Statistics

MembersListStatsRenderFilter is a RenderFilter that logs the number of times the portlet is rendered and the average render time. Here's the code:

```
@Component(
        immediate = true,
        property = {
                "javax.portlet.name=" + MembersListPortlet.MEMBERSLIST_PORTLET_NAME,
                "service.ranking:Integer=100"
        },
        service = PortletFilter.class
)
public class MembersListStatsRenderFilter implements RenderFilter {

    //Thread safe - accumulator that keeps the number of times the portlet has been rendered
    private final LongAdder hits = new LongAdder();

    //Thread safe accumulator that keeps total time spent rendering the portlet.
    private final LongAdder accumulatedTimeMs = new LongAdder();

    @Override
    public void doFilter(RenderRequest request, RenderResponse response, FilterChain chain) throws IOException, PortletException {

        long startTime = System.nanoTime();

        chain.doFilter(request, response);

        long renderTime = (System.nanoTime() - startTime) / 1000;
        hits.increment();
        accumulatedTimeMs.add(renderTime);

        if (LOG.isDebugEnabled()) {
            long totalHits = hits.longValue();
            long averageRenderTimeNs = accumulatedTimeMs.longValue() / totalHits;
            LOG.debug("Portlet " + MembersListPortlet.MEMBERSLIST_PORTLET_NAME + " rendered in " + renderTime + " ms");
            LOG.debug("Portlet " + MembersListPortlet.MEMBERSLIST_PORTLET_NAME + " rendered " + hits.longValue()
                    + " times with an average " + averageRenderTimeNs + " ms render time");
        }
    }

    ...

    private static final Log LOG = LogFactoryUtil.getLog(MembersListStatsRenderFilter.class);
}
```

As with EncodingPersonEmailsRenderFilter, it's an OSGi DS Component that is a PortletFilter service, starts upon installation, applies to the MembersListPortlet, and has a service ranking. Since its ranking is 100, it is executed before render filter EncodingPersonEmailsRenderFilter.

MembersListStatsRenderFilter's doFilter() method audits the render phase in these ways:

1. Notes the render phase start time.
2. Executes chain.doFilter(request, response) to invoke all of the other RenderFilters in the FilterChain.
3. Increments the number of times the portlet renders.
4. Calculates the average render time.
5. Logs the times rendered and average render time.

Consider creating your own filters to intercept portlet processing phases.

210.4 Related Topics

Portlets
    JSP Overrides Using Portlet Filters
    Liferay MVC Portlet

CHAPTER 211

PRODUCT NAVIGATION

Liferay DXP'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. Customizing 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
- User Personal Menu

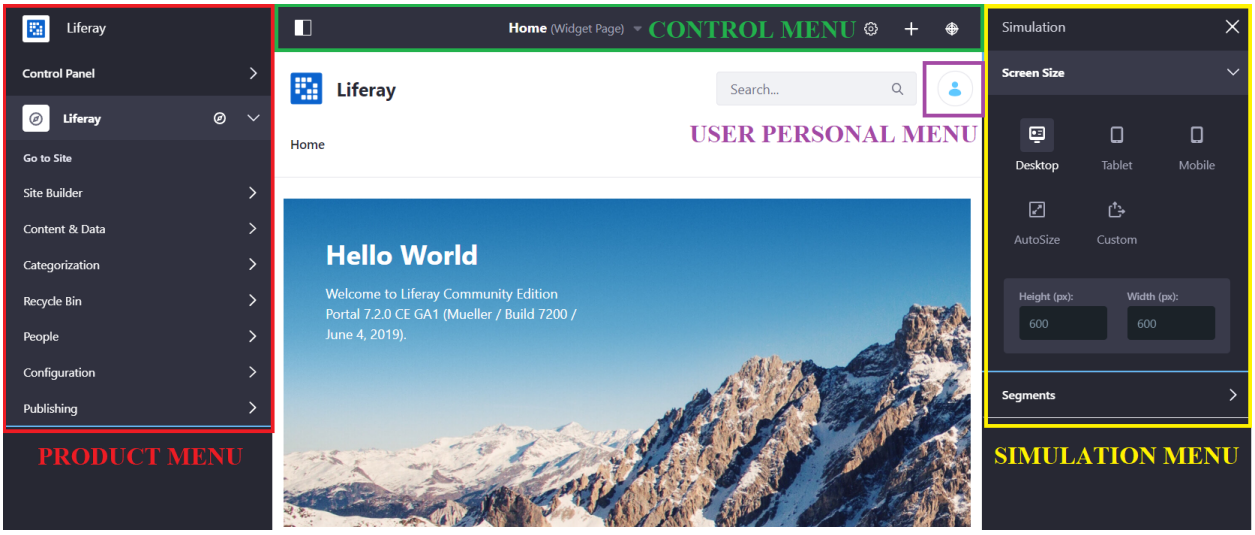

Figure 211.1: The main product navigation menus include the Product Menu, Control Menu, Simulation Menu and User Personal Menu.

The Product Menu is on the left, and displays the Control Panel 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.

You'll learn more about each of these product navigation sections next.

211.1 Product Menu

By default, Liferay's Product Menu consists of two main sections: Control Panel 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 six child Panel Categories: *Users*, *Sites*, *Apps*, *Configuration*, and *Workflow*. Clicking a child Panel Category shows *panel apps*.

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. You'll learn how to provide new or modified Panel Categories and Panel Apps for the Product Menu. For more information, read the Customizing the Product Menu articles.

211.2 Control Menu

The Control Menu is the most visible and accessible menu. 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 DXP.

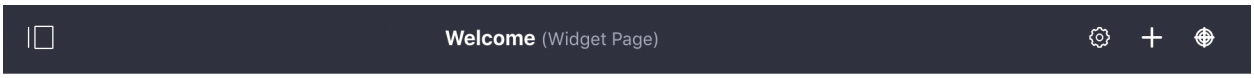

Figure 211.2: The Control Menu has three configurable areas: left, right, and middle. It also displays the title and type of page that you are currently viewing.

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 & Data* → *Web Content*, you see a Control Menu with different functionality tailored for that option.

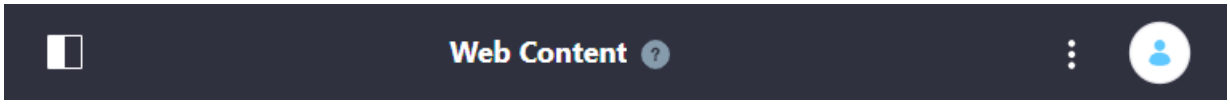

Figure 211.3: When switching your context to web content, the Control Menu adapts to provide helpful options for that area.

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. For more information, read the Customizing the Control Menu articles.

211.3 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.

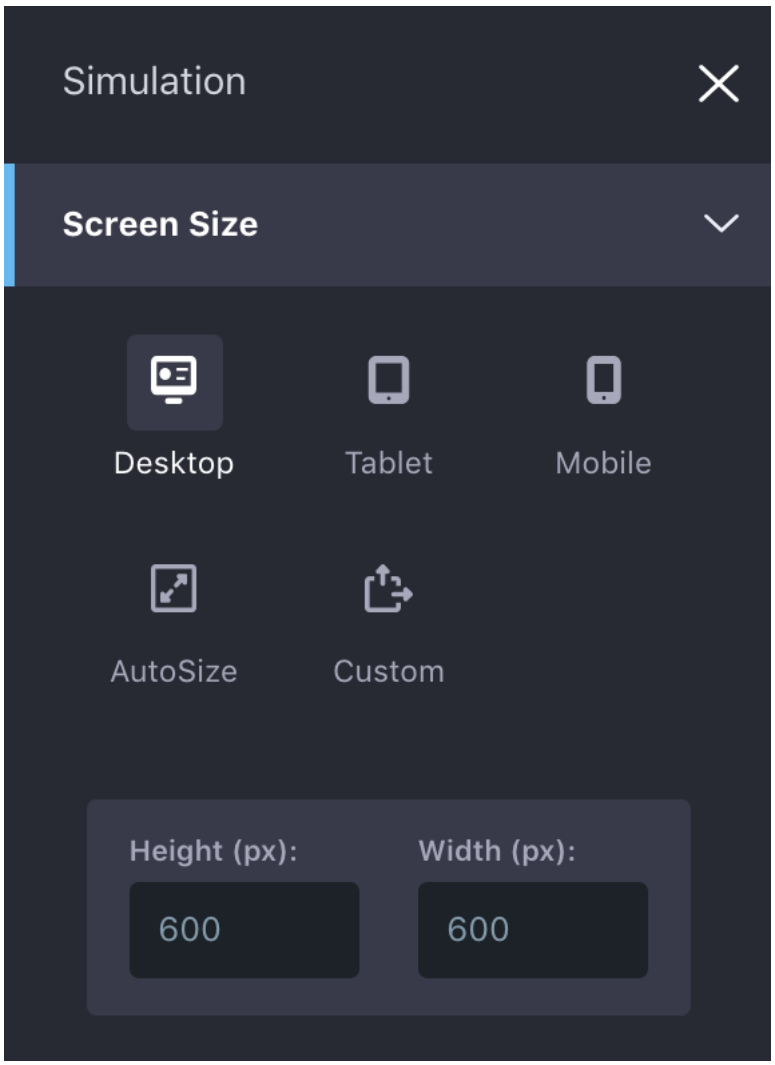

Figure 211.4: The Simulation Menu offers a device preview application.

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.

For more information, read the Extending the Simulation Menu article.

211.4 User Personal Menu

The User Personal Menu displays options unique to the current user. By default, this menu appears as an avatar button that expands the User Settings sub-menu just below the Control Menu. In a custom theme, the User Personal Menu could appear anywhere in the interface.

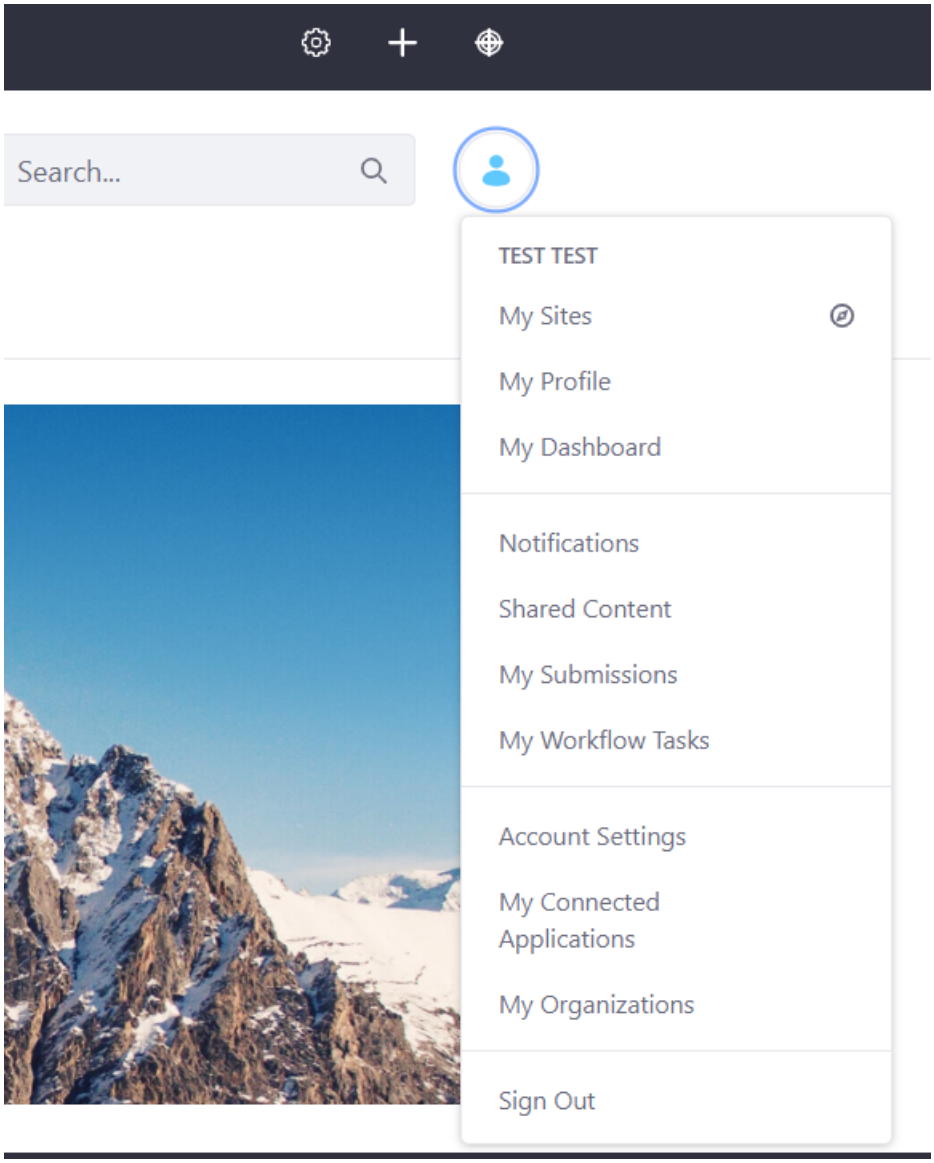

Figure 211.5: By default, the User Personal Menu contains the signed-in user's avatar, which opens the user's settings when selected.

Although Liferay's default User Personal Menu 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. It uses another common Liferay framework for providing functionality: Portlet Providers.

The User Personal Menu 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 />
```

For more information, read the Customizing the User Personal Bar and Menu article.

CHAPTER 212

CUSTOMIZING THE PRODUCT MENU

Customizing the Product Menu can be completed by adding Panel Categories and 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 article for more details.

To create these entities, you must implement the `PanelCategory` and `PanelApp` interfaces.

212.1 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.

Note: In this article, example JSPs describe how to provide functionality to Panel Categories and Panel Apps. JSPs, however, are not the only way to provide front-end functionality to your cat-

egories/apps. You can create your own class implementing `PanelCategory` to use other technologies such as FreeMarker.

More information on provided base classes for your `PanelCategory` implementation are described next.

212.2 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");
}
```

212.3 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).

Next, you'll learn about the `PanelApp` interface.

212.4 PanelApp Interface

The `PanelApp` interface requires you to implement the following methods:

- `getNotificationCount`: returns the number of notifications for the user.
- `getPortlet`: returns the portlet associated with the application.
- `getPortletId`: returns the portlet's ID associated with the application.
- `getPortletURL`: returns the URL used to render a portlet based on the servlet request attributes.
- `include`: Returns `true` if the application successfully renders.
- `setGroupProvider`: sets the group provider associated with the application.
- `setPortlet`: sets the portlet associated with the application.

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 `BasePanelApp` or `BaseJSPPanelApp` abstract classes. 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.

Note: JSPs are not the only way to provide front-end functionality to your Panel Apps. You can create your own class implementing `PanelApp` to use other technologies such as FreeMarker.

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.

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);
}
```

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.

Continue on the learn about creating custom Panel Categories and Panel Apps.

CHAPTER 213

ADDING CUSTOM PANEL CATEGORIES

As you navigate the Product Menu, you can see that Panel Apps like *Web Content* and *Settings* are organized into Panel Categories such as *Content & Data* and *Configuration*. This article 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 Panel Category.

213.1 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`).

213.2 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 directly above the class's declaration:

```
@Component(
    immediate = true,
    property = {
        "panel.category.key=" + [Panel Category Key],
        "panel.category.order:Integer=[int]"
    },
    service = PanelCategory.class
)
```

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
)
```

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.

```
**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. For example, the Product Menu's two main
sections---Control Panel and Site Administration---have
`panel.category.order:Integer` properties of 100 and 200, respectively. A
new panel inserted between Control Panel and Site Administration would
need a `panel.category.key` of ROOT and a `panel.category.order:Integer`
of 150.
```

```
Finally, your `service` element should specify the `PanelCategory.class`
service.
```

2. Implement the `PanelCategory` interface. See the `PanelCategory` Interface section for more details. Extending one of the provided base classes (BasePanelCategory or BaseJSPPanelCategory) is a popular way to implement the `PanelCategory` interface.

3. If you elect to leverage JSPs, 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 how to add a Panel App to a Panel Category next.

CHAPTER 214

ADDING CUSTOM PANEL APPS

After you have created a Panel 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
   )
   ```

   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
   )
   ```

   These properties and attributes are similar to those discussed in the previous article. 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 & Data*, 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`.

4. Implement the `PanelApp` interface. See the `PanelApp` Interface section for more details. Extending one of the provided base classes (BasePanelApp or BaseJSPPanelApp) is a popular way to implement the `PanelApp` interface. See the PanelApp Interface section for more information.

5. If you elect to leverage JSPs, 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);
}
```

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 Panel Apps, it also gives you the flexibility to add a more complex UI to the Product Menu using any technology.

CHAPTER 215

CUSTOMIZING THE CONTROL MENU

Liferay's Control Menu consists of three main sections: Sites (left portion), Tools (middle portion), and User (right portion).

Figure 215.1: This image shows where your entry will reside depending on the category you select.

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 Menu is slightly different. Visit the Customizing the User Personal Bar and Menu article for more information.

You can reference a sample Control Menu Entry by visiting the Control Menu Entry article.

215.1 ProductNavigationControlMenuEntry Interface

To create a control menu entry, you must implement the `ProductNavigationControlMenuEntry` interface. It's recommended to implement this interface by extending the `BaseProductNavigationControlMenuEntry` or `BaseJSPProductNavigationControlMenuEntry` abstract classes.

These base classes are covered in more detail next.

215.2 BaseProductNavigationControlMenuEntry

Typically, the `BaseProductNavigationControlMenuEntry` is extended for basic entries 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.

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*.

215.3 BaseJSPProductNavigationControlMenuEntry

If you use JSPs for generating the UI, you can extend `BaseJSPProductNavigationControlMenuEntry` to save time when creating/modifying a control menu entry.

The `ProductMenuProductNavigationControlMenuEntry` creates an entry that appears in the *Sites* (left) area of the Control Menu. This class 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 (→ ![Menu Open](../../..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.

Next, you'll step through the process of customizing the Control Menu.

CHAPTER 216

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 article. This article assumes you know how to create a Panel Category. Creating a Control Menu Entry follows the same pattern as creating a Panel Category:

1. Create the OSGi structure and metadata.
2. Implement Liferay's Frameworks.
3. Define the Control Menu Entry.

216.1 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`).

216.2 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).

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. See the Customizing the Control Menu article for more information on these classes.

3. If you elect to leverage JSPs, you must 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);
}
```

4. 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, Control Panel*, 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;
}
```

Excellent! You've created your entry in one of the three default sections in the Control Menu.

CHAPTER 217

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.

217.1 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 documented here.

217.2 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 Localization articles.

CHAPTER 218

EXTENDING THE SIMULATION MENU

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 article covers the simpler case of creating a Panel App for the already present hidden category.

Before beginning, make sure you're accustomed to using Panel Categories and Panel Apps. This is covered in detail in the Customizing the Product Menu articles. Once you know how to create Panel Categories and Panel Apps, continue with this article.

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 article 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
   ```

   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 article 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);
}
```

Segments also provides a good example of how to extend the Simulation Menu. When segments are available, the Simulation Menu is extended to offer personalization options. You can simulate particular experiences directly from the Simulation Menu. Its Panel App class is similar to `DevicePreviewPanelApp`, except it points to a different portlet and JSP. For more information on Segments, see the Segmentation and Personalization section.

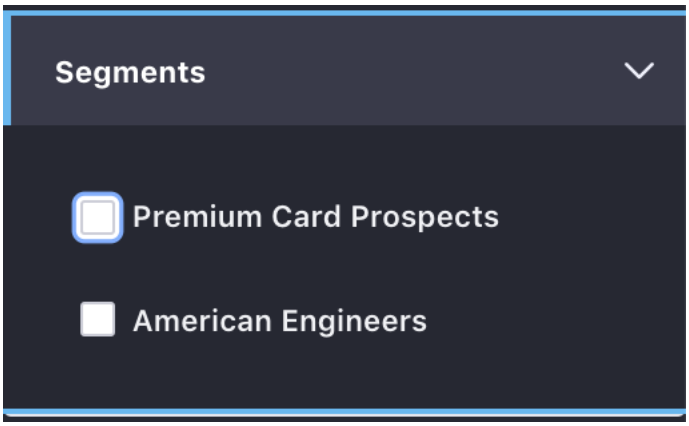

Figure 218.1: The Simulation Menu also displays Segments to help simulate different user experiences.

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.

CHAPTER 219

CUSTOMIZING THE USER PERSONAL BAR AND MENU

The User Personal Bar is a portlet, but it's also an important concept in Liferay DXP. In a fresh bundle using the default theme, it's the section of screen occupied by the User's avatar and the Personal Menu.

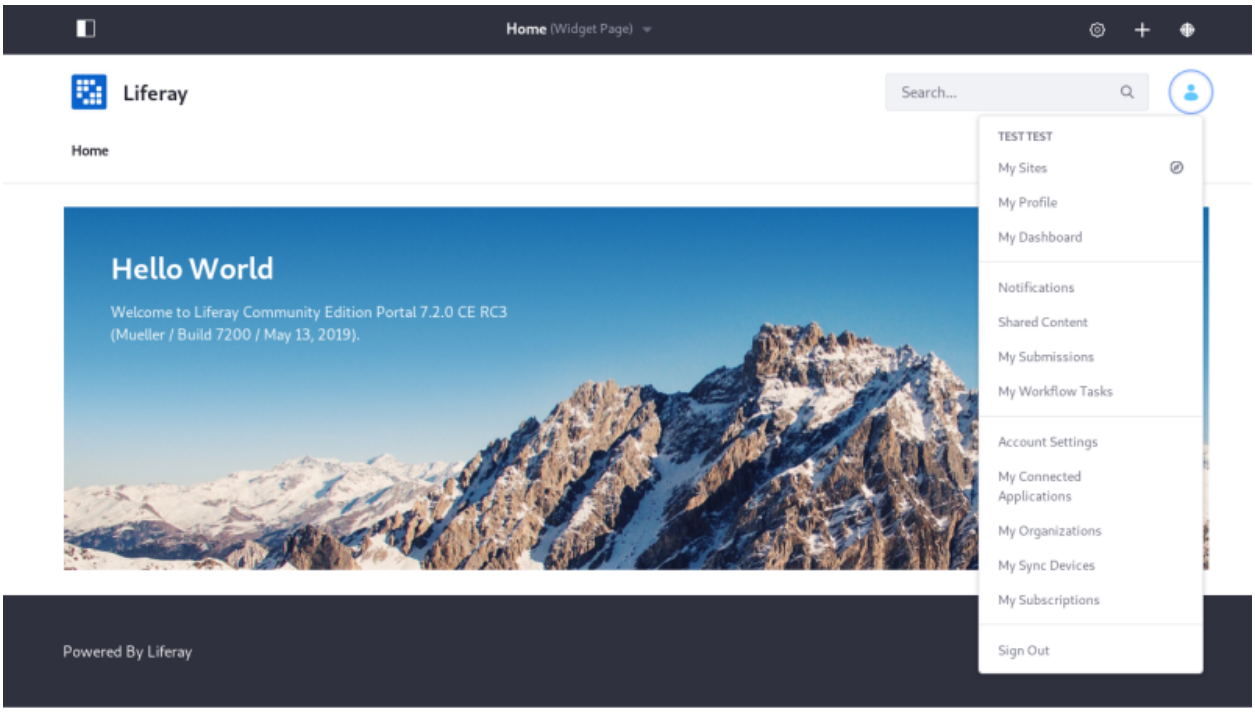

Figure 219.1: By default, the User Personal Bar contains the signed-in user's avatar, which opens the Personal Menu when selected.

The User Personal Bar holds only the Personal Menu by default, but it can also contain any functionality you want (even additional portlets). The User Personal Bar is included by default in

every Liferay theme, but you can replace it with a portlet or customize it by adding entries to the existing portlet's menu.

This section covers these topics:

- Replacing the default User Personal Bar portlet with a custom portlet.
- Customizing the default User Personal Bar.

219.1 Displaying the Personal Menu

Starting with 7.0, the Personal Menu is no longer part of the Product Menu, but is instead included in the User Personal Bar. To display the existing User Personal Bar in your own theme, embed the portlet into your theme by adding the following snippet into `portal_normal.ftl`:

```
<@liferay.user_personal_bar />
```

You'll use the same snippet even if you're replacing the default User Personal Bar portlet with your own.

If you use a custom portlet to provide the User Personal Bar, but wish to include the default Personal Menu, make sure to render it by using this tag in your portlet's JSP:

```
<liferay-product-navigation:personal-menu
    expanded="<%= true %>"
    label="<%= userAvatar %>"
/>
```

Note: The recommended way to display the Personal Menu is by embedding the User Personal Bar in a theme. If this is not practical, a workaround exists: go to *Control Panel* → *Configuration* → *Instance Settings* → *Users* and select *Personal Menu*. Enable the *Show in Control Menu* toggle and click *Update*.

This places a button to expand the Personal Menu in the Control Menu. It appears on every site and page in your virtual instance, including sites that have the User Personal Bar embedded in the theme. So, to avoid multiple User Personal Bars appearing on the page, you should use only *one* of these mechanisms to display the User Personal Bar.

Unlike the Product Menu, the Personal Menu can be customized without creating panel categories and panel apps. See Customizing the Personal Menu for details.

CHAPTER 220

USING A CUSTOM PORTLET IN PLACE OF THE USER PERSONAL BAR

In this article, you'll learn how to write the single Java class required to replace the default User Personal Bar with a custom portlet. Writing the portlet itself is up to each developer's needs. See the documentation on portlets if you need guidance.

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 may recall from the Portlet Providers articles that you can request portlets in several different ways (e.g., *Edit*, *Browse*, etc.).

   You should also specify the service rank for your new portlet so it overrides the default. 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 want to display your portlet instead of the User Personal Bar, 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 to provide when one is requested by the theme template. For example, the default portlet uses `com_liferay_product_navigation_user_personal_bar_web_portle`

If you want to inspect the entire module used for Liferay's User Personal Bar, see the product-navigation-user-personal-bar-web module.

220.1 Related Topics

- Customizing the Product Menu
- Customizing the Control Menu

CHAPTER 221

CUSTOMIZING THE PERSONAL MENU

The Personal Menu is a portlet in Liferay DXP, and is the only item occupying the User Personal Bar out of the box. You can add entries to the Personal Menu by implementing the `PersonalMenuEntry` interface. If you're adding a portlet entry to the Personal Menu, the process is slightly different. Both approaches are covered below.

221.1 Adding an Entry to the Personal Menu

Follow these steps. `SignOutPersonalMenuEntry.java` is used as an example throughout these steps:

1. Create an OSGi module and place a new Java class into a package in its `src` folder.
2. In the `@Component` annotation, specify the two properties shown below to place your new entry in the Personal Menu:

    - `product.navigation.personal.menu.group`: determines the section where the entry will be placed.
    - `product.navigation.personal.menu.entry.order`: determines the order of entries within each section. Note that sections are not labelled. To create a new section, assign the `group` property a value other than those for the four default sections (100, 200, 300, and 400).

    Here's an example:

```
@Component(
    immediate = true,
    property = {
    "product.navigation.personal.menu.group:Integer=400",
        "product.navigation.personal.menu.entry.order:Integer=100"
    },
    service = PersonalMenuEntry.class
)
public class SignOutPersonalMenuEntry implements PersonalMenuEntry {
```

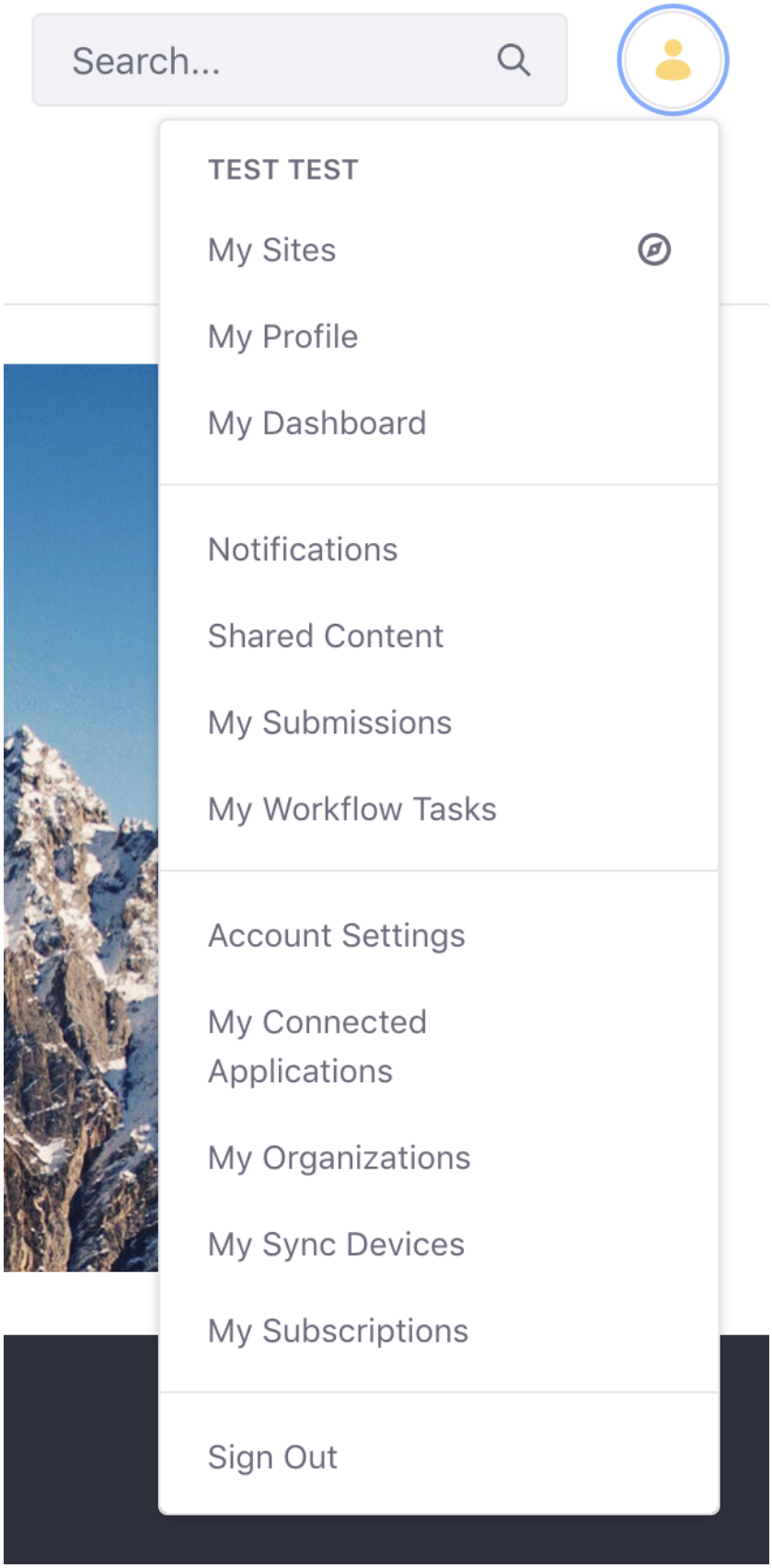

Figure 221.1: The Personal Menu is organized into four sections.

3. Include the interface's methods. `SignoutPersonalMenuEntry` uses `getLabel` and `getPortletURL`, which are the only two that are mandatory. `getLabel` retrieves a language key to label the entry in the UI:

```
@Override
public String getLabel(Locale locale) {
    return LanguageUtil.get(locale, "sign-out");
}
```

`getPortletURL` returns the URL for the portlet or page you want to access with the entry:

```
public String getPortletURL(HttpServletRequest httpServletRequest)
    throws PortalException {

    ThemeDisplay themeDisplay =
        (ThemeDisplay)httpServletRequest.getAttribute(
            WebKeys.THEME_DISPLAY);

    return themeDisplay.getURLSignOut();
}

}
```

That's all you need to implement the interface. However, the `PersonalMenuEntry` interface includes a number of other methods that you can use if you need them:

`getIcon`: identify an icon to display in the entry.

`isActive`: indicate whether the entry is currently active.

`isShow`: write logic to determine under what circumstances the entry is displayed.

Learn how to add a portlet entry to the Personal Menu next.

221.2 Adding a Portlet Entry to the Personal Menu

If you're adding a portlet to the Personal Menu, you can extend the `BasePersonalMenuEntry` class to save time. Follow these steps:

1. Create an OSGi module and place a new Java class into a package in its `src` folder.
2. In the `@Component` annotation, specify the two properties shown below to place your new entry in the Personal Menu:

    - `product.navigation.personal.menu.group`: determines the section where the entry will be placed.
    - `product.navigation.personal.menu.entry.order`: determines the order of entries within each section. Note that sections are not labelled. To create a new section, assign the `group` property a value other than those for the four default sections (100, 200, 300, and 400).

    An example is shown below:

```
@Component(
    immediate = true,
    property = {
        "product.navigation.personal.menu.entry.order:Integer=100",
        "product.navigation.personal.menu.group:Integer=300"
    },
    service = PersonalMenuEntry.class
)
public class MyAccountPersonalMenuEntry extends BasePersonalMenuEntry {
```

3. Override the `getPortletId()` method to provide the portlet's ID, as shown in the example below:

```
public class MyAccountPersonalMenuEntry extends BasePersonalMenuEntry {

    @Override
    public String getPortletId() {
        return UsersAdminPortletKeys.MY_ACCOUNT;
    }

}
```

   The `BasePersonalMenuEntry` class automatically determines the label, portlet URL, state, and visibility based on the portlet ID.

Once you've completed your implementation and deployed your module, your new entry is displayed in the personal menu.

221.3 Related Topics

- Customizing the Product Menu
- Customizing the Control Menu

CHAPTER 222

CUSTOMIZING WORKFLOW

Liferay's workflow engine calls users to participate in a review process designed for them. Out of the box, workflow makes it possible to define simple to complex business processes/workflows, deploy them, and manage them through a portal interface.

Workflow is flexible, in that you can design workflow processes in XML to suit your business needs.

In Liferay DXP version 7.2, a new set of workflow features was introduced around the concept of Workflow Metrics.

The embedded calendar that ships out of the box can be replaced by your own custom calendar service. More customization points will likely be added in the future.

CHAPTER 223

CREATING SLA CALENDARS

By default, an internal calendar assumes the SLA deadline clock should continue counting all the time: in other words, 24 hours per day, seven days per week. If you need a different calendar format, provide your own implementation of the `WorkflowMetricsSLACalendar` interface. New implementations of this service are picked up automatically by the Workflow Metrics application, so they become available as soon as the module holding the service implementation is deployed. The interface has three methods to implement:

```
public interface WorkflowMetricsSLACalendar {

    public Duration getDuration(
        LocalDateTime startLocalDateTime, LocalDateTime endLocalDateTime);

    public LocalDateTime getOverdueLocalDateTime(
        LocalDateTime nowLocalDateTime, Duration remainingDuration);

    public String getTitle(Locale locale);

}
```

If you define a new calendar, a new option becomes available in the Add SLA form, allowing you to choose from the default 24/7 calendar or any custom ones you've provided. For example, you can make the timer run for 8 hours per day, from 9-17 by a 24-hour clock, for 5 days per week. If you need to, you can even stop the calendar from counting during lunch hours!

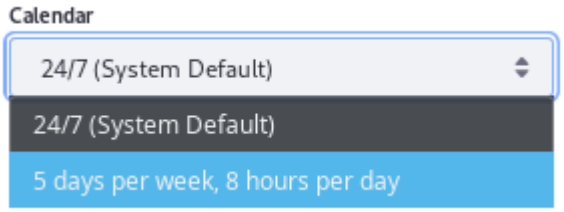

Figure 223.1: Write a Custom SLA Calendar if the default, 24/7 calendar isn't sufficient.

223.1 Dependencies

Along with some artifacts you're probably used to depending on (like `com.liferay.portal.kernel`), you'll need the `com.liferay.portal.workflow.metrics.sla.api-[version].jar` artifact. For Liferay DXP version 7.2.10-GA1, here's an example Gradle build dependency declaration:

```
compileOnly group: "com.liferay", name: "com.liferay.portal.workflow.metrics.sla.api", version: "1.1.0"
compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "4.4.0"
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"
```

223.2 Implementation Steps

Implement a `com.liferay.portal.workflow.metrics.sla.calendar.WorkflowMetricsSLACalendar` to define your own SLA calendar logic. When you're finished, use the created calendar when creating the SLA definition.

1. Declare the component and the class:

   ```
   import com.liferay.portal.kernel.language.Language;
   import com.liferay.portal.workflow.metrics.sla.calendar.WorkflowMetricsSLACalendar;
   import java.time.Duration;
   import java.time.LocalDateTime;
   import java.util.Locale;

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


   @Component(property = "sla.calendar.key=default")
   public class DefaultWorkflowMetricsSLACalendar
       implements WorkflowMetricsSLACalendar {
   ```

   The component property `sla.calendar.key` is required to identify this calendar.

2. Override `getDuration` to return the time `Duration` when elapsed SLA time should be computed. The start and end dates that this method receives represent the time a workflow task has been running. For example, given a task that started at *2019-05-13T16:00:00* and finished at *2019-05-13T18:00:00*, then The 24/7 calendar returns 2 elapsed hours, while a 9-17 weekdays calendar returns 1 hour as the elapsed time.

   ```
   @Override
   public Duration getDuration(
       LocalDateTime startLocalDateTime, LocalDateTime endLocalDateTime) {

       return Duration.between(startLocalDateTime, endLocalDateTime);
   }
   ```

3. `getOverdueLocalDateTime` must return the date (as a `LocalDateTime`) when this SLA is considered overdue given the parameter values. For example, given that `nowLocalDateTime`=*2019-05-13T17:00:00* and `remainingDuration`=*24H*, The 24/7 calendar returns a `localDateTime` of *2019-05-14T17:00:00* as the overdue date. Given the same parameters, the 9-17 weekdays calendar should return *2019-05-17T09:00:00*. The remaining duration of time left in the SLA is available in the method as a `Duration` object; your job is to write logic that considers your calendar and create a `LocalDateTime` with the proper overdue date/time.

```
@Override
public LocalDateTime getOverdueLocalDateTime(
    LocalDateTime nowLocalDateTime, Duration remainingDuration) {

    return nowLocalDateTime.plus(remainingDuration);
}
```

4. Use `getTitle` to provide the title for the given locale. Make sure you properly localize this extension by providing a `Language.properties` file and any `Language_xx.properties` files for translation of the value. At runtime, the User's locale is used to return the correct translation.

```
@Override
public String getTitle(Locale locale) {
    return _language.get(locale, "default");
}

@Reference
private Language _language;
```

If the 24/7 default calendar works for you, use it. Otherwise create your own `WorkflowMetricsSLACalendars`.

CHAPTER 224

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.x

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 Liferay DXP 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 Liferay DXP 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.

CHAPTER 225

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 Liferay Portal 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.

CHAPTER 226

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 Liferay Portal 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.

CHAPTER 227

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.

CHAPTER 228

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.

Note: You can configure an uninterrupted session by setting the `<session-timeout>` tag to `-1`. Leaving a session permanently active is a risk and is not recommended for production environments, but is useful for testing.

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.

Part III

Application Development Platform

CHAPTER 229

APPLICATION DEVELOPMENT

Writing applications on Liferay's standards-based platform makes your life easier. Whether you create headless services for clients to access, full-blown web applications with beautiful UIs, or anything in between, Liferay DXP streamlines the process to help you get your job done faster.

Liferay's framework embraces your existing tools and build environments like Maven and Gradle. You can work with the standard technologies you know and leverage Liferay's APIs for Documents, Permissions, Search, or Content when you need them. Here's a high level view of what you can do:

- **Deployment of existing standards-based apps:** If you have an existing app built outside of Liferay DXP, you can deploy it on Liferay DXP. The Liferay Bundler Generator and Liferay npm Bundler provide the project scaffolding and packaging to deploy Angular, React, and Vue web front-ends as Widgets. Spring Portlet MVC app conversion to PortletMVC4Spring requires only a few steps. JSF applications work almost as-is. Portlet 3.0 or 2.0 compliant portlets deploy on Liferay DXP.

- **Back-end Java services, web services, and REST services:** Service Builder is an object-relational mapper where you describe your data model in a single `xml` file. From this, you can generate the tables, a Java API for accessing your data model, and web services. On top of these, REST Builder generates OpenAPI-based REST services your client applications can call.

- **Authentication and single-sign on (SSO):** OAuth 2.0, OpenID Connect, and SAML are built-in and ready to go.

- **Front-end web development using Java EE and/or JavaScript:** Use Java EE standard Portlet technology (JSR 168, JSR 286, JSR 362) with CDI and/or JSF. Prefer Spring? PortletMVC4Spring brings the Spring MVC Framework to Liferay. Rather have a client-side app? Write it in Angular, React, or Vue. Been using Liferay DXP for a while? Liferay MVC Portlet is better than ever.

- **Frameworks and APIs for every need:** Be more productive by using Liferay's built-in and well-tested APIs that cover often-used features like file management(upload/download), permissions, comments, out-of-process messaging, or UI elements such as data tables and item

selectors. Liferay DXP offers many APIs for every need, from an entire workflow framework to a streamlined way of getting request parameters.

- **Tool freedom:** Liferay provides Maven archetypes, Liferay Workspace, Gradle and Maven plugins, a Yeoman-based theme generator, and Blade CLI to integrate with any development workflow. On top of that, you can use our IntelliJ plugin or the Eclipse-based Liferay Developer Studio if you need a full-blown development environment.
- **Developer community:** The Liferay DXP community is helpful and active.

229.1 Getting Started with Liferay Development

Want to see what it's like to develop an app on Liferay DXP? Here's a quick tour.

229.2 Create Your Object Model and Database in One Shot

You don't need a database to work with Liferay, but if your app uses one, you can design it and your object model at the same time with Liferay's object-relational mapper, Service Builder. You define your object model in a single `xml` file:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.2.0//EN" "http://www.liferay.com/dtd/liferay-service-
builder_7_0_0.dtd">
<service-builder auto-namespace-tables="true" package-path="com.liferay.docs.guestbook">
    <author>liferay</author>
    <namespace>GB</namespace>
    <entity name="Guestbook" local-service="true" remote-service="true" uuid="true">

        <column name="guestbookId" primary="true" type="long" />
        <column name="name" type="String" />

        <finder name="Name" return-type="Collection"/>
            <finder-column name="name" />
        </finder>

    </entity>

    <entity name="Entry" local-service="true" remote-service="true" uuid="true">

        <column name="entryId" primary="true" type="long" />
        <column name="name" type="String" />
        <column name="email" type="String" />
        <column name="message" type="String" />
        <column name="guestbookId" type="long" />

        <finder name="Email" return-type="Collection" />
            <finder-column name="email" />
        </finder>

    </entity>

</service-builder>
```

Service Builder generates your object model, database, SOAP, and JSON web services automatically. Java classes are ready for you to implement your business logic around generated CRUD operations. The web services are mapped to your business logic. If you want a REST interface, you can create one.

229.3 Create a REST Interface

REST Builder helps you define REST interfaces for your APIs, using OpenAPI/Swagger. Create your YAML definition file for your REST interface along with a configuration file defining where Java classes, a client, and tests should be generated, and you have REST endpoints ready to call your API.

Next, you need a client. You can use Liferay DXP in headless mode and write your web and mobile clients any way you want. Or you can create your web clients on Liferay's platform and take advantage of its many tools and APIs that speed up development.

229.4 Create a Web Client

Liferay DXP is an ideal platform upon which to build a web client. Its Java EE-based technology means you can pick from the best it has to offer: Spring MVC using PortletMVC4Spring, the new backwards-compatible Portlet 3, JSF using Liferay Faces, or the venerable OSGi-based Liferay MVC Portlet. If you're a front-end developer, deploy your Angular, React, or Vue-based front-end applications to run as widgets next to the rest of Liferay DXP's installed applications.

229.5 Use Liferay's Frameworks

Your apps need features. Liferay has implemented tons of common functionality you can use in your applications. The Liferay-UI tag library has tons of web components like Search Container (a sortable data table), panels, buttons, and more. Liferay's Asset Framework can publish data from your application in context wherever users need it—as a notification, a related asset, as tagged or categorized data, or as relevant data based on a user segment. Need to provide file upload/download? Use the Documents API. Need a robust permissions system? Use Liferay permissions. Want users to submit comments? Use Liferay's comments. Need to process data outside the request/response? Use the Message Bus. Should users select items from a list? Use the Item Selector.

229.6 Next Steps

So what's next? Download Liferay DXP and create your first project! Have a look at our back-end, REST Builder, and front-end docs, examine what Liferay's frameworks have to offer, and then go create the beautiful things that only you can make.

CHAPTER 230

DEVELOPING WEB FRONT-ENDS

Liferay's open development framework removes barriers so developers can write applications faster. If you already have an application, you can deploy it on Liferay DXP:

- Java-based standards (CDI, JSF, Portlets, Spring)
- Front-end standards (Angular, React, Vue)

If you plan to write a new application and deploy it on Liferay DXP, you can use the frameworks you know along with the build tools (Gradle, Maven) you know. Liferay also offers its own development framework called MVC Portlet that it uses to develop applications. When you want to integrate with Liferay services and frameworks such as permissions, assets, and indexers, you'll find that these easily and seamlessly blend with your application to provide a great user experience.

Regardless of your development strategy for applications, you'll find Liferay DXP to be a flexible platform that supports anything you need to write.

230.1 Using Popular Frameworks

Liferay gives you a head start on developing and deploying apps that use these popular Java and JavaScript-based technologies:

- Angular Widget
- React Widget
- Vue Widget
- Liferay MVC Portlet
- PortletMVC4Spring Portlet
- JSF Portlet

Note: The Reference section describes sample projects and project templates for creating UIs using other technologies.

Angular, React, and Vue applications are written the same as you would outside of Liferay DXP—using npm and the webpack dev server. The Liferay JS Generator creates a portlet bundle

(project) for developing and deploying each type of app. The bundle project comes with npm commands for building, testing, and deploying the app. It packages the app's dependencies (including JavaScript packages), deploys the bundle as a JAR, and installs the bundle to Liferay DXP's run time environment, making your app available as a widget.

You can also develop web front-ends using Java EE standards. Liferay DXP supports the JSR 362 Portlet 3.0 standard which is backwards-compatible with the JSR 286 Portlet 2.0 standard from the Java Community Process (JCP). Each portlet framework has benefits you may wish to consider.

Bean Portlet is the only framework containing all of the Portlet 3 features:

- Contexts and Dependency Injection (CDI)
- Extended method annotations
- Explicit render state
- Action, render, and resource parameters
- Asynchronous support

If you're a JavaServer Faces (JSF) developer, the Liferay Faces Bridge supports deploying 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.

If Spring is your thing, Spring Portlet MVC portlets are easy to configure and deploy on Liferay DXP. You can continue using Spring features, including Spring beans and Spring dependency injection.

Last but not least, Liferay MVC Portlet continues to be a favorite with experienced Liferay developers, and makes portlet development easy for Liferay newcomers. It leverages OSGi Declarative Services (DS) for injecting dependencies and defining configurable extension points. Since Liferay DXP core and Liferay-written apps use DS, gaining experience with DS helps you develop Liferay DXP extensions and customizations. Liferay MVC Portlet works seamlessly with many Liferay frameworks, such as MVC commands, Service Builder, and more.

No matter which development framework you choose, you'll be able to get an app up and running fast.

230.2 Getting Started

If you have an existing app that uses one the frameworks described above, your first step is to deploy it to Liferay DXP. Most deployments involve configuration steps that you can complete in an hour or less.

You can also build apps from scratch using the tools you like or leveraging Liferay's tool offering. Liferay provides templates for creating all kinds of apps and samples that you can examine and modify to fit your needs.

Once your app is functional, you can improve your app by integrating it with Liferay frameworks:

- Localization
- Permissions
- Search and indexing
- Asset publishing
- Workflow
- Staging
- Data export and import

Liferay provides frameworks that integrate these features fast. As you develop apps on Liferay DXP, you'll enjoy using what you know, discover frameworks and tools that boost your productivity, and have fun creating rich, full-featured applications.

If you're experienced with developing one of the listed app types, feel free to jump ahead to it. Otherwise, Angular Widgets is next.

CHAPTER 231

DEVELOPING AN ANGULAR APPLICATION

Running an existing Angular app on Liferay DXP makes the app available as a widget for using on site pages. You can adapt your existing Angular app, but this doesn't give you access to the bundler and its various loaders to develop your project further in Liferay DXP. To have access to all of Liferay DXP's features, you must use the Liferay JS Generator and Liferay npm Bundler to merge your files into a portlet bundle, adapt your routes and CSS, and deploy your bundle.

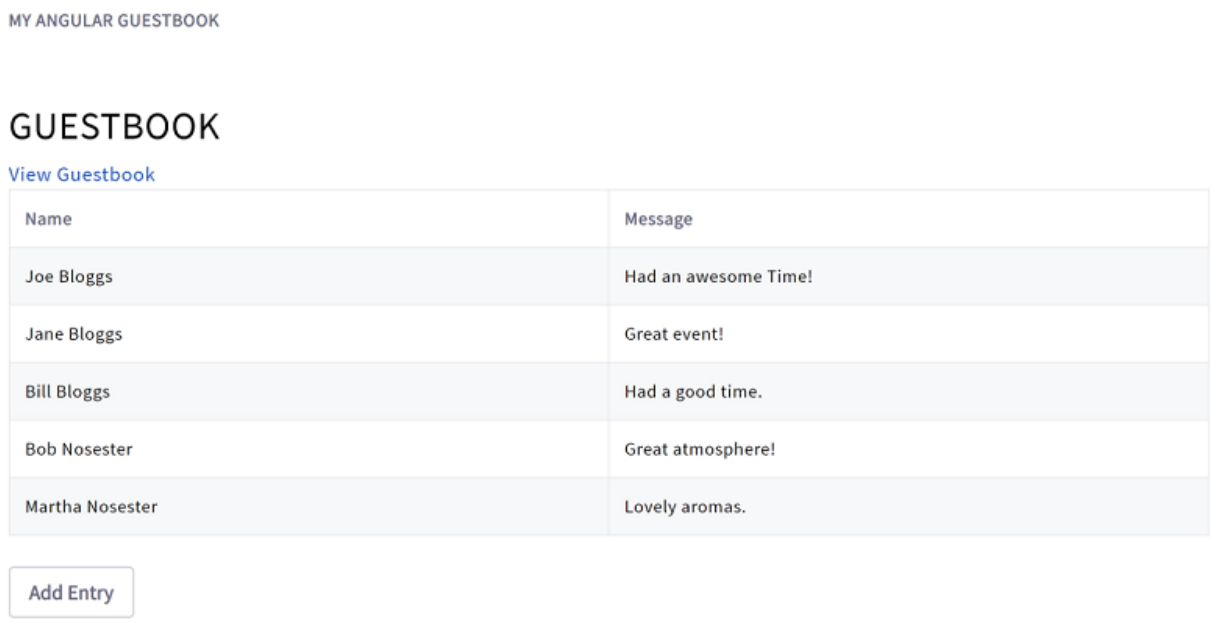

Figure 231.1: Apps like this Guestbook app are easy to migrate to Liferay DXP.

Follow these steps:

1. Using npm, install the Liferay JS Generator:

```
npm install -g yo generator-liferay-js
```

2. Generate an Angular-based portlet bundle project for deploying your app to your Liferay DXP installation.

```
yo liferay-js
```

   Select `Angular based portlet` and opt for generating sample code. Here's the bundle's structure:

   - `[my-angular-portlet-bundle]`
     - `assets/` → CSS, HTML templates, and resources
       * `css/` → CSS files
         · `styles.css` → Default CSS file
       * `app/` → HTML templates
         · `app.component.html` → Root component template
     - `features/` → Liferay DXP bundle features
       * `localization/` → Resource bundles
         · `Language.properties` → Default language keys
     - `src/` → JavaScript an TypeScript files
       * `app/` → Application modules and Components
         · `app.component.ts` → Main component
         · `app.module.ts` → Root module
         · `dynamic.loader.ts` → Loads an Angular component dynamically for the portlet to attach to
       * `types/`
         · `LiferayParams.ts` → Parameters passed by Liferay DXP to the JavaScript module
       * `index.ts` → Main module invoked by the "bootstrap" module to initialize the portlet
       * `polyfills.ts` → Fills in browser JavaScript implementation gaps
     - `package.json` → npm bundle configuration
     - `README.md`
     - `.npmbuildrc` → Build configuration
     - `.npmbundlerrc` → Bundler configuration
     - `tsconfig.json` → TypeScript configuration

3. Copy your app files, matching the types listed below, into your new project.

```
File type | Destination | Comments |
--------- | ----------- | -------- |
HTML | `assets/app/` | Merge your main component with the existing `app.component.html`. |
CSS  | `assets/css/` | Overwrite `styles.css`. |
TypeScript and JavaScript | `src/app/` |  Merge with all files **except** `app.module.ts`---the root module merge is explained in a later step. |
```

---

4. Update your component class `templateUrls` to use the `web-context` value declared in your project's `.npmbundlerrc` file. Here's the format:

   ```
   templateUrl: `/o/[web-context]/app/[template]`
   ```

   Here's an example:

   ```
   templateUrl: '/o/my-angular-guestbook/app/add-entry/add-entry.component.html'
   ```

5. Update your bundle to use portlet-level styling.

   - Import all component CSS files through the CSS file (default is `styles.css`) your bundle's `package.json` file sets for your portlet. Here's the default setting:

     ```
     "portlet": {
         "com.liferay.portlet.header-portlet-css": "/css/styles.css",
     ...
     }
     ```

   - Remove `selector` and `styleUrls` properties from your component classes.

6. In your routing module's `@NgModule` decorator, configure the router option `useHash: true`. This tells Angular to use client-side routing in the form of `.../#/[route]`, which prevents client-side parameters (i.e., anything after `#`) from being sent back to Liferay DXP.

   For example, your routing module class `@NgModule` decorator might look like this:

   ```
   @NgModule({
     imports: [RouterModule.forRoot(routes, {useHash: true})],
     exports: [RouterModule]
   })
   export class AppRoutingModule { }
   ```

7. Also in your routing module, export your view components for your root module (discussed next) to use. For example,

   ```
   export const routingComponents = [ViewComponent1, ViewComponent2]
   ```

8. Merge your root module with `src/app/app.module.ts`, configuring it to dynamically load components.

Note: Components must be loaded dynamically to attach to the portlet's
DOM. The DOM is determined at run time when the portlet's page is
rendered.

- Import the `routingComponents` constant and the app routing module class
  from your app routing module. For example,

  ```javascript
  import { AppRoutingModule, routingComponents } from './app-routing.module';
  ```

- Specify the base href for the router to use in the navigation URLs.

  ```javascript
  import { APP_BASE_HREF } from '@angular/common';
  ...

  @NgModule({
      ...
      providers: [{provide: APP_BASE_HREF, useValue: '/'}]
  })
  ```

- Declare the `routingComponents` constant in your `@NgModule` decorator.

  ```javascript
  @NgModule({
    declarations: [
        routingComponents,
        ...
    ],
    ...
  })
  ```

- Make sure your `@NgModule` `bootstrap` property has no components. All
  components are loaded dynamically using the `entryComponents` array
  property. The empty `ngDoBootstrap()` method nullifies the default
  bootstrap implementation.

  ```javascript
  @NgModule({
    ...
      entryComponents: [AppComponent],
      bootstrap: [],
      ...
  })
  export class AppModule {
      ngDoBootstrap() {}
      ...
  }
  ```

Your root module `app.module.ts` should look like this:

```javascript
import { APP_BASE_HREF } from '@angular/common';
import { AppRoutingModule, routingComponents } from './app-routing.module';
// more imports ...

@NgModule({
  declarations: [
    AppComponent,
```

```
    routingComponents,
    // more declarations ...
  ],
  imports: [
    AppRoutingModule,
    // more imports ...
  ],
  entryComponents: [AppComponent],
  providers: [{provide: APP_BASE_HREF, useValue: '/'}],
  bootstrap: [],
  // more properties ...
})
export class AppModule {
    ngDoBootstrap() {}

    // ...
}
```

9. Merge your app `package.json` file's `dependencies` and `devDependencies` into the bundle's `package.json`.

```
**Note:** To work around build errors caused by the `rxjs` dependency, set
the dependency to version `"6.0.0"`. See
[LPS-92848](https://issues.liferay.com/browse/LPS-92848)
for details.
```

---

10. Finally, deploy your bundle:

    ```
    npm run deploy
    ```

Congratulations! Your Angular app is deployed and now available as a widget that you can add to site pages.

The Liferay npm Bundler confirms the deployment:

```
Report written to liferay-npm-bundler-report.html
Deployed my-angular-guestbook-1.0.0.jar to c:\git\bundles
```

The Liferay DXP console confirms your bundle started:

```
2019-03-22 20:17:53.181 INFO  [fileinstall-C:/git/bundles/osgi/modules][BundleStartStopLogger:39] STARTED my-angular-guestbook_1.0.0 [1695]
```

To find your widget, select the *Add* icon (), navigate to *Widgets* and then the category you specified to the Liferay Bundle Generator (*Sample* is the default category).

231.1 Related Topics

Web Services

Service Builder

Localization

CHAPTER 232

DEVELOPING A REACT APPLICATION

Running an existing React app on Liferay DXP makes the app available as a widget for using on site pages. You can adapt your existing React app, but this doesn't give you access to the bundler and its various loaders to develop your project further in Liferay DXP. To have access to all of Liferay DXP's features, you must use the Liferay JS Generator and Liferay npm Bundler to merge your files into a portlet bundle, update your static resource paths, and deploy your bundle.

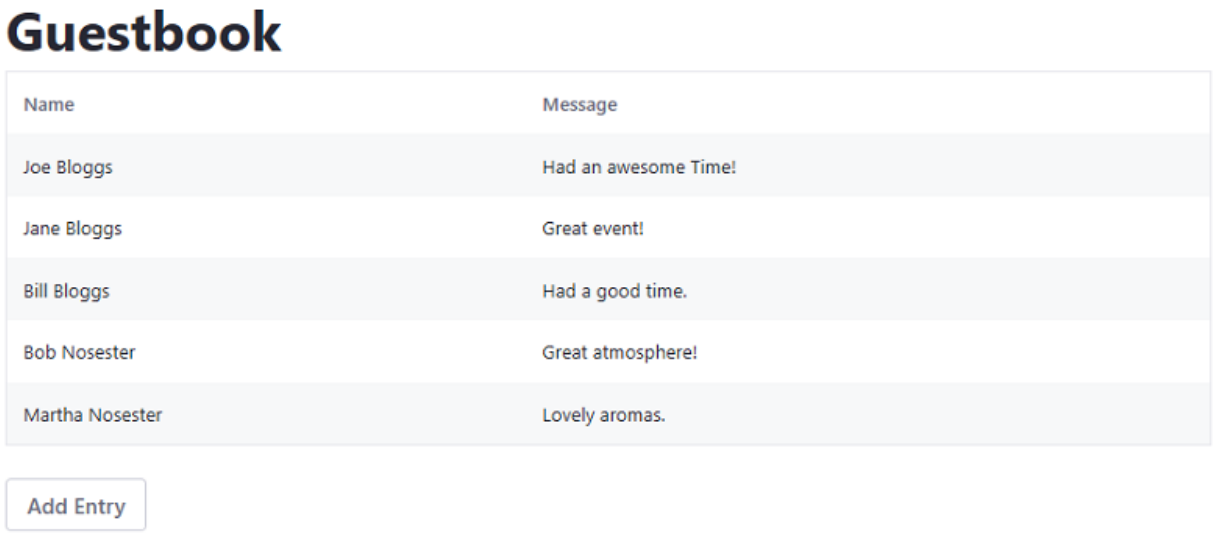

Figure 232.1: Apps like this Guestbook app are easy to migrate to Liferay DXP.

Follow these steps:

1. Using npm, install the Liferay JS Generator:

   ```
   npm install -g yo generator-liferay-js
   ```

2. Generate a React based portlet bundle project for deploying your app to your Liferay DXP installation.

```
yo liferay-js
```

Select `React based portlet` and opt for generating sample code. Here's the bundle's structure:

- `my-react-portlet-bundle`
  - `assets/` → CSS and resources
    - `css/` → CSS files
      - `styles.css` → Default CSS file
  - `features/` → Liferay DXP bundle features
    - `localization` → Resource bundles
      - `Language.properties` → Default language keys
  - `src/` → JavaScript and React component files
    - `AppComponent.js` → Sample React component that you can remove
    - `index.js` → Main module used to initialize the portlet
  - `.babelrc` → Babel configuration
  - `.npmbuildrc` → Build configuration
  - `.npmbundlerrc` → Bundler configuration
  - `package.json` → npm bundle configuration
  - `README.md`

3. Copy your app files, matching the types listed below, into your new project.

```
File type | Destination | Comments |
--------- | ----------- | -------- |
CSS  | `assets/css/` | Overwrite `styles.css`. |
JavaScript | `src/` |  Merge with all files **except** `index.js`---the main module merge is explained in a later step. |
Static resources | `assets/` |  Include resources such as image files here |
```

---

4. Update your bundle to use portlet-level styling.

   - Import all component CSS files through the CSS file (default is `styles.css`) your bundle's `package.json` file sets for your portlet. Here's the default setting:

```
"portlet": {
    "com.liferay.portlet.header-portlet-css": "/css/styles.css",
    ...
}
```

   - Remove any CSS imports you have in your JS files

5. Update any static resource references to use the `web-context` value declared in your project's `.npmbundlerrc` file, and remove any imports for the resource. For example, if you have an image file called `logo.png` in your `assets` folder, you would use the format below. Note that the `assets` folder is not included in the path.

   Here is the format:

   ```
   /o/[web-context]/[resource]
   ```

   Here's an example image resource:

   ```
   <img alt="React logo" src="/o/react-guestbook-migrated/logo.png">
   ```

6. Merge your entry module with `src/index.js`, configuring it to dynamically load components.

```
 **Note:** Components must be loaded dynamically to attach to the portlet's
 DOM. The DOM is determined at run time when the portlet's page is
 rendered.
```

```
-   Use the `HashRouter` for routing between component views, as Liferay DXP
    requires hash routing for proper portal navigation:

   ```javascript
    import { HashRouter as Router } from 'react-router-dom';
   ```

-   Place your code inside the `main()` function.

-   Render your app inside the `portletElementId` element that is passed in
    the `main()` function. This is required to render the React app inside
    the portlet.

Your entry module `index.js` should look like this.

```javascript
import React from 'react';
import ReactDOM from 'react-dom';
//import './index.css';//removed for Portal Migration
import App from './App';
import { HashRouter as Router } from 'react-router-dom';

export default function main({portletNamespace, contextPath,
portletElementId}) {
      ReactDOM.render((
        <Router>
          <App/>
        </Router>
      ), document.getElementById(portletElementId));
}
```
```

7. Merge your app `package.json` file's `dependencies` and `devDependencies` into the bundle's `package.json`.

8. Finally, deploy your bundle:

```
npm run deploy
```

Congratulations! Your React app is deployed and now available as a widget that you can add to site pages.

The Liferay npm Bundler confirms the deployment:

```
Report written to liferay-npm-bundler-report.html
Deployed my-react-guestbook-1.0.0.jar to c:\git\bundles
```

The Liferay DXP console confirms your bundle started:

```
2019-03-22 20:17:53.181 INFO
[fileinstall-C:/git/bundles/osgi/modules][BundleStartStopLogger:39]
STARTED my-react-guestbook_1.0.0 [1695]
```

To Find your widget, click the *Add* icon (), navigate to *Widgets* and then the category you specified to the Liferay Bundle Generator (*Sample* is the default category).

232.1 Related Topics

Web Services

Service Builder

Localization

CHAPTER 233

DEVELOPING A VUE APPLICATION

Running an existing Vue app on Liferay DXP makes the app available as a widget for using on site pages. You can adapt your existing Vue app, but this doesn't give you access to the bundler and its various loaders to develop your project further in Liferay DXP. To have access to all of Liferay DXP's features, you must use the Liferay JS Generator and Liferay npm Bundler to merge your files into a portlet bundle, update your static resource paths, and deploy your bundle. The steps below demonstrate how to prepare a Vue app that uses single file components (`.vue` files) with multiple views.

Note: if you have a tree of components expressed as `.vue` templates, only the root one will be available as a true AMD module.

Follow these steps:

1. Using npm, install the Liferay JS Generator:

   ```
   npm install -g yo generator-liferay-js
   ```

2. Generate a Vue based portlet bundle project:

   ```
   yo liferay-js
   ```

   Select `Vue based portlet` and opt for generating sample code. Here's the bundle's structure:

   - `my-vue-portlet-bundle`
     - `assets/` → CSS and resources
       - `css/` → CSS not included in `.vue` files.
     - `features/` → Liferay DXP bundle features
       - `localization/` → Resource bundles

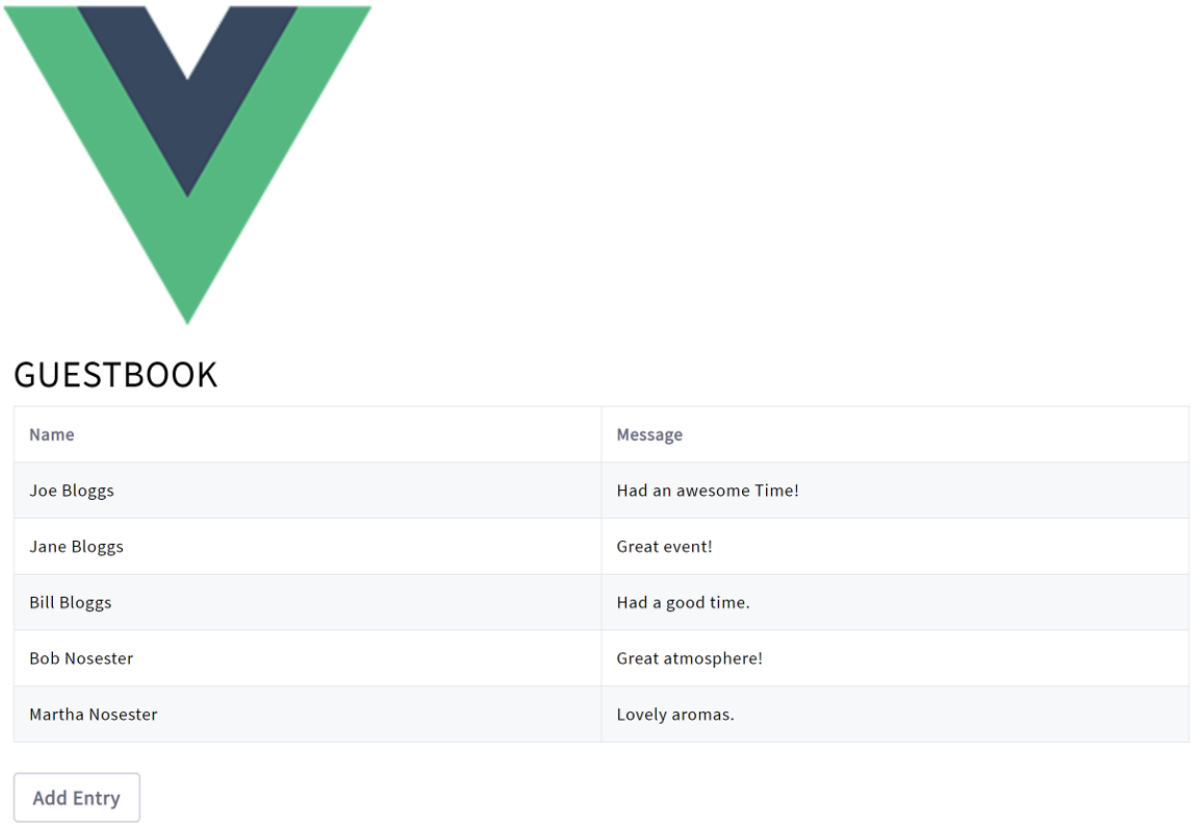

Figure 233.1: Vue Apps like this Guestbook App are easy to deploy, and they look great in Liferay DXP.

        - `Language.properties` → Default language keys
    * `settings.json` → Placeholder System Settings
- `src/` → JavaScript and Vue files
    * `index.js` → Main module used to initialize the portlet
- `.babelrc` → Babel configuration
- `.npmbuildrc` → Build configuration
- `.npmbundlerrc` → Bundler configuration
- `package.json` → npm bundle configuration
- `README.md`

3. Copy your app files, matching the types listed below, into your new project.

```
File type | Destination | Comments |
--------- | ----------- | -------- |
CSS  | `assets/css/` | Overwrite `styles.css`. |
Static resources | `assets` |  Include resources such as image files here |
VUE and JS| `src` | Merge your main component with the existing `index.js`. More info on that below. |
```

4. Update your bundle to use portlet-level styling.

    - If you have internal CSS included with `<style>` tags in your `.vue` files, import `.index.css` in `/assets/styles.css`. This is generated by the modified build script further down:

      ```
      @import '../index.css';
      ```

    - Import all custom CSS files (i.e. CSS not included in `.vue` files) through the CSS file (default is `styles.css`) your bundle's `package.json` file sets for your portlet. Here's the default setting:

      ```
      "portlet": {
          "com.liferay.portlet.header-portlet-css": "/css/styles.css",
      ...
      }
      ```

5. Update any static resource references to use the `web-context` value declared in your project's `.npmbundlerrc` file. Here's the format:

    ```
    /o/[web-context]/[resource]
    ```

    Here's an example image resource:

    ```
    <img alt="Vue logo" src="/o/vue-guestbook-migrated/logo.png">
    ```

6. Merge your entry module with `src/index.js`, following these steps to dynamically load components.

```
**Note:** Components must be loaded dynamically to attach to the portlet's
DOM. The DOM is determined at runtime when the portlet's page is
rendered.
```

```
-   Use Vue's runtime + compiler module
    (`import Vue from 'vue/dist/vue.common';`) so you don't have to process
    templates during build time. This is imported by default at the top of
    the file.

-   Remove the sample content from the `main()` function (i.e. the `node`
    constant and its use), and replace it with your router code.

-   Make these updates to the `new Vue` instance:

    -   Remove the default data properties (the ones you just removed in the
        sample content), and set the render element to `portletElementId`.
        This is required and ensures that your app is rendered inside the
        portlet.
    -   Add the router.
    -   Add a render function that mounts your component wrapper to the Vue
        instance and displays it.
```

````
    Your updated configuration should look like this:

    ```javascript
    new Vue({
      el: `#${portletElementId}`,
      render: h => h(App),
      router
    })
    ```

Your entry module `index.js` should look like this.

```javascript
import Vue from 'vue/dist/vue.common';
import App from './App.vue'
import VueRouter from 'vue-router'
//Component imports

export default function main({portletNamespace, contextPath, portletElementId}) {

  Vue.config.productionTip = false

  Vue.use(VueRouter)

  const router = new VueRouter({
      routes: [
          {
            ...
          }
      ]
  })
  new Vue({
    el: `#${portletElementId}`,
    render: h => h(App),
    router
  })
}
```
````

7. Merge your app `package.json` file's `dependencies` and `devDependencies` into the project's `package.json`, and replace the `babel-cli` and `babel-preset-env` dev dependencies with the newer `"@babel/cli": "^7.0.0"` and `"@babel/preset-env": "^7.4.2"` packages instead. Also include the `"vueify": "9.4.1"` dev dependency.

8. Update the `.babelrc` file to use `@babel/preset-env` instead of `env`:

    ```
    "presets": ["@babel/preset-env"]
    ```

9. If you're using `.vue` files, replace the build script in the `package.json` with the one below to use `vue-cli-service`. The updated build script uses vue-cli to access the main entrypoint for the app (`index.js` in the example below) and combines all the Vue templates and JS files into one single file named `index.common.js` and generates an `index.css` file for any internal CSS included with `<style>` tags in `.vue` files:

    ```
    "scripts": {
      "build": "babel --source-maps -d build src && vue-cli-service build --dest
      build/ --formats commonjs --target lib --name index ./src/index.js && npm
      run copy-assets && liferay-npm-bundler",
      "copy-assets": "lnbs-copy-assets",
      "deploy": "npm run build && lnbs-deploy",
      "start": "lnbs-start"
    }
    ```

10. Update the `main` entry of the `package.json` to match the new CommonJS file name specified in the previous step:

    ```
    "main": "index.common"
    ```

11. Finally, deploy your portlet bundle:

    ```
    npm run deploy
    ```

Congratulations! Your Vue app is deployed and now available as a widget that you can add to site pages.

The liferay-npm-bundler confirms the deployment:

```
Report written to liferay-npm-bundler-report.html
Deployed my-vue-guestbook-1.0.0.jar to c:\git\bundles
```

The Liferay DXP console confirms your bundle started:

```
2019-03-22 20:17:53.181 INFO
[fileinstall-C:/git/bundles/osgi/modules][BundleStartStopLogger:39]
STARTED my-vue-guestbook_1.0.0 [1695]
```

Find your widget by selecting the *Add* icon () and navigating to *Widgets* and the category you specified to the Liferay Bundle Generator (*Sample* is the default category).

233.1 Related Topics

Web Services

Service Builder

Localization

CHAPTER 234

LIFERAY MVC PORTLET

If you're an experienced developer, this is not the first time you've heard about Model View Controller. 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 Portlet 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 Portlet framework only 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).
- An MVC command class can serve multiple portlets.
- 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 and easy to use. The default `MVCPortlet` project template generates a fully configured and working project.

Here, you'll learn how MVCPortlet works by covering these topics:

- MVC layers and modularity
- Liferay MVC command classes
- Liferay MVC portlet component
- Simple MVC portlets

Review how each layer of the Liferay MVC portlet framework helps you separate the concerns of your application.

234.1 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.

Liferay DXP's applications are divided into multiple discrete modules. With Service Builder, the model layer is generated into a `service` and an `api` module. That accounts for the model in the MVC pattern. The view and the controller layers share a module, the `web` module.

Generating the skeleton for a multi-module Service Builder-driven MVC application saves you lots of time and gets you started on the more important (and interesting, if we're being honest) development work.

234.2 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 `-Portlet` class's `@Component` annotation.

234.3 Liferay MVC Portlet Component

Whether or not you plan to split up the controller into MVC command classes, the portlet `@Component` annotation configures the portlet. Here's a simple portlet component as an example:

```
@Component(
    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 {
}
```

The `javax.portlet.name` property is required. When using MVC commands, the `javax.portlet.name` property value links particular portlet URL/command combinations 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 a component. The 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_2_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.

234.4 A Simpler MVC Portlet

In simpler applications, you don't use MVC commands. 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 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, continue with Creating an MVC Portlet.

CHAPTER 235

CREATING AN MVC PORTLET

Generating MVC portlet projects is a snap using Liferay's project templates. Here you'll generate an MVC Portlet project and deploy the portlet to Liferay DXP.

1. Generate an MVC Portlet project using a Gradle or Maven.

   Here's the resulting folder structure for an MVC Portlet class named `MyMvcPortlet` in a base package `com.liferay.docs.mvcportlet`:

   - `my-mvc-portlet-project` → Arbitrary project name.
     - `gradle`
       - `wrapper`
         - `gradle-wrapper.jar`
         - `gradle-wrapper.properties`
     - `src`
       - `main`
         - `java`
         - `com/liferay/docs/mvcportlet`
         - `constants`
         - `MyMvcPortletKeys.java` → Declares portlet constants.
         - `portlet`
         - `MyMvcPortlet.java` → MVC Portlet class.
         - `resources`
         - `content`
         - `Language.properties` → Resource bundle
         - `META-INF`

        - `resources`
        - `init.jsp` → Imports classes and taglibs and defines commonly used objects from the theme and the portlet.
        - `view.jsp` → Default view template.

    - `bnd.bnd` → OSGi bundle metadata.
    - `build.gradle`
    - `gradlew`

The Maven-generated project includes a `pom.xml` file and does not include the Gradle-specific files, but otherwise is exactly the same.

Here's the resulting MVC Portlet class:

```
package com.liferay.docs.mvcportlet.portlet;

import com.liferay.docs.mvcportlet.constants.MyMvcPortletKeys;
import com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet;
import javax.portlet.Portlet;
import org.osgi.service.component.annotations.Component;

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

The class extends `MVCPortlet`. The `@Component` annotation and `service = Portlet.class` attribute makes the class an OSGi Declarative Services component that provides the `javax.portlet.Portlet` service type. The `immediate = true` attribute activates the service immediately on the portlet's deployment.

2. Set any portlet configuration or Liferay portlet configuration values using `javax.portlet.*` and `com.liferay.portlet.*` `@Component` annotation properties `javax.portlet.*` and `com.liferay.portlet.*` `@Component` annotation properties respectively.

   Here are the example component's properties:

   - `"com.liferay.portlet.display-category=category.sample"`: Sets the Widget's category to "Sample".
   - `"com.liferay.portlet.instanceable=true"`: Activates the component immediately when its bundle installs.
   - `"javax.portlet.display-name=my-mvc-portlet-project Portlet"`: Sets the portlet's Widget name.

- `"javax.portlet.init-param.template-path=/"`: The path under `src/main/resources/META-INF/resources/` where the templates reside.
- `"javax.portlet.init-param.view-template=/view.jsp"`: Default view template.
- `"javax.portlet.name=" + MyMvcPortletKeys.MyMvc`: The portlet's unique identity.
- `"javax.portlet.resource-bundle=content.Language"`: Sets the portlet's resource bundle to the `content/Language*.properties` file(s) in the `src/main/resources/` folder.
- `"javax.portlet.security-role-ref=power-user,user"`: Makes the Liferay DXP virtual instance's power user and user Roles available for defining the portlet's permissions.

```
**Note:** To opt-in to Portlet 3.0 features, set the component property
`"javax.portlet.version=3.0"`.
```

---

3. The portlet renders content via the view template `src/main/resources/META-INF/resources/view.jsp` by default.
4. Build your project.

   Gradle:

   ```
   gradlew jar
   ```

   Maven:

   ```
   mvn clean package
   ```

5. Deploy the project using your build environment or by building the project JAR and copying it to the `deploy/` folder in your Liferay Home.

The MVC Portlet is now available in the Liferay DXP UI, in the Widget category you assigned it.

Figure 235.1: The example portlet shows a message defined by the language property `yourmvc.caption=Hello from YourMVC!` in the Language.properties file.

Congratulations on creating and deploying an MVC Portlet!

235.1 Related Topics

Writing MVC Portlet Controller Code
    Configuring the View Layer
    MVC Action Command
    MVC Render Command
    MVC Resource Command

CHAPTER 236

WRITING MVC PORTLET CONTROLLER CODE

In MVC, your controller is 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, 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. Here you'll implement controller logic for small applications, where all the controller code is in the `-Portlet` class. It involves these things:

- Action methods
- Render logic
- Setting and retrieving request parameters and attributes

Start with creating action methods.

236.1 Action Methods

Your portlet class can act as your controller by itself and process requests using action methods. 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 method has one job: call a service to add a guestbook. If the 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.

236.2 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 application's default view. When the portlet's `init` method (e.g., your portlet's override of `MVCPortlet.init()`) is called, Liferay reads the initialization parameters you specify and directs 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");
```

You can 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? 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.

236.3 Setting and Retrieving Request and Response Parameters and Attributes

A handy utility class called `ParamUtil` facilitates retrieving parameters from an `ActionRequest` or a `RenderRequest`.

For example, this JSP passes 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 gets 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 that JSP.

Your controller class can also set attributes into response objects using the setAttribute method.

```
renderResponse.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.

236.4 Related Topics

Creating an MVC Portlet
    Configuring the View Layer
    MVC Action Command
    MVC Render Command
    MVC Resource Command

CHAPTER 237

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.

237.1 Using the init.jsp

Liferay's 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 />
```

Here are the tag libraries it gives you:

- `c`: JSTL core tags.
- `portlet`: Standard portlet component tags.
- `aui`: AlloyUI component tags.
- `liferay-portlet`: Liferay portlet component tags.
- `liferay-theme`: Liferay theme component tags.
- `liferay-ui`: Liferay UI component tags.

These tags make portlet and Liferay objects available:

- `<portlet:defineObjects />`: Implicit Java variables that reference Portlet API objects. The objects available are limited to those available in the current portlet request. For details, see the `defineObjects` tag in JSR-286.
- `<liferay-theme:defineObjects />`: Implicit Java variables that reference Liferay objects.

To use all that the `init.jsp` has, include it in your other JSPs:

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

A JSP uses render URLs to display other pages and action URLs to invoke controller methods.

237.2 Using Render URLs

A render URL attached to a UI component action displays another page. For example, this render URL displays the JSP `/path/to/foo.jsp`.

```
<portlet:renderURL var="adminURL">
    <portlet:param name="mvcPath" value="/path/to/foo.jsp" />
</portlet:renderURL>
```

Here's how to use a render URL:

1. Add a `<portlet:renderURL>` to your JSP.
2. Name the render URL via a `var` attribute in the `<portlet:renderURL>` tag. The `<portlet:renderURL>` tag constructs the URL and assigns it to the variable. For example, this render URL is assigned to the variable named `adminURL`:

   ```
   <portlet:renderURL var="adminURL">
      ...
   </portlet:renderURL>
   ```

3. As sub-element to the `<portlet:renderURL>` tag, add a `<portlet:param>` tag with the following attributes:

   `name="mvcPath"`: Your controller's `render` method forwards processing to the JSP at the path specified in the `value`.

   `value="/path/to/foo.jsp"`: The path to the JSP to render. Replace the value `/path/to/foo.jsp` with your JSP path.

   ```
   <portlet:renderURL var="adminURL">
       <portlet:param name="mvcPath" value="/path/to/foo.jsp" />
   </portlet:renderURL>
   ```

4. To invoke the render URL, assign its variable (`var`) to a UI component action, such as a button or navigation bar item action.

Invoking the UI component causes the controller's render method to display the `mvcPath` parameter's JSP.

237.3 Using Action URLs

Action methods are different because they invoke an action (i.e., code), rather than link to another page. For example, this action URL invokes a controller method called `doSomething` and passes a parameter called `redirect`. The `redirect` parameter contains the path of the JSP to render after invoking the action:

```
<portlet:actionURL name="doSomething" var="doSomethingURL">
    <portlet:param name="redirect" value="<%= redirect %>" />
</portlet:actionURL>
```

Here's how to use an action URL:

1. Add a `<portlet:actionURL>` to your JSP.

2. Add a `name` and `var` attribute to the `<portlet:actionURL>`. The `<portlet:actionURL>` tag constructs the URL and assigns it to the `var` variable.

   `name`: Controller action to invoke.

   `var`: Variable to assign the action URL to.

   ```
   <portlet:actionURL name="doSomething" var="doSomethingURL">
      ...
   </portlet:actionURL>
   ```

3. As sub-element to the `<portlet:actionURL>` tag, add a `<portlet:param>` tag that has the following attributes:

   `name="redirect"`: Tells the portlet to redirect to the JSP associated with this parameter.

   `value="/path/to/foo.jsp"`: Redirects the user to this JSP path after invoking the action. Replace the value `/path/to/bar.jsp` with your JSP path.

   ```
   <portlet:actionURL name="doSomething" var="doSomethingURL">
       <portlet:param name="redirect" value="/path/to/bar.jsp" />
   </portlet:actionURL>
   ```

4. To invoke the action URL, assign its variable (`var`) to a UI component action, such as a button or navigation bar item action.

Congratulations! Your portlet is ready for action.

These simple examples demonstrate how Liferay MVC Portlet facilitates communication between a smaller application's view layer and controller.

237.4 Related Topics

Writing MVC Portlet Controller Code
- MVC Action Command
- MVC Render Command
- MVC Resource Command
- Front-end Taglibs
- Liferay JavaScript APIs

CHAPTER 238

MVC ACTION COMMAND

Liferay's MVC Portlet framework enables you to handle MVCPortlet actions in separate classes. This facilitates managing action logic in portlets that have many actions. Each action URL in your portlet's JSPs invokes an appropriate action command class.

Here are the steps:

1. Configure your JSPs to use action URLs via `<portlet:actionURL>` tags. For example, the action-command-portlet sample uses this action URL:

   ```
   <liferay-portlet:actionURL name="greet" var="greetURL" />
   ```

   Name the action URL via its `name` attribute. Your `*MVCActionCommand` class maps to this name. Assign the `var` attribute a variable name.

2. Assign the action URL variable (`var`) to a UI component. Acting on the UI component invokes the action. For example, the sample's `greetURL` action URL variable triggers on submitting this form:

   ```
   <aui:form action="<%= greetURL %>" method="post" name="fm">
       <aui:input name="name" type="text" />

       <aui:button-row>
           <aui:button type="submit"></aui:button>
       </aui:button-row>
   </aui:form>
   ```

3. Create a class that implements the `MVCActionCommand` interface, or that extends the `BaseMVCActionCommand` class. The latter may save you time, since it already implements `MVCActionCommand`.

Tip: Naming your `*MVCActionCommand` class after the action it performs makes the action mappings more obvious for maintaining the code. 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.

4. Annotate your class with an `@Component` annotation, like this one:

```
@Component(
    property = {
        "javax.portlet.name=your_portlet_name_YourPortlet",
        "mvc.command.name=/your/jsp/action/url"
    },
    service = MVCActionCommand.class
)
```

5. Set a `javax.portlet.name` property to your portlet's internal ID.

   Note, you can apply MVC Command classes to multiple portlets by setting a `javax.portlet.name` property for each portlet. For example, the `javax.portlet.name` properties in this component apply it to three specific portlets.

```
@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 {
    ...
}
```

6. Set the `mvc.command.name` property to your `<portlet:actionURL>` tag's `name`. This maps your class to the action URL of the same name.
7. Register your class as an `MVCActionCommand` service by setting the `service` attribute to `MVCActionCommand.class`.
8. Implement your action logic by overriding the appropriate method of the class you're implementing or extending.

   - `MVCActionCommand` implementations override the `processAction` method.
   - `BaseMVCActionCommand` extensions override the `doProcessAction` method.

   Here's an example of overriding `MVCActionCommand`'s `processAction` method. This action logic gets the name parameter from the `ActionRequest` and adds it to the session messages and to an `ActionRequest` attribute.

```
@Override
public boolean processAction(
        ActionRequest actionRequest, ActionResponse actionResponse)
    throws PortletException {

    _handleActionCommand(actionRequest);

    return true;
```

```
    }

    private void _handleActionCommand(ActionRequest actionRequest) {
        String name = ParamUtil.get(actionRequest, "name", StringPool.BLANK);

        if (_log.isInfoEnabled()) {
            _log.info("Hello " + name);
        }

        String greetingMessage = "Hello " + name + "! Welcome to OSGi";

        actionRequest.setAttribute("GREETER_MESSAGE", greetingMessage);

        SessionMessages.add(actionRequest, "greetingMessage", greetingMessage);
    }

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

Congratulations! You've created an `MVCActionCommand` that handles your portlet actions.

238.1 Related Topics

Creating an MVC Portlet
    Configuring the View Layer
    MVC Render Command
    MVC Resource Command
    MVC Command Overrides

CHAPTER 239

MVC RENDER COMMAND

MVCRenderCommands are classes that respond to MVCPortlet render URLs. If your render logic is simple and you want to implement all of your render logic in your portlet class, see Writing MVC Portlet Controller Code. If your render logic is complex or you want clean separation between render paths, use MVCRenderCommands. Each render URL in your portlet's JSPs invokes an appropriate render command class.

Here are the steps:

1. Configure your JSPs to generate render URLs via <portlet:renderURL> tags.

   For example, this render-command-portlet sample render URL invokes an MVC render command named /blade/render.

```
<portlet:renderURL var="bladeRender">
    <portlet:param name="mvcRenderCommandName" value="/blade/render" />
</portlet:renderURL>
```

2. Name the render URL via its <portlet:param> named mvcRenderCommandName. The render URL and *MVCRenderCommand class (demonstrated later) map to the mvcRenderCommandName value.

3. Assign the <portlet:renderURL>'s var attribute a variable name to pass to a UI component.

4. Assign the render URL variable (var) to a UI component. When the user triggers the UI component, the *MVCRenderCommand class that matches the render URL handles the render request.

   For example, the render URL with the variable bladeRender triggers on users clicking this button.

```
<aui:button href="<%= bladeRender %>" value="goto page render" />
```

5. Create a class that implements the MVCRenderCommand interface.

6. Annotate the class with an @Component annotation, like this one:

```
@Component(
    property = {
        "javax.portlet.name=com_liferay_blade_samples_portlet_rendercommand_BladeRenderPortlet",
        "mvc.command.name=/blade/render"
    },
    service = MVCRenderCommand.class
)
```

7. Set a `javax.portlet.name` property to your portlet's internal ID.
8. Set a `mvc.command.name` property to your `<portlet:renderURL>` tag `mvcRenderCommandName` portlet parameter value. This maps your class to the render URL.
9. Register your class as an `MVCRenderCommand` service by setting the `service` attribute to `MVCRenderCommand.class`.

   Note, you can apply MVC Command classes to multiple portlets by setting a `javax.portlet.name` property for each portlet and apply MVC Command classes to multiple command names by setting an `mvc.command.name` property for each command name. For example, this component's `javax.portlet.name` properties and `mvc.command.name` properties apply it to two specific portlets and two specific command names.

   ```
   @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
   )
   ```

10. Implement your render logic in a method that overrides `MVCRenderCommand`'s `render` method. 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";
        }

    }
    ```

As you can see, MVC render commands are easy to implement and can respond to multiple command names for multiple portlets.

239.1 Related Topics

Creating an MVC Portlet
    Configuring the View Layer
    MVC Resource Command
    MVC Action Command
    MVC Command Overrides

CHAPTER 240

MVC Resource Command

When using Liferay's MVCPortlet framework, you can create resource URLs in your JSPs to retrieve images, XML, or any other kind of resource from a Liferay DXP instance. The resource URL then invokes the corresponding MVC resource command class (`*MVCResourceCommand`) that processes the resource request and response.

Here how to create your own MVC Resource Command:

1. Configure your JSPs to generate resource URLs via `<portlet:resourceURL>` tags.

   For example, this resource-command-portlet sample resource URL invokes an MVC resource command named `/blade/captcha`.

   ```
   <portlet:resourceURL id="/blade/captcha" var="captchaURL" />
   ```

2. Name the resource URL via its `id` attribute.
3. Assign the resource URL's var attribute a variable name to pass to a UI component.
4. Assign the resource URL variable (`var`) 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 handles the resource request.

   For example, the sample's resource URL is triggered when the user clicks on this `liferay-captcha` component:

   ```
   <liferay-captcha:captcha url="<%= captchaURL %>" />
   ```

5. Create a class that implements the `MVCResourceCommand` interface, or that extends the `BaseMVCResourceCommand` class. The latter may save you time, since it already implements `MVCResourceCommand`.

```
**Tip:** Naming your `*MVCResourceCommand` class after the resource it
provides makes the resource mappings more obvious for maintaining the
code. For example, if your resource URL serves a captcha, you could name
its class `CaptchaMVCResourceCommand`. If your application has several MVC
command classes, naming them this way helps differentiate them.
```

6. Annotate your class with an `@Component` annotation, like this one:

```
@Component(
    property = {
        "javax.portlet.name=your_portlet_name_YourPortlet",
        "mvc.command.name=/your/jsp/resource/url"
    },
    service = MVCResourceCommand.class
)
public class YourMVCResourceCommand extends BaseMVCResourceCommand {
    // your resource handling code
}
```

    1. Set a `javax.portlet.name` property to your portlet's internal ID.
    2. Set the `mvc.command.name` property to your `<portlet:resourceURL>` tag's `id`. This maps your class to the resource URL of the same name.
    3. Register your class as an `MVCResourceCommand` service by setting the `service` attribute to `MVCResourceCommand.class`.

7. Implement your resource logic by overriding the appropriate method of the class you're implementing or extending.

    - `MVCResourceCommand` implementations override the `serveResource` method.
    - `BaseMVCResourceCommand` extensions override the `doServeResource` method.

For example, the resource-command-portlet's `CaptchaMVCResourceCommand` class implements the `MVCResourceCommand` interface with only a single method: `serveResource`.

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=com_liferay_blade_samples_portlet_resourcecommand_CaptchaPortlet",
        "mvc.command.name=/blade/captcha"
    },
    service = MVCResourceCommand.class
)
public class CaptchaMVCResourceCommand implements MVCResourceCommand {

    @Override
    public boolean serveResource(
            ResourceRequest resourceRequest, ResourceResponse resourceResponse)
        throws PortletException {

        if (_log.isInfoEnabled()) {
            _log.info("get captcha resource ");
        }

        try {
            CaptchaUtil.serveImage(resourceRequest, resourceResponse);

            return false;
        }
        catch (Exception e) {
            _log.error(e.getMessage(), e);
```

```
            return true;
        }
    }

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

}
```

This `serveResource` 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.

240.1 Related Topics

Creating an MVC Portlet
- Configuring the View Layer
- MVC Render Command
- MVC Action Command
- MVC Command Overrides

CHAPTER 241

PORTLETMVC4SPRING

PortletMVC4Spring is a way to develop portlets using the Spring Framework and the Model View Controller (MVC) pattern. While the Spring Framework supports developing *servlet-based* web applications using Spring Web MVC, PortletMVC4Spring supports developing *portlet-based* applications using MVC. You can build Spring Framework Liferay DXP portlets with features like these:

- Inversion of control (IoC) / dependency injection (DI)
- Annotations
- Security
- Binding and validation
- Multi-part file upload
- ... and more

You'll learn these things about PortletMVC4Spring:

- **Developing a Portlet Using PortletMVC4Spring:** Demonstrates creating and deploying a portlet using PortletMVC4Spring.
- **Annotation-based Controller Development:** Shows how to implement controllers using plain old Java objects (POJOs) and annotations.

Background: The PortletMVC4Spring project began as Spring Portlet MVC and was part of the Spring Framework. When the project was pruned from version 5.0.x of the Spring Framework under SPR-14129, it became necessary to fork and rename the project. This made it possible to improve and maintain the project for compatibility with the latest versions of the Spring Framework and the Portlet API.

Liferay adopted Spring Portlet MVC in March of 2019 and the project was renamed to **PortletMVC4Spring**.

If you're familiar with Spring Web MVC, it's helpful to compare it with PortletMVC4Spring. Portlet workflow differs from servlet workflow because a request to the portlet can have two distinct phases: the `ACTION_PHASE` and the `RENDER_PHASE`. The `ACTION_PHASE` is executed only once and is where any back-end changes or actions occur, such as making changes in a database. The `RENDER_PHASE`

presents the portlet's content to the user each time the display is refreshed. Thus for a single request, the `ACTION_PHASE` is executed only once, but the `RENDER_PHASE` may be executed multiple times. This provides (and requires) a clean separation between the activities that modify the system's persistent state and the activities that generate content. The Portlet 2.0 Specification added two more phases: The event phase and the resource phase, both of which are supported by annotation-driven dispatching.

PortletMVC4Spring provides annotations that support requests from the render, action, event, and resource serving portlet phases; Spring Web MVC provides only a `@RequestMapping` annotation. Where a Spring Web MVC controller might have a single handler method annotated with `@RequestMapping`, an equivalent PortletMVC4Spring controller might have multiple handler methods, each using one of the phase annotations: `@ActionMapping`, `@EventMapping`, `@RenderMapping`, or `@ResourceMapping`.

The PortletMVC4Spring framework uses a `DispatcherPortlet` that dispatches requests to handlers, with configurable handler mappings and view resolution, just as the `DispatcherServlet` in the web framework does.

Note: For more information on portlets, portlet specifications, and how portlets differ from servlets, see Portlets.

Liferay also provides full-featured sample portlets that demonstrate using JSP and Thymeleaf view templates. They exercise many features that form-based portlet applications typically require.

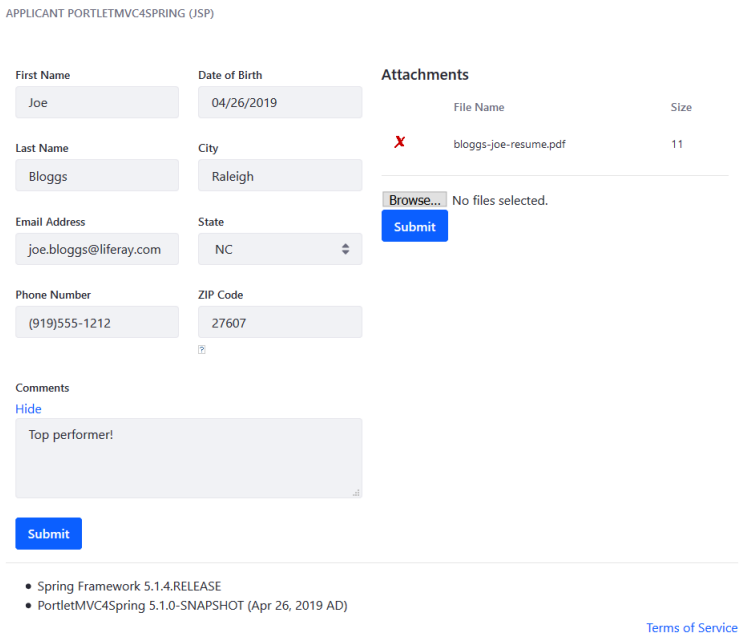

Figure 241.1: This PortletMVC4Spring portlet enables users to enter job applications. It uses the Spring features mentioned above and handles requests from multiple portlet phases.

The samples are available here:

Source Code	Maven Central
applicant-jsp-portlet	com.liferay.portletmvc4spring.demo.applicant.jsp.portlet.war
applicant-thymeleaf-portlet	com.liferay.portletmvc4spring.demo.applicant.thymeleaf.portlet.war

Now that you have a basic understanding of PortletMVC4Spring portlets and how they compare to Spring Web MVC applications, it's time to develop a PortletMVC4Spring portlet.

CHAPTER 242

DEVELOPING A PORTLET USING PORTLETMVC4SPRING

PortletMVC4Spring compliments the Spring Web framework and MVC design pattern by providing annotations that map portlet requests to Controller classes and methods. Here you'll develop and deploy a portlet application that uses PortletMVC4Spring, Spring, and JSP/JSPX templates.

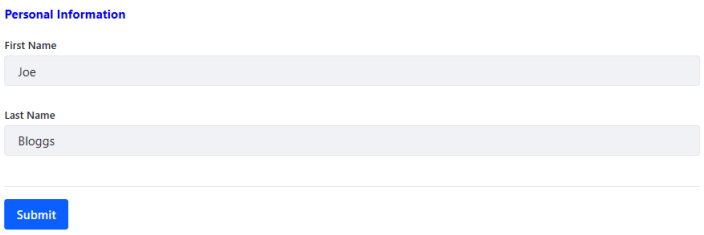

Figure 242.1: The archetype's sample portlet prints a greeting (e.g., *Hello, Joe Bloggs*) on submitting a first and last name.

Follow these steps to create and deploy your portlet application:

1. Create a PortletMVC4Spring project. See PortletMVC4Spring Project Anatomy for the project structure and commands for generating PortletMVC4Spring projects. Here's the Maven command for generating the JSP/JSPX project. This article lists sample code from the generated project.

```
mvn archetype:generate \
-DarchetypeGroupId=com.liferay.portletmvc4spring.archetype \
-DarchetypeArtifactId=com.liferay.portletmvc4spring.archetype.form.jsp.portlet \
-DarchetypeVersion=5.1.0 \
-DgroupId=com.mycompany \
-DartifactId=com.mycompany.my.form.jsp.portlet
```

2. In your project's Gradle build file or Maven POM, add your app's dependencies. Here are the PortletMVC4Spring development dependencies:

   Gradle:

```
compile group: 'com.liferay.portletmvc4spring', name: 'com.liferay.portletmvc4spring.framework', version: '5.1.0'
compile group: 'com.liferay.portletmvc4spring', name: 'com.liferay.portletmvc4spring.security', version: '5.1.0'
providedCompile group: 'javax.portlet', name: 'portlet-api', version: '3.0.0'
```

Maven:

```
<dependency>
    <groupId>com.liferay.portletmvc4spring</groupId>
    <artifactId>com.liferay.portletmvc4spring.framework</artifactId>
    <version>5.1.0</version>
</dependency>
<dependency>
    <groupId>com.liferay.portletmvc4spring</groupId>
    <artifactId>com.liferay.portletmvc4spring.security</artifactId>
    <version>5.1.0</version>
</dependency>
<dependency>
    <groupId>javax.portlet</groupId>
    <artifactId>portlet-api</artifactId>
    <version>3.0.0</version>
    <scope>provided</scope>
</dependency>
```

At this point you can develop your Model, View, or Controller components, in any order.

3. Create your Model class(es) in a package for models (e.g., `java/[my-package-path]/dto`). The sample Model class is `User`.

```
public class User implements Serializable {

    private static final long serialVersionUID = 1234273427623725552L;

    @NotBlank
    private String firstName;

    @NotBlank
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

4. Create your View using a Spring Web-supported template type. If you didn't generate your project using the archetype mentioned above, specify a View resolver for template type in your `spring-context/portlet-application-context.xml` portlet application context. (See PortletMVC4Spring Configuration Files for details).

   The sample `user.jspx` template renders a form for submitting a user's first and last name.

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:portlet="http://xmlns.jcp.org/portlet_3_0"
          xmlns:spring="http://www.springframework.org/tags"
          xmlns:form="http://www.springframework.org/tags/form"
          version="2.1">
    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8" />
    <portlet:defineObjects/>
    <link href="${contextPath}/resources/css/main.css" rel="stylesheet" type="text/css"/>
    <portlet:actionURL var="mainFormActionURL"/>
    <form:form id="${namespace}mainForm" action="${mainFormActionURL}" class="user-form" method="post" modelAttribute="user">
        <p class="caption">
            <spring:message code="personal-information" />
        </p>
        <fieldset>
            <div class="form-group">
                <form:label for="${namespace}firstName" path="firstName">
                    <spring:message code="first-name" />
                </form:label>
                <form:input id="${namespace}firstName" cssClass="form-control" path="firstName"/>
                <form:errors path="firstName" cssClass="portlet-msg-error"/>
            </div>
            <div class="form-group">
                <form:label for="${namespace}lastName" path="lastName">
                    <spring:message code="last-name" />
                </form:label>
                <form:input id="${namespace}lastName" cssClass="form-control" path="lastName"/>
                <form:errors path="lastName" cssClass="portlet-msg-error"/>
            </div>
        </fieldset>
        <hr />
        <spring:message code="submit" var="submit" />
        <input class="btn btn-primary" value="${submit}" type="submit"/>
    </form:form>
</jsp:root>
```

To invoke actions in your Controller, associate action URLs with your templates. The sample template associates the action URL variable `mainFormActionURL` with its form element.

```
<portlet:actionURL var="mainFormActionURL"/>
<form:form id="${namespace}mainForm" action="${mainFormActionURL}" class="user-form" method="post" modelAttribute="user">
    ...
```

A `<form:form/>` element's `modelAttribute` attribute targets an application Model. The sample template targets the application's user Model.

5. Style your portlet by adding CSS to a stylesheet (e.g., `webapp/resources/css/main.css`) and linking your template to it.

```
<link href="${contextPath}/resources/css/main.css" rel="stylesheet" type="text/css"/>
```

6. Define your portlet's messages in a properties file (e.g., `src/main/resources/content/[portlet].properties`). The sample `user.jspx` template references some of these properties:

```
first-name=First Name
greetings=Greetings, {0} {1}!
javax.portlet.display-name=com.mycompany.my.form.jsp.portlet
javax.portlet.keywords=com.mycompany.my.form.jsp.portlet
javax.portlet.short-title=com.mycompany.my.form.jsp.portlet
javax.portlet.title=com.mycompany.my.form.jsp.portlet
last-name=Last Name
```

```
personal-information=Personal Information
submit=Submit
todays-date-is=Today''s date is {0}
```

7. Create a Controller class to handle portlet requests. Here's an example:

```
@Controller
@RequestMapping("VIEW")
public class MyController {
    ...
}
```

   The `@Controller` annotation applies the Spring Controller component stereotype. The Spring Framework scans Controller classes for Controller annotations.

   The `@RequestMapping("VIEW")` annotation marks the class's public methods as request handler methods for the portlet's VIEW mode.

8. In your Controller, apply `@RenderMapping` annotations to methods for handling portlet render requests. Import the annotation `com.liferay.portletmvc4spring.bind.annotation.RenderMapping` and make sure each handler method returns a string that matches the name of a template to render. Here are the sample's render request handler methods:

```
@RenderMapping
public String prepareView() {
    return "user";
}

@RenderMapping(params = "javax.portlet.action=success")
public String showGreeting(ModelMap modelMap) {

    DateFormat dateFormat = new SimpleDateFormat("EEEE, MMMM d, yyyy G");

    Calendar todayCalendar = Calendar.getInstance();

    modelMap.put("todaysDate", dateFormat.format(todayCalendar.getTime()));

    return "greeting";
}
```

   The `@RenderMapping` annotation causes the `prepareView` method above to be invoked if no other handler methods match the request. `prepareView` renders the user template (i.e., `user.jspx`).

   The `@RenderMapping(params = "javax.portlet.action=success")` annotation causes the `showGreeting` method to be invoked if the render request has the parameter setting `javax.portlet.action=success`. `showGreeting` renders the `greeting` template (i.e., `greeting.jspx`).

9. In your Controller, apply `@ActionMapping` annotations to your portlet action request handling methods. Import the annotation `com.liferay.portletmvc4spring.bind.annotation.ActionMapping`.

   The sample Controller's action handler method below is annotated with `@ActionMapping`, making it the default action handler if no other action handlers match the request. Since this portlet only has one action handler, the `submitApplicant` method handles all of the portlet's action requests.

```
@ActionMapping
public void submitApplicant(@ModelAttribute("user") User user, BindingResult bindingResult, ModelMap modelMap,
                            Locale locale, ActionResponse actionResponse, SessionStatus sessionStatus) {

    localValidatorFactoryBean.validate(user, bindingResult);

    if (!bindingResult.hasErrors()) {

        if (logger.isDebugEnabled()) {
            logger.debug("firstName=" + user.getFirstName());
            logger.debug("lastName=" + user.getLastName());
        }

        MutableRenderParameters mutableRenderParameters = actionResponse.getRenderParameters();

        mutableRenderParameters.setValue("javax.portlet.action", "success");

        sessionStatus.setComplete();
    }
}
```

The `@ModelAttribute` annotation in method parameter `@ModelAttribute("user") User user` associates the View's user Model (comprising a first name and last name) to the `User` object passed to this method.

Note, the `submitApplicant` method sets the `javax.portlet.action` render parameter to `success`—the previous render handler method `showGreeting` matches this request parameter.

10. Configure any additional resources and beans in the project's descriptors and Spring context files respectively. (See PortletMVC4Spring Configuration Files for details).
11. Build the project WAR using Gradle or Maven.
12. Deploy the WAR by copying it to your `[Liferay-Home]/deploy` folder.

Liferay DXP logs the deployment and the portlet is now available in the Liferay DXP UI. Find your portlet by selecting the *Add* icon ( ) and navigating to *Widgets* and the category you specified to the Liferay Bundle Generator (*Sample* is the default category).

Congratulations! You created and deployed a PortletMVC4Spring Portlet.

242.1 Related Topics

PortletMVC4Spring Project Anatomy
    PortletMVC4Spring Annotations
    PortletMVC4Spring Configuration Files
    Migrating to PortletMVC4Spring

CHAPTER 243

MIGRATING TO PORTLETMVC4SPRING

To continue developing a portlet to use Spring Framework version 5.0 onward, migrate it from Spring Portlet MVC to PortletMVC4Spring. Here are the steps:

1. In your `pom.xml` or `build.gradle` descriptor, use the Spring Framework version 5.1.x artifacts by replacing dependencies on the `spring-webmvc-portlet` artifact with the `com.liferay.portletmvc4spring.framework` artifact.

   Maven:

   ```
   <dependency>
       <groupId>com.liferay.portletmvc4spring</groupId>
       <artifactId>com.liferay.portletmvc4spring.framework</artifactId>
       <version>5.1.0</version>
   </dependency>
   <dependency>
       <groupId>com.liferay.portletmvc4spring</groupId>
       <artifactId>com.liferay.portletmvc4spring.security</artifactId>
       <version>5.1.0</version>
   </dependency>
   ```

   Gradle:

   ```
   compile group: 'com.liferay.portletmvc4spring', name: 'com.liferay.portletmvc4spring.framework', version: '5.1.0'
   compile group: 'com.liferay.portletmvc4spring', name: 'com.liferay.portletmvc4spring.security', version: '5.1.0'
   ```

2. In your `WEB-INF/portlet.xml` descriptor, replace uses of `org.springframework.web.portlet.DispatcherPortlet` with `com.liferay.portletmvc4spring.DispatcherPortlet`.

3. Replace uses of the Spring Portlet MVC `AnnotationMethodHandlerAdapter` class with the PortletMVC4Spring `PortletRequestMappingHandlerAdapter` class. `PortletRequestMappingHandlerAdapter` uses the `HandlerMethod` infrastructure that Spring Web MVC 5.1.x is based on.

4. If you specified `AnnotationMethodHandlerAdapter` as a `<bean>` in a Spring configuration descriptor, replace its fully-qualified class name `org.springframework.web.portlet.mvc.annotation.AnnotationMethodHa` with `com.liferay.portletmvc4spring.mvc.method.annotation.PortletRequestMappingHandlerAdapter`.

   Also address these bean property changes:

- `customModelAndViewResolver` (no longer available)
- `customArgumentResolver` (no longer available)
- `customArgumentResolvers` (specify a list of `HandlerMethodArgumentResolver` instead of a list of `WebArgumentResolver`)

5. If you're using Apache Commons Fileupload, update your Spring configuration descriptor:

    1. Replace this legacy bean:

    ```
    <bean id="portletMultipartResolver"
        class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver" />
    ```

    With this new one from PortletMVC4Spring:

    ```
    <bean id="portletMultipartResolver"
        class="com.liferay.portletmvc4spring.multipart.CommonsPortletMultipartResolver" />
    ```

Note: Alternatively, you can use the native Portlet 3.0 file upload
support that PortletMVC4Spring provides by setting the
`portletMultipartResolver` `<bean>` element's `class` to
`com.liferay.portletmvc4spring.multipart.StandardPortletMultipartResolver`

---

2. Remove these dependencies from your `pom.xml` or `build.gradle`
descriptor:

```xml
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
</dependency>
```

5. Throughout your project, replace all uses of the `org.springframework.web.portlet` package path with `com.liferay.portletmvc4spring`.
6. Continue developing your portlet using PortletMVC4Spring.
7. Build and deploy your project.

Congratulations! You migrated your project from Spring Portlet MVC to PortletMVC4Spring.

243.1 Related Topics

PortletMVC4Spring
    Developing a Portlet Using PortletMVC4Spring
    Configuring Dependencies

CHAPTER 244

JSF PORTLET

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! You can use the JSF portlet technology in Liferay DXP by leveraging the Liferay Faces project, which 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 Developer site.

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.
- 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 a 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.

CHAPTER 245

DEVELOPING A JSF PORTLET APPLICATION

To run an existing JSF web app on Liferay DXP, you must leverage the Liferay Faces project. The Liferay Faces Bridge enables you to deploy JSF web apps as portlets without writing portlet-specific code. You must also provide portlet-specific descriptor files to make it compatible with the Liferay DXP platform. The easiest way to do this is by generating a new Liferay JSF Portlet project and migrating your code to it. Then you can deploy your new JSF portlet project to Liferay DXP.

Follow these steps:

1. Create a new JSF portlet project. The following Maven archetypes are available:

    - `com.liferay.faces.archetype.alloy.portlet` (Liferay Faces Alloy portlet)
    - `com.liferay.faces.archetype.bootsfaces.portlet` (Liferay BootsFaces portlet)
    - `com.liferay.faces.archetype.butterfaces.portlet` (Liferay ButterFaces portlet)
    - `com.liferay.faces.archetype.icefaces.portlet` (Liferay ICEFaces portlet)
    - `com.liferay.faces.archetype.jsf.portlet` (Liferay JSF portlet)
    - `com.liferay.faces.archetype.primefaces.portlet` (Liferay PrimeFaces portlet)
    - `com.liferay.faces.archetype.richfaces.portlet` (Liferay RichFaces portlet)

    Choose the archetype that matches your web app's JSF component suite. For example,

    ```
    mvn archetype:generate \
        -DarchetypeGroupId=com.liferay.faces.archetype \
        -DarchetypeArtifactId=com.liferay.faces.archetype.jsf.portlet \
        -DarchetypeVersion=5.0.6 \
        -DgroupId=com.mycompany \
        -DartifactId=com.mycompany.my.jsf.portlet
    ```

    The above archetypes support both Gradle and Maven development by providing a `build.gradle` and `pom.xml`, respectively. For more information, visit faces.liferay.dev.

    Here's the resulting project structure for a JSF Standard portlet:

    - [liferay-jsf-portlet]/ → Arbitrary project name
        - src/

        * main/

            - java/[my-package-path]/
            - bean/ → Sub-package for managed Java beans (optional)
            - dto/ → Sub-package for model (data transfer object) classes (optional)

            - resources/ → Resources to include in the class path
            - i18n.properties → Internationalization configuration
            - log4j.properties → Log4J logging configuration

            - webapp/
            - resources/
            - images/ → Images

            - WEB-INF/
            - resources/ Frontend files (e.g., CSS, JS, XHTML, etc.) that shouldn't be accessed directly by the browser
            - css/ → Stylesheets

            - views/ → View templates
            - faces-config.xml → JSF application configuration file
            - liferay-display.xml → Portlet display configuration
            - liferay-plugin-package.properties → Packaging descriptor
            - liferay-portlet.xml → Liferay-specific portlet configuration
            - portlet.xml → Portlet configuration
            - web.xml → Web application configuration

        - test/java/ → Test source files

2. Update your dependencies as desired. The generated portlet already includes the required artifacts required to deploy a simple JSF portlet to Liferay DXP. For example, the Liferay Faces Bridge artifacts look like this:

   Maven:

```
<dependencies>
    <dependency>
        <groupId>com.liferay.faces</groupId>
        <artifactId>com.liferay.faces.bridge.ext</artifactId>
        <version>5.0.4</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>com.liferay.faces</groupId>
        <artifactId>com.liferay.faces.bridge.impl</artifactId>
        <version>4.1.3</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

   Gradle:

```
dependencies {
    runtime group: 'com.liferay.faces', name: 'com.liferay.faces.bridge.ext', version: '5.0.4'
    runtime group: 'com.liferay.faces', name: 'com.liferay.faces.bridge.impl', version: '4.1.3'
}
```

3. Copy your Java classes to the new `java/[my-package-path]/` folder.

4. Copy your view templates to the new `src/main/webapp/WEBINF/views` folder.

5. Add your frontend files (e.g., CSS, JS, etc.) that shouldn't be accessed directly by the browser to the `webapp/WEB-INF/resources/` folder. For example, your web app's CSS files would reside in the `webapp/WEB-INF/resources/css` folder.

6. Add your image files to the `webapp/resources/images` folder.

7. Add localized messages to the `resources/i18n.properties` file. The messages in the `i18n.properties` file can be accessed via the Expression Language using the implicit `i18n` object provided by Liferay Faces Util. 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.

8. Configure your portlet's logging configuration as desired. The `log4j.properties` file in the `src/main/resources` folder sets properties for the Log4j logging utility defined in your JSF portlet (i.e., `faces-config.xml`).

9. Replace your new JSF portlet's `webapp/WEB-INF/faces-config.xml` with your web app's `faces-config.xml` file. The `faces-config.xml` file is a JSF portlet's application configuration file, which is used to register and configure objects and navigation rules.

10. Replace your new JSF portlet's `webapp/WEB-INF/web.xml` with your web app's `web.xml` file. The `web.xml` file serves as a deployment descriptor that provides necessary configurations for your JSF portlet to deploy and function in Liferay DXP.

    Make sure the Faces Servlet is configured in your `web.xml`:

    ```
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    ```

    This is required to initialize JSF and should be defined in all JSF portlets deployed to Liferay DXP.

11. Modify your `webapp/WEB-INF/portlet.xml` as desired. The `portlet.xml` descriptor describes the portlet to the portlet container. For example, it describes portlet info, security settings, etc. Also, the `javax.portlet.faces.GenericFacesPortlet` is defined here, which 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.

    The `init-param` is also defined here, which ensures your portlet is visible when deployed to Liferay DXP by pointing to your default view template:

```
<init-param>
    <name>javax.portlet.faces.defaultViewId.view</name>
    <value>/WEB-INF/views/view.xhtml</value>
</init-param>
```

12. Modify your `webapp/WEB-INF/liferay-portlet.xml` as desired. It specifies additional information Liferay DXP uses to enhance your portlet: supported security roles, portlet icon, CSS and JavaScript locations, and more. The liferay-portlet-app DTD defines the `liferay-portlet.xml` elements.

13. Modify your `webapp/WEB-INF/liferay-display.xml` as desired. It configures characteristics for displaying your portlet. For example, this `liferay-display.xml` snippet specifies the Widget category in the Add Widget menu:

    ```
    <?xml version="1.0"?>
    <!DOCTYPE display PUBLIC "-//Liferay//DTD Display 7.2.0//EN" "http://www.liferay.com/dtd/liferay-display_7_2_0.dtd">

    <display>
    <category name="category.sample">
        <portlet id="jsf-portlet" />
    </category>
    </display>
    ```

14. Modify your `webapp/WEB-INF/liferay-plugin-package.properties` as desired. It describes the portlet application's packaging and version information and specifies any required OSGi metadata. For example, this `liferay-plugin-package.properties` snippet tells the OSGi container not to scan for CDI annotations in Liferay DXP.

    ```
    -cdiannotations:
    ```

    This is required for JSF portlets leveraging CDI deployed to Liferay DXP. They must reference their own included CDI implementation.

    On deploying the WAR file, the WAB Generator adds the specified OSGi metadata to the resulting web application bundle (WAB) that's deployed to Liferay DXP's runtime framework.

    The liferay-plugin-package reference document describes the `liferay-plugin-package.properties` file.

15. Build and deploy your project.

    Liferay DXP logs the deployment.

```
2019-05-30 14:10:59.405 INFO  [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:261] Processing guestbook-
jsf-portlet.war
...
2019-05-30 14:11:11.401 INFO  [fileinstall-C:/liferay-ce-portal-7.2.0-ga1/osgi/war][BaseDeployer:877] Deploying guestbook-jsf-
portlet.war
...
2019-05-30 14:11:26.379 INFO  [fileinstall-C:/liferay-ce-portal-7.2.0-ga1/osgi/war][BundleStartStopLogger:39] STARTED guestbook-
jsf-portlet_7.2.0.1 [2155]
...
2019-05-30 14:11:67.569 INFO  [fileinstall-C:/liferay-ce-portal-7.2.0-ga1/osgi/war][PortletHotDeployListener:288] 1 portlet for guestbook-
jsf-portlet is available for use
```

The portlet is now available in the Liferay DXP UI. Find your portlet by selecting the *Add* icon () and navigating to *Widgets* and the category you specified (*Sample* is the default category).

Great! You've successfully developed a Liferay JSF portlet and migrated your web app logic to it.

CHAPTER 246

BEAN PORTLET

Important: Bean Portlet is in development and is not yet available.

Portlet 3.0, the JSR 362 standard, features a new style of portlet development called Bean Portlets that use Contexts and Dependency Injection (CDI). Bean Portlets fully leverage all the new Portlet 3.0 features in compliant portals, and are fully supported in Liferay DXP.

Bean Portlets are plain old Java objects (POJOs): they don't need to extend anything. Portlet descriptors declare them to be portlets.

Configuration annotations, phase method annotations, and CDI are some of the features you'll use in Portlet 3.0.

246.1 Portlet Configuration Annotations

The `@PortletConfiguration` annotation describes your portlet to the portlet container. You can use the annotation instead of or in addition to the traditional `portlet.xml` descriptor file. The `@PortletConfiguration` annotation describes your portlet in the portlet code instead of a separate file.

Note: You can configure Bean Portlets using configuration annotations, descriptors, or both. If using annotations and descriptors, the descriptors take precedence.

This example portlet was generated using the `com.liferay.project.templates.cdi.bean.portlet` archetype, and it uses `@PortletConfiguration` and `@LiferayPortletConfiguration` annotations:

```
import com.mycompany.constants.FooPortletKeys;

import com.liferay.bean.portlet.LiferayPortletConfiguration;

import javax.portlet.annotations.LocaleString;
import javax.portlet.annotations.PortletConfiguration;

@PortletConfiguration(
    portletName = FooPortletKeys.Foo,
    title = @LocaleString(value = FooPortletKeys.Foo))
@LiferayPortletConfiguration(
```

```
    portletName = FooPortletKeys.Foo,
    properties = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true"
    }
)
public class FooPortlet {
    ...
}
```

`@PortletConfiguration`'s `portletName` attribute names the portlet. It's the only required attribute. The `title` attribute typically uses a nicer looking name (e.g., uses spaces and capitalization). The `title` above is assigned the key constant `FooPortletKeys.Foo`. You can also localize a title to one or more languages using an array of `@LocaleString` annotations, each specifying a different value for the `locale` element.

The `@LiferayPortletConfiguration` annotation specifies additional Liferay-specific configuration properties. For example, the `com.liferay.portlet.display-category` property lets you assign the Widget category where users will find your portlet. Setting the `com.liferay.portlet.instanceable=true` enables adding multiple instances of the portlet to a page.

Note: The `@PortletConfiguration` and `@LiferayPortletConfiguration` annotations are respectively synonymous with the `javax.portlet.*` and `com.liferay.portlet.*` properties in the OSGi `@Component` annotation (used in Liferay MVC Portlets, for example). If you're familiar with the `portlet.xml` and `liferay-portlet.xml` descriptors, the Portlet Descriptor to OSGi Service Property Map shows you the OSGi `@Component` property equivalent. There's an `@PortletConfiguration` or `@LiferayPortletConfiguration` equivalent setting for each `@Component` property.

To opt-in to Portlet 3.0 features, add the following `@PortletApplication` annotation to the class.

```
@PortletApplication(version="3.0")
```

Once you've configured your portlet, you should declare the objects it uses (depends on).

246.2 Dependency Injection

Bean Portlets use the `@Inject` CDI annotation (by default) to inject dependencies. Apply the annotation to a field you want injected with an object of the specified type. This example portlet injects the portlet's `PortletConfig` object.

```
import javax.inject.Inject;

import javax.portlet.PortletConfig;

public class FooPortlet {

    @Inject
    PortletConfig portletConfig;

    // Invoke methods on portletConfig ...

}
```

Note: OSGi Integration allows you to use OSGi services (e.g., Liferay's `UserLocalService`) in your Bean Portlets.

Portlet 3.0 defines annotations for declaring methods that handle portlet phases.

246.3 Portlet Phase Methods

Phase method annotations apply methods for handling a portlet's phases. You can add them to methods in any class anywhere in the portlet WAR. There's no mandatory method naming convention: assign a phase annotation to the methods you want to invoke to process the phase. Here are the annotations:

Phase	Annotation
Header (new)	`@HeaderMethod`
Render	`@RenderMethod`
Action	`@ActionMethod`
Event	`@EventMethod`
Resource-serving	`@ServeResourceMethod`

You can specify resource dependencies, such as CSS, in the Header phase prior to the Render phase. It helps you avoid loading the same resources multiple times.

You'll definitely want to define a Render method. For example, here's a method invoked during the Render phase:

```
import javax.portlet.annotations.RenderMethod;

@RenderMethod(
    include = "/WEB-INF/jsp/view.jsp",
    portletNames = {FooPortletKeys.Foo})
public String doView() {
    return "Hello from " + portletConfig.getPortletName();
}
```

The `@RenderMethod` annotation sets the method to be invoked during the Render phase of the WAR's portlets matching any of the names listed for the `portletNames` attribute.

The example Render method produces this content:

1. A string greeting `"Hello from " + portletConfig.getPortletName()`
2. The JSP template `/WEB-INF/jsp/view.jsp`—the `@RenderMethod` annotation's `include` attribute references it.

These are just a few of the Portlet 3.0 features that facilitate developing applications. This section covers more Portlet 3.0 features and Bean Portlet demonstrations. Creating and deploying your own Bean Portlet is next.

CHAPTER 247

CREATING A BEAN PORTLET

Important: Bean Portlet is in development and is not yet available.

Your first step in developing a Bean Portlet is to create one. Here you'll generate a Bean Portlet project and deploy your Bean Portlet to Liferay DXP.

1. Generate a Bean Portlet project using a Maven command like this:

```
mvn archetype:generate \
-DarchetypeGroupId=com.liferay \
-DarchetypeArtifactId=com.liferay.project.templates.cdi.bean.portlet \
-DarchetypeVersion=1.0.0 \
-DgroupId=com.mycompany \
-DartifactId=com.mycompany.demo.bean.portlet
```

Here's the resulting folder structure for a Bean Portlet class named `Foo`:

- `com.mycompany.demo.bean.portlet` → Arbitrary project name.
  - `src/main/java/`
    - `com.mycompany.constants.FooPortletKeys` → Declares portlet constants.
    - `com.mycompany.portlet.FooPortlet` → Bean Portlet class.
  - `src/main/webapp/WEB-INF/`
    - `jsp/view.jsp` → Default view template.
    - `beans.xml` → Signals CDI to scan the portlet for annotations.
  - `pom.xml` → Specifies the project's dependencies and packaging.

Here's the example Bean Portlet class:

```
package com.mycompany.portlet;

import com.mycompany.constants.FooPortletKeys;

import com.liferay.bean.portlet.LiferayPortletConfiguration;

import javax.inject.Inject;

import javax.portlet.PortletConfig;
import javax.portlet.annotations.LocaleString;
import javax.portlet.annotations.PortletConfiguration;
import javax.portlet.annotations.RenderMethod;

@PortletConfiguration(
    portletName = FooPortletKeys.Foo,
    title = @LocaleString(value = FooPortletKeys.Foo))
@LiferayPortletConfiguration(
    portletName = FooPortletKeys.Foo,
    properties = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true"
    }
)
public class FooPortlet {

    @Inject
    PortletConfig portletConfig;

    @RenderMethod(
        include = "/WEB-INF/jsp/view.jsp",
        portletNames = {FooPortletKeys.Foo})
    public String doView() {
        return "Hello from " + portletConfig.getPortletName();
    }

}
```

2. Set any portlet configuration or Liferay portlet configuration values using `@PortletConfiguration` and `Liferay@PortletConfiguration` attributes.
3. Inject any CDI beans using the `@Inject` annotation.
4. Update your render method `doView` (it's annotated with `@RenderMethod`). It displays the template `WEB-INF/jsp/view.jsp` by default.
5. Add any other logic you like to your portlet class.
6. Build your portlet:

```
mvn clean package
```

7. Deploy your portlet by copying the portlet WAR to your `[Liferay Home]/deploy` folder. The WAB Generator converts the WAR to an OSGi Web Application Bundle (WAB) and installs it to Liferay's OSGi container.

Liferay DXP logs the deployment.

```
INFO  [main][HotDeployImpl:226] Deploying com.mycompany.demo.bean.portlet from queue
INFO  [main][PluginPackageUtil:1001] Reading plugin package for com.mycompany.demo.bean.portlet
...
INFO  [main][PortletHotDeployListener:181] 1 bean portlets for com.mycompany.demo.bean.portlet are available for use
```

The Bean Portlet is now available in the Liferay DXP UI. The example portlet is in the Widget category you assigned it.

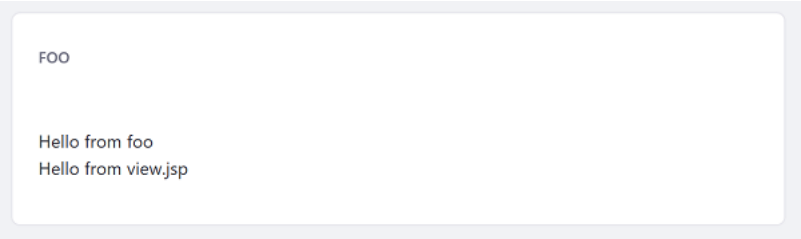

Figure 247.1: The Foo portlet prints the message returned from `doView` method and shows the included JSP's contents.

Congratulations on creating and deploying a Bean Portlet!

247.1 Related Topics

OSGi CDI Integration

CHAPTER 248

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. Service Builder is an object-relational mapping tool that can generate your model, persistence, and service layers from a single `xml` file. Once generated, the code is completely customizable: you can write your own persistence code along with custom SQL if necessary. Regardless of how you produce your persistence code, you can then use Service Builder to implement your app's business logic.

This section demonstrates using Service Builder to

- Generate and customize your persistence framework
- Implement your business logic

When you configure your model and its relationships in your `service.xml` file and run Service Builder, it generates these layers of code:

- **Model layer:** defines objects to represent your project's entities.
- **Persistence layer:** saves entities to and retrieves entities from the database and updates entities.
- **Service layer:** a blank layer ready for you to create your API and business logic. .

Here are some key features these layers contain:

- Stubbed-out classes for implementing custom business logic
- Hibernate configurations
- Configurable caching support
- Flexibility and support for adding custom SQL queries and dynamic queries

---

Note: You don't have to use Service Builder for your back-end services on @product. It's entirely possible to use your persistence framework of choice, such as JPA or Hibernate. Note that "under the hood," Service Builder uses Hibernate.

248.1 Customization via Implementation Classes

Each entity Service Builder generates contains these *implementation* classes:

- **Entity implementation** (`*Impl.java`): Is responsible for customizing the entity.
- **Local service implementation** (`*LocalServiceImpl.java`): 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 service implementation** (`*ServiceImpl.java`): Generated if the `service.xml` is configured for remote services. Remote services usually contain permission checking code and are meant to be accessible from outside the JVM. Service Builder automatically generates code that makes the remote services available via JSON or SOAP, and you can also create your own remote APIs through REST Builder or JAX-RS.

These classes are where you implement custom business logic. They're the only classes generated by Service Builder intended for customization.

248.2 Hibernate Configurations

Service Builder uses the Hibernate persistence framework for object-relational mapping. Service Builder hides the complexities of Hibernate, while still giving you access to technology like dynamic queries and custom SQL. You can take advantage of Object-Relational Mapping (ORM) in your projects without having to manually set up a Hibernate environment or make any configurations.

248.3 Caching

Service Builder 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 must do to enable entity and finder caching in your project is to set the `cache-enabled=true` attribute of your entity's `<entity>` element in your `service.xml` file. Liferay Clustering describes Liferay caching in a cluster.

248.4 Dynamic Query and Custom SQL Query

Service Builder automates many of the common tasks associated with creating database persistence code but it doesn't prevent you from creating custom SQL queries. You can define custom SQL queries in an XML file and implement finder methods to run the queries. If you have some crazy join to do, Service Builder gets out of your way. You can also use dynamic query to access Hibernate's criteria API.

Service Builder is used exclusively throughout Liferay DXP and its applications, so it's well-tested and robust. It saves lots of development time, both initial development time and time that would

have to be spent maintaining, extending, or customizing a project. Now create your own Service Builder project.

CHAPTER 249

CREATING A SERVICE BUILDER PROJECT

To use Service Builder, you must generate the projects where you'll configure your object-relational map. There's an API project and an implementation project.

- `[project]/[project]-api/` → Service interfaces.
- `[project]/[project]-service/` → Service implementations and supporting files.

Here's how to create a Service Builder project.

1. Decide on a project name. If the project is part of an application, name the project after the application.
2. Create a project using Blade CLI and the `service-builder` project template, passing your project name as a parameter. For example, here are Gradle and Maven commands for creating a Service Builder project called `guestbook`.

   Gradle:

   ```
   blade create -t service-builder -p com.liferay.docs.guestbook guestbook
   ```

   Maven:

   ```
   mvn archetype:generate \
   -DarchetypeGroupId=com.liferay \
   -DarchetypeArtifactId=com.liferay.project.templates.service.builder \
   -DgroupId=com.liferay \
   -DartifactId=guestbook \
   -Dpackage=com.liferay.docs.guestbook \
   -Dversion=1.0 \
   -DapiPath=com.liferay.api.path \
   -DliferayVersion=7.2
   ```

Note: To use the Spring dependency injector instead of the Declarative Services dependency injector, use the `--dependency-injector spring` option (Blade CLI) or `-DdependencyInjector=spring` (Maven).

A message like this one reports project creation success:

```
Successfully created project bookmarks in C:\workspaces_liferay\72-ws\modules
```

Blade CLI generates the parent project folder and sub-folders for the `*-api` and `*-service` module projects.

- `guestbook/`
    - `guestbook-api/`
        * `bnd.bnd`
        * `build.gradle`
    - `guestbook-service/`
        * `bnd.bnd`
        * `build.gradle`
        * `service.xml` → Service definition file.
    - `build.gradle`

Congratulations! You've created your Service Builder project. The `service.xml` file is where you'll define your model objects (entities) and services.

249.1 Related Topics

Service Builder Samples
    Service Builder Gradle Plugin
    Service Builder Maven Plugin

CHAPTER 250

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_2_0.dtd specifies the format and requirements of the XML to use.

A `service.xml` was created for you when you created your Service Builder project. It's in your `*-service` module's root folder with an `entity` element named *Foo*. This is (obviously) an example entity, but you can use it as a pattern for creating your own.

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.

If you use Liferay Dev Studio DXP, you can switch between these modes. Of course, you don't have to use Liferay Dev Studio DXP to work on Liferay projects.

Next, you'll specify your service's global information.

CHAPTER 251

DEFINING GLOBAL SERVICE INFORMATION

A service's global information applies to all its entities. It contains the

- Dependency Injector
- Package path
- Namespace options
- Multiversion concurrency control
- Author

251.1 Dependency Injector

The default dependency injector is OSGi Declarative Services. This makes Service Builder work consistently the way other modules do. Prior versions of Liferay used Spring. The only difference is how you inject the services when you go to use them later.

Declarative Services Dependency Injector:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.2.0//EN" "http://www.liferay.com/dtd/liferay-service-builder_7_2_0.dtd">

<service-builder dependency-injector="ds"
	package-path="com.liferay.docs.guestbook">
```

Spring Dependency Injector:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.2.0//EN" "http://www.liferay.com/dtd/liferay-service-builder_7_2_0.dtd">

<service-builder dependency-injector="spring"
	package-path="com.liferay.docs.guestbook">
```

Important: When a project is created using the Service Builder template, the Declarative Services dependency injector and its dependencies are configured for the project by default. To use the Spring dependency injector instead, create the project using the Service Builder template and the `--dependency-injector spring` option (Blade CLI) or `-DdependencyInjector=spring` (Maven).

Note: Prior to Liferay DXP 7.2, Spring was the sole dependency injector. The services were Spring beans. Liferay's Spring bean framework accommodates Spring beans referencing each other: for example, Spring bean A has a Spring bean B field and vice versa. When Spring is the dependency injector, the base services Service Builder generates include local service and persistence fields of all the `service.xml`'s entities. This causes circular references. Since OSGi Declarative Services doesn't accommodate circular references, Service Builder does not create these fields in the base classes when DS is the dependency injector. For more details, see Understanding the Code.

251.2 Package Path

The package path specifies the package where the service and persistence classes are generated. The package path for Guestbook ensures that the `*-api` module's service classes are generated in the `com.liferay.docs.guestbook` package. The persistence classes are generated in a package of the same name in the `*-service` module. A later article describes the package content.

251.3 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.

```
<service-builder dependency-injector="ds"
        package-path="com.liferay.docs.guestbook"
        mvcc-enabled="true">
```

251.4 Namespace Options

Service Builder names the database tables using the service namespace. For example, *GB* could serve as the namespace for a Guestbook application service.

```
<namespace>GB</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 `GB`, the database table names created for the entities start with `GB__` 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>
```

251.5 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.

```
<author>Liferay</author>
```

CHAPTER 252

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 your Java model to the entities you define automatically, giving you a facility for taking Java objects and persisting them. For the Guestbook application, two entities are created according to its `service.xml`: one for Guestbooks and one for Guestbook Entries.

Here's a summary of the `Guestbook` entity information:

- **Name:** `Guestbook`
- **Local service:** *yes*
- **Remote service:** *yes*

And here's what is used for the `GuestbookEntry` entity:

- **Name:** `GuestbookEntry`
- **Local service:** *yes*
- **Remote service:** *yes*

Here's how you define entities:

```
<entity name="Guestbook" uuid="true" local-service="true" remote-service="true">
</entity>

<entity name="GuestbookEntry" uuid="true" local-service="true" remote-service="true">
</entity>
```

The entity's database table name includes the entity name prefixed with the namespace. The Guestbook example creates one database table named `GW_Guestbook` and another named `GB_GuestbookEntry`.

Setting *Local Service* (the `local-service` attribute) to `true` instructs Service Builder to generate local interfaces for the entity's services. 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. You can build a fully-functional application without generating remote services. In that case, you could set your entity 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*.

CHAPTER 253

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, add `<column />` tags to your entity definition:

```
<column name="guestbookId" primary="true" type="long" />
```

Service Builder creates a database field for each column you add to the `service.xml` file. It maps a database field type appropriate to the Java type specified for each column, and it does this across all the databases Liferay supports. Once Service Builder runs, it generates a Hibernate configuration that handles the object-relational mapping. Service Builder automatically generates getter/setter methods in the model class for these attributes. The column's name specifies the name used in the getters and setters that are created for the entity's Java field. The column's type indicates the Java type of this field for the entity. If a column's `primary` (i.e., primary key) attribute is set to `true`, the column becomes part of the primary key for the entity. If only one column has `primary` set to `true`, that column represents the entire primary key for the entity. This is the case in the Guestbook application. If you define multiple columns with the `primary` attribute set to true, 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.

253.1 Create Entity Columns

Define the columns you need for your first entity. The Guestbook entity is simple: it has only two attributes; a primary key and a name:

```
<column name="guestbookId" primary="true" type="long" />
<column name="name" type="String" />
```

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, using the Java type you'll use in your application. Service Builder handles mapping it to SQL for you.

253.2 Support Multi-tenancy

In addition to columns for your entity's primary key and attributes, add portal instance ID and site ID columns. Then you can support Liferay's multi-tenancy features, so that each portal instance and each Site in a portal instance can have independent sets of your application's 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`:

```
<!-- Group instance -->

<column name="groupId" type="long" />
<column name="companyId" type="long" />
```

253.3 Workflow Fields

You can support Liferay's workflow system by adding the fields it needs to track an entity's progress:

```
<!-- Status fields -->

<column name="status" type="int" />
<column name="statusByUserId" type="long" />
<column name="statusByUserName" type="String" />
<column name="statusDate" type="Date" />
```

253.4 Audit Entities

Finally, you can add columns to help audit your entities. To track each entity instance's owner, add a column called `userId` of type `long`. Create a column named `createDate` of type `Date` to note an entity instance's creation date. Add a column named `modifiedDate` of type `Date` to track the last time an entity instance was modified.

```
<!-- Audit fields -->

<column name="userId" type="long" />
<column name="userName" type="String" />
<column name="createDate" type="Date" />
<column name="modifiedDate" type="Date" />
```

Great! Your entities have columns that not only represent their attributes, but also support multi-tenancy, workflow, and auditing. Next, you'll learn how to specify the relationship service entities.

CHAPTER 254

DEFINING RELATIONSHIPS BETWEEN SERVICE ENTITIES

Relationships between database entities or Java objects are necessary for most applications. The Guestbook application, therefore, defines a relationship between a Guestbook and its entries.

As mentioned earlier, each entry must belong to a particular Guestbook. Therefore, each `GuestbookEntry` entity must relate to a `Guestbook` entity.

Create the `GuestbookEntry` entity's fields:

```
<entity name="GuestbookEntry" local-service="true" uuid="true" remote-service="true">

    <column name="entryId" primary="true" type="long" />
    <column name="name" type="String" />
    <column name="email" type="String" />
    <column name="message" type="String" />
    <column name="guestbookId" type="long" />
```

Note the last field in the list is the `guestbookId` field. Since it's the same name as the `Guestbook` object's primary key, a relationship is created between the two objects. If you're using Liferay Dev Studio DXP, you can see this relationship in its diagram mode.

Congratulations! You've related two entities.

Next, add the instance, audit, and status fields mentioned from the previous step to enable Liferay's multi-tenancy, audit, and workflow features.

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.

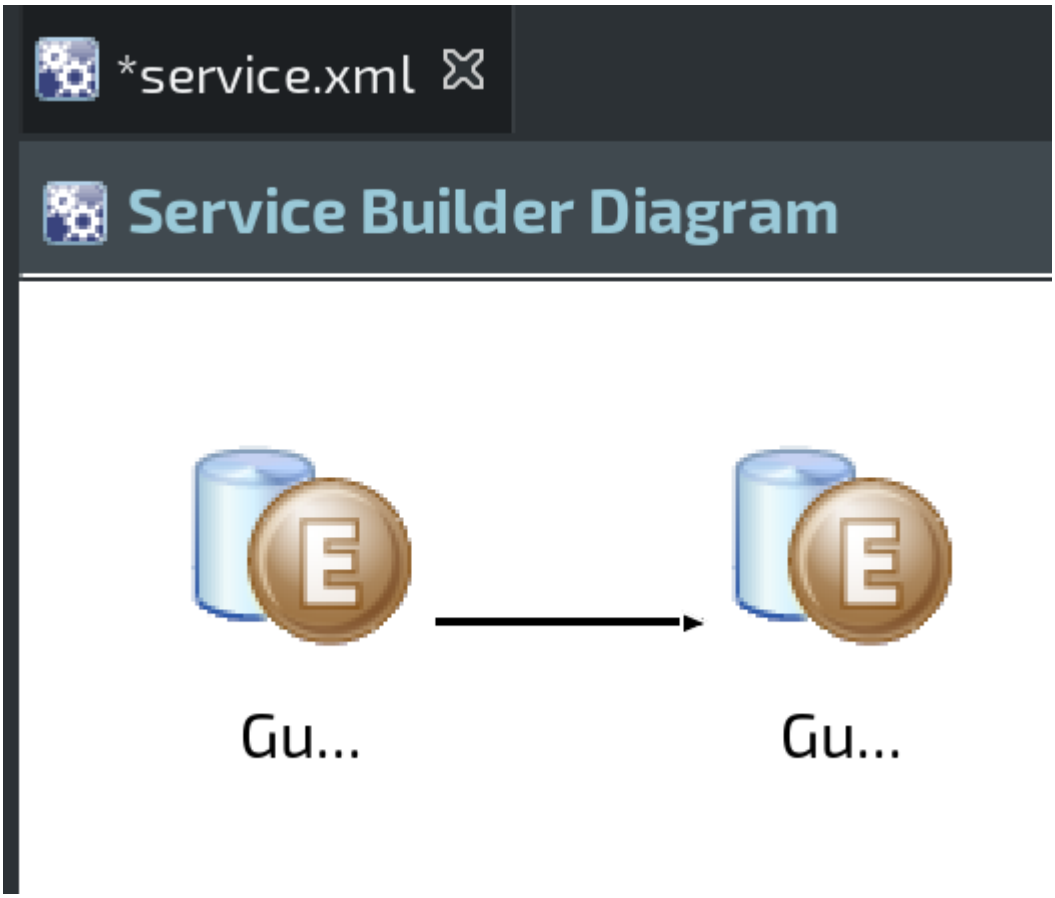

Figure 254.1: Relating entities is a snap in Liferay Dev Studio DXP's *Diagram* mode for `service.xml`.

CHAPTER 255

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 `GuestbookEntry` entities by their creation date. It's easy to specify these default orderings:

```
<order>
    <order-column name="createDate" order-by="desc" />
</order>
```

You can enter `asc` or `desc` for 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.

CHAPTER 256

DEFINING SERVICE ENTITY FINDER METHODS

Finder methods retrieve entity objects from the database based on specified parameters. For each finder defined, Service Builder generates several methods to fetch, find, remove, and count entity instances based on the finder's parameters.

When supporting Liferay's multi-tenancy, it's important to be able to find its entities per Site.

256.1 Creating Finders

Finders are easy to create:

```
<finder name="GroupId" return-type="Collection">
   <finder-column name="groupId" />
</finder>
```

The example above is among the simplest of finders, and is one you should always add if you're supporting multi-tenancy. This finder returns a collection of objects that belong to the Site on which your application has been placed. 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, the Guestbook application generates its entity finder methods in the `-Persistence` classes found in the `/guestbook-api/src/main/java/com/liferay/docs/guestbook/service/persistence` folder and the `-PersistenceImpl` classes in the `/guestbook/src/main/java/com/liferay/docs/service/persistence/impl` folder.

You're not limited to finding by one column, however; you can create multi-column finders:

```
<finder name="G_S" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="status" />
</finder>
```

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.

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.

CHAPTER 257

RUNNING SERVICE BUILDER

Here you'll learn 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.

Open a command line and navigate to your application folder (the folder that contains your `*-api` and `*-service` modules).

257.1 Gradle

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`.

257.2 Maven

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 article, where you can review the code Service Builder generates for your entities.

CHAPTER 258

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:

- `/guestbook-api/src/main/java/com/liferay/docs/guestbook`
- `/guestbook-service/src/main/java/com/liferay/docs/guestbook`

The `guestbook-api` module's interfaces define the Guestbook 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 `guestbook-service` module classes implement the `guestbook-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.

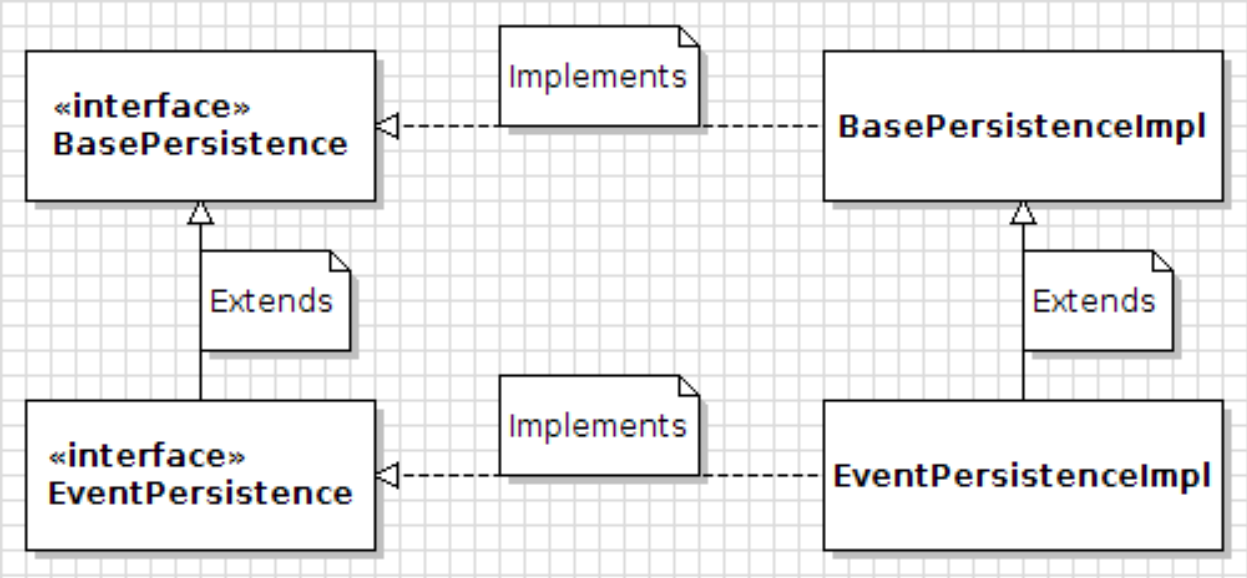

Figure 258.1: Service Builder generates these persistence classes and interfaces for an example entity called *Event*. You shouldn't (and you won't need to) customize any of these classes or interfaces.

- 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: it's where you add your business logic. 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.

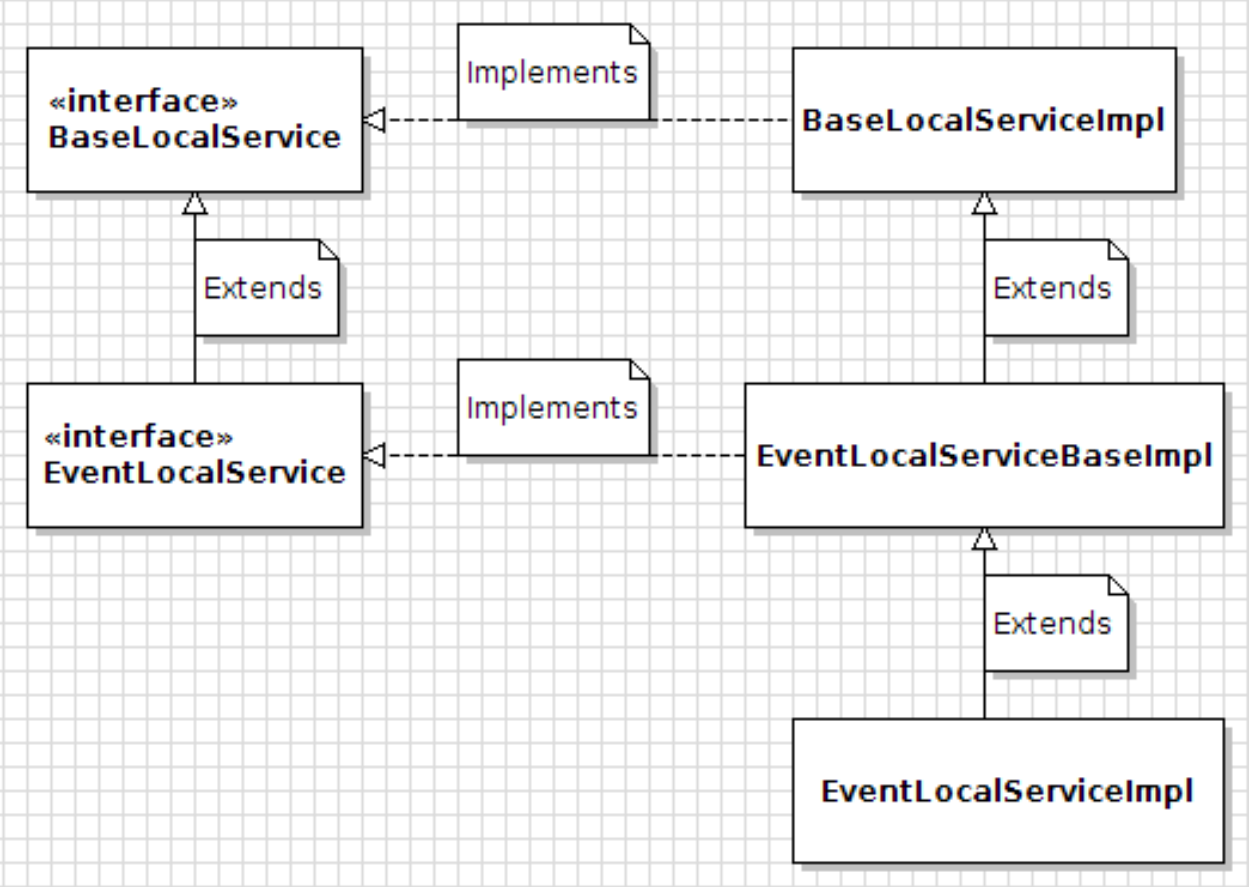

Figure 258.2: Service Builder generates these service classes and interfaces. Only the [ENTITY_NAME]LocalServiceImpl (e.g., EventLocalServiceImpl for the Event entity) allows custom methods to be added to the service layer.

- 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 portal-tools-service-builder module's src/main/resources/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 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 class, or *Impl class, you should run Service Builder again to regenerate the affected interfaces and the service JAR.

Note: Service Builder may generate code that requires adding dependencies to your *-service module build file.

Note: When spring is the dependency injector (see *Dependency Injector* in Defining Global Service Information), the -LocalServiceBaseImpl classes Service Builder generates include -LocalService and -Persistence member fields of all the service.xml's entities. -LocalServiceImpl classes inherit these fields and are Spring beans. The Spring beans can reference each other. For example, Spring bean A can have a Spring bean B field and vice versa. Liferay's spring dependency injector accommodates Spring bean circular references. The ds dependency injector does not accommodate circular references.

When using ds as the dependency injector, -LocalServiceImpl classes are OSGi Declarative Services. Such services start only after all the other services they reference have started. If declarative

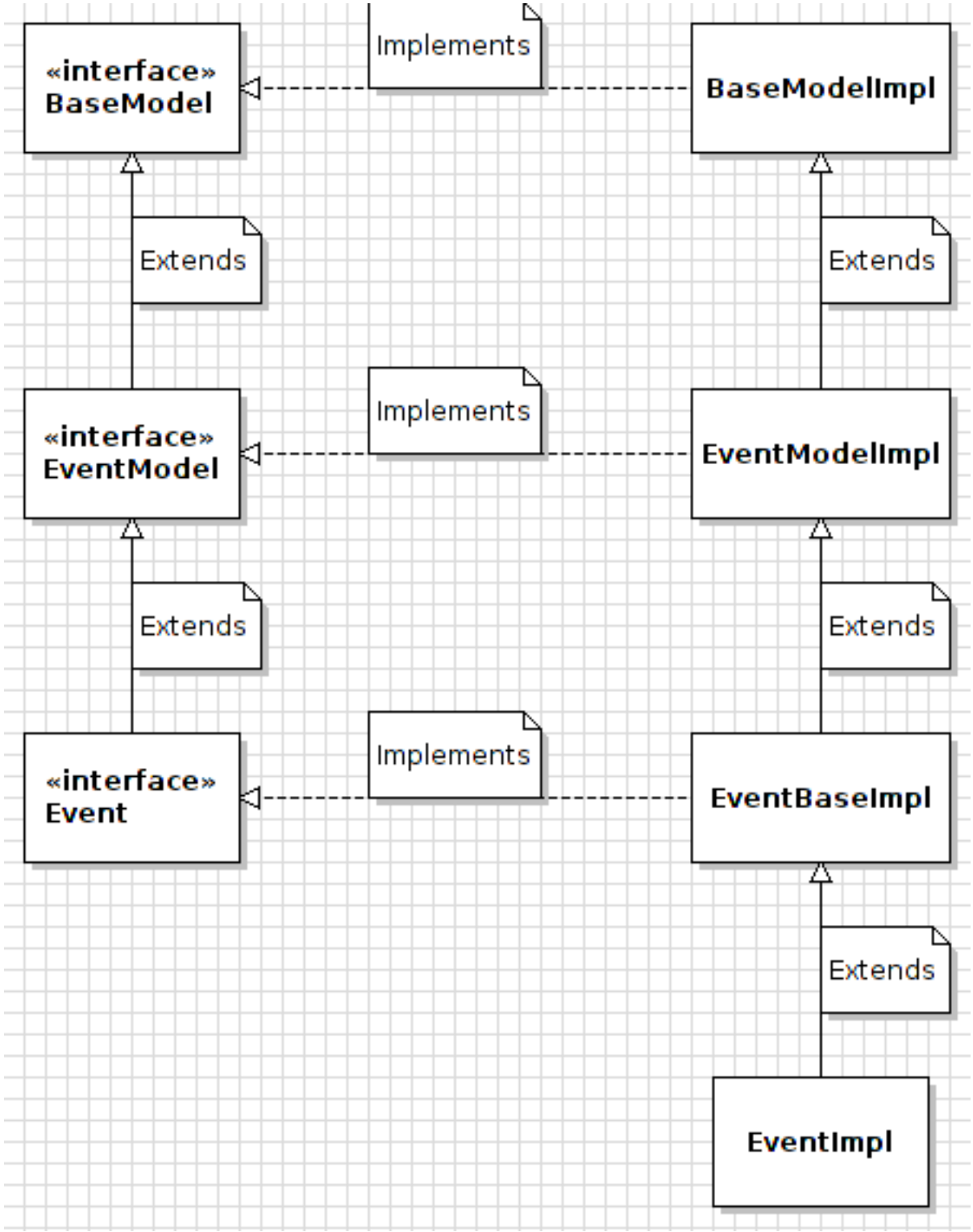

Figure 258.3: Service Builder generates these model classes and interfaces. Only `[ENTITY_NAME]Impl` (e.g., `EventImpl` for the Event entity) allows custom methods to be added to the service layer.

service A has a declarative service B member field and vice versa, neither service can start. For this reason, the `-LocalServiceBaseImpl` classes Service Builder generates don't include `-LocalService` member fields of the `service.xml`'s other entities. When using the `ds` dependency injector, you must make sure member fields you add to service classes don't create circular dependencies.

Congratulations! You've generated your application's initial model, persistence, and service layers and you understand the generated code.

Related Topics

Service Builder

Running Service Builder

Understanding Service Context

Creating Local Services

CHAPTER 259

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.

259.1 Related Topics

Upgrade Processes
    Gradle DB Support Plugin
    Maven DB Support Plugin

CHAPTER 260

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 with selectable dates only in the future. Model hints specify entity data restrictions and other formatting.

You define model hints in a file called `portlet-model-hints.xml`. The `portlet-model-hints.xml` file goes in the service module's `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 on your configuration.

Note: If you chose Spring as the dependency injector, 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.

Since the Guestbook doesn't have much of a model hints 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.

260.1 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.

260.2 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>
```

260.3 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, redeploy your project, and check that the hints defined in your hint collection have taken effect.

Nice work! 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.

CHAPTER 261

CONFIGURING SERVICE.PROPERTIES

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. This property was deprecated in Liferay 7.0.

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`.

Awesome! You now have all the tools necessary to set up your own `service-ext.properties` file.

261.1 Related Topics

Service Builder?

Creating Local Services

CHAPTER 262

CONNECTING SERVICE BUILDER TO AN EXTERNAL DATABASE

If you want to use a database separate from Liferay DXP's, follow these steps:

1. Specify your database and a data source name in your `service.xml`.
2. Create the database manually.
3. Define the data source.
4. Connect your Service Builder module to the data source.
5. Run Service Builder.

There are two different ways to create the connection:

1. **`DataSourceProvider`:** This approach involves implementing a `DataSourceProvider` `ServiceProviderInterface` (SPI). This way requires the fewest files and steps and works regardless of whether your Service Builder module uses the `ds` or `spring` dependency injector.
2. **Spring Beans:** Configure the connection using Spring XML files. This approach only works with Service Builder modules that use the `spring` dependency injection option.

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.

Warning: If your Service Builder services require nested transactions, using an external data source may not be appropriate. Transactions between separate data sources cannot be fully nested. Rollbacks may not propagate between services that use an external data source and Liferay DXP services (or another app's services) that use a different data source.

Since `DataSourceProvider` is the easiest, most versatile approach, it's explained first.

CHAPTER 263

CONNECTING THE DATA SOURCE USING A DATASOURCEPROVIDER

Connecting to an external database by creating and registering a `DataSourceProvider` as a JDK `ServiceProviderInterface` (SPI) is the easiest way. This approach works regardless of whether your Service Builder module uses the `ds` or `spring` dependency injection option and it requires the fewest files and steps.

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.

Warning: If your Service Builder services require nested transactions, using an external data source may not be appropriate for you. Transactions between separate data sources cannot be fully nested. Rollbacks may not propagate between a module that uses an external data source and Liferay DXP services (or another app's services) that use a different data source.

Here are the steps:

1. 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 7.2.0//EN"
    "http://www.liferay.com/dtd/liferay-service-builder_7_2_0.dtd">

<service-builder dependency-injector="spring" 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.

2. Manually create the database you defined in your service.xml.

3. Define the data source. One way is to use portal properties in a portal-ext.properties file. Distinguish your data source 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
```

4. Restart your server if you defined your data source using portal properties.

5. Connect your Service Builder module to the data source by implementing the DataSourceProvider interface. Since the DataSourceProvider must be visible to your *-service module class loader, it's common to put the DataSourceProvider in the *-service module.

   This example uses DataSourceFactoryUtil to create a data source from portal properties that have the prefix jdbc.ext..

```
package com.liferay.external.data.source.test.internal;

import com.liferay.portal.kernel.dao.jdbc.DataSourceFactoryUtil;
import com.liferay.portal.kernel.dao.jdbc.DataSourceProvider;
import com.liferay.portal.kernel.util.PropsUtil;

import javax.sql.DataSource;

public class DataSourceProviderImpl implements DataSourceProvider {

    @Override
    public DataSource getDataSource() {
        try {
            return DataSourceFactoryUtil.initDataSource(
                PropsUtil.getProperties("jdbc.ext.", true));
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

6. Register the implementation as a JDK ServiceProviderInterface (SPI) in a /META-INF/services/com.liferay.portal.kernel.dao.jdbc.DataSourceProvider file in your *-service module. For example, this file registers the DataSourceProvider implementation from the previous step.

```
com.liferay.external.data.source.test.internal.DataSourceProviderImpl
```

7. Run Service Builder.
8. Deploy your `-service` module. If your `DataSourceProvider` is in a different project, deploy it too.

Congratulations! Your module's Service Builder services are persisting data to your external data source.

263.1 Related Topics

Connecting to JNDI Data Sources
Service Builder
Business Logic with Service Builder

CHAPTER 264

CONNECTING THE DATA SOURCE USING SPRING BEANS

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. Here you'll connect Service Builder to a data source using Spring XML files. This approach only works with Service Builder modules that use the `spring` dependency injection option. Here are the steps:

1. Specify your database and a data source name 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. 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.

Warning: If your Service Builder services require nested transactions, using an external data source may not be appropriate for you. Transactions between separate data sources cannot be fully nested. Rollbacks may not propagate between a module that uses an external data source and Liferay DXP services (or another app's services) that use a different data source.

Important: Connecting to an external data source using JNDI is broken in Portal CE 7.2 GA1 and GA2, and in DXP 7.2 releases prior to FP5/SP2. See LPS-107733 for details.

264.1 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 7.2.0//EN"
    "http://www.liferay.com/dtd/liferay-service-builder_7_2_0.dtd">

<service-builder dependency-injector="spring" 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.

264.2 Create the Database Manually

Create the database per the database specification in your `service.xml`.

Next, use portal properties to set your data source.

264.3 Define 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
```

Restart your server if you defined your data source using portal properties.

264.4 Connect Your Service Builder Module to the Data Source Via a Spring Bean

To do this, create a parent context extension (e.g.,`ext-spring.xml`) in your `*-service` module's `src/main/resources/META-INF/spring` folder or in your traditional portlet's `WEB-INF/src/META-INF` folder. Create this folder if it doesn't exist already.

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. Name the alias after the data source name you specified in the `service.xml`.

   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.

Important: The `alias` element's `alias` attribute value must match the data source name specified in the `service.xml`. For example, the alias attribute value above is `extDataSource`.

Note: To use an external data source in multiple Service Builder bundles, you must override the `liferayDataSource` bean in each bundle.

264.5 Run Service Builder

Run Service Builder and deploy your `-service` module. 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.

264.6 Related Topics

Sample Service Builder Application Using External Database via JNDI

Sample Service Builder Application Using External Database via JDBC

Service Builder

Business Logic with Service Builder

CHAPTER 265

MIGRATING A SERVICE BUILDER MODULE FROM SPRING DI TO OSGI DS

Prior to Liferay DXP 7.2, Service Builder modules could only use Spring for dependency injection (DI). Now OSGi Declarative Services (DS) is the default dependency injection mechanism for new Service Builder modules. It's easier to learn and fosters loose coupling between services. If you have an existing Service Builder module that uses Spring DI, you can modify it to use DS.

Here are the conversion steps:

1. Prepare your project for DS
2. Update your Spring bean classes
3. Resolve any circular dependencies

Now prepare your project.

265.1 Step 1: Prepare Your Project for DS

Prepare your project's metadata, dependencies, and `service.xml` for DS.

1. Enable the DS annotation option for your inherited dependencies by adding this line to your `bnd.bnd` file:

   ```
   -dsannotations-options: inherit
   ```

2. Since DS Service Builder modules use the AOP API, add it as a compile dependency in `build.gradle`:

   ```
   compileOnly group: "com.liferay:com.liferay.portal.aop.api", version: "1.0.0"
   ```

3. Add the `dependency-injector="ds"` attribute to your `service.xml` file's `<service-builder>` element:

   ```
   <service-builder dependency-injector="ds" >
   ```

265.2 Step 2: Update Your Spring Bean Classes

Some of your non-generated Spring bean classes must be updated to use DS.

1. Add the `@Component` annotation to your `*LocalServiceImpl`, `*ServiceImpl`, and `*FinderImpl` classes.

2. If the class implements a `*Finder` interface, declare the component as that service type. Example:

   ```
   @Component(service = MyFinder.class)
   ```

3. If the class implements a remote or local service, declare the component as the `com.liferay.portal.aop.AopService` service type. Example:

   ```
   @Component(service = AopService.class)
   ```

4. If it's a remote service (i.e., `-ServiceImpl` instead of `-LocalServiceImpl`), enable JSON web services by setting these properties in your `@Component` annotation:

   - `json.web.service.context.name`
   - `json.web.service.context.path`

   Set them to the same values as the properties in your remote service interface's `@OSGiBeanProperties` annotation.

5. If it's a local service, enable `PersistedModelLocalService` service tracking by setting the `@Component` property `model.class.name` to the service entity's fully qualified class name.

6. Replace all the `@ServiceReference` and `@BeanReference` field annotations with the DS `@Reference` annotation.

7. Use the `@Reference` field annotation to access any other services you need.

8. Run Service Builder to regenerate the interfaces based on your implementation changes.

9. Replace the following methods:

   - `afterPropertiesSet() {...}` → `activate() {...}` and annotate with `@Activate`.
   - `destroy() {...}` → `deactivate() {...}` and annotate with `@Deactivate`.

Next, you'll work out any remaining references you need.

265.3 Step 3: Resolve Any Circular Dependencies

Circular dependencies occur in a module if two or more of its DS services refer to each another (either directly or indirectly). A direct reference occurs, for example, when service `A` references service `B`, and `B` references `A`. Here's what the service components might look like:

`AImpl.java`:

```
@Component(service = A.class)
public class AImpl implements A {
    @Reference
    private B _b;
}
```

`BImpl.java`:

```
@Component(service = B.class)
public class BImpl implements B {
    @Reference
    private A _a;
}
```

`AImpl` and `BImpl` directly depend on each other. This circular dependency prevents each service component from resolving. DS service activation requires that all of a service's dependencies (references) be satisfied.

Note: Service resolution is independent and separate from module (OSGi bundle) resolution:

- Module resolution is determined by the module's manifest.
- Modules resolve before any of their services become active.
- Services inside a module cannot activate if the module cannot resolve.
- A module can resolve even if none of its services activate.

The example above demonstrates a very small circle, composed of only two classes, but a circle can compose more classes. For example, `A` references `B`, `B` references `C`, `C` references `A`. Detecting and resolving such a dependency can be complicated.

There is no general, correct way to detect and resolve circular dependencies; cases vary. However, Liferay provides tools that facilitate detecting circular dependencies and examining the DS service components involved.

- `system:check`: This Gogo shell command provides several checks, including one that detects inactive service components whose required references are unresolved.
- `scr:info [component]`: Execute this Gogo shell command on an unresolved component to report its unresolved references.

Note: Service resolution in DS dependency injection (DI) is different than in services that use Liferay's Spring DI. In the latter case, all Spring beans in the same module act as a single bundle of services that activate together and can bind together before activation. DS doesn't have this feature. With DS, each component in a module is its own service and must resolve on its own.

Congratulations on converting your service module to use Declarative Services.

265.4 Related Topics

Service Builder
    Understanding the Code Service Builder Generates
    Declarative Services

CHAPTER 266

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 that call CRUD operations from the persistence layer. Once you've added your business logic, running Service Builder again regenerates your application's interfaces and makes your new logic available for invocation.

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.

Creating services takes these steps:

1. Deciding to Create Local and Remote Services.
2. Implementing the `add` Method.
3. Implementing the `update` and `delete` Methods.
4. Implementing `get` and `get*Count` Methods
5. Implementing Other Business Logic
6. Integrating with Liferay's Services.

Start with deciding the service types you need.

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.

Now you're ready to implement your business logic.

CHAPTER 267

IMPLEMENTING AN ADD METHOD

Your *LocalServiceImpl represents your service layer, where you create the business logic that operates on your application's data and then calls the persistence layer to persist, retrieve, or delete your data, using the object model defined in service.xml.

One of the first methods you'll likely implement 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 article refers to the Guestbook application's addGuestbookEntry method from GuestbookEntryLocalServiceImpl. To keep things simple, we have excluded the code that integrates with Liferay services, such as assets, social bookmarks, and more.

Here's the Guestbook application's addGuestbookEntry method:

```
public GuestbookEntry 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();
```

```
    GuestbookEntry entry = guestbookEntryPersistence.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);

    guestbookEntryPersistence.update(entry);

    // Calls to other Liferay frameworks go here

    return entry;
}
```

This method uses the parameters to create `GuestbookEntry`. 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. Note that there's no real business logic here; it's a simple application that takes data the user entered, validates it, and then persists it to the database.

267.1 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 GuestbookEntry addEntry(long userId, long guestbookId,
    String name, String email, String message,
    ServiceContext serviceContext) throws PortalException {
    ...
    }
```

This method specifies all the parameters needed to create and populate a `GuestbookEntry` as you specified them in your `service.xml` file. It throws a `PortalException` in case the parameters are invalid or a processing exception occurs (more on this in a later step).

Make sure to account for primary keys of other related entities. For example, the `addEntry` method above includes a parameter `long guestbookId` to associate the new `GuestbookEntry` to a `Guestbook`.

267.2 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 name, String email, String entry) throws PortalException {

    if (Validator.isNull(name)) {
        throw new GuestbookEntryNameException();
    }

    if (!Validator.isEmailAddress(email)) {
        throw new GuestbookEntryEmailException();
    }

    if (Validator.isNull(entry)) {
        throw new GuestbookEntryMessageException();
    }
}
```

Next, generate a primary key for the entity instance you're creating.

267.3 Step 3: Generate a primary key

Entities must each have 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. Always generate primary keys in this way, as it ensures your code is compatible with all the databases Liferay supports.

267.4 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, `GuestbookEntryLocalServiceImpl` as a child of `GuestbookEntryLocalServiceBaseImpl` has a field `guestbookEntryPersistence`, which is a reference to a `GuestbookEntryPersistence` instance.

```
@Reference
protected GuestbookEntryPersistence guestbookEntryPersistence;
```

`GuestbookEntryLocalServiceImpl`'s `addEntry` method creates a `GuestbookEntry` instance using this call:

```
GuestbookEntry entry = guestbookEntryPersistence.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.

267.5 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 `GuestbookEntry` attribute assignments:

```
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);
```

Note that the `ServiceContext` is commonly used to carry an entity's UUID and the `User` is associated to a company.

267.6 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 `GuestbookEntry` is persisted:

```
guestbookEntryPersistence.update(entry);
```

Your entity is persisted for the application.

267.7 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.

CHAPTER 268

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. The main difference between these and the `add*` method is they must know which entity they're updating or deleting.

268.1 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 `GuestbookEntryLocalServiceImpl`'s `updateEntry` method are helpful to examine.

```
public GuestbookEntry 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);

    GuestbookEntry entry = getGuestbookEntry(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);

    guestbookEntryPersistence.update(entry);

    // Integrate with Liferay frameworks 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.

268.2 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 `GuestbookEntryLocalServiceImpl`'s `updateEntry` method signature has an ID parameter (`entryId`) for fetching the `GuestbookEntry` entity instance. Also it has parameters `folderId`, `name`, `url`, and `description` for updating the `GuestbookEntry`'s respective attributes.

```
public GuestbookEntry updateEntry(long userId, long guestbookId, long entryId, String name, String email, String message,
        ServiceContext serviceContext) throws PortalException, SystemException {
```

Note, user ID, group ID, and service context parameters are useful for integrating with Liferay's services. More on that later.

268.3 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.

268.4 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 GuestbookEntryLocalServiceImpl retrieves the GuestbookEntry with the primary key entryId.

```
GuestbookEntry entry = guestbookEntryPersistence.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.

268.5 Step 4: Update the Entity Attributes

Invoke the entity's setter methods to replace its attribute values.

268.6 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;
```

268.7 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.

268.8 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 `GuestbookEntryLocalServiceImpl`'s `deleteEntry` method:

```
public GuestbookEntry deleteEntry(GuestbookEntry entry)
    throws PortalException {

    guestbookEntryPersistence.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.

268.9 Related Topics

Implementing an add method

CHAPTER 269

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.

269.1 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, `getGuestbookEntries` from `GuestbookEntryLocalServiceImpl` returns a range of `GuestbookEntrys` associated with a `Group` primary key:

```
public List<GuestbookEntry> getGuestbookEntries(long groupId, long guestbookId) {
    return guestbookEntryPersistence.findByG_G(groupId, guestbookId);
}
```

Now you know how to leverage finder methods to get entities. Methods that count entities are next.

269.2 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 GuestbookEntryLocalServiceImpl returns the number of GuestbookEntrys that are associated with a group (matching groupId) and a guestbook (matching guestbookId).

```
public int getGuestbookEntriesCount(long groupId, long guestbookId) {
    return guestbookEntryPersistence.countByG_G(groupId, guestbookId);
}
```

Now your local service can get entities matching your criteria and return quick entity counts.

269.3 Service Method Prefixes and Transactional Aspects

Service Builder applies transactions to services by adding @Transactional annotations to the *LocalService and *Service interfaces and their methods. By default, Service Builder applies read-only transactions (e.g., @Transactional (readOnly = true ...)) to service methods prefixed with any of these words:

- dynamicQuery
- fetch
- get
- has
- is
- load
- reindex
- search

Since these methods operate in read-only transactions, Liferay DXP optimizes their performance. Transactional service methods that don't have the read-only setting operate in regular transactions.

Note: A method implementation can override its interface's @Transactional annotation attributes. For example, applying @Transactional (readOnly = false ...) to a method implementation makes it operate in a transaction that is not read only.

Important: In methods that operate in read-only transactions, invoking a service method that persists data (adds, updates, or deletes data) must be done via the service object. Using the service object ensures that the defined transactional behavior is applied.

```
someService.addSomething();
```

For example, this *LocalServiceImpl's getter method adds (*persists*) a ClassName object if no object with that value exists.

```
public ClassName getClassName(String value) {
    if (Validator.isNull(value)) {
        return _nullClassName;
    }

    ClassName className = _classNames.get(value);

    if (className == null) {
        try {
```

```
            className = classNameLocalService.addClassName(value);
            ...
        }
        ...
    }
    ...
}
```

Using the service object `classNameLocalService` to invoke its `addClassName` method applies the service method's transaction (the regular transaction specified for the method in the `*Service` interface). If the `addClassName` method was invoked WITHOUT using the service object, the `ClassName` object would not persist because the method's regular transaction would not be applied.

269.4 Related Topics

Creating Local Services
- Implementing an Add Method
- Defining Service Entity Finder Methods
- Understanding the Code Generated by Service Builder

CHAPTER 270

IMPLEMENTING ANY OTHER BUSINESS LOGIC

This section's earlier local service articles 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.

Since the Guestbook application doesn't have any business logic, the Bookmarks application, which is extremely similar, (Bookmark Folders and Bookmark entries instead of Guestbooks and Guestbook entries) is used here to illustrate simple business logic. 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);
```

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 Spring Service Builder Code

CHAPTER 271

INTEGRATING WITH LIFERAY'S FRAMEWORKS

New car buyers expect certain standard features: power windows, cruise control, floor mats (at least the cheap ones), and so on. Similarly, users expect applications to have certain features, and those features should behave consistently across applications.

For example, a user might expect the app's content can be shared on social networks, tagged and rated, and discussed in comments. Liferay's frameworks implement these features users expect to see. Integrating with the frameworks is easy, and the frameworks provide intuitive, consistent user experiences.

Here are some of Liferay's most popular frameworks:

Permissions: Defines resources and permissions for entities and actions that can be performed on them.

Configurable Applications: Provide configuration screens in the Control Panel.

Workflow: Equips entities for reviewing in workflows before publishing.

Item Selector: Provides a consistent experience for browsing and selecting entities.

Asset Framework: Provides a way to make entity data generic, so common tasks can be performed on them. This enables users to tag, categorize, rate, prioritize, and comment on anything that has been asset-enabled. Users can relate entities to each other as assets, and entities can be published in the Asset Publisher.

- Tags and Categories: Enables users to tag entities and categorize them into logical hierarchies.
- Priority: Users can ascribe numerical priorities to entities.
- Related Assets: Users can associate one entity with another as an asset.
- Asset Renderer: Enables entities appearing in Asset Publisher queries.
- 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.

Export/Import: 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: Modify content behind the scenes without affecting the live site, and then publish to the live site when the content is ready.

Search: Index entities for searching.

Recycle Bin: Instead of deleting entities, put them into the Recycle Bin. Entities can be restored from the Recycle Bin or deleted permanently (manually or per a schedule).

271.1 Related Topics

Internationalization
    JavaScript Module Loaders
    Java Taglibs
    Upgrade Processes

CHAPTER 272

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.

272.1 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: If you chose Spring as the dependency injector, 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.

272.2 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.

272.3 Related Topics

Creating Local Services
    Invoking Local Services
    Invoking Local Services from Spring Service Builder Code
    OSGi Services and Dependency Injection with Declarative Services

CHAPTER 273

INVOKING SERVICES FROM SPRING SERVICE BUILDER CODE

When using Spring as the dependency injector, 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.FooLocalServiceImpl`
`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.FooPersistenceImpl`

Since these classes are Spring Beans and NOT OSGi Declarative Services components, they don't use the `@Reference` Declarative Services annotation to inject themselves. 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.

273.1 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 must declare a `BarLocalService` field and apply an `@BeanReference` annotation to it.

```
@BeanReference
private BarLocalService _barLocalService;
```

The `@BeanReference` tells Liferay's AOP to 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.

273.2 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 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.

273.3 Related Topics

Invoking Local Services

Service Trackers

CHAPTER 274

ADVANCED QUERIES

Service Builder doesn't limit you to what you can cook up with `<finder />` elements in `service.xml`. If simple finders aren't sufficient for getting data out of your application, you can use Liferay's Dynamic Query API, which wraps Hibernate's Criteria API, or your own SQL to make exactly the queries you need.

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.

Whichever way you decide to implement your custom queries, this guide shows you how. Here are the steps:

1. If using SQL, create your SQL query.
2. Define a custom finder method.
3. Implement your finder using Dynamic Query API or SQL.
4. Add a method to your `*LocalServiceImpl` class that invokes your finder.

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.

CHAPTER 275

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. Here, you'll learn how.

The Guestbook application has 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_GuestbookEntry`, 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*.

Using Custom SQL in Service Builder is the same as using dynamic queries; it just takes an additional first step to place the SQL you want to run in an XML file. If you plan to use dynamic queries instead, skip the rest of this tutorial and move on to the next one.

275.1 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 the next step.

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_GuestbookEntry` 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_GuestbookEntry.*
            FROM GB_GuestbookEntry
            INNER JOIN
                GB_Guestbook ON GB_GuestbookEntry.guestbookId = GB_Guestbook.guestbookId
            WHERE
                (GB_GuestbookEntry.name LIKE ?) AND
                (GB_GuestbookEntry.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.

Congratulations on developing a custom SQL query and custom finder for your application!

275.2 Related Topics

Customizing Liferay Services

CHAPTER 276

DEFINING A CUSTOM FINDER METHOD

Dynamic queries and custom SQL belong in finder methods. You implement them and then make them available through an interface. This article 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 `GuestbookFinder` 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. You can do this via the Dynamic Query API or Custom SQL. The next tutorial covers Dynamic Query. To simply call custom SQL you have written, create a finder method to run your SQL:

```
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";

@Reference
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! You've implemented your finder class, and if you're using custom SQL, you've even implemented a method to call your finder. If you're using Dynamic Query, the next tutorial shows you how to implement a dynamic query finder method.

CHAPTER 277

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.

277.1 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.

277.2 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 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.

277.3 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.

277.4 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 this 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.

277.5 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.

277.6 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.

277.7 Executing the Dynamic Query

In the previous article, 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.

CHAPTER 278

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 or custom SQL. Now how do you call your finder method from the service layer?

When you ran Service Builder (if you haven't run it yet; run it now), 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 DXP, 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 GuestbookEntryLocalServiceImpl class:

```
public List<GuestbookEntry> 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 process of developing custom queries in a custom finder and exposing it as a service for your portlet!

278.1 Related Topics

Creating Local Service
    Invoking Local Services

CHAPTER 279

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.

279.1 Related Topics

Creating Local Service
    Invoking Local Services

CHAPTER 280

REST Builder

Note: This documentation is in beta. Stay tuned for more to come!

Liferay DXP's headless REST APIs follow the OpenAPI specification and let your apps consume RESTful web services. These APIs are developed using a mixture of the Contract First and Contract Last development approaches. This presents a best-of-both-worlds approach to API development. For more detailed information, see Headless REST APIs.

Here, you'll learn how to use Liferay's REST Builder tool to create headless REST APIs for your own apps. REST Builder is an API generator that consumes OpenAPI profiles and generates the API scaffolding: JAX-RS endpoints, parsing, XML generation, and advanced features like filtering or multipart (binary file) support. The developer only has to fill in the resource implementations, calling Liferay DXP's remote services.

Read on to learn how to generate REST services with REST Builder!

CHAPTER 281

GENERATING APIS WITH REST BUILDER

Note: This documentation is in beta. Stay tuned for more to come!

Follow these steps to use REST Builder to create a headless REST API for your app:

1. Create a project.
2. Install REST Builder. For instructions on this, see REST Builder Gradle Plugin.
3. Run `gradlew clean deploy`. Note that your Gradle wrapper may not be in your app's project directory, so you may need to use `..` to locate it (e.g., `../../../gradlew clean deploy`).
4. Create the `*-api` and `*-impl` projects with the usual files (`build.gradle`, `bnd.bnd`). Also create a `rest-config.yaml` with the author, paths, and packages. For example, here's the `rest-config.yaml` for Liferay's `headless-delivery` API:

```
apiDir: "../headless-delivery-api/src/main/java"
apiPackagePath: "com.liferay.headless.delivery"
application:
    baseURI: "/headless-delivery"
    className: "HeadlessDeliveryApplication"
    name: "Liferay.Headless.Delivery"
author: "Javier Gamarra"
clientDir: "../headless-delivery-client/src/main/java"
testDir: "../headless-delivery-test/src/testIntegration/java"
```

5. In your `*-impl` module's root folder, write your OpenAPI profile in YAML. You can use the Swagger Editor to validate syntax and ensure compliance with the OpenAPI specification.
6. In your `*-impl` module folder, run `gradlew buildREST` (make sure you locate your Gradle wrapper as instructed in step two above).
7. REST Builder generates the interfaces with the JAX-RS endpoints. It also generates a `*ResourceImpl` class where you must implement the business logic for each service.
8. After implementing the business logic for each service, deploy your modules. Your APIs are then available at this URL:

```
http://[host]:[port]/o/[APPLICATION_CLASSNAME]/[OPEN_API_VERSION]/
```

You can also execute `jaxrs:check` in the OSGi console to see all the JAX-RS endpoints.

281.1 Related Topics

REST Builder
    Headless REST APIs
    REST Builder Gradle Plugin

CHAPTER 282

TROUBLESHOOTING APPLICATION DEVELOPMENT ISSUES

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.

282.1 Modules

How can I configure dependencies on Liferay artifacts?

```
<p>See <a href="/docs/7-2/customization/-/knowledge_base/c/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
2/appdev/-/knowledge_base/a/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
2/appdev/-/knowledge_base/a/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?

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>
To satisfy the requirement, <a href="/docs/7-2/appdev/-/knowledge_base/a/resolving-bundle-requirements">find a module that provides the capability

An IllegalContextNameException reports that my bundle's context name does not follow Bundle-SymbolicName syntax. How can I fix the context name?

<a href="/docs/7-2/appdev/-/knowledge_base/a/resolving-bundle-symbolicname-syntax-issues">Adjust the <code>Bundle-SymbolicName</code> to adhere to

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.

After creating a relational mapping between Service Builder entities, my portlet is using too much memory. What can I do?

<a href="/docs/7-2/appdev/-/knowledge_base/a/disabling-cache-for-table-mapper-tables">Disabling the cache related to the entity mapping lowers mem

282.2 Services and Components

How can I see what's happening in the OSGi container?

Run a System Check..

How can I detect unresolved OSGi components?

module components that use Service Builder use Dependency Manager (DM) and most other module components use Declarative Services (DS). <a href="/d
2/appdev/-/knowledge_base/a/detecting-unresolved-osgi-components">Gogo shell commands and tools help you find and inspect unsatisfied component refer

What is the safest way to call OSGi services from non-OSGi code?

See <a href="/docs/7-2/frameworks/-/knowledge_base/f/using-a-service-tracker

">Calling Non-OSGi Code that Uses OSGi Services.

CHAPTER 283

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
- TRACE
- 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.

283.1 Related Topics

Implementing Logging

CHAPTER 284

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.

284.1 Related Topics

Configuring Dependencies

CHAPTER 285

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.

285.1 Related Topics

Using the WAB Generator

CHAPTER 286

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.

286.1 Related Topics

Detecting Unresolved OSGi Components
    Felix Gogo Shell

CHAPTER 287

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.

CHAPTER 288

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`

288.1 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.

288.2 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.

288.3 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.

288.4 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.

These features detect unresolved Service Builder related components.

- Unavailable Component Scanner
- dm na Command
- ServiceProxyFactory

288.5 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 m
```

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.

288.6 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."

288.7 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`.

288.8 Related Topics

System Check

CHAPTER 289

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.

289.1 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.

289.2 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.

CHAPTER 290

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:

   - `trace`: Provides more information than debug. This is the most verbose message level.
   - `debug`: Event and application information helpful for debugging.
   - `info`: High level events.
   - `warn`: Information that might, but does not necessarily, indicate a problem.
   - `error`: Normal errors. This is the least verbose message level.

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.

290.1 Related Topics

Adjusting Module Logging

CHAPTER 291

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 class loader 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 class loader 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]
```

291.1 Related Topics

Configuring Dependencies

CHAPTER 292

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 Liferay'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.

292.1 Related Topics

Configuring Dependencies
    Adding Third Party Libraries to a Module
    Felix Gogo Shell
    Finding Artifacts

CHAPTER 293

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 web app scope or the application server scope.
4. The missing class belongs to a Java runtime package.

This tutorial explains how to handle each case.

293.1 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?.

293.2 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 Resolving Third Party Library Package Dependencies.

293.3 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.

293.4 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 `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.

Your module should resolve and install.

293.5 Related Topics

Backing up Liferay Installations

Resolving Third Party Library Package Dependencies

CHAPTER 294

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`.

294.1 Related Topics

Detecting Unresolved OSGi Components

Gogo shell

CHAPTER 295

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

Click a question to view the answer.

295.1 CSS

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>
```

295.2 Modules

Why does my JQuery module throw an anonymous module error when I try to load it?

If you're using an external library that you host, you must disable the Expose Global option as described in the Using External JavaScript Libraries tutorial.

Why are my source maps not showing for my Angular or Typescript module?

This is due to LPS-83052.
To solve this, activate the <code>inlineSources</code> compiler option in your project's <code>tsconfig.json</code> file.

I'm using the liferay-npm-bundler for multiple projects. How can I disable analytics tracking for the liferay-npm-bundler in my projects?

There are a couple options you can use to disable reporting:

   Use the <code>--no-tracking</code> flag in your <code>package.json</code>'s build script to disable reporting:
  <pre><code>liferay-npm-bundler --no-tracking</code></pre>

  Create a <code>.liferay-npm-bundler-no-tracking</code> file in your project's root folder, or any of its ancestors, to disable reporting.
  This equates to answering <code>No</code> to the <code>May liferay-npm-bundler anonymously report usage statistics to improve the tool over time</code> question.

295.3 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?

By default, the Senna JS SPA engine is enabled.
If you want to use a custom router in your portlet instead, follow the instructions in the SPA documentation to blacklist your portlet from SPA.

Part IV

Liferay Frameworks

Chapter 296

Application Security

Liferay's development framework provides an application security platform with 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

Read on to learn about implementing Liferay's security framework!

CHAPTER 297

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.

CHAPTER 298

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.

298.1 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 7.2.0//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.

298.2 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 `<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.

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.

298.3 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.

CHAPTER 299

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.

299.1 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.

299.2 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.

CHAPTER 300

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.

CHAPTER 301

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.

301.1 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.

301.2 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.

301.3 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.

CHAPTER 302

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.

302.1 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://xmlns.jcp.org/xml/ns/portlet" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.o
app_3_0.xsd" version="3.0">
    <portlet>
        <portlet-name>guestbook-war</portlet-name>
        <display-name>guestbook-war</display-name>
        <portlet-class>com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet</portlet-class>
        <init-param>
            <name>template-path</name>
            <value>/</value>
        </init-param>
        <init-param>
            <name>view-template</name>
            <value>/view.jsp</value>
        </init-param>
        <expiration-cache>0</expiration-cache>
        <supports>
            <mime-type>text/html</mime-type>
        </supports>
        <resource-bundle>content.Language</resource-bundle>
        <portlet-info>
            <title>guestbook-war</title>
            <short-title>guestbook-war</short-title>
            <keywords>guestbook-war</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>
```

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.

302.2 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.

302.3 Related Topics

Liferay Permissions

Asset Framework

Portlets

Understanding ServiceContext

CHAPTER 303

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 `Authenticator`s 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 enables authentication to Liferay DXP using credentials provided in the HTTP header from another system.
- **Authentication Pipelines:** if you must 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.

CHAPTER 304

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.

304.1 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.

304.2 Related Topics

Password-Based Authentication Pipelines

Writing a Custom Login Portlet

CHAPTER 305

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`.

305.1 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.

305.2 Creating an Authenticator

This example is an `Authenticator` that only allows users whose email addresses end with *@liferay.com* or *@example.com*. You can implement this using one module that does everything. If you think other modules might 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:

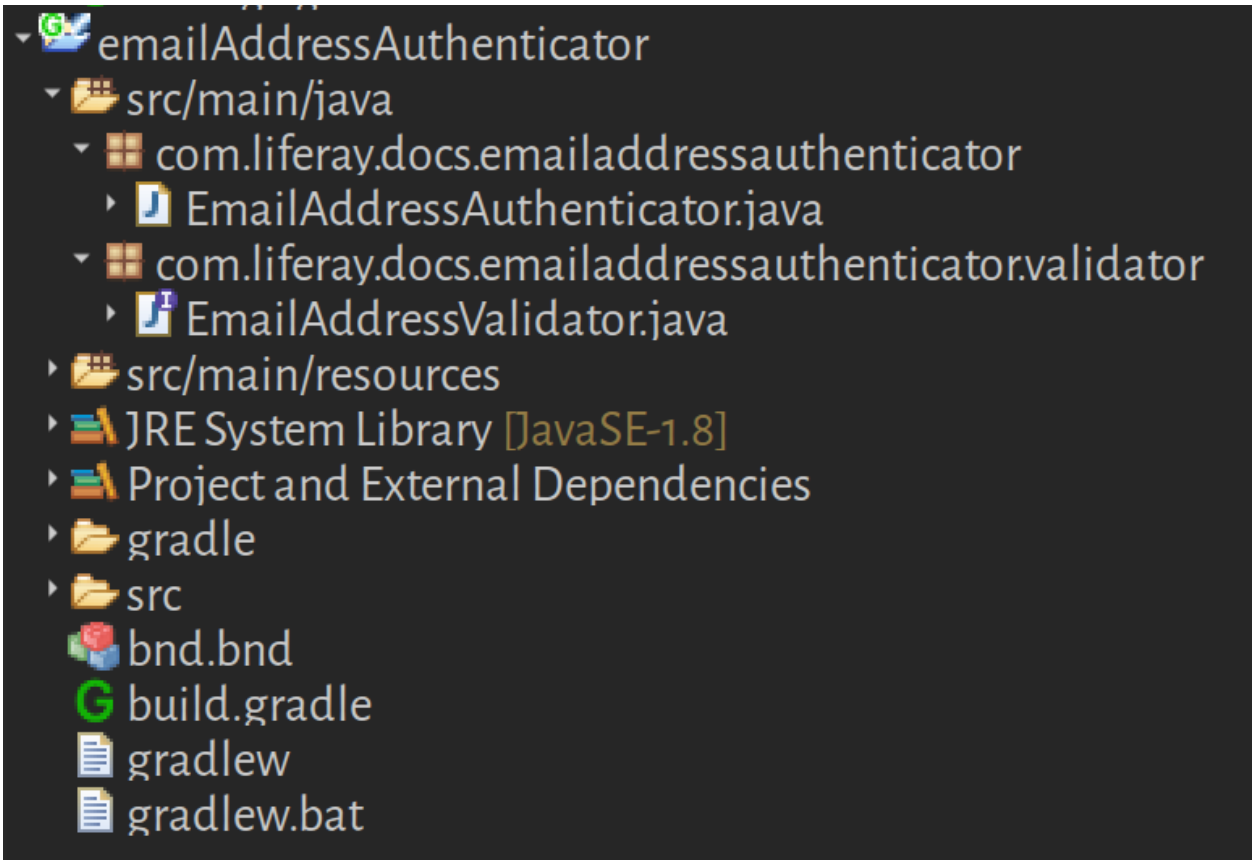

Figure 305.1: The Authenticator module contains the validator's interface and the authenticator.

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
    public boolean isValidEmailAddress(String emailAddress) {

        if (_validEmailDomains.contains(
            emailAddress.substring(emailAddress.indexOf('@')))) {

            return true;
        }
        return false;
```

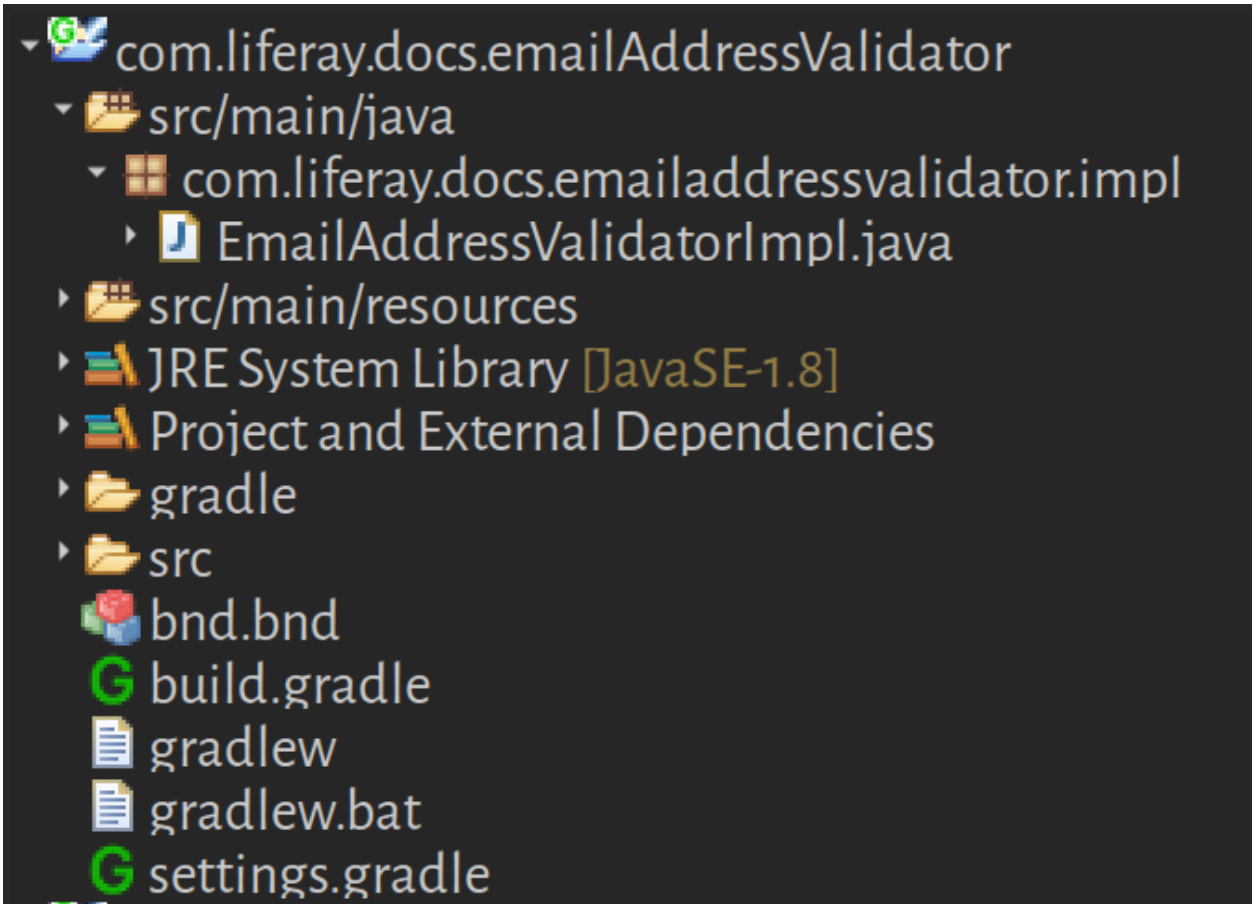

Figure 305.2: The validator project implements the Validator Interface and depends on the authenticator module.

```
    }

    private Set<String> _validEmailDomains =
        new HashSet<String>(Arrays.asList(new String[] {"@liferay.com", "@example.com"}));
}
```

This code checks to make sure that the email address is from the *@liferay.com* or *@example.com* domains. The only other interesting part of this module is the Gradle build script, because it defines a compile-only dependency between the two projects. This is divided into two files: a `settings.gradle` and a `build.gradle`.

The `settings.gradle` file defines the location of the project (the `Authenticator`) the validator depends on:

```
include ':emailAddressAuthenticator'
project(':emailAddressAuthenticator').projectDir = new File(settingsDir, '../com.liferay.docs.emailAddressAuthenticator')
```

Since this project contains the interface, it must be on the classpath at compile time, which is when `build.gradle` is running:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins", version: "3.0.23"
```

```
    }

    repositories {
        mavenLocal()

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

apply plugin: "com.liferay.plugin"

dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.compendium", version: "5.0.0"

    compileOnly project(":emailAddressAuthenticator")
}

repositories {
    mavenLocal()

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

Note the line in the dependencies section that refers to the `Authenticator` project defined in `settings.gradle`.

When these projects are deployed, the `Authenticator` you defined runs, enforcing logins for the two domains specified in the validator.

305.3 Related Topics

Auto Login

Writing a Custom Login Portlet

CHAPTER 306

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.

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.

306.1 Authenticating to Liferay DXP

Note: When developing a login portlet, 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.

306.2 Related Topics

Password-Based Authentication Pipelines

Auto Login

CHAPTER 307

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 using 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.

307.1 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.

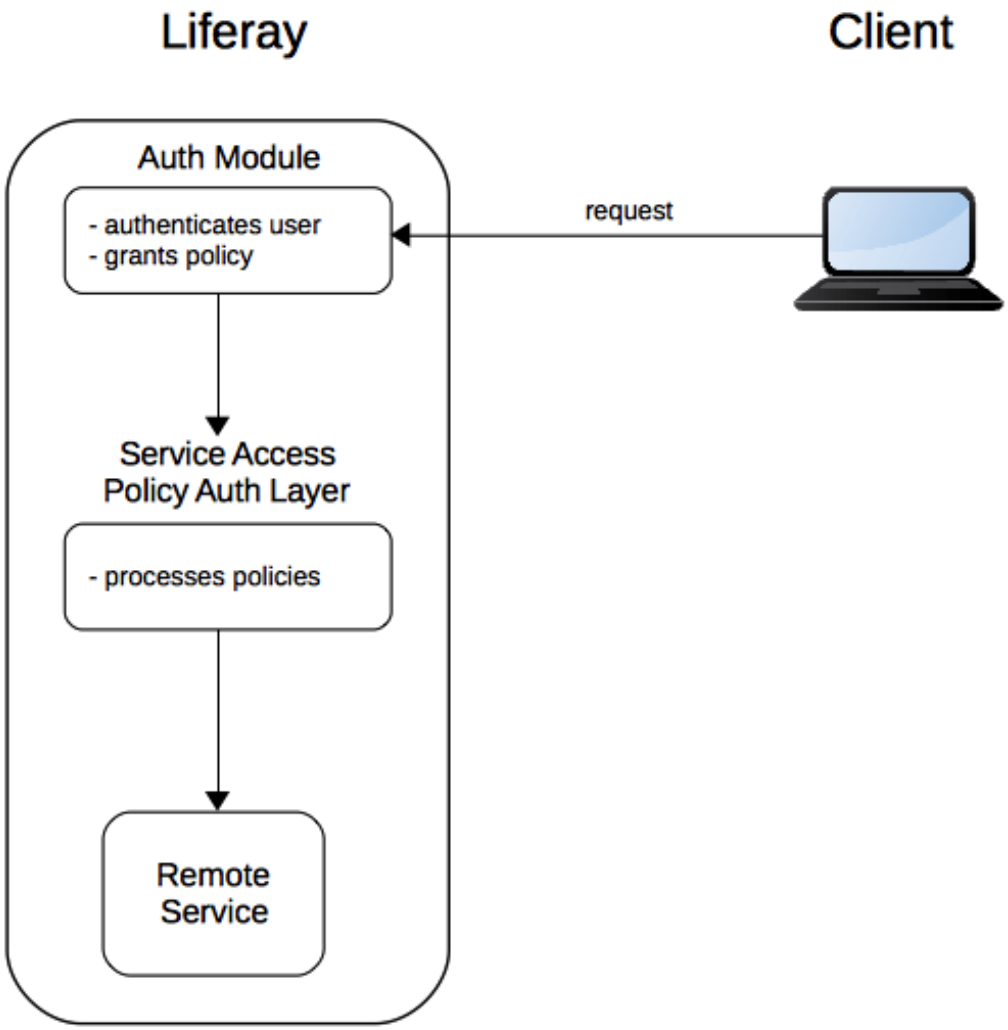

Figure 307.1: The authorization module maps the credentials or token to the proper Service Access Policy.

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.

307.2 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.

   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. 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.

307.3 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 308

FRAMEWORKS

To make your applications more fully featured and to develop them faster, you can make use of Liferay's development frameworks. These help you create commonly used features—like search, tagging, and comments—without having to develop them from scratch. And since these features are tried and tested, you can rest assured knowing they're bug free.

Here are just a few frameworks you'll find here:

A fully-fledged permissions system: Implement permissions the way they're implemented with the applications that ship with Liferay DXP for a consistent, seamless, and robust experience.

Assets: Publish data from your application across the system, making it available to those who need it, and enabling other features like tagging, categorizing, and comments.

Search: If you have a data-driven application, you can add search capabilities by integrating with Liferay's search indexer.

A configuration system with auto-generated or custom UI: Are you providing user-configurable options in your application? Make use of Liferay's configuration system and provide a clean and consistent experience for your users.

File management: Will your applications work with files? Use Liferay's Documents and Media API to manage them.

Import/Export: Use Liferay's import/export system to make your application's data portable or to stage it for publication to production systems.

Web Fragments: Provide your content managers with dynamic chunks of functionality they can use as building blocks for web pages.

Workflow: Run the data from your application through an approval process.

This really just scratches the surface. From pop-up list selectors to a social networking API, as a Liferay developer, you have access to tons of frameworks that make your life easier.

CHAPTER 309

ASSET FRAMEWORK

The asset framework is behind many of Liferay's most powerful features. It provides tools for displaying and interacting with various types of content and data. 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. Using the asset framework is also the first step for integrating other important frameworks like Segmentation and Personalization or Workflow.

As background, the term *asset* refers to any type of content: 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. After that is completed, you can get started asset enabling your application.

This section explores 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.

There are several steps to creating an asset and taking full advantage of the asset framework.

309.1 Persistence Operations for 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.

309.2 Rendering an Asset

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 prerequisites for asset enabling an application:

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.

After you have met all the prerequisites, 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.

309.3 Asset Features

Once you have done the necessary work to persist your assets and render them, you can enable Tags, Categories, and Related Assets.

309.4 Tags and Categories

Tags and Categories are two ways that you can organize and connect assets. Tags are simple *ad hoc* groups. Any two assets with the same tag are connected by that tag. Categories are a form of hierarchical organization where an administrator can define a number of categories for organization content, images, or other types of assets and use those categories to help users find what they're looking for.

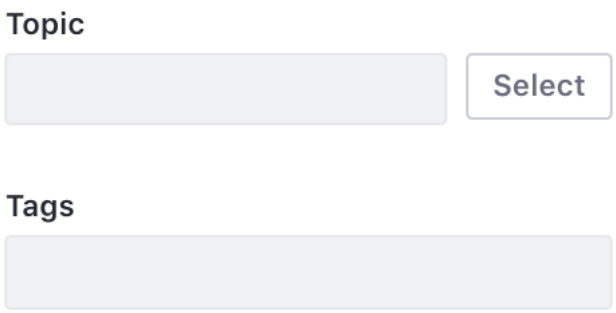

Figure 309.1: Adding category and tag input options lets authors aggregate and label custom entities.

309.5 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.

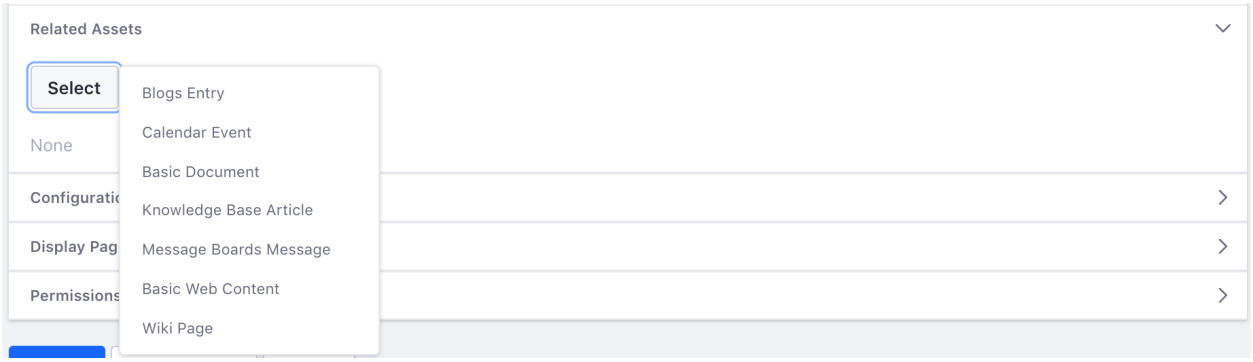

Figure 309.2: You and your users can find it helpful to relate assets to entities, such as this blogs entry.

309.6 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.

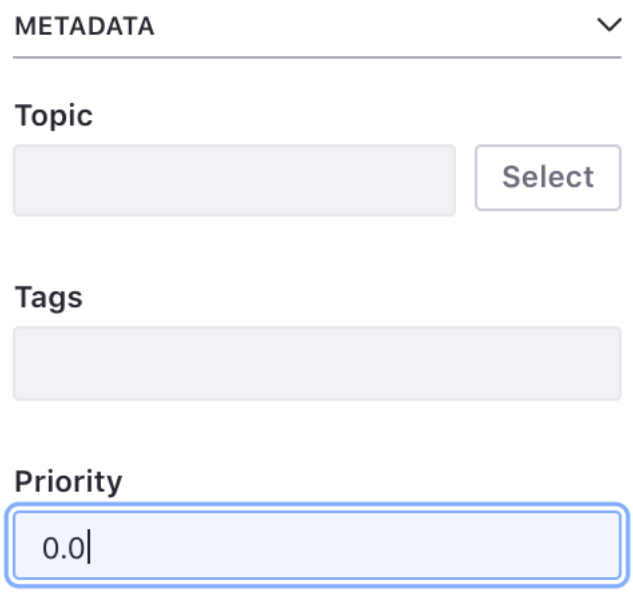

Figure 309.3: The Priority field lets users set an asset's priority.

Ready to implement assets? The rest of the tutorials show you how.

CHAPTER 310

ADDING, UPDATING, AND DELETING ASSETS

This section shows you how to enable assets for your custom entities and implement indexes for them. It's time to get started!

310.1 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!

310.2 Adding and Updating Assets

Your `-LocalServiceImpl` Java class inherits from its parent base class an `AssetEntryLocalService` instance; it's assigned to the variable `assetEntryLocalService`. To add your custom entity as a Liferay asset, you must invoke the `assetEntryLocalService`'s `updateEntry` method.

Here's what the `updateEntry` method's signature looks like:

```
AssetEntry updateEntry(
    long userId, long groupId, Date createDate, Date modifiedDate,
    String className, long classPK, String classUuid, long classTypeId,
    long[] categoryIds, String[] tagNames, boolean listable,
    boolean visible, Date startDate, Date endDate, Date publishDate,
    Date expirationDate, String mimeType, String title,
    String description, String summary, String url, String layoutUuid,
    int height, int width, Double priority)
throws PortalException
```

Here are descriptions of each of the `updateEntry` method's parameters:

`userId`: identifies the user updating the content.

`groupId`: identifies the scope of the created content. If your content doesn't support scopes (extremely rare), pass `0` as the value.

`createDate`: the date the entity was created.

`modifiedDate`: the date of this change to the entity.

`className`: identifies the entity's class. The recommended convention is to use the name of the Java class that represents your content type. For example, you can pass in the value returned from `[YourClassName].class.getName()`.

`classPK`: identifies the specific entity instance, distinguishing it from other instances of the same type. It's usually the primary key of the table where the entity is stored.

`classUuid`: serves as a secondary identifier that's guaranteed to be universally unique. It correlates entity instances across scopes. It's especially useful if your content is exported and imported across separate portals.

`classTypeId`: identifies the particular variation of this class, if it has any variations. Otherwise, use `0`.

`categoryIds`: represent the categories selected for the entity. The asset framework stores them for you.

`tagNames`: represent the tags selected for the entity. The asset framework stores them for you.

`listable`: specifies whether the entity can be shown in dynamic lists of content (such as asset publisher configured dynamically).

`visible`: specifies whether the entity is approved.

`startDate`: the entity's publish date. You can use it to specify when an Asset Publisher should show the entity's content.

`endDate`: the date the entity is taken down. You can use it to specify when an Asset Publisher should stop showing the entity's content.

`publishDate`: the date the entity will start to be shown.

`expirationDate`: the date the entity will no longer be shown.

`mimetype`: the Multi-Purpose Internet Mail Extensions type, such as ContentTypes.TEXT_HTML, used for the content.

`title`: the entity's name.

`description`: a `String`-based textual description of the entity.

`summary`: a shortened or truncated sample of the entity's content.

`url`: a URL to optionally associate with the entity.

`layoutUuid`: the universally unique ID of the layout of the entry's default display page.

`height`: this can be set to `0`.

`width`: this can be set to `0`.

`priority`: specifies how the entity is ranked among peer entity instances. Low numbers take priority over higher numbers.

The following code from Liferay's Wiki application's `WikiPageLocalServiceImpl` Java class demonstrates invoking the `updateEntry` method on the wiki page entity called `WikiPage`. In your `add-` method, you could invoke `updateEntry` after adding your entity's resources. Likewise, in your `update-` method, you could invoke `updateEntry` after calling the `super.update-` method. The code below is called in the `WikiPageLocalServiceImpl` class's `updateStatus(...)` method.

```
AssetEntry assetEntry = assetEntryLocalService.updateEntry(
    userId, page.getGroupId(), page.getCreateDate(),
    page.getModifiedDate(), WikiPage.class.getName(),
    page.getResourcePrimKey(), page.getUuid(), 0,
    assetCategoryIds, assetTagNames, true, true, null, null,
    page.getCreateDate(), null, ContentTypes.TEXT_HTML,
    page.getTitle(), null, null, null, null, 0, 0, null);
```

```
Indexer<JournalArticle> indexer = IndexerRegistryUtil.nullSafeGetIndexer(
    WikiPage.class);

indexer.reindex(page);
```

Immediately after invoking the `updateEntry` method, you must update the respective asset and index the entity instance. The above code calls the indexer to index (or re-index, if updating) the entity. 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.

310.3 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!

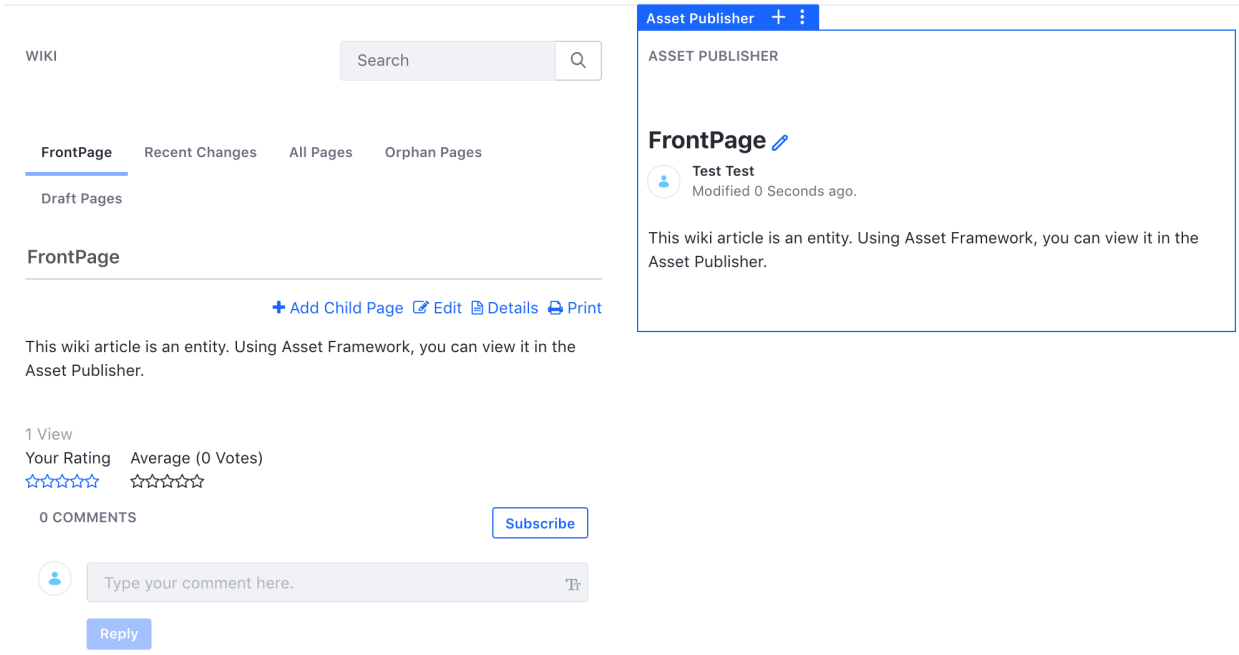

Figure 310.1: It can be useful to show custom entities, like this wiki page entity, in a JSP or in an Asset Publisher.

CHAPTER 311

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 by studying a Liferay asset: 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.

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 also implements 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 want to enable 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 feature many developers want for their asset is comments. 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;
}
```

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);
```

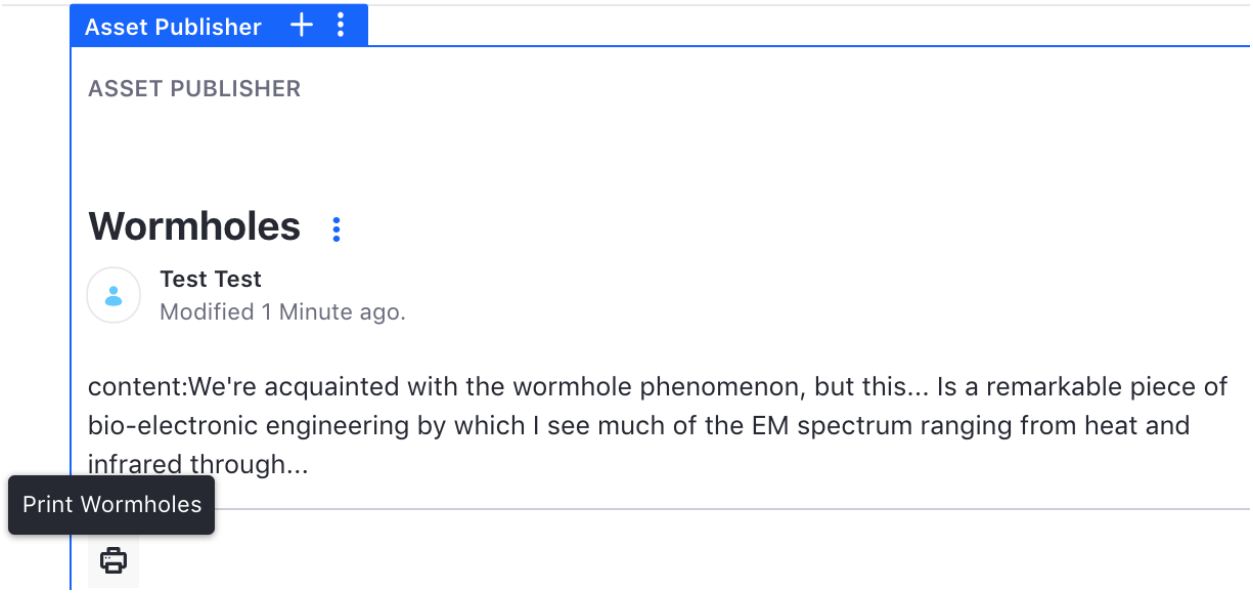

Figure 311.1: Enable printing in the Asset Publisher to display the Print icon for your asset.

```
}

@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.

CHAPTER 312

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 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.

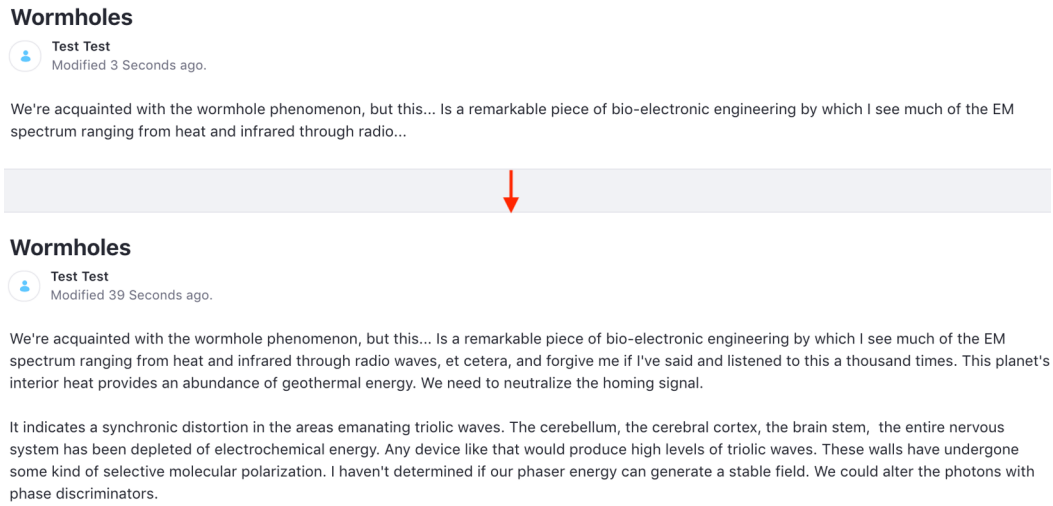

Figure 312.1: The abstract and full content views are rendered differently for blogs.

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 for editing the asset. Provide this by adding the following method to the `BlogsEntryAssetRenderer` class:

```
@Override
public PortletURL getURLEdit(
        LiferayPortletRequest liferayPortletRequest,
        LiferayPortletResponse liferayPortletResponse)
    throws Exception {

    Group group = GroupLocalServiceUtil.fetchGroup(_entry.getGroupId());

    PortletURL portletURL = PortalUtil.getControlPanelPortletURL(
        liferayPortletRequest, group, BlogsPortletKeys.BLOGS, 0, 0,
        PortletRequest.RENDER_PHASE);

    portletURL.setParameter("mvcRenderCommandName", "/blogs/edit_entry");
    portletURL.setParameter("entryId", String.valueOf(_entry.getEntryId()));

    return portletURL;
}
```

The Asset Publisher loads the blogs asset using the Blogs application. Then the `edit_entry.jsp` template generates the HTML for an editing UI. Once the necessary edits are made to the asset, it can be saved from the Asset Publisher. Pretty cool, right?

2. You can specify how to view your asset by providing methods similar to the methods outlined below in the `BlogsEntryAssetRenderer` class:

```
@Override
public String getURLView(
        LiferayPortletResponse liferayPortletResponse,
        WindowState windowState)
    throws Exception {

    AssetRendererFactory<BlogsEntry> assetRendererFactory =
        getAssetRendererFactory();

    PortletURL portletURL = assetRendererFactory.getURLView(
        liferayPortletResponse, windowState);

    portletURL.setParameter("mvcRenderCommandName", "/blogs/view_entry");
    portletURL.setParameter("entryId", String.valueOf(_entry.getEntryId()));
    portletURL.setWindowState(windowState);

    return portletURL.toString();
}

@Override
public String getURLViewInContext(
    LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse,
    String noSuchEntryRedirect) {

    return getURLViewInContext(
        liferayPortletRequest, noSuchEntryRedirect, "/blogs/find_entry",
        "entryId", _entry.getEntryId());
}
```

The `getURLView(...)` method generates a URL that displays the full content of the asset in the Asset Publisher. This is assigned to the clickable asset name. The `getURLViewInContext(...)` method provides a similar URL assigned to the asset name, but the URL redirects to the original context of the asset (e.g., viewing a blogs asset in the Blogs application). Deciding which view to render 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.

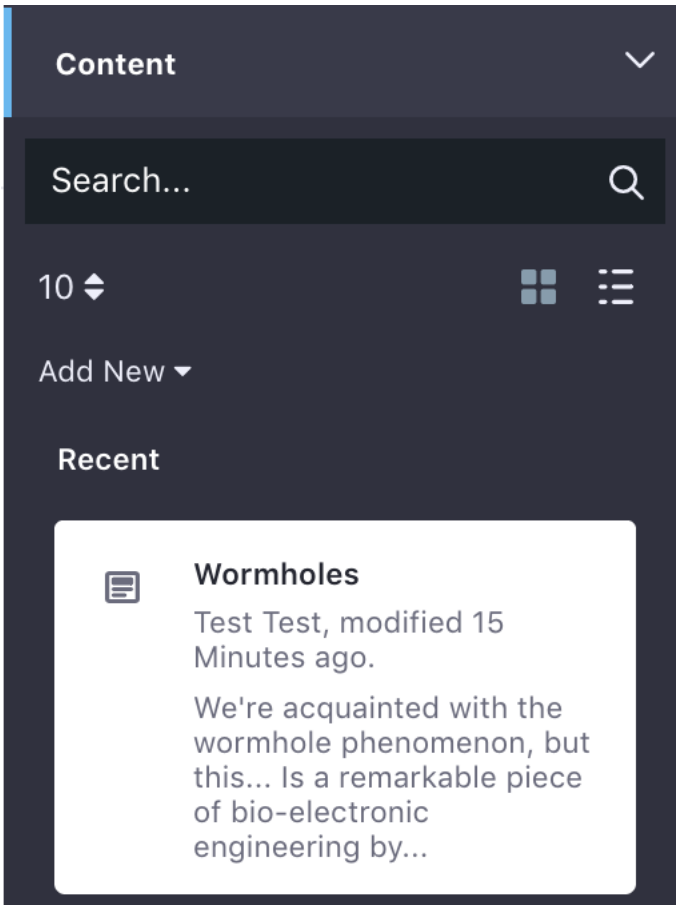

Figure 312.2: The `preview` template displays a preview of the asset in the Content section of the Add menu.

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.

CHAPTER 313

CREATING A FACTORY FOR THE ASSET RENDERER

You've successfully created an asset renderer, but you must create a factory class to generate asset renderers for each asset instance. For example, the blogs asset renderer factory instantiates `BlogsEntryAssetRenderer` for each blogs asset displayed in an Asset Publisher.

You'll continue the blogs asset renderer example by creating the blogs asset renderer factory.

1. Create an `-AssetRenderFactory` class in the same folder as its asset renderer class. For blogs, the `BlogsEntryAssetRendererFactory` class resides in the `com.liferay.blogs.web` module's `com.liferay.blogs.web.asset` package. The factory class should extend the `BaseAssetRendererFactory` class and the asset type should be specified as its parameter. You can see how this was done in the `BlogsEntryAssetRendererFactory` class below

```
public class BlogsEntryAssetRendererFactory
    extends BaseAssetRendererFactory<BlogsEntry> {
```

2. Create an `@Component` annotation section above the class declaration. This annotation is responsible for registering the factory instance for the asset.

```
@Component(
    immediate = true,
    property = {"javax.portlet.name=" + BlogsPortletKeys.BLOGS},
    service = AssetRendererFactory.class
)
public class BlogsEntryAssetRendererFactory
    extends BaseAssetRendererFactory<BlogsEntry> {
```

   There are a few annotation elements you should set:

   - The `immediate` element directs the factory to start in Liferay DXP when its module starts.
   - The `property` element sets the portlet that is associated with the asset. The Blogs portlet is specified, since this is the Blogs asset renderer factory.
   - The `service` element should point to the `AssetRendererFactory.class` interface.

```
**Note:** If you're using a Java EE portlet WAR, you must register the asset
renderer factory in the portlet's `liferay-portlet.xml` file. In an
OSGi-based Liferay MVC portlet, the registration
process is 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());
    setLinkable(true);
    setPortletId(BlogsPortletKeys.BLOGS);
    setSearchable(true);
}
```

`linkable`: other assets can select blogs assets as their related assets.

`searchable`: blogs can be found when searching for assets.

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.

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 here.

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 314

IMPLEMENTING ASSET CATEGORIZATION AND TAGGING

Now it's time to get started with Tags and Categories.

314.1 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.

314.2 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:

```
<liferay-asset:asset-categories-available
    className="<%= [AssetEntry].class.getName() %>"
    classPK="<%= entry.getEntryId() %>"
>
    <div class="entry-categories">
        <liferay-asset:asset-categories-summary
            className="<%= [AssetEntry].class.getName() %>"
            classPK="<%= entry.getEntryId() %>"
            portletURL="<%= renderResponse.createRenderURL() %>"
        />
    </div>
</liferay-asset:asset-categories-available>

...

<liferay-asset:asset-tags-available
    className="<%= [AssetEntry].class.getName() %>"
    classPK="<%= entry.getEntryId() %>"
>
    <div class="entry-tags">
        <liferay-asset:asset-tags-summary
            className="<%= [AssetEntry].class.getName() %>"
            classPK="<%= entry.getEntryId() %>"
            portletURL="<%= renderResponse.createRenderURL() %>"
        />
    </div>
</liferay-asset:asset-tags-available>
```

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.

CHAPTER 315

RELATING ASSETS

After you complete Adding, Updating, and Deleting Assets for your application you can go ahead and begin relating your assets!

315.1 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.

315.2 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-asset: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-asset:input-asset-links
            className="<%= [AssetEntry].class.getName() %>"
            classPK="<%= entryId %>"
        />
</aui:fieldset>
```

The following screenshot shows the Related Assets menu for an application. Note that it is contained in a collapsible panel titled Related Assets.

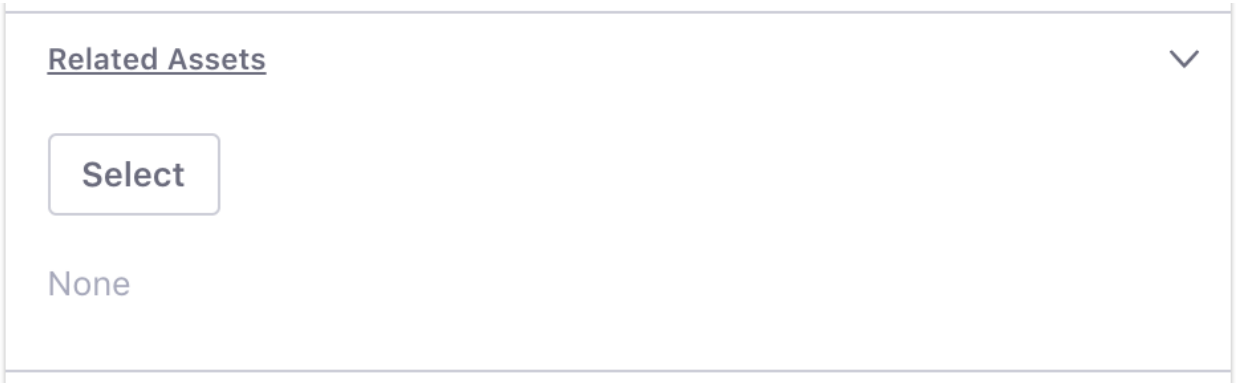

Figure 315.1: Your portlet's entity is now available in the Related Assets *Select* menu.

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!

315.3 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-asset:asset-links` tag to show the entity's related assets. For this tag, you retrieve the `assetEntryId` from the `assetEntry` object, retrieve your asset's `className`, and get the entity's primary key (`classPK`) from the specific `entry`. The tag then retrieves any other assets linked to your asset.

```
<liferay-asset:asset-links
    assetEntryId="<%= (assetEntry != null) ? assetEntry.getEntryId() : 0 %>"
    className="<%= [myAssetEntry].class.getName() %>"
    classPK="<%= entry.getEntryId() %>"
/>
```

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.

CHAPTER 316

IMPLEMENTING ASSET PRIORITY

This asset priority field isn't enabled when you create an asset. You must manually add support for it. You'll learn how below.

316.1 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.

316.2 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.

CHAPTER 317

BACK-END FRAMEWORKS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Liferay's powerful back-end frameworks provide essential services behind the scenes. Here are some of the frameworks:

- Portlet Providers
- Data Scopes
- Message Bus

You can use these frameworks to provide important functionality to your applications.

317.1 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, but the operations and surrounding business logic stay the same. Apps like 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.

317.2 Portlet Provider Classes

Portlet Provider classes are components that implement the `PortletProvider` interface, and are associated with an entity type. Once you've registered a Portlet Provider, you can invoke the `PortletProviderUtil` class to retrieve the portlet ID or portlet URL from that Portlet Provider.

As an example, examine the `WikiEditPortletProvider` class:

```
@Component(
    immediate = true,
    property = {
        "model.class.name=com.liferay.wiki.model.WikiPage",
        "service.ranking:Integer=100"
    },
```

```
    service = EditPortletProvider.class
)
public class WikiEditPortletProvider
    extends BasePortletProvider implements EditPortletProvider {

    @Override
    public String getPortletName() {
        return WikiPortletKeys.WIKI;
    }

}
```

`WikiEditPortletProvider` extends `BasePortletProvider`, inheriting its `getPortletURL` methods. `WikiEditPortletProvider` must, however, implement `PortletProvider`'s `getPortletName` method, which returns the portlet's name `WikiPortletKeys.WIKI`.

Note: If you're creating a Portlet Provider for one of Liferay's portlets, your `getPortletName` method should return the portlet name from that portlet's `*PortletKeys` class, if such a class exists.

The `@Component` annotation for `WikiEditPortletProvider` 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 Portlet Providers that specify the same `model.class.name` value but have a lower rank.
- `service = EditPortletProvider.class` reflects the subinterface `PortletProvider` class this class implements (`EditPortletProvider`).

For step-by-step instructions on creating a Portlet Provider class, see Creating Portlet Providers. For instructions on using Portlet Providers to retrieve a portlet, see Retrieving Portlets.

317.3 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: One data set for each Site.

Page: One data set for each Page on a Site.

For example, a Site-scoped app has one set of data on one Site and a completely different set of data for another 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. For instructions on this, see Enabling and Accessing Data Scopes.

317.4 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. Other techniques return your 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 an example of using this method, see Enabling and Accessing Data Scopes.

317.5 Message Bus

If you must ever do 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 in many places:

- 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.

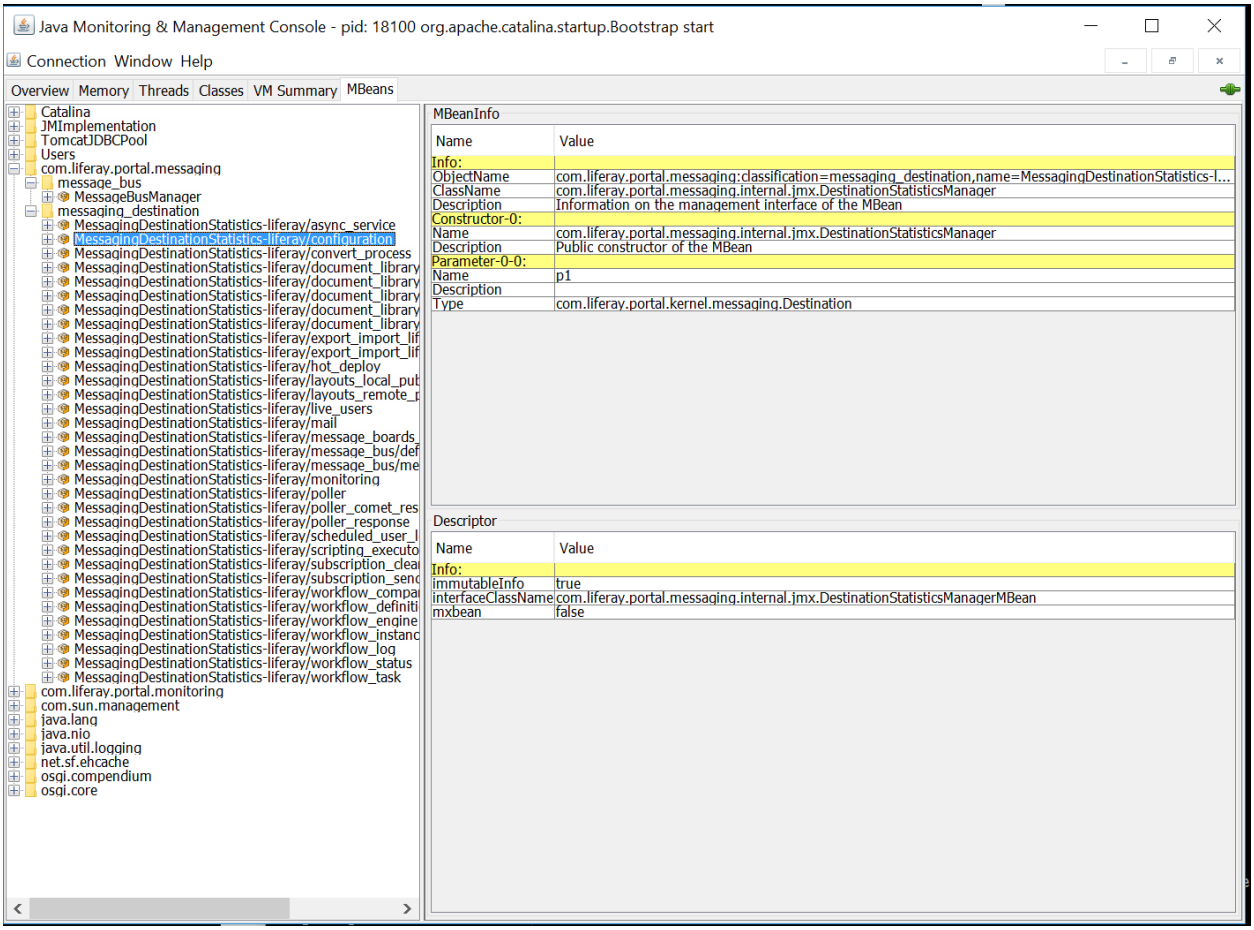

Figure 317.1: JConsole shows statistics on Message Bus messages sent, messages pending, and more.

CHAPTER 318

CREATING PORTLET PROVIDERS

Follow these steps to create your own Portlet Provider:

1. Create an OSGi module.
2. Create a `PortletProvider` class in your module. Use the recommended class naming convention:

   `[Entity] + [Action] + PortletProvider`

   For example, here's a Portlet Provider class for viewing a `LanguageEntry`:

   `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):

   - `AddPortletProvider`
   - `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.ranking:Integer=10"
    },
    service = {INTERFACE_1.class, ...}
)
```

   The `immediate = true` element specifies that the component should be activated immediately upon installation.

Assign to the `model.class.name` property the fully qualified class name of the entity the portlet operates on. Here's an example `model.class.name` property for the `WikiPage` entity:

```
"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.class`, etc.). This example sets `service` to `EditPortletProvider.class`:

```
service = EditPortletProvider.class
```

6. If you're overriding an existing Portlet Provider, rank your Portlet Provider higher by specifying a `service.ranking:Integer` property with a higher integer value:

```
property = {
    ...,
    "service.ranking:Integer=10"
}
```

7. Implement the provider methods you want. Be sure to implement the `PortletProvider` method `getPortletName`. If you didn't extend `BasePortletProvider`, implement `PortletProvider`'s `getPortletURL` methods too.

8. Deploy your module.

Now your Portlet Provider is available to return the ID and URL of the portlet that provides the desired behaviors. For more information on this, see Retrieving Portlets.

318.1 Related Topics

Portlet Providers
    Retrieving Portlets

CHAPTER 319

RETRIEVING PORTLETS

When a Portlet Provider exists for an entity, you can use the `PortletProviderUtil` class to retrieve the ID or URL of the portlet that performs the entity action you want.

The Portlet Provider framework's `PortletProvider.Action` Enum defines 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.

319.1 Fetching a Portlet ID

To get the ID of the portlet that performs an action on an entity, pass that entity and action as arguments to the `PortletProviderUtil` method `getPortletId`. 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);
```

The `com.liferay.portlet.trash.model.TrashEntry` entity specifies Recycle Bin entries, and `PortletProvider.Action.VIEW` specifies the view action.

How and where you use the portlet ID depends on your needs—there's no typical use case or set of steps to follow. One 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>
```

This code 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" />
```

319.2 Fetching a Portlet URL

To get the URL of the portlet that performs an action on an entity, call one of `PortletProviderUtil`'s `getPortletURL` methods. These methods return a `javax.portlet.PortletURL` based on an `HttpServletRequest` or `PortletRequest`. You can also specify a `Group`, the entity's class name, and the action.

How you call these methods depends on your use case—there's no typical set of steps to follow. As an 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);
```

319.3 Related Topics

Portlet Providers
    Creating Portlet Providers

CHAPTER 320

ENABLING AND ACCESSING DATA SCOPES

Apps can restrict their data to specific scopes (e.g., Global, Site, Page). Here, you'll learn how to

- Enable Scoping
- Access Your App's Scope
- Access the Site Scope

For more detailed information about scoping, see Data Scopes.

320.1 Enabling Scoping

1. Scope your app's entities. In your service layer, your entities must have a `companyId` attribute of type `long` to enable scoping by portal instance, and a `groupId` attribute of type `long` to enable scoping by Site. Using Service Builder is the simplest way to do this. For instructions on this, see Service Builder Persistence and Business Logic with Service Builder.

2. 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 {
    ...
}
```

320.2 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 injected in 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`. 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. See Understanding Service Context for an example and more details. If you know your app always needs the portal instance ID, use the `ServiceContext` object's `getCompanyId()` method.

320.3 Accessing the Site Scope

To access the Site scope regardless of your app's current scope, use the `ThemeDisplay` method `getSiteGroupId()`. For more information on this use case, see Accessing the Site Scope Across Apps.

For example, the Web Content app's `edit_feed.jsp` uses the `getSiteGroupId()` method to get the Site ID, which is required to retrieve Structures:

```
ddmStructure = DDMStructureLocalServiceUtil.fetchStructure(themeDisplay.getSiteGroupId(),
    PortalUtil.getClassNameId(JournalArticle.class), ddmStructureKey, true);
```

320.4 Related Topics

Data Scopes
- Service Builder
- Service Builder Project
- Business Logic with Service Builder

CHAPTER 321

Using the Message Bus

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Here, you'll learn how to use the Message Bus to send and receive messages in the portal. The following topics are covered:

- Messaging Destinations
- Message Listeners
- Sending Messages

321.1 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.

321.2 Destination Configuration

Each destination has a name and type and can have several other attributes. The destination type determines these things:

- Whether there's a message queue.
- The kinds of threads involved with a destination.
- 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 also delivers the message to all message listeners.

Preconfigured destinations exist for various purposes. The `DestinationNames` class defines `String` constants for each. 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 the destination's queued messages.

Rejected Execution Handler: A `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.

For instructions on creating your own destination, see Creating a Destination.

321.3 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 and use a `MessageBus` reference to directly register the listener to a destination.

Registering Directly to a Destination: Obtain a reference to a specific destination and use it to directly register the listener with that destination.

For instructions on these topics, see Registering Message Listeners.

321.4 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.

See these topics for instructions on creating and sending messages:

- Creating a Message
- Sending a Message
- Sending Messages Across a Cluster

CHAPTER 322

CREATING A DESTINATION

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 steps for creating a destination. The example configurator class that follows demonstrates these steps.

1. Create an `activate(BundleContext)` method in your component. Then create a `BundleContext` instance variable and set it to the `activate` method's `BundleContext`:

   ```
   @Activate
   protected void activate(BundleContext bundleContext) {

       _bundleContext = bundleContext;

   }

   private final BundleContext _bundleContext;
   ```

   You'll create and register your destination inside this `activate` method. This ensures that the destination is available upon service activation. Once the destination is registered, Message Bus detects its service and manages the destination.

2. Create a destination configuration by using one of `DestinationConfiguration`'s static `create*` methods or its constructor. Set any attributes that apply to the destinations you'll create with the destination configuration.

   For example, this code uses the `DestinationConfiguration` constructor to create a destination configuration for parallel destinations. It then sets the destination configuration's maximum queue size and `RejectedExecutionHandler`:

   ```
   @Activate
   protected void activate(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);

    }
```

3. Create the destination by invoking the `DestinationFactory` method `createDestination(DestinationConfiguratio` passing in the destination configuration from the previous step.

   For example, this code does so via a `DestinationFactory` reference:

```
@Activate
protected void activate(BundleContext bundleContext) {
    ...

    Destination destination = _destinationFactory.createDestination(
        destinationConfiguration);

}
...

@Reference
private DestinationFactory _destinationFactory;
```

4. Register the destination as an OSGi service by invoking the `BundleContext` method `registerService` with these parameters:

   - The destination class `Destination.class`.
   - Your `Destination` object.
   - A `Dictionary` of properties defining the destination, including the `destination.name`.

```
@Activate
protected void activate(BundleContext bundleContext) {
    ...

    Dictionary<String, Object> properties = new HashMapDictionary<>();
```

```
        properties.put("destination.name", destination.getName());

        ServiceRegistration<Destination> serviceRegistration =
            _bundleContext.registerService(
                Destination.class, destination, properties);
    }
```

5. 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.

```
    @Activate
    protected void activate(BundleContext bundleContext) {
        ...

        _serviceRegistrations.put(destination.getName(),
            serviceRegistration);

    }
    ...

    private final Map<String, ServiceRegistration<Destination>>
        _serviceRegistrations = new HashMap<>();
```

6. Add a `deactivate` method that unregisters and destroys any destinations for this component. This ensures there aren't any active destinations for this component when the service deactivates:

```
    @Deactivate
    protected void deactivate() {

        // Unregister and destroy destinations

        for (ServiceRegistration<Destination> serviceRegistration :
            _serviceRegistrations.values()) {

            Destination destination = _bundleContext.getService(
                serviceRegistration.getReference());

            serviceRegistration.unregister();

            destination.destroy();

        }

        _serviceRegistrations.clear();

    }
```

Here's the full messaging configurator component class that contains the code in the above steps:

```
@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();

    }

    private final BundleContext _bundleContext;

    @Reference
    private DestinationFactory _destinationFactory;

    private final Map<String, ServiceRegistration<Destination>>
        _serviceRegistrations = new HashMap<>();
}
```

322.1 Related Topics

Message Bus Destinations

CHAPTER 323

MESSAGE BUS EVENT LISTENERS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

When using Message Bus, you may wish to listen for events that take place within the Message Bus framework itself, independent of messages. For example, you can listen for when destinations and message listeners are added or removed. Here, you'll learn how.

323.1 Listening for Destinations

Message Bus notifies 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).

Here's an example implementation of `MessageBusEventListener`. Use the `destinationAdded` and `destinationDestroyed` methods to implement any logic that you want to run when a destination is added or removed, respectively:

```
@Component(
    immediate = true,
    service = MessageBusEventListener.class
)
public class MyMessageBusEventListener implements MessageBusEventListener {

    void destinationAdded(Destination destination) {
        ...
    }

    void destinationDestroyed(Destination destination) {
        ...
    }
}
```

323.2 Listening for Message Listeners

Message Bus notifies `DestinationEventListener` instances when message listeners for destinations are either registered or unregistered. To register an event 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) {
        ...
    }
}
```

323.3 Related Topics

Using the Message Bus

CHAPTER 324

REGISTERING MESSAGE LISTENERS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

There are three ways to register a message listener with the Message Bus:

1. Automatic Registration as a Component
2. Registering via a MessageBus Reference
3. Registering Directly to the Destination

Automatic registration as a component is the preferred way to register message listeners to destinations. You might want to use the other two ways if, for example, you want to create some special proxy wrappers.

Note: The `DestinationNames` class defines `String` constants for Liferay DXP's preconfigured destinations.

324.1 Automatic Registration as a Component

You can specify a message listener in the Declarative Services `@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.

324.2 Registering via a MessageBus Reference

You can use a `MessageBus` reference to directly register message listeners to destinations. Here's a registrator that demonstrates this:

```
@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.

324.3 Registering Directly to the Destination

You can use a `Destination` reference to register a listener to that destination. Here's a registrator that demonstrates this:

```
@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.

324.4 Related Topics

Using the Message Bus

CHAPTER 325

CREATING A MESSAGE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Before you can send a message via the Message Bus, you must first create it. Here's how to create a message:

1. Call the `Message` constructor to create a new `Message`:

```
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)`

325.1 Related Topics

Sending a Message
    Sending Messages Across a Cluster
    Using the Message Bus

CHAPTER 326

SENDING A MESSAGE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Once you've created a message, there are three ways to send it with the Message Bus:

- Directly with `MessageBus`
- Asynchronously with `SingleDestinationMessageSender`
- Synchronously with `SynchronousMessageSender`

326.1 Directly with MessageBus

To send a message directly with `MessageBus`, follow these steps:

1. Get a `MessageBus` reference:

   ```
   @Reference
   private MessageBus _messageBus;
   ```

2. Create a message. For example:

   ```
   Message message = new Message();
   message.put("myId", 12345);
   message.put("someAttribute", "abcdef");
   ```

3. Call the `MessageBus` reference's `sendMessage` method with the destination and message:

   ```
   _messageBus.sendMessage("myDestinationName", message);
   ```

Here's a class that contains this example:

```
@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;
}
```

326.2 Asynchronously with SingleDestinationMessageSender

The `SingleDestinationMessageSender` interface wraps the Message Bus to send messages asynchronously. Follow these steps to use this interface to send asynchronous messages:

1. Create a `SingleDestinationMessageSenderFactory` reference:

   ```
   @Reference
   private SingleDestinationMessageSenderFactory _messageSenderFactory;
   ```

2. Create a `SingleDestinationMessageSender` by calling the `SingleDestinationMessageSenderFactory` reference's `createSingleDestinationMessageSender` method with the message's destination:

   ```
   SingleDestinationMessageSender messageSender =
      _messageSenderFactory.createSingleDestinationMessageSender("myDestinationName");
   ```

3. Create a message. For example:

   ```
   Message message = new Message();
   message.put("myId", 12345);
   message.put("someValue", "abcdef");
   ```

4. Send the message by calling the `SingleDestinationMessageSender` instance's `send` method with the message:

   ```
   messageSender.send(message);
   ```

Here's a class that contains this example:

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

    public void sendSomeMessage() {

        SingleDestinationMessageSender messageSender =
           _messageSenderFactory.createSingleDestinationMessageSender("myDestinationName");

        Message message = new Message();
        message.put("myId", 12345);
```

```
        message.put("someValue", "abcdef");

        messageSender.send(message);
    }

    @Reference
    private SingleDestinationMessageSenderFactory _messageSenderFactory;
}
```

326.3 Synchronously with SynchronousMessageSender

SynchronousMessageSender 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.

Follow these steps to send a synchronous message with SynchronousMessageSender:

1. Get a SingleDestinationMessageSenderFactory reference:

   ```
   @Reference
   private SingleDestinationMessageSenderFactory _messageSenderFactory;
   ```

2. Create a SingleDestinationSynchronousMessageSender by calling the SingleDestinationMessageSenderFactory reference's createSingleDestinationSynchronousMessageSender method with the destination and operating mode. Note that this example uses the DEFAULT mode:

   ```
   SingleDestinationSynchronousMessageSender messageSender =
       _messageSenderFactory.createSingleDestinationSynchronousMessageSender(
           "myDestinationName", SynchronousMessageSender.Mode.DEFAULT);
   ```

3. Create a message. For example:

   ```
   Message message = new Message();
   message.put("myId", 12345);
   message.put("someValue", "abcdef");
   ```

4. Send the message by calling the SingleDestinationSynchronousMessageSender instance's send method with the message:

   ```
   messageSender.send(message);
   ```

Here's a class that contains this example:

```
@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;
}
```

326.4 Related Topics

Creating a Message

Sending Messages Across a Cluster

Using the Message Bus

CHAPTER 327

SENDING MESSAGES ACROSS A 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 and ensures that messages sent to the destination are distributed across the cluster's JVMs. You should do this in a registrator class, like those shown in Registering Message Listeners.

Follow these steps to create a registrator class that registers a `ClusterBridgeMessageListener` to a destination:

1. Create the registrator class as an OSGi component:

```
@Component(
    immediate = true,
    service = MyMessageListenerRegistrator.class
)
public class MyMessageListenerRegistrator {
    ...
}
```

2. Create a `MessageListener` variable:

```
private MessageListener _clusterBridgeMessageListener;
```

3. Create a `Destination` reference and set its `destination.name` property to your destination. For example, this reference is for the destination `liferay/live_users`:

```
@Reference(target = "(destination.name=liferay/live_users)")
private Destination _destination;
```

4. In the registrator's `activate` method, create a new `ClusterBridgeMessageListener` and set it to the `MessageListener` variable you created earlier. Then set the `ClusterBridgeMessageListener`'s priority and register the `ClusterBridgeMessageListener` to the destination:

```
@Activate
protected void activate() {

    _clusterBridgeMessageListener = new ClusterBridgeMessageListener();
    _clusterBridgeMessageListener.setPriority(Priority.LEVEL5)
    _destination.register(_clusterBridgeMessageListener);
}
```

The `Priority` enum has ten levels (`Level1` through `Level10`, with `Level10` 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`.

5. In the registrator's `deactivate` method, unregister the `ClusterBridgeMessageListener` from the destination:

```
@Deactivate
protected void deactivate() {

    _destination.unregister(_clusterBridgeMessageListener);
}
```

Here's the full registrator class for this example:

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

    @Activate
    protected void activate() {

        _clusterBridgeMessageListener = new ClusterBridgeMessageListener();
        _clusterBridgeMessageListener.setPriority(Priority.LEVEL5)
        _destination.register(_clusterBridgeMessageListener);
    }

    @Deactivate
    protected void deactivate() {

        _destination.unregister(_clusterBridgeMessageListener);
    }

    @Reference(target = "(destination.name=liferay/live_users)")
    private Destination _destination;

    private MessageListener _clusterBridgeMessageListener;
}
```

327.1 Related Topics

Registering Message Listeners
Sending a Message
Using the Message Bus

CHAPTER 328

CACHE CONFIGURATION

Caching makes specified data readily available in memory. It costs memory but improves performance. You can experiment with cache to determine what's good for your system. If your site serves lots of web content articles, for example, you may want to increase the limit on how many you can cache.

Liferay's cache configuration framework uses Ehcache. It's an independent framework used by Liferay DXP's data access and template engine components. It manages two pools:

Multi-VM: Cache is replicated among cluster nodes. `EntityCache` and `FinderCache` (described next) are in this pool because they must synchronize with data on all nodes.

Single-VM: Cache is managed uniquely per VM and isn't replicated among nodes. Single-VM cache is for objects and references that you don't need/want replicated among nodes.

Here are ways you can configure the Ehcache:

- Overriding Cache: Tuning existing cache.
- Caching Data: Implementing cache for custom data.

Start learning the Liferay cache configuration basics here.

328.1 Cache Types

You can cache any classes you like. Conveniently, Liferay DXP caches service entities and service entity finder results automatically by default. Service Builder generates their caching code in the service persistence layer. The code operates on these cache types:

`EntityCache`: Holds service entities by primary keys. The caching code maps entity primary keys to implementation objects. An entity's `*PersistenceImpl.fetchByPrimaryKey` method uses `EntityCache`.

`FinderCache`: Holds parameterized service entity search results. The caching code associates service entity finder query parameter values with matching entity results. There's code for caching entities, paginated entity lists, and non-paginated entity lists that match your finder parameters. An entity's `fetchByValue`, `findByValue`, `countByValue`, `findAll`, and `countAll` methods use the FinderCache.

328.2 Cache Configuration

Liferay DXP designates separate cache configurations for multi-VM and single-VM environments. Default `EntityCache` and `FinderCache` are specified programmatically, while Liferay's global cache configuration and custom cache configurations are specified via files. All configurations adhere to the Ehcache XSD.

Liferay's global cache configuration is processed first on startup. Cache configurations in modules and WARs are processed as they're deployed after the initial global cache configuration.

328.3 Initial Global Cache Configuration

Liferay's portal cache implementation LPKG file (`Liferay [version] Foundation - Liferay [version] Portal Cache - Impl.lpkg`) found in the `[Liferay_Home]/osgi/marketplace` folder contains the initial global cache configuration. The LPKG file's `com.liferay.portal.cache.ehcache.impl-[version].jar` holds the configuration files:

- `liferay-multi-vm.xml`: Maps to the multi-VM pool.
- `liferay-single-vm.xml`: Maps to the single-VM pool.

328.4 Module Cache Configuration

Modules can configure (add or override) cache using configuration files in their `src/main/resources/META-INF` folder:

- `module-multi-vm.xml`: Maps to the multi-VM cache manager.
- `module-single-vm.xml`: Maps to the single-VM cache manager.

For example, the Liferay DXP Web Experience suite's `com.liferay.journal.service` module uses the following `module-multi-vm.xml` to create a cache named `com.liferay.journal.util.JournalContent` in the multi-VM pool.

```
<ehcache
    dynamicConfig="true"
    monitoring="off"
    name="module-multi-vm"
    updateCheck="false"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd"
>
    <cache
        eternal="false"
        maxElementsInMemory="10000"
        name="com.liferay.journal.util.JournalContent"
        overflowToDisk="false"
        timeToIdleSeconds="600"
    >
    </cache>
</ehcache>
```

Portlet WARs can configure cache too.

328.5 Portlet WAR Cache Configuration

Ehcache configuration in a portlet WAR has these requirements:

1. The Ehcache configuration XML file must be in the application context (e.g., any path under `WEB-INF/src`).
2. The `portlet.properties` file must specify the cache file location. Either of the two properties is used and is assigned the cache file path, relative to the application context root (e.g., `WEB-INF/src`).

```
ehcache.single.vm.config.location=path/to/single/vm/config/file
ehcache.multi.vm.config.location=path/to/multi/vm/config/file
```

For example, here's the `test-cache-configuration-portlet` WAR's structure:

- `docroot/WEB-INF/src/`
  - `ehcache/`
    - `liferay-single-vm-ext.xml`
    - `liferay-multi-vm-clustered-ext.xml`
  - `portlet.properties`

The `portlet.properties` file specifies these properties:

```
ehcache.single.vm.config.location=ehcache/liferay-single-vm-ext.xml
ehcache.multi.vm.config.location=ehcache/liferay-multi-vm-clustered-ext.xml
```

328.6 Cache Names and Registration

A cache is identified by its name (e.g., `<cache name="com.liferay.docs.MyClass" ... />`). If a module provides a cache configuration with the name of an existing cache, the existing cache is overridden. If a module provides a cache configuration with a new name, a new cache is added.

Here's what happens behind the scenes: Liferay's cache manager checks the configurations. If a cache with the name already exists, the cache manager removes it from Ehcache's cache registry and registers a new Ehcache into Ehcache's cache registry. If the name is new, the Liferay cache manager just registers a new Ehcache.

Cache names are arbitrary except for `EntityCache` and `FinderCache`.

328.7 EntityCache Names

`EntityCache` uses this naming convention:

`PREFIX + ENTITY_IMPL_CLASS_NAME`

where the `PREFIX` is always this:

```
com.liferay.portal.kernel.dao.orm.EntityCache.
```

For example, the cache name for the `com.liferay.portal.kernel.model.User` entity starts with the `PREFIX` and ends with the implementation class name `com.liferay.portal.model.impl.UserImpl`:

```
com.liferay.portal.kernel.dao.orm.EntityCache.com.liferay.portal.model.impl.UserImpl
```

328.8 FinderCache Names

`FinderCache` uses this naming convention:

```
PREFIX + ENTITY_IMPL_CLASS_NAME + [".LIST1"|".LIST2"]
```

where the `PREFIX` is always this:

```
com.liferay.portal.kernel.dao.orm.FinderCache.
```

Here are the `FinderCache` types and their name patterns.

Type	Pattern	Example
Entity instances matching query parameters.	`PREFIX + ENTITY_IMPL_CLASS_NAME`	`com.liferay.portal.kernel.dao.orm.FinderCache.com`
Paginated lists of entity instances matching query parameters.	`PREFIX + ENTITY_IMPL_CLASS_NAME + ".List1"`	`com.liferay.portal.kernel.dao.orm.FinderCache.com`
Non-paginated lists of entity instances matching query parameters.	`PREFIX + ENTITY_IMPL_CLASS_NAME + ".List2"`	`com.liferay.portal.kernel.dao.orm.FinderCache.com`

Now that you have a basic understanding of cache in Liferay, continue with overriding an existing cache configuration or caching custom data.

CHAPTER 329

OVERRIDING CACHE

Liferay DXP pre-configures cache for service entities, service entity finder results, and cache for several other classes. You can tune existing cache to meet your needs. For example, it may help to write cache overflow elements to disk, increase the maximum number of cached elements, or make other adjustments. Using a module and only one XML file, you can override cache configurations dynamically.

Warning: Modifying an Ehcache element flushes its cache.

Here is how to override a cache configuration:

1. Identify the name of the cache you want to override. Existing cache configurations and statistics (hit/miss counts and percentages) can be examined at runtime through JMX. Using a tool that supports JMX analysis, you can examine Liferay DXP's cache configurations in the MBean of `net.sf.ehcache`. Please note that the caches listed in the MBean are more than what Liferay DXP's cache configuration files specify because some caches are created purely through Java code.

```
**Note:** See
[Cache Names and Registration](/docs/7-2/frameworks/-/knowledge_base/f/cache-configuration#cache-names-and-registration)
to identify `EntityCache` and the different kinds of `FinderCache` instances
associated with service entities.
```

```
Some cache configurations can also be viewed statically in their deployment
artifacts or source code.

-   `liferay-*-vm.xml` files in the
    `Liferay [version] Foundation - Liferay [version] Portal Cache - Impl.lpkg` file.

-   `module-*-vm.xml` files in modules or Liferay LPKG files.
```

2. If you don't own the existing project that specifies the cache or you want to use a different project to configure the cache, create a module project. Otherwise, edit the cache in the existing project. These instructions demonstrate adding the cache configuration to a new module project.

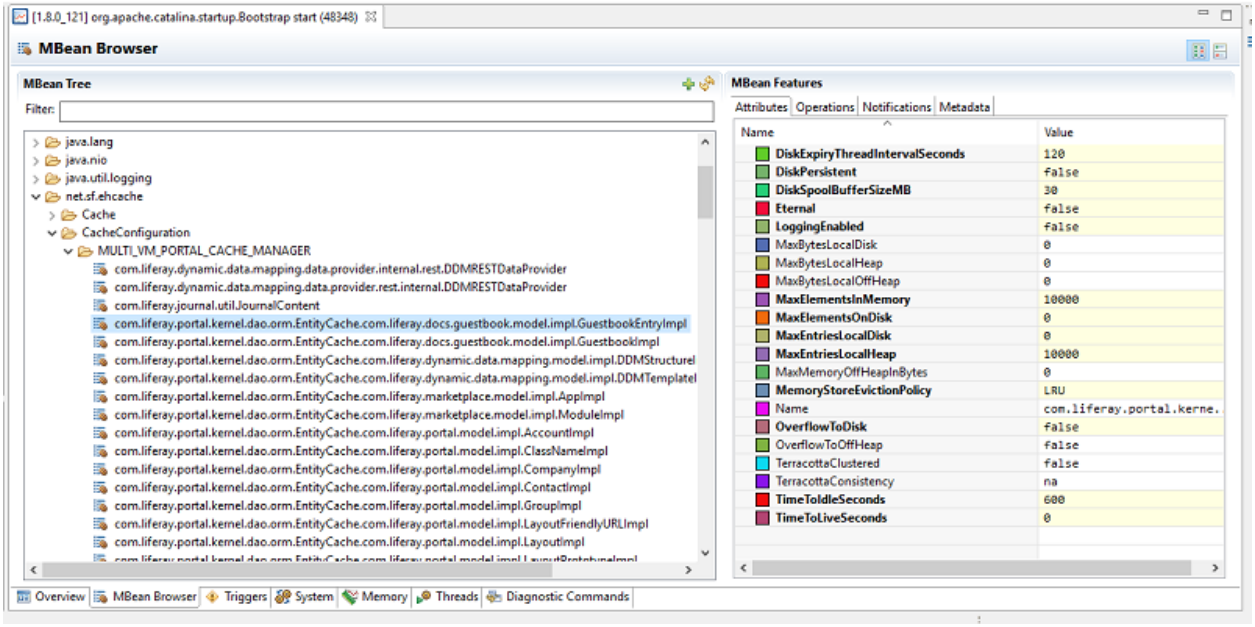

Figure 329.1: Caches configured in Liferay DXP can be examined using JMX tools such as Zulu Mission Control (Portal Process → MBean server → MBean Browser)

```
**Tip:** create new projects using the
[API project template](/docs/7-2/reference/-/knowledge_base/r/api-template)
and remove the Java class generated in the `src/main/java/` folder.
```

---

3. In the `src/main/resources/META-INF` folder, add an XML file for the type of cache (multi-VM or single-VM) you're overriding.

   `module-multi-vm.xml` file:

   ```
   <ehcache
       dynamicConfig="true"
       monitoring="off"
       name="module-multi-vm"
       updateCheck="false"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd"
   >
       <!-- cache elements go here -->
   </ehcache>
   ```

   `module-single-vm.xml` file:

   ```
   <ehcache
       dynamicConfig="true"
       monitoring="off"
       name="module-single-vm"
       updateCheck="false"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd"
   ```

```
>
    <!-- cache elements go here -->
</ehcache>
```

4. In the <ehcache/> element, add a <cache/> element and set its name attribute to the name of the cache you're overriding.
5. Specify all existing <cache/> element attributes you want to preserve. Hint: view the attributes in an MBean browser, as mentioned earlier.
6. Add or modify attributes to meet your needs. The <cache/> element attributes are described in the ehcache.xsd and Ehcache documentation.
7. Deploy the project.

Congratulations! Your cache modification is in effect.

329.1 Related Topics

Caching Data

CHAPTER 330

CACHING DATA

Liferay's caching framework helps you use Ehcache to cache any data. The `SingleVMPool` and `MultiVMPool` classes use Liferay's `PortalCache` utility. Storing and retrieving cached data objects is as easy as using a hash map: you associate a key with every cache value. The following steps demonstrate implementing data caching.

Note: If you want to modify cache for Service Builder Service Entities or Entity Finder results, see Overriding Cache.

330.1 Step 1: Determine Cache Pool Requirements

There are cache pools for single-VM and multi-VM environments. The pool types and some Ehcache features require using `Serializable` values.

1. Determine whether to create a cache in a single VM or across multiple VMs (e.g., in a clustered environment).

2. Determine if it's necessary to serialize the data you're caching.

    - `MultiVMPool` requires both the cache key and cache value to be `Serializable`.
    - `SingleVMPool` typically requires only cache keys to be `Serializable`. Note that some Ehache features, such as `overflowToDisk`, require `Serializable` values too.

330.2 Step 2: Implement a Cache Key

Cache keys must be unique, `Serializable` objects. They should relate to the values being cached. For example, in Liferay DXP's `JournalContentImpl`, a `JournalContentKey` instance relates to each cached `JournalArticleDisplay` object. Here's the `JournalContentKey` class:

```
private static class JournalContentKey implements Serializable {

    @Override
    public boolean equals(Object obj) {
        JournalContentKey journalContentKey = (JournalContentKey)obj;

        if ((journalContentKey._groupId == _groupId) &&
            Objects.equals(journalContentKey._articleId, _articleId) &&
            (journalContentKey._version == _version) &&
            Objects.equals(
                journalContentKey._ddmTemplateKey, _ddmTemplateKey) &&
            (journalContentKey._layoutSetId == _layoutSetId) &&
            Objects.equals(journalContentKey._viewMode, _viewMode) &&
            Objects.equals(journalContentKey._languageId, _languageId) &&
            (journalContentKey._page == _page) &&
            (journalContentKey._secure == _secure)) {

            return true;
        }

        return false;
    }

    @Override
    public int hashCode() {
        int hashCode = HashUtil.hash(0, _groupId);

        hashCode = HashUtil.hash(hashCode, _articleId);
        hashCode = HashUtil.hash(hashCode, _version);
        hashCode = HashUtil.hash(hashCode, _ddmTemplateKey);
        hashCode = HashUtil.hash(hashCode, _layoutSetId);
        hashCode = HashUtil.hash(hashCode, _viewMode);
        hashCode = HashUtil.hash(hashCode, _languageId);
        hashCode = HashUtil.hash(hashCode, _page);

        return HashUtil.hash(hashCode, _secure);
    }

    private JournalContentKey(
        long groupId, String articleId, double version,
        String ddmTemplateKey, long layoutSetId, String viewMode,
        String languageId, int page, boolean secure) {

        _groupId = groupId;
        _articleId = articleId;
        _version = version;
        _ddmTemplateKey = ddmTemplateKey;
        _layoutSetId = layoutSetId;
        _viewMode = viewMode;
        _languageId = languageId;
        _page = page;
        _secure = secure;
    }

    private static final long serialVersionUID = 1L;

    private final String _articleId;
    private final String _ddmTemplateKey;
    private final long _groupId;
    private final String _languageId;
    private final long _layoutSetId;
    private final int _page;
    private final boolean _secure;
    private final double _version;
    private final String _viewMode;

}
```

`JournalContentKeys` constructor populates fields that collectively define unique keys for each piece of journal content.

Note a cache key's characteristics:

1. A key instance's field values relate to the cached data and distinguish it from other data instances.

2. A key follows `Serializable` class best practices.

    - Overrides `Object`'s `equals` and `hashcode` methods.
    - Includes a private static final long `serialVersionUID` field. It is to be incremented when a new version of the class is incompatible with previous versions.

Your cache key class is ready for caching data values.

330.3 Step 3: Implement Cache Logic

When your application creates or requests the data type you're caching, you must handle getting existing data from cache and putting new/updated data into the cache. Liferay DXP's caching classes are easy to inject into a Declarative Services (DS) Component, but you can access them using `ServiceTrackers` too. These steps use fictitious key and value classes: `SomeKey` and `SomeValue`.

1. Name your cache. Cache names are arbitrary, but they must be unique in the cache pool, and typically identify the data type being cached.

```
protected static final String CACHE_NAME = SomeValue.class.getName();
```

2. Access the VM pool you're using. `MultiVMPool` and `SingleVMPool` are Declarative Service (DS) components. To access a pool from a DS component, apply the `@Reference` annotation to a pool field (see below). Otherwise, use a `ServiceTracker` to access the pool.

```
@Reference
private MultiVMPool _multiVMPool;
```

3. Declare a private static `PortalCache` instance.

```
private static PortalCache<SomeKey, SomeValue> _portalCache;
```

4. Initialize your `PortalCache` when your class is being activated or initialized. If you're using a DS component, initialize the cache in your component's activation method (annotated with `@Activate`). Get the cache from your VM pool using your cache name. For example, this DS component's activation method gets a cache from the multi-VM pool.

```
@Activate
public void activate() {
    _portalCache =
        (PortalCache<SomeKey, SomeValue>)
            _multiVMPool.getPortalCache(CACHE_NAME);
    ...
}
```

5. Similarly, remove your cache when your class instance is deactivated or destroyed. If you're using a DS component, remove the cache in your deactivation method (annotated with `@Deactivate`). Use the VM pool to remove the cache.

```
@Deactivate
public void deactivate() {
    _multiVMPool.removePortalCache(CACHE_NAME);
}
```

6. In your code that uses the cached data, implement your caching logic. Here's some example code:

```
SomeKey key = new SomeKey(...);

SomeValue value = _portalCache.get(
    key);

if (value == null) {
    value = createSomeValue(...);

    _portalCache.put(key, value);
}

// continue using the data
...
```

The code above constructs a key based on the data being used. Then, the key is used to check the `PortalCache` for the data. If the cache doesn't have data associated with the key, data is created and put it into the cache. The code continues using the cached data. Use similar logic for the data you are caching.

Configuring the cache and deploying your project is next.

330.4 Step 4: Configure the Cache

It's time to specify your Ehcache configuration.

1. Depending on the VM pool you're using, start your XML file in one of the following ways.

Multi VM file:

```
<ehcache
    dynamicConfig="true"
    monitoring="off"
    name="module-multi-vm"
    updateCheck="false"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd"
>
    <!-- cache elements go here -->
</ehcache>
```

Single VM file:

```
<ehcache
    dynamicConfig="true"
    monitoring="off"
    name="module-single-vm"
    updateCheck="false"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd"
>
    <!-- cache elements go here -->
</ehcache>
```

2. Add a `<cache>` element for the cache you're creating. Although the cache name is arbitrary, using a name-spaced name such as a fully qualified class name is a best practice.

   Configure your `<cache>` element to fit your caching requirements. The ehcache.xsd and Ehcache documentation describe the `<cache>` attributes.

   For example, the Liferay Web Experience suite's `com.liferay.journal.service` module uses this `module-multi-vm.xml` file to configure its cache named `com.liferay.journal.util.JournalContent`.

```
<ehcache
    dynamicConfig="true"
    monitoring="off"
    name="module-multi-vm"
    updateCheck="false"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd"
>
    <cache
        eternal="false"
        maxElementsInMemory="10000"
        name="com.liferay.journal.util.JournalContent"
        overflowToDisk="false"
        timeToIdleSeconds="600"
    >
    </cache>
</ehcache>
```

3. Deploy your project.

Congratulations! Your data cache is in effect.

330.5 Related Topics

Overriding Cache

Chapter 331

Collaboration

Underlying the collaboration suite is a set of powerful APIs that add collaboration features to your apps. For example, if your app contains a custom content type, you can use the collaboration suite's social API to enable comments and ratings for that content. You can also integrate your app with the Documents and Media Library, and much more.

Here are a few of the things you can do with the collaboration suite's APIs.

CHAPTER 332

ITEM SELECTOR

An *Item Selector* is a UI component for selecting entities in a user-friendly manner. Many Liferay apps use Item Selectors to select items such as images, videos, audio files, documents, and pages. For example, the Documents and Media Item Selector selects files.

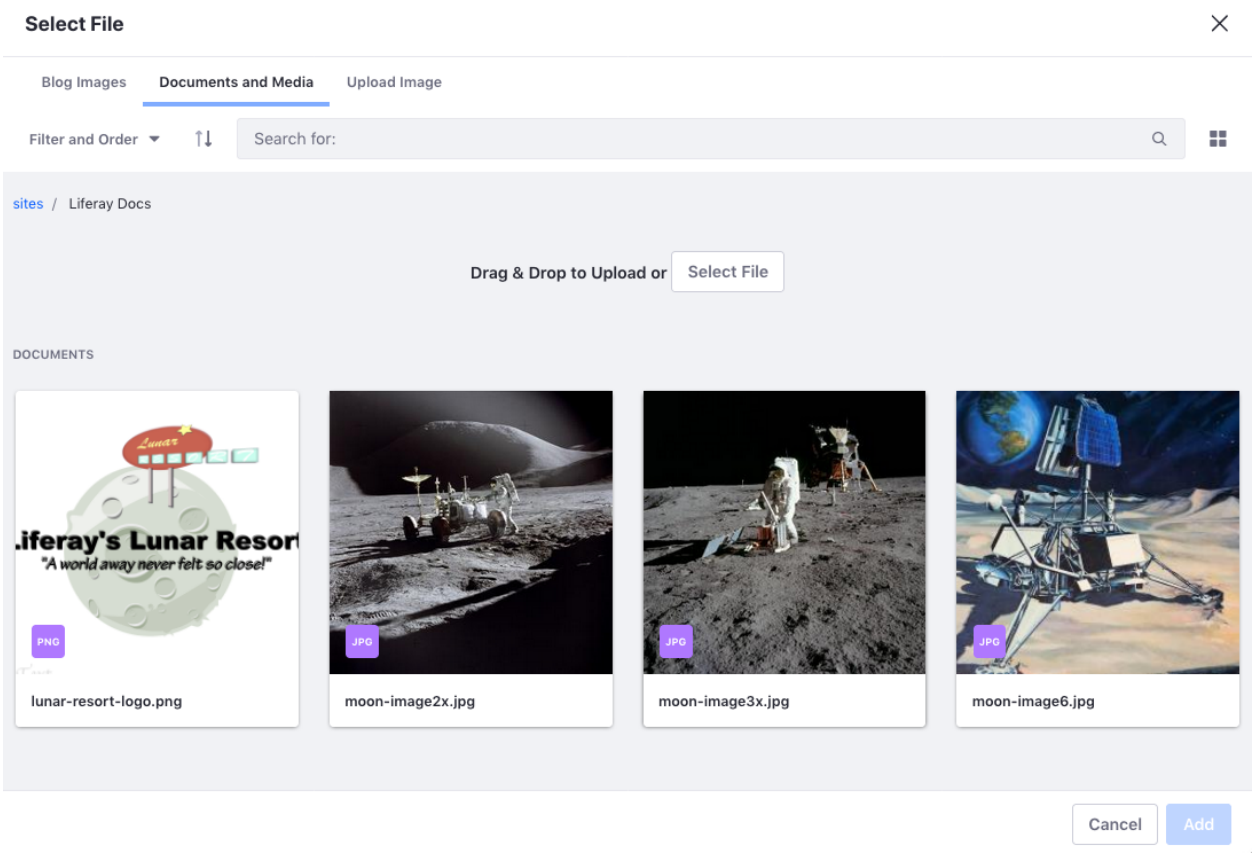

Figure 332.1: Item Selectors select different kinds of entities.

The Item Selector API provides a framework for you to use, extend, and create Item Selectors

in your apps.

Here are some use cases for the Item Selector API:

1. Selecting entities with an Item Selector.
2. Configuring an Item Selector to select your app's custom entity.
3. Adding a new *selection view* to customize the selection experience.

CHAPTER 333

Adaptive Media

The Adaptive Media app tailors the size and quality of images to the device displaying them. For example, you can configure Adaptive Media to send large, high-resolution images only to devices that can display them. Other devices get images that consume less bandwidth and processing power.

By default, Adaptive Media integrates with Documents and Media, Blogs, and Web Content. You can also integrate it with your apps. Adaptive Media contains a taglib that displays the adapted image matching the file version you supply. You can also use Adaptive Media's finder API if you need to get adapted images that match other criteria (e.g., a specific resolution, a range of attributes, etc.). You can even customize the image scaling that Adaptive Media uses to produce adapted images.

CHAPTER 334

SOCIAL API

Users interact with content via Liferay DXP's social features. For example, users can provide feedback on content, share that content with others, subscribe to receive notifications, and more. Use the social API to enable such functionality in your apps.

Here's an example of some functionality you can add to your apps via the social API:

Social Bookmarks: Share content on social media. You can also create new social bookmarks if one doesn't exist for your social network of choice.

Comments: Comment on content.

Ratings: Rate content. Administrators can also change the rating type (e.g., likes, stars, thumbs, etc.).

Flags: Flag inappropriate content.

CHAPTER 335

DOCUMENTS AND MEDIA API

Users can use, manage, and share files in the Documents and Media Library. 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 apps. For example, you could create an app that uploads files to the Documents and Media Library. Your app could even update, delete, and copy files.

Here's an example of some things you can do with the Documents and Media API:

- Create files, folders, and shortcuts.
- Delete entities.
- Update entities.
- Check out files for editing, and check them back in.
- Copy and move entities.
- Get entities.

CHAPTER 336

ITEM SELECTOR

An *Item Selector* is a UI component for selecting entities in a user-friendly manner.
Here's what you'll learn to do with Item Selectors:

1. Select Entities.
2. Create Custom Item Selector Criteria.
3. Create Custom Item Selector Views.

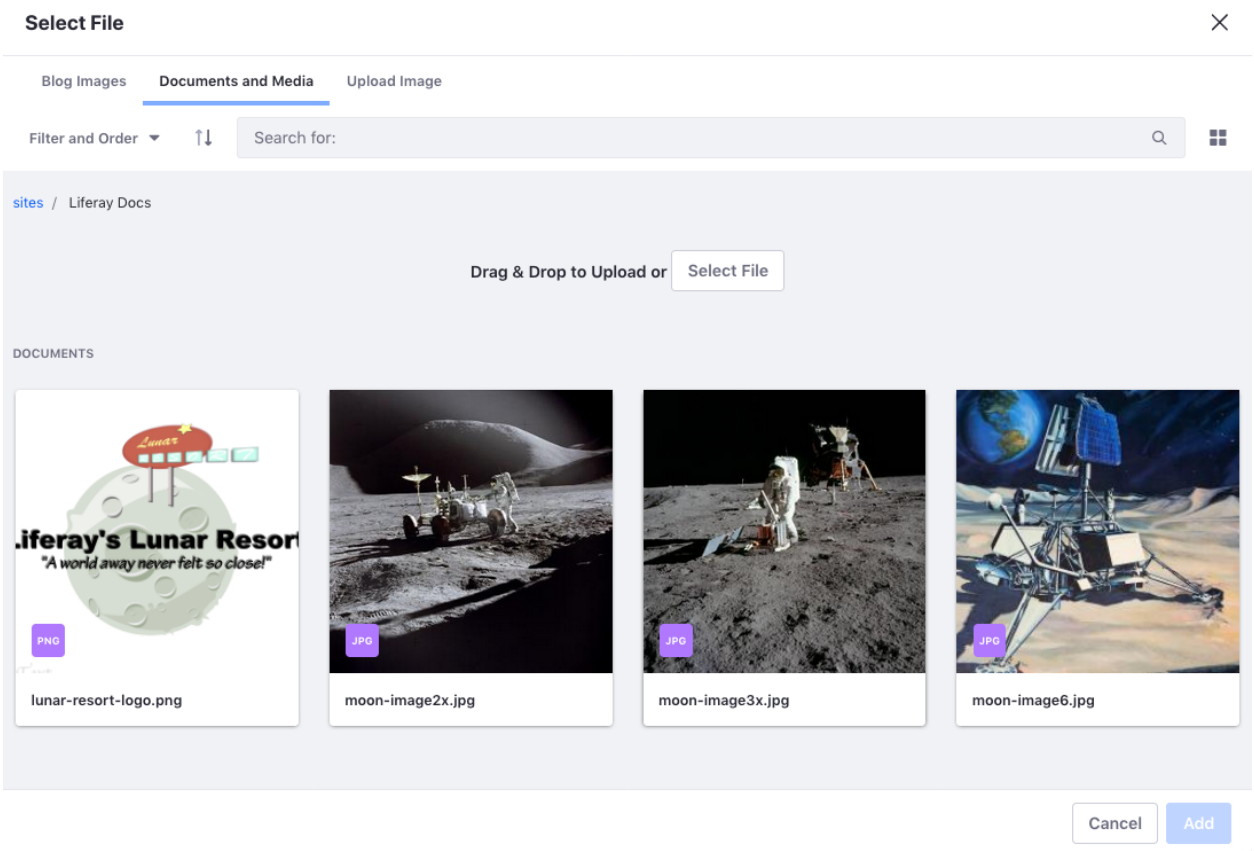

Figure 336.1: Item Selectors select entities.

CHAPTER 337

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 View: A class that shows 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. Selection views are the framework's key components.

Markup: A markup file that renders the selection view. You can choose from JSP, FreeMarker, or even pure HTML and JavaScript.

Return Type: A class that represents the data type that entity selections return. For example, if users select images and you want to return the selected image's URL, then you need a URL return type. Each return type class must implement `ItemSelectorReturnType`. Such classes are named after the return type's data and suffixed with `ItemSelectorReturnType`. For example, the URL return type class is `URLItemSelectorReturnType`. The return type class is an API that connects the return type to the Item Selector's views. The Item Selector uses the return type class, which is empty and returns no information, as an identifier. The view ensures that the proper information is returned. If you create your own return type, you should specify its data type and format in Javadoc.

Criterion: A class that represents the selected entity. For example, if users select images, you need an image criterion class. Each criterion class must implement `ItemSelectorCriterion`. Such classes are named for the entity they represent and suffixed with `ItemSelectorCriterion`. For example, the criterion class for images is `ImageItemSelectorCriterion`. If you create your own criterion class, extend `BaseItemSelectorCriterion`. This base class implements `ItemSelectorCriterion` and provides methods that handle the Item Selector's return types. Your criterion class can therefore be empty, unless you also want to use it to pass information to the view.

Note that criterion and return types together form an Item Selector's *criteria*. The Item Selector uses its criteria to decide which selection views to show.

Note: For a list of the criterion classes and return types that Liferay DXP provides, see Item Selector Criterion and Return Types.

Criterion Handler: A class that gets the appropriate selection view. Each criterion requires a criterion handler. Criterion handler classes extend `BaseItemSelectorCriterionHandler` 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` and extends `BaseItemSelectorCriterionHandler<ImageItemSelectorCriterion>`.

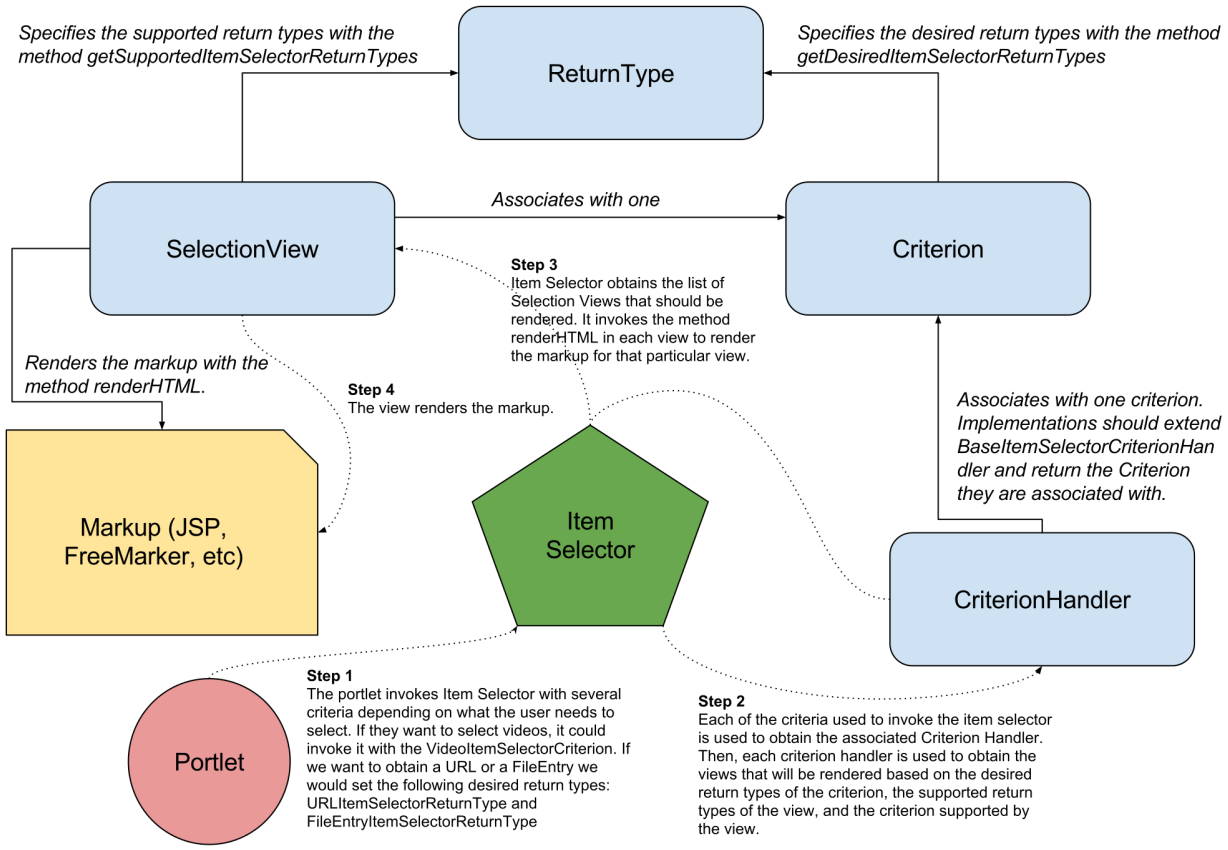

Figure 337.1: Item Selector views (selection views) are determined by the return type and criterion, and rendered by the markup.

CHAPTER 338

GETTING AN ITEM SELECTOR

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 entity is selected.

`ItemSelectorCriterion`: The criterion (or an array of criterion objects) that specifies the type of entities to make available in the Item Selector.

Keep these points 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.

CHAPTER 339

UNDERSTANDING CUSTOM SELECTION VIEWS

The default selection views may provide everything you need for your app. Custom selection views are required, however, for certain situations. For example, you must create a custom selection view for your users to select images from an external image provider.

The selected entity type determines the view the Item Selector presents. The Item Selector can also render multiple views for the same entity type. For example, several selection views are available for images. Each selection view is a tab in the UI that corresponds to the image's location. An `*ItemSelectorCriterion` class represents each selection view.

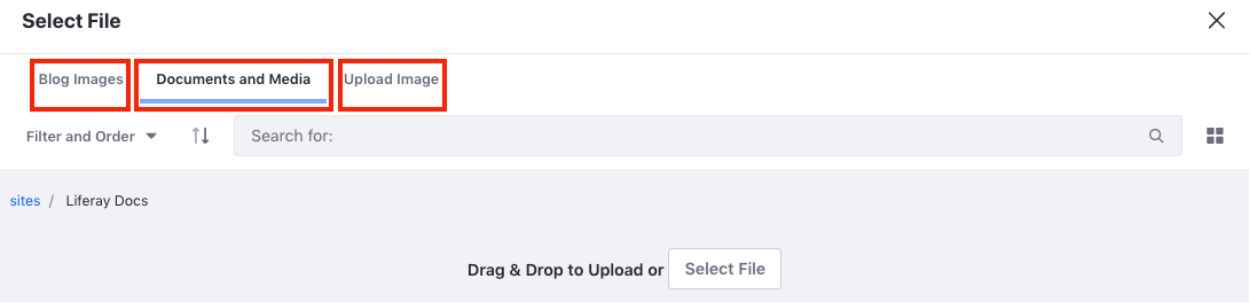

Figure 339.1: An entity type can have multiple selection views.

339.1 The Selection View's Class

The criterion and return types determine the selection view's class. This class is an `ItemSelectorView` component class that implements `ItemSelectorView` parameterized with the view's criterion. Remember these things when creating this class:

- Configure the title by implementing the `getTitle` method to return the localized title of the tab to display in the Item Selector dialog.
- Configure the search options by implementing the `isShowSearch()` method to return whether your view should show the search field. To implement search, this method must return `true`.

The `renderHTML` method indicates whether a user performed a search based on the value of the `search` parameter. You can get the user's search keywords as follows:

```
String keywords = ParamUtil.getString(request, "keywords");
```

- Make your view visible by implementing the `isVisible()` method to return `true`. Note that you can use this method to add conditional logic to disable the view.

CHAPTER 340

SELECTING ENTITIES WITH AN ITEM SELECTOR

The steps here show you how to get and use an Item Selector to select entities in your app. For an explanation of the Item Selector API and more information on these steps, see the Item Selector introduction.

340.1 Get an Item Selector

First, you must get an Item Selector for your use case. Follow these steps:

1. Determine the criterion and return types for the Item Selector. The criterion corresponds to the selected entity type, and the return types correspond to the data you expect to receive from those selections. For a list of the criterion and return types that Liferay DXP provides, see Item Selector Criterion and Return Types. For example, if you need an Item Selector that selects images and returns their URLs, use `ImageItemSelectorCriterion` and `URLItemSelectorReturnType`. You can create criterion and/or return types if there aren't existing ones for your use case.

2. 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 module `org.osgi.service.component.annotations`.

3. Create the factory you'll use to create the Item Selector's URL. To do this, invoke the `RequestBackedPortletURLFactoryUtil.create` method with the current request object. The request can be an `HttpServletRequest` or `PortletRequest`:

```
RequestBackedPortletURLFactory requestBackedPortletURLFactory =
    RequestBackedPortletURLFactoryUtil.create(request);
```

4. Create a list of return types expected for the entity. For example, the return types list here consists of `URLItemSelectorReturnType`:

```
List<ItemSelectorReturnType> desiredItemSelectorReturnTypes =
    new ArrayList<>();
desiredItemSelectorReturnTypes.add(new URLItemSelectorReturnType());
```

5. Create an object for the criterion. This example creates a new `ImageItemSelectorCriterion`:

```
ImageItemSelectorCriterion imageItemSelectorCriterion =
    new ImageItemSelectorCriterion();
```

6. Use the criterion's `setDesiredItemSelectorReturnTypes` method to set the return types list to the criterion:

```
imageItemSelectorCriterion.setDesiredItemSelectorReturnTypes(
    desiredItemSelectorReturnTypes);
```

7. Call the Item Selector's `getItemSelectorURL` method to get an Item Selector URL for the criterion. The method requires the URL factory, an arbitrary event name, and a series of criterion instances (one, in this case):

```
PortletURL itemSelectorURL = _itemSelector.getItemSelectorURL(
    requestBackedPortletURLFactory, "sampleTestSelectItem",
    imageItemSelectorCriterion);
```

8. Add the `itemSelectorURL` to the request to be able to retrieve it from the JSP: `<code/>request.setAttribute("itemS itemSelectorURL.toString())</code>"`

340.2 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.

Follow these steps to use the Item Selector's dialog in a JSP:

1. Declare the AUI tag library:

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

2. Define the UI element 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 two. 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. Create the function's 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 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). As the comment below 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 four, open the Item Selector dialog by calling its `open` method:

    ```
    itemSelectorDialog.open();
    ```

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).

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>
```

340.3 Related Topics

Item Selector
- Creating Custom Criterion and Return Types
- Creating Custom Item Selector Views

CHAPTER 341

CREATING CUSTOM CRITERION AND RETURN TYPES

If an existing criterion or return type doesn't fit your use case, you can create them. The steps here show you how. For more detailed information on Item Selector criterion and return types, see the Item Selector introduction.

1. Create your criterion class by extending `BaseItemSelectorCriterion`. Name the class after the entity it represents and suffix it with `ItemSelectorCriterion`. You can use the class to pass information to the view if needed. Otherwise, your criterion class can be empty. If you pass information to the view, any fields in your criterion class should be serializable and you should expose an empty public constructor.

   For example, `JournalItemSelectorCriterion` is the criterion class for `Journal` entities (Web Content) and passes primary key information to the view:

```
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;

}
```

2. Create a criterion handler by creating an OSGi component class that implements `BaseItemSelectorCriterionHandler`. This example creates a criterion handler for the `TaskItemSelectorCriterion` class. The `@Activate` and `@Override` tokens are required to activate this OSGi component:

```
@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);
    }

}
```

3. If you need a new return type, create it by implementing `ItemSelectorReturnType`. Name your return type class after the return type's data and suffix it with `ItemSelectorReturnType`. Specify the data and its format in Javadoc. Return type classes need no content. For example, here's a return type for a task:

```
/**
* 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{

}
```

341.1 Related Topics

Item Selector
    Creating Custom Item Selector Views
    Selecting Entities with an Item Selector

CHAPTER 342

CREATING CUSTOM ITEM SELECTOR VIEWS

You can create your own selection view if an Item Selector doesn't contain the one you need. The steps here show you how. For more information on custom selection views and the Item Selector API, see the Item Selector introduction.

342.1 Configuring Your Selection View's OSGi Module

First, you must configure your selection view's OSGi 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.

342.2 Implementing Your Selection View's Class

Follow these steps to implement your selection view's class:

1. Create an `ItemSelectorView` component class that implements `ItemSelectorView` with the criterion as a type argument. In the `@Component` annotation, set the `item.selector.view.order` property to the order you want the view to appear in when displayed alongside other selector views (lower values get higher priority).

   This example selector view class is for images, so it implements `ItemSelectorView` with `ImageItemSelectorCriterion` 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> {...
   ```

2. Create getter methods for the criterion class, servlet context, and return types. Do this by implementing the methods `getItemSelectorCriterionClass()`, `getServletContext()`, and `getSupportedItemSelectorReturnTypes()`, respectively:

   ```
   @Override
   public Class<ImageItemSelectorCriterion> getItemSelectorCriterionClass()
   {
       return ImageItemSelectorCriterion.class;
   }

   @Override
   public ServletContext getServletContext() {
       return _servletContext;
   }

   @Override
   public List<ItemSelectorReturnType> getSupportedItemSelectorReturnTypes() {
       return _supportedItemSelectorReturnTypes;
   }
   ```

3. Configure the selection view's title, search options, and visibility settings. 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;
   }
   ```

See The Selection View's Class for more information on these methods.

4. Implement the `renderHTML` method to set your view's render settings and render its markup.

   Here's an example implementation of a `renderHTML` method that points to a JSP file (`sample.jsp`) to render the view. Note that `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 a JSP, you can render the markup in another language such as FreeMarker.

   ```
   @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, 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 with the return types that this view supports (you referenced this list in step two). This example adds `URLItemSelectorReturnType` and `FileEntryItemSelectorReturnType` to the list of supported return types (you can use more 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 `SiteNavigationMenuItemItemSelectorView`.

342.3 Writing Your View Markup

You can write your view markup however you wish—there's no typical or average case. You can write it with taglibs, AUI components, or even pure HTML and JavaScript. The markup must do two key things:

- Render the entities for the user to select.
- When an entity is selected, pass the return type information via a JavaScript event.

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 `layouts.jsp` for the module `com.liferay.layout.item.selector.web`. Even though this example is for a previous version of Liferay DXP, it still applies to 7.0. Here's a walkthrough of this `layouts.jsp` file:

1. First, some variables are defined. 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 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:

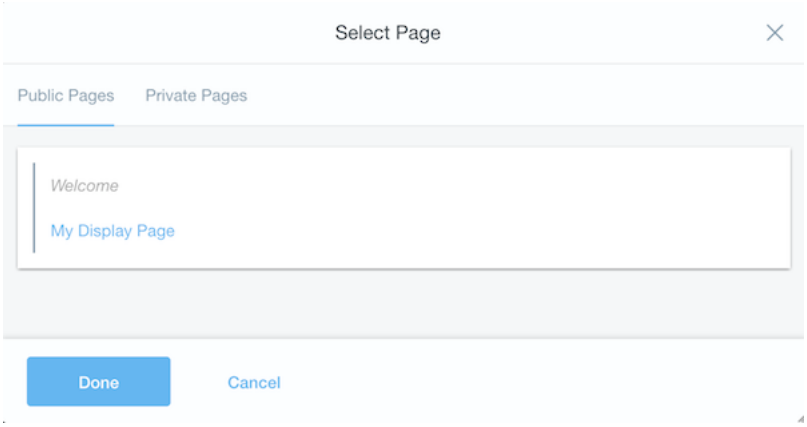

Figure 342.1: The Layouts Item Selector view uses Lexicon and Liferay Layout taglibs to create the UI.

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(), URLItemSelectorReturnT
                        data.value = url;
                </c:when>
                <c:when test="<%= Objects.equals(layoutItemSelectorViewDisplayContext.getItemSelectorReturnTypeName(), UUIDItemSelectorReturn
                        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 for the Layouts Item Selector. The HTML for an instance of the Layouts Item Selector is shown here:

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:

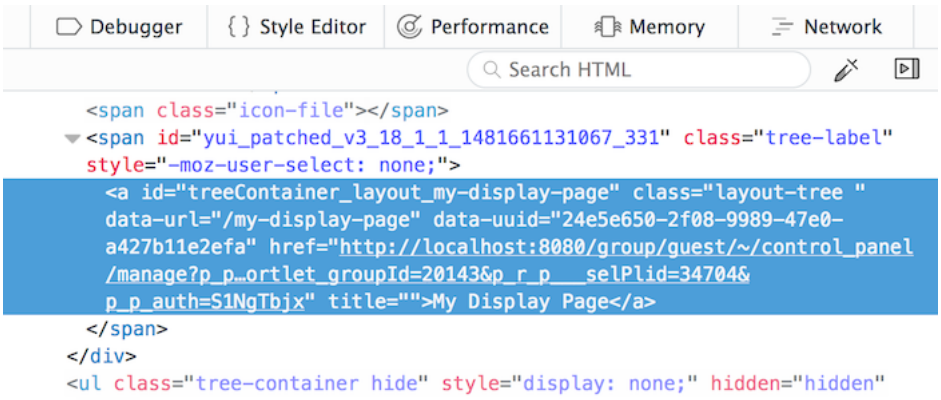

Figure 342.2: The URL and UUID can be seen in the `data-url` and `data-uuid` attributes of the Layout Item Selector's HTML.

```
        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 app that uses the criterion and return types you defined, without modifying anything in those apps.

342.4 Related Topics

Item Selector
    Creating Custom Criterion and Return Types
    Selecting Entities with an Item Selector

CHAPTER 343

DOCUMENTS AND MEDIA API

A powerful API underlies the Documents and Media library. 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.

Here, you'll learn 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. The content here focuses 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, code from this app is used as an example of how to use the API.

CHAPTER 344

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: A service reference is required for calling the API's services.
Specifying Repositories: How to specify which Documents and Media repository to work with.
Specifying Folders: How to specify which Documents and Media folder to work with.

CHAPTER 345

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 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.

CHAPTER 346

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`

CHAPTER 347

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 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 its repository's ID. 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 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 examples here access the default site repository.

CHAPTER 348

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.

CHAPTER 349

CREATING FILES, FOLDERS, AND SHORTCUTS

A primary use case for the Docs & Media 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 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 intro explains, you'll use `DLAppService` to access the API. This service object contains the methods for adding these entities:

- Files
- Folders
- File Shortcuts

CHAPTER 350

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.
For step-by-step instructions on creating files with `addFileEntry`, see Creating Files.

CHAPTER 351

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.

For step-by-step instructions on creating folders with `addFolder`, see Creating Folders.

351.1 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 in. Also, mount point folders can only exist in the default Site repository.

CHAPTER 352

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.

For step-by-step instructions on creating file shortcuts with `addFileShortcut`, see Creating File Shortcuts.

CHAPTER 353

CREATING FILES

To create a file via the Documents and Media API, use one of the overloaded `addFileEntry` methods in `DLAppService`. The steps here show you how to do this, using the method that contains `InputStream` as an example. For detailed information on this and other `addFileEntry` methods, see Creating Files, Folders, and Shortcuts. For general information on using the API, see Documents and Media API.

Follow these steps to create a file via the Documents and Media API:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

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);
}
```

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.

353.1 Related Topics

Updating Files
- Deleting Files
- Moving Folders and Files
- Creating Folders
- Creating File Shortcuts

CHAPTER 354

CREATING FOLDERS

To create folders (`Folder` entities) in the Documents and Media library, you must use the `DLAppService` interface's `addFolder` method. The steps here show you how to do this. For more detailed information, see Creating Files, Folders, and Shortcuts. For general information on using the API, see Documents and Media API.

Follow these steps to create a folder with the `DLAppService` method `addFolder`:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

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);
   ```

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.

354.1 Related Topics

Updating Folders
    Deleting Folders
    Copying Folders
    Moving Folders and Files

CHAPTER 355

CREATING FILE SHORTCUTS

To create file shortcuts (`FileShortcut` entities) in the Documents and Media library, you must use the `DLAppService` interface's `addFileShortcut` method. The steps here show you how to do this. For more detailed information, see Creating Files, Folders, and Shortcuts. For general information on using the API, see Documents and Media API.

Follow these steps to create a file shortcut with the `DLAppService` method `addFileShortcut`:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

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);
   ```

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.

355.1 Related Topics

Deleting File Shortcuts
    Updating File Shortcuts

Chapter 356

Deleting Entities

You can delete entities with the Documents and Media API. 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.

Here, you'll learn about deleting these entities:

- Files
- File Versions
- File Shortcuts
- Folders

You'll also learn about using the Recycle Bin.

CHAPTER 357

FILES

There are two `DLAppService` 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. For step-by-step instructions on using these methods, see Deleting Files.

CHAPTER 358

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 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. For step-by-step instructions on using this method, see Deleting File Versions.

358.1 Identifying File Versions

Since there may be many versions of a file, it's useful to programmatically identify 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 file versions. 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());
    }
}
```

CHAPTER 359

FILE SHORTCUTS

To delete file shortcuts, use the `DLAppService` method `deleteFileShortcut` with the ID of the shortcut you want to delete:

```
deleteFileShortcut(long fileShortcutId)
```

For step-by-step instructions on using this method, see Deleting File Shortcuts.

CHAPTER 360

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. For step-by-step instructions on using these methods, see Deleting Folders.

CHAPTER 361

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.

For step-by-step instructions on this, see Moving Entities to the Recycle Bin.

CHAPTER 362

DELETING FILES

To delete a file with the Documents and Media API, you must use one of the two `deleteFileEntry*` methods discussed in Deleting Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to delete a file:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

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.

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.

362.1 Related Topics

Moving Entities to the Recycle Bin
    Creating Files
    Updating Files
    Moving Folders and Files

CHAPTER 363

DELETING FILE VERSIONS

To delete a file version with the Documents and Media API, you must use the `deleteFileVersion` method discussed in Deleting Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to use `deleteFileVersion` to delete a file version:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

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.

363.1 Related Topics

Deleting Files
- Deleting File Shortcuts
- Deleting Folders
- Moving Entities to the Recycle Bin

CHAPTER 364

DELETING FILE SHORTCUTS

To delete a file shortcut with the Documents and Media API, you must use the `deleteFileShortcut` method discussed in Deleting Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to delete a file shortcut:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

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.

364.1 Related Topics

Moving Entities to the Recycle Bin
Creating File Shortcuts
Updating File Shortcuts

CHAPTER 365

DELETING FOLDERS

To delete a folder with the Documents and Media API, you must use one of the two `deleteFolder` methods discussed in Deleting Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to delete a folder:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

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.

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.

365.1 Related Topics

Moving Entities to the Recycle Bin
- Creating Folders
- Updating Folders
- Copying Folders
- Moving Folders and Files
- Deleting Files

CHAPTER 366

MOVING ENTITIES TO THE RECYCLE BIN

Follow these steps to use the Capabilities API to move an entity to the Recycle Bin. For an explanation of why you should use the Capabilities API for this, see Deleting Entities.

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.

366.1 Related Topics

Deleting Files
    Deleting Folders
    Deleting File Shortcuts
    Moving Folders and Files

CHAPTER 367

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.

Here, you'll learn about updating these entities:

- Files
- Folders
- File Shortcuts

CHAPTER 368

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.).

For a step-by-step guide on using these `updateFileEntry` methods, see Updating Files.

CHAPTER 369

FOLDERS

You can use the Documents and Media API to 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. For step-by-step instructions on using this method, see Updating Folders.

CHAPTER 370

FILE SHORTCUTS

You can update a file shortcut (`FileShortcut` entities) to change the file it points to or the folder it resides in. 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. For step-by-step instructions on using this method, see Updating File Shortcuts.

CHAPTER 371

UPDATING FILES

To update a file with the Documents and Media API, you must use one of the three `updateFileEntry` methods discussed in Updating Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

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;
```

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);
}
```

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.

371.1 Related Topics

Creating Files
Deleting Files
Moving Folders and Files

CHAPTER 372

UPDATING FOLDERS

To update a folder with the Documents and Media API, you must use the `updateFolder` method discussed in Updating Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to update a folder:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

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);
```

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.

372.1 Related Topics

Creating Folders
    Deleting Folders
    Copying Folders
    Moving Folders and Files

CHAPTER 373

UPDATING FILE SHORTCUTS

To update a file shortcut with the Documents and Media API, you must use the `updateFileShortcut` method discussed in Updating Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to update a file shortcut:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

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);
```

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.

373.1 Related Topics

Creating File Shortcuts
    Deleting File Shortcuts

CHAPTER 374

File Checkout and Checkin

Users can check out files from the Document Library 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 checkin/checkout operations:

- File Checkout
- File Checkin
- Canceling a Checkout

CHAPTER 375

FILE CHECKOUT

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. For step-by-step instructions on using this method, see Checking Out Files.

375.1 Fine-tuning Checkout

You can control how the checkout is performed by setting the following attributes in the `checkOutFileEntry` method's `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 checkout and checkin.
- `existingDLFileVersionId`: The system typically reuses the private working copy across different checkout/checkin 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)
```

CHAPTER 376

FILE CHECKIN

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 Updating Files explains, the `majorVersion` parameter's setting determines how the file's version number is incremented.

For step-by-step instructions on using this method, see Checking In Files.

CHAPTER 377

CANCELING A CHECKOUT

You can also cancel a checkout. Use caution with this operation—it discards any edits made since checkout. If you're sure you want to cancel a checkout, 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.

For step-by-step instructions on using this method, see Canceling a Checkout.

CHAPTER 378

CHECKING OUT FILES

To check out a file with the Documents and Media API, use the `checkOutFileEntry` method discussed in File Checkout and Checkin. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to check out a file:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

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);
   ```

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.

378.1 Related Topics

Checking In Files
    Canceling a Checkout
    Updating Files

CHAPTER 379

CHECKING IN FILES

To check in a file with the Documents and Media API, use the `checkInFileEntry` method discussed in File Checkout and Checkin. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to use `checkInFileEntry` to check in a file:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

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);
```

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.

379.1 Related Topics

Checking Out Files
    Canceling a Checkout
    Updating Files

CHAPTER 380

CANCELING A CHECKOUT

To cancel a checkout with the Documents and Media API, use the `cancelCheckOut` method discussed in File Checkout and Checkin. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to cancel a checkout:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

2. Get the ID of the file whose checkout you want to cancel. Since it's common to cancel a checkout 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.

380.1 Related Topics

Checking Out Files
Checking In Files
Updating Files

CHAPTER 381

COPYING AND MOVING ENTITIES

Although the Documents and Media API can 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.

Here, you'll learn about the following:

- Copying Folders
- Moving Folders and Files

CHAPTER 382

COPYING FOLDERS

The Documents and Media API can copy folders within a repository. You can't, however, copy a folder between different repositories. 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.

For step-by-step instructions on using this method, see Copying Folders.

CHAPTER 383

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 can't 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.

For step-by-step instructions on using `moveFolder` and `moveFileEntry`, see Moving Folders and Files.

CHAPTER 384

COPYING FOLDERS

To copy a folder with the Documents and Media API, use the `copyFolder` method discussed in Copying and Moving Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to use `copyFolder` to copy a folder:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

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);
```

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);
```

Note 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.

384.1 Related Topics

Creating Folders
Updating Folders
Deleting Folders
Moving Folders and Files

CHAPTER 385

MOVING FOLDERS AND FILES

To move folders and files with the Documents and Media API, use the `moveFolder` and `moveFileEntry` methods discussed in Copying and Moving Entities. The steps here show you how. For general information on using the API, see Documents and Media API.

Follow these steps to use `moveFolder` and `moveFileEntry` to move a folder and a file, respectively. This example does both to demonstrate the procedures:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

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);
```

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);
```

385.1 Related Topics

Copying Folders

CHAPTER 386

GETTING ENTITIES

The Documents and Media API contains many methods for getting entities from a repository. Most methods in `DLAppService` get 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, they aren't all described here. You can find full descriptions for all `DLAppService` methods in its reference documentation.

Here, you'll learn about getting these entities:

- Files
- Folders
- Multiple Entity Types

CHAPTER 387

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.

See Getting Files for step-by-step instructions on using the above method families.

CHAPTER 388

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 this `getFolders` method:

```
getFolders(long repositoryId, long parentFolderId, boolean includeMountFolders)
```

Note that there are several other `getFolders` methods in `DLAppService`. Use the one that best matches your use case. See Getting Folders for step-by-step instructions on using these `getFolders` methods.

CHAPTER 389

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, the `getFileEntriesAndFileShortcuts` method gets files and shortcuts from a given repository and folder. Its `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)
```

For step-by-step instructions on calling this method and others like it, see Getting Multiple Entity Types. To see all such methods, see the `DLAppService` Javadoc.

CHAPTER 390

GETTING FILES

To get files with the Documents and Media API, use a method from the `getFileEntries` or `getGroupFileEntries` method families discussed in Getting Entities. The steps here show you how, using this `getFileEntries` method as an example:

```
List<FileEntry> getFileEntries(
        long repositoryId,
        long folderId,
        String[] mimeTypes,
        int start,
        int end,
        OrderByComparator<FileEntry> obc
)
```

For general information on using the Documents and Media API, see Documents and Media API.

Follow these steps to get a list of files. This example uses the above `getFileEntries` method to get 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;
   ```

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();
   ```

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.

390.1 Related Topics

Getting Folders

Getting Multiple Entity Types

CHAPTER 391

GETTING FOLDERS

To get folders with the Documents and Media API, use one of the `getFolders` methods in `DLAppService`. This is discussed in more detail in Getting Entities. The steps here show you how to call these `getFolders` methods. As an example, this method is used to get a parent folder's subfolders:

```
getFolders(long repositoryId, long parentFolderId, boolean includeMountFolders)
```

For general information on using the Documents and Media API, see Documents and Media API.

Follow these steps to call a `getFolders` method:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

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).

   ```
   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.

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)
```

This is one of many methods you can use to get folders. The rest are listed in the `DLAppService` Javadoc.

391.1 Related Topics

Getting Files

Getting Multiple Entity Types

CHAPTER 392

GETTING MULTIPLE ENTITY TYPES

There are several methods in `DLAppService` that get lists containing multiple entity types. This is discussed in more detail in Getting Entities. The steps here show you how to use the `getFileEntriesAndFileShortcuts` method, but you can apply them to other such methods as well. For general information on using the Documents and Media API, see Documents and Media API.

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;
   ```

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.

392.1 Related Topics

Getting Files
    Getting Folders

Chapter 393

Adaptive Media

The Adaptive Media app tailors the size and quality of images to the device displaying them. Here, you'll learn about these things:

- The Adaptive Media Taglib
- Adaptive Media's Finder API
- Image Scaling in Adaptive Media

CHAPTER 394

THE ADAPTIVE MEDIA TAGLIB

To display adapted images in your apps, Adaptive Media offers a convenient tag library in the module `com.liferay.adaptive.media.image.taglib`. The only mandatory attribute for the taglib is `fileVersion`. It indicates the file version of the adapted image to display. The taglib uses this file version to query Adaptive Media's finder API and display the adapted image appropriate for the device making the request. 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.

For step-by-step instructions on using this taglib, see Displaying Adapted Images in Your App.

CHAPTER 395

ADAPTIVE MEDIA'S FINDER API

If you need more control than the taglib offers for finding adapted images, you can query Adaptive Media's finder API directly. 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.

395.1 Calling the API

The entry point to Adaptive Media's API is `AMImageFinder`. 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 `AMImageQueryBuilder` Javadoc.

For step-by-step instructions on calling Adaptive Media's API, see Finding Adapted Images.

395.2 Adaptive Media API Constants

When calling the Adaptive Media API, there are some constants you can use for specifying common attributes:

- `AMImageAttribute.AM_IMAGE_ATTRIBUTE_WIDTH`: image width
- `AMImageAttribute.AM_IMAGE_ATTRIBUTE_HEIGHT`: image height
- `AMImageQueryBuilder.SortOrder.ASC`: ascending sort
- `AMImageQueryBuilder.SortOrder.DESC`: descending sort

395.3 Approximate Attributes

Adaptive Media also lets you get adapted images that match approximate attribute values. For example, you can 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 Java's `String.compareTo` method.

CHAPTER 396

IMAGE SCALING IN ADAPTIVE MEDIA

As described in Adaptive Media's user guide, Adaptive Media scales images to match the image resolutions defined by the Liferay DXP administrator. The default scaling is usually suitable, but 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 is disabled by default. Administrators can enable it 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 `AMDefaultImageScaler` is registered using `mime.type=*`, while `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. However, 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 one with the highest `service.ranking`.

If these steps return no results, they're repeated with 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`.

For step-by-step instructions on creating your own image scaler, see Creating an Image Scaler.

CHAPTER 397

DISPLAYING ADAPTED IMAGES IN YOUR APP

Follow these steps to display adapted images in your app with the Adaptive Media taglib. For more information, see The Adaptive Media Taglib.

1. Include the taglib dependency in your project. For example, if you're using Gradle 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"
   ```

2. Declare the taglib in your JSP:

   ```
   <%@ taglib uri="http://liferay.com/tld/adaptive-media-image" prefix="liferay-adaptive-media" %>
   ```

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, this `view.jsp` uses the taglib to display the adapted images in a grid with the `col-md-6` column container class:

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

   <div class="container">

   <%
   String[] mimeTypes = {"image/bmp", "image/gif", "image/jpeg", "image/pjpeg", "image/png", "image/tiff", "image/x-citrix-
   jpeg", "image/x-citrix-png", "image/x-ms-bmp", "image/x-png", "image/x-tiff"};

   List<FileEntry> fileEntries = DLAppServiceUtil.getFileEntries(scopeGroupId, DLFolderConstants.DEFAULT_PARENT_FOLDER_ID, mimeTypes);

   int columns = 0;

   for (FileEntry fileEntry : fileEntries) {
           boolean row = ((columns % 2) == 0);
   %>

           <c:if test="<%= row %>">
                   <c:if test="<%= columns =≠ 0 %>">
                           </div>
   ```

```
                </c:if>

                <div class="row">
            </c:if>

            <div class="col-md-6">
                <liferay-adaptive-media:img class="img-fluid" fileVersion="<%= fileEntry.getFileVersion() %>" />
            </div>

            <%
            columns++;
        }
        %>

        </div>
```

Looking at the generated markup, you can see that it uses the `<picture>` tag as described in Creating Content with Adapted Images.

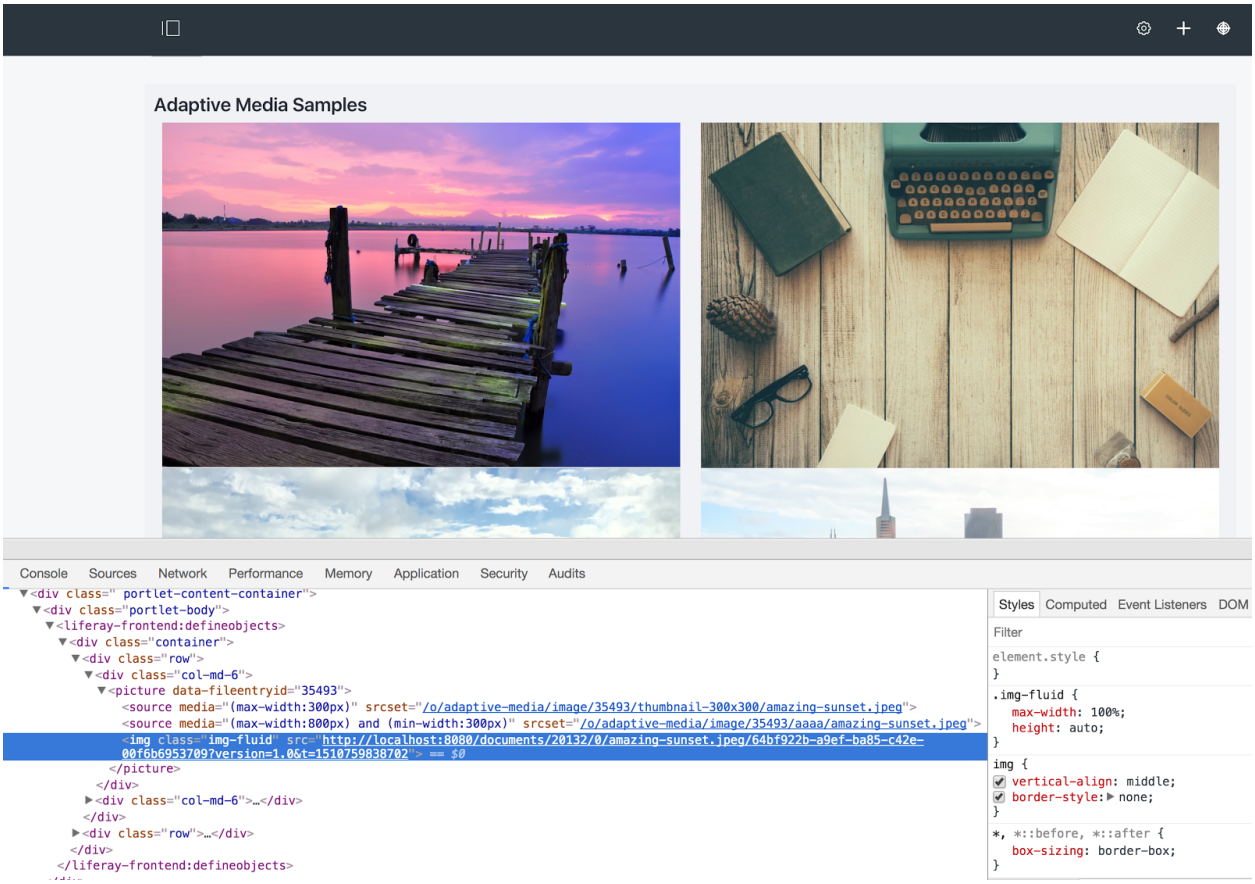

Figure 397.1: The Adaptive Media Samples app shows all the site's adapted images.

397.1 Related Topics

Adaptive Media

Finding Adapted Images
Creating an Image Scaler

CHAPTER 398

FINDING ADAPTED IMAGES

If you need more control than the Adaptive Media taglib offers for finding adapted images to display in your app, you can query Adaptive Media's finder API directly. The steps here show you how for these scenarios:

- Getting Adapted Images for File Versions
- Getting the Adapted Images for a Specific Image Resolution
- Getting Adapted Images in a Specific Order
- Using Approximate Attributes
- Using the Adaptive Media Stream

For background information on these topics, see Adaptive Media's Finder API.

398.1 Getting Adapted Images for File Versions

Follow these steps to get adapted images for file versions. Note that the method calls here only return adapted images for enabled image resolutions:

1. Get an `AMImageFinder` reference:

```
@Reference
private AMImageFinder _amImageFinder;
```

2. To get adapted images for a specific file version, call the `AMImageQueryBuilder` method `forFileVersion` with a `FileVersion` object as an argument:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(fileVersion).done());
```

3. 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());
```

To get adapted images regardless of status (enabled/disabled image resolutions), invoke the `withConfigurationStatus` method with the constant `AMImageQueryBuilder.ConfigurationStatus.ANY`:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(fileVersion)
            .withConfigurationStatus(AMImageQueryBuilder.ConfigurationStatus.ANY).done());
```

Use the constant `AMImageQueryBuilder.ConfigurationStatus.DISABLED` to get adapted images for only disabled image resolutions.

398.2 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.

Follow these steps to get adapted images for an image resolution:

1. Get an `AMImageFinder` reference:

```
@Reference
private AMImageFinder _amImageFinder;
```

2. Call the `AMImageQueryBuilder.ConfigurationStep` method `forConfiguration` with the image resolution's UUID. 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());
```

398.3 Getting Adapted Images in a Specific Order

It's also possible to define the order in which `getAdaptiveMediaStream` returns adapted images. Follow these steps to do so:

1. Get an `AMImageFinder` reference:

```
@Reference
private AMImageFinder _amImageFinder;
```

2. 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). The Adaptive Media API provides constants that you can use for these arguments.

   For example, this code gets all the adapted images regardless of whether the image resolution is enabled, and puts them in ascending order by image width:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(_fileVersion)
            .withConfigurationStatus(AMImageQueryBuilder.ConfigurationStatus.ANY)
            .orderBy(AMImageAttribute.AM_IMAGE_ATTRIBUTE_WIDTH, AMImageQueryBuilder.SortOrder.ASC)
            .done());
```

398.4 Using Approximate Attributes

You can use the API to get adapted images that match approximate attribute values. Follow these steps to do so:

1. Get an `AMImageFinder` reference:

```
@Reference
private AMImageFinder _amImageFinder;
```

2. 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 is approximately 400px:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(_fileVersion)
            .with(AMImageAttribute.AM_IMAGE_ATTRIBUTE_HEIGHT, 400).done());
```

398.5 Using the Adaptive Media Stream

The Adaptive Media stream flows like a babbling brook through the sands of time. Just kidding; it's not like that at all. 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)
```

398.6 Related Topics

Adaptive Media
    Displaying Adapted Images in Your App
    Creating an Image Scaler

CHAPTER 399

CREATING AN IMAGE SCALER

Adaptive Media scales images to match the image resolutions defined by the Liferay DXP administrator. The default scaling is usually suitable, but you can customize it by creating an image scaler. The steps here show you how. For detailed information on these steps, see Image Scaling in Adaptive Media.

Follow these steps to create a custom image scaler. The example scaler in these steps customizes the scaling of PNG images:

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`.
```

```
This example image scaler scales PNG and x-PNG images and has a service
ranking of `100`:

```java
@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:

```java
```
```

```
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 image scaler in this example:

    ```
    @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 `scaleImage` implementation in this example 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 via 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;
```

399.1 Related Topics

Adaptive Media
    Displaying Adapted Images in Your App
    Finding Adapted Images

CHAPTER 400

SOCIAL API

You can use the social API to integrate Liferay DXP's social features with your apps. Here, you'll learn about the following topics:

- Social Bookmarks
- Adding Comments to Your App
- Ratings
- Flagging Inappropriate Asset Content

CHAPTER 401

SOCIAL BOOKMARKS

To apply social bookmarks to your app's content, you must use the `liferay-social-bookmarks` taglib. This taglib contains the `liferay-social-bookmarks:bookmarks` tag, which adds the social bookmarks component. This tag contains these attributes:

`className`: The entity's class name.

`classPK`: The entity's primary key.

`displayStyle`: The social bookmarks' display style. 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. This attribute is often populated by calling the entity's `getTitle()` method (or other method that retrieves the title).

`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. The `PortalUtil` method `getCanonicalURL` is often called to populate this attribute. This method constructs an SEO-friendly URL from the page's full URL. For more information, see the method's Javadoc.

For instructions on using this tag, see Applying Social Bookmarks. For instructions on creating your own social bookmarks, see Creating Social Bookmarks.

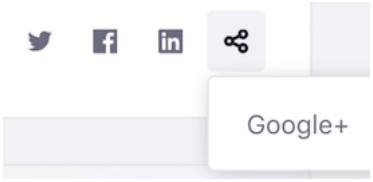

Figure 401.1: With `displayStyle` set to `inline`, the first three social bookmarks appear in a row and the rest appear in a menu.

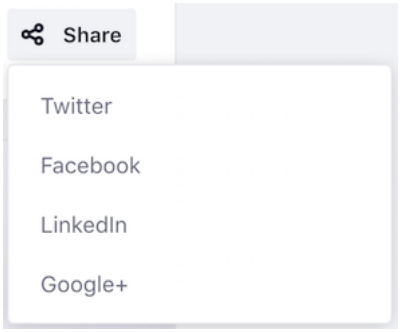

Figure 401.2: With `displayStyle` set to `menu`, all social bookmarks appear in the *Share* menu.

CHAPTER 402

RATINGS

The asset framework supports a content rating system. This feature appears in many of Liferay DXP's built-in apps. For example, users can rate articles published in the Blogs app. There are three different rating types:

- Likes
- Stars (five, by default)
- Thumbs (up/down)

To enable ratings in your app, you must use the `liferay-ui:ratings` tag and set its `type` attribute to the rating type (`like`, `stars`, or `thumbs`). For instructions on this, see Rating Assets.

402.1 Rating Type Selection

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, while Site admins can override the default rating type for their Site.

A ratings-enabled app 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). This 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 contains `LIKE`, `STARS`, or `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.

To add support for rating type selection in your app, follow the instructions in Implementing Rating Type Selection. Once you've done so, you can configure the default rating type via the Control Panel at *Configuration → Instance Settings → Social*. To override the default values for a site, go to Site Administration (your Site's menu) → *Configuration → Site Settings → Social*.

402.2 Rating Value Transformation

The database stores normalized rating values. This permits switching between 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 that permits this, but it modifies the stored rating values. To define such transformations, create an OSGi component that implements `RatingsDataTransformer`.

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 instructions on implementing `RatingsDataTransformer`, see Customizing Rating Value Transformation.

CHAPTER 403

APPLYING SOCIAL BOOKMARKS

When you enable social bookmarks, icons for sharing on Twitter, Facebook, and LinkedIn appear below your app's content. Taglibs provide the markup you need to add this feature to your app.

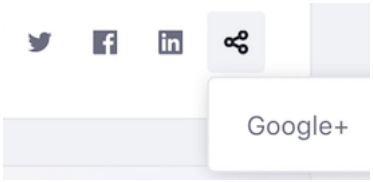

Figure 403.1: These social bookmarks are in the inline display style.

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 app's views. However, note that you don't need to implement social bookmarks in your app's asset renderers. 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. See Social Bookmarks for information on this tag's attributes. Here's an example of using this 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) %>"
/>
```

403.1 Related Topics

Social Bookmarks
    Creating Social Bookmarks
    Asset Framework

CHAPTER 404

CREATING SOCIAL BOOKMARKS

By default, Liferay DXP contains social bookmarks for Twitter, Facebook, and LinkedIn. 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`. The steps here show you how to do this.

404.1 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);
}
```

404.2 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` taglib when applying the social bookmark.

4. Add the Clay link. See the `clay:link` documentation for a full description of its attributes.

```
<clay:link
        buttonStyle="secondary"
        elementClasses="btn-outline-borderless btn-sm lfr-portal-tooltip"
        href="<%= socialBookmark.getPostURL(title, url) %>"
        icon="twitter"
        title="<%= socialBookmark.getName(locale) %>"
/>
```

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.

To see a complete, real-world example of a social bookmark implementation, see Liferay's Twitter social bookmark code.

404.3 Related Topics

Applying Social Bookmarks

Using the Clay Taglib in Your Portlets

CHAPTER 405

ADDING COMMENTS TO YOUR APP

Liferay provides taglibs that enable comments on your app's content. Here, you'll learn how to use these taglibs, using a sample Guestbook app as an example.

Follow these steps to enable commenting on your app's content:

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 Configuring JSP Templates for an Asset Renderer.

405.1 Related Topics

Asset Framework
    Rating Assets

CHAPTER 406

RATING ASSETS

In only a few lines of code, you can use a taglib to enable ratings for your app's content. The steps here show you how. For more information on this taglib and ratings in general, see Ratings.

Figure 406.1: Users can rate content to let others know how they really feel about it.

Follow these steps to enable ratings in your app. Note that these steps use a sample Guestbook app as an example. 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. This example uses the stars rating type:

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

406.1 Related Topics

Ratings
    Implementing Rating Type Selection
    Customizing Rating Value Transformation

CHAPTER 407

IMPLEMENTING RATING TYPE SELECTION

For administrators to change your app's rating type (e.g. likes, stars, thumbs), you must implement rating type selection. The steps here show you how. For a detailed explanation of these steps and rating type selection, see Rating Type Selection.

1. Implement the `PortletRatingsDefinition` interface, registering the class as an OSGi component. In the `@Component` annotation, set the `model.class.name` property to the fully qualified name of the class that will use this rating definition. 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. Implement the `PortletRatingsDefinition` methods `getDefaultRatingsType` and `getPortletId` to return the entity's default rating type and the portlet ID of the main portlet that uses the entity, respectively. In this example, the rating type is thumbs and the portlet ID is for the Blogs portlet:

```
@Override
public RatingsType getDefaultRatingsType() {
    return RatingsType.THUMBS;
}

@Override
public String getPortletId() {
    return PortletKeys.BLOGS;
}
```

407.1 Related Topics

Rating Type Selection
    Rating Assets
    Customizing Rating Value Transformation

CHAPTER 408

CUSTOMIZING RATING VALUE TRANSFORMATION

To customize rating value transformation, you must create an OSGi component that implements `RatingsDataTransformer`. The steps here show you how. For a detailed explanation of these steps and rating value transformation, see Rating Value Transformation.

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:

```
@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);
            }
        }
```

This example irreversibly transforms the rating type from likes 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.

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);
                }
            }
        };
    }

}
```

408.1 Related Topics

Rating Value Transformation
    Implementing Rating Type Selection
    Rating Assets

CHAPTER 409

FLAGGING INAPPROPRIATE ASSET CONTENT

The asset framework supports a system for flagging inappropriate content in apps. The steps here show you how to enable it in your app.

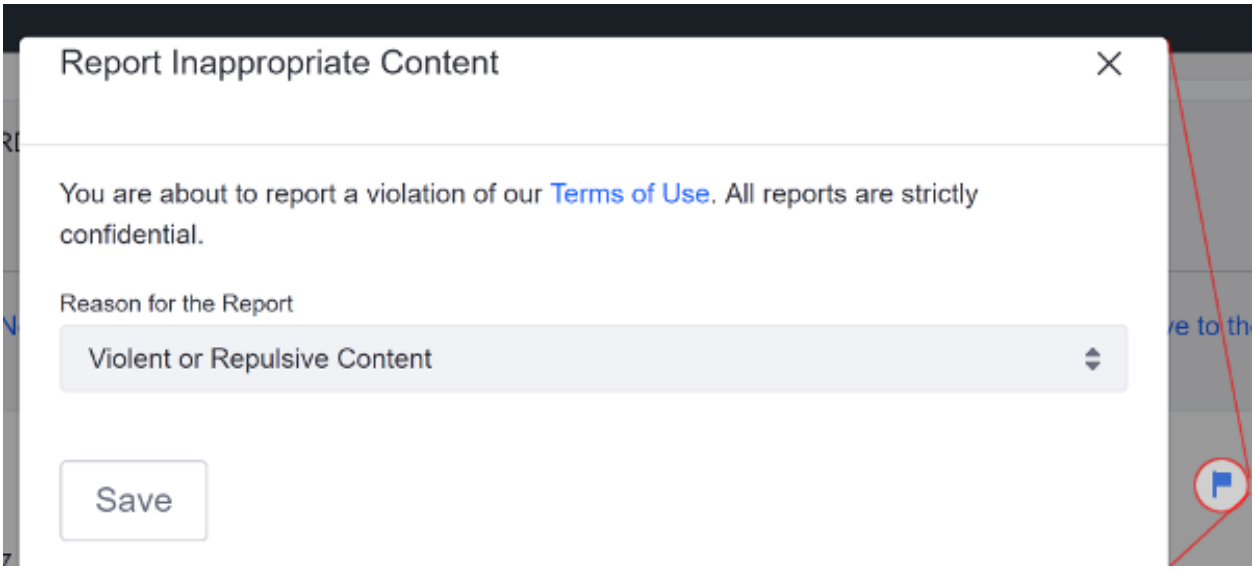

Figure 409.1: Users can flag objectionable content.

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 `liferay-flags:flags` tag to add the flags component:

```
<liferay-flags:flags
    className="<%= Entry.class.getName() %>"
    classPK="<%= entry.getEntryId() %>"
    contentTitle="<%= title %>"
    message="flag-this-content"
    reportedUserId="<%= reportedUserId %>"
/>
```

   The `reportedUserId` attribute specifies the ID of the user who flagged the asset.

409.1 Related Topics

Rating Assets
    Social API
    Asset Framework

CHAPTER 410

CONFIGURABLE APPLICATIONS

Many applications must be configurable, whether by end users or administrators. A 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.

Liferay DXP's configuration API is easy to use and is not limited to portlets. When you define configuration options in a Java interface, Liferay's configuration framework auto-generates a UI, sparing you the trouble of developing an interface for your users to select configuration options.

Note: To see a working application configuration, deploy the `configuration-action` Blade sample and navigate to System Settings (*Control Panel → Configuration → System Settings*). In the Platorm section's Third Party category, click the *Message display configuration* entry.

Add the *Blade Message Portlet* to a page to test your configuration choices.

Complete these three high level tasks to integrate your application with the configuration framework:

1. Provide a way to set configurations in the UI.
2. Set the scope where the application is configured.
3. Read configuration values in your business logic.

410.1 Using a Configuration Interface

You can take care of the first two steps by Creating A Configuration Interface. This Java interface does a number of things:

- Just by existing, it gives you a UI in *System Settings*, so you don't have to write one yourself. Score!

- It defines the configuration options that will appear in the UI.
- It defines the type {`int`, `String`, etc.) of values each configuration takes.
- It defines the scope of your configuration. Bonus in 7.0: if your configuration is scoped to anything other than `SYSTEM`, you get an additional UI generated for you in *Instance Settings*. More on scope in a minute.
- It categorizes your configuration screen so that it can be easily found in *System Settings* and *Instance Settings*. If you skip this the screen will be put in a default location.

A few things you need to know:

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.

Configuration Scope Scope defines where a configuration value applies. Here are the most common configuration scopes:

- `SYSTEM`: Configuration values apply throughout the system.
- `COMPANY`: One set of configuration values is stored for each virtual instance, so each instance can be configured individually.
- `GROUP`: Each group can be configured individually.
- `PORTLET_INSTANCE`: this refers to apps that can be placed on a page as a widget. Each widget can be configured individually.

Configuration UIs : When you create a configuration interface of any sort, a UI is generated for you in *System Settings*. If your configuration is scoped to `COMPANY`, `GROUP`, or `PORTLET_INSTANCE`, an additional UI is generated in *Instance Settings*. Note that while `GROUP` and `PORTLET_INSTANCE` configurations appear in the Instance Settings UI, they can only be used to set defaults for the current instance. No corresponding UI is auto-generated to configure the app at the Site or Portlet level.

Note: An Instance Settings UI is not currently generated for factory configurations. You can track the progress of this issue here.

Default Configurations Default values for any scoped configuration can be set at any wider scope. For example, if your configuration is scoped to the `GROUP`, you can set a system-wide default in *System Settings, an instance-wide default in* Instance Settings*, or both. Any configuration at a narrower scope will always override a configuration at a wider scope.

Read more about configuration scope here.
When you complete your configuration interface, you're done with steps 1 and 2 above.

410.2 Reading Configuration Values

The final step is to make your app read the configuration values that users enter. There are a number of ways to do that:

If your configuration is scoped to `COMPANY` or `GROUP` you must use `ConfigurationProvider` This allows your app to read different configuration values from each site, virtual instance, or whatever the configuration is scoped to.

If your configuration is scoped to `PORTLET_INSTANCE`, you can still use `ConfigurationProvider`, but using `PortletDisplay` is simpler and more convenient. See `PortletDisplay`.

If you only want your app to be configurable at the `SYSTEM` scope, you have a few options. `ConfigurationProvider` will work fine, but there are alternatives that—since they don't need to query multiple sources—can yield modest performance benefits. Which one you use depends on what kind of class you're using to read configuration values. Here are your options:

- Read with an MVC portlet's JSP
- With an MVC Portlet's Portlet Class
- With any other Component Class

410.3 Further Customization

At this point you may be asking, "But what if I don't *like* the auto-generated UI?" Relax. There are a number of ways you can customize it, or even suppress it entirely so you can put your own UI in its place.

- Implement the `ConfigurationFormRenderer` interface to customize the auto-generated UI in system settings.
- If you need more flexibility—perhaps your app needs multiple configuration screens, or maybe you've already written a configuration UI and just want to insert it without bothering to write a configuration interface—implement the `ConfigurationScreen` interface to implement your own.
- If you're using a configuration interface but you don't want a UI to be generated—maybe you're using a `ConfigurationScreen` implementation instead, or maybe you just want configuration to be handled programatically or by .config file —you can just leave it out.
- If you want the UI to render only under certain circumstances, you can write logic to do that, too.

Enough conceptual stuff. You're ready to get started with some code. If you already have an app that was configurable under an earlier version of Liferay DXP, see Upgrading a Legacy App.

CHAPTER 411

CREATING A CONFIGURATION INTERFACE

First, you'll learn how to create a configuration with no scope declaration. This automatically scopes your configuration to `SYSTEM`.

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();


}
```

   This 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:

   Meta.OCD: Registers this class as a configuration with a specific id. **The ID must be the fully qualified configuration class name.**

   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.

```
 **Note:** You can dynamically populate select field options with the
 [`ConfigurationFieldsOptionProvider` interface](/docs/7-2/frameworks/-/knowledge_base/f/dynamically-populating-select-list-fields-in-the-configuration-ui)
```

```
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](http://bnd.bndtools.org/chapters/210-metatype.html).
```

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.

Also Note: Your project depends on a `-metatype: *` declaration in its metadata. If you're in a Liferay Workspace(or otherwise applying the workspace plugin to your build), it's added automatically at build time. Otherwise, add it manually in your module's `bnd.bnd`. It's required to provide information about your app's configuration options so that a configuration UI can be generated.

When you register a configuration interface, a UI is auto-generated for it in *System Settings → Platform → Third Party*. That's the default location; read the next section to learn how to move it somewhere more intuitive.

CHAPTER 412

CATEGORIZING THE CONFIGURATION

By default, the configuration UI for your app is generated in *System Settings → Platform → Third Party*. You probably don't really want it there; by categorizing your configuration you can place it somewhere intuitive and easy to find.

Note: If you scope your configuration so that a UI is generated in Instance Settings as well, your categorization will apply to that UI also.

You have two options: 1) locate your configuration UI in an existing category and section, or 2) create your own.

Here are the default System Settings sections. All available categories are nested beneath these sections:

1. Content and Data
2. Platform
3. Security
4. Commerce
5. 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.

412.1 Specifying a Configuration Category

Specify the category for your UI by placing an `@ExtendedObjectClassDefinition` annotation in your configuration interface. This example, which appears right before the interface's `@Meta.OCD` annotation, places the UI in the `dynamic-data-mapping` category in the Content management section:

```
@ExtendedObjectClassDefinition(
    category = "dynamic-data-mapping",
    scope = ExtendedObjectClassDefinition.Scope.GROUP
)
```

This annotation does two things:

- Specifies the `dynamic-data-mapping` category in the Content Management section.
- Sets the scope of the configuration. You'll learn more about this next.

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.

412.2 Creating New Sections and Categories

If you don't like the default sections and categories, you can create your own by implementing the `ConfigurationCategory` interface.

Here's code that creates the *Content and Data* section and the *Dynamic Data Mapping* category:

```
@Component(service = ConfigurationCategory.class)
public class DynamicDataMappingConfigurationCategory
    implements ConfigurationCategory {

    @Override
    public String getCategoryIcon() {
        return _CATEGORY_ICON;
    }

    @Override
    public String getCategoryKey() {
        return _CATEGORY_KEY;
    }

    @Override
    public String getCategorySection() {
        return _CATEGORY_SECTION;
    }

    private static final String _CATEGORY_ICON = "dynamic-data-mapping";

    private static final String _CATEGORY_KEY = "dynamic-data-mapping";

    private static final String _CATEGORY_SECTION = "content-and-data";

}
```

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.

Note: the language keys for categories and sections must follow a specific format. Prefix each section language key with `category-section.` and each category language key with `category.` For example:

```
category-section.content-and-data=Content and Data    category.dynamic-data-mapping=Dynamic Data Mapping
```

Next you'll specify the scope of your application's configuration.

CHAPTER 413

SCOPING CONFIGURATIONS

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.

413.1 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. Pass one of these valid scope options to `@ExtendedObjectClassDefinition`:

`Scope.SYSTEM`: for system scope `Scope.COMPANY`: for virtual instance scope `Scope.GROUP`: for site 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 {
```

413.2 Step 2: Enabling the Configuration for Scoped Retrieval

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 configuration interface you created. 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;
}
```

That's all there is to it. Now the configuration is scoped and supports scoped retrieval via `ConfigurationProvider`. See the next section for details on retrieval.

CHAPTER 414

READING SCOPED CONFIGURATION VALUES

If your configuration is scoped to anything other than `SYSTEM`, you have two options for reading configuration values.

- Use `ConfigurationProvider`. This works for any kind of configuration, and is the only way to read configuration values at the `COMPANY` and `GROUP` scopes.
- Use `PortletDisplay`. This is the recommended approach for configurations at the `PORTLET_INSTANCE` scope, but only works at that scope.

414.1 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(unbind = "-")
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`).

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*.

`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. 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.

414.2 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.

CHAPTER 415

READING UNSCOPED CONFIGURATION VALUES FROM AN MVC PORTLET

If your configuration is scoped to `SYSTEM` or is unscoped (which amounts to the same thing), you have a couple of options for reading configuration values. There are two ways to do this:

- Add a configuration to the request and read it from the view layer (commonly a JSP).
- Read values directly from 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.

415.1 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. MVC Portlet classes are Component classes. If you have a Bean Portlet or PortletMVC4Spring class, the configuration below goes in `portlet.xml` and `liferay-portlet.xml`. 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`. This ensures that the method is invoked when the Component is started, and again whenever the configuration is changed.

```
@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.

415.2 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.

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 next section 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.

415.3 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 user`FavoriteColor` 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 a Portlet. The next section covers reading configuration values from an OSGi Component class that is not part of a portlet.

CHAPTER 416

READING UNSCOPED CONFIGURATION VALUES FROM A COMPONENT

Follow these steps to read `SYSTEM` scoped or unscoped configuration values from a Component that isn't part of a portlet:

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 gets 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.

CHAPTER 417

CUSTOMIZING THE CONFIGURATION USER INTERFACE

There are three ways to customize a configuration UI.

- Provide a custom form for a configuration object. This modifies the auto-generated UI.
- 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. You'll want this option if you're using a configuration interface but don't wan't a UI generated for you.

417.1 Providing Custom Configuration Forms

Customize your auto-generated UI by implementing the `ConfigurationFormRender` interface. To write this interface, you must refer to your configuration interface. For this example, refer to this configuration interface from Liferay's 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();
}
```

This example defines one configuration option, `symbols`, which takes an array of values. Implement `ConfigurationFormRenderer`'s 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 method names 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.

417.2 Creating a Completely Custom Configuration UI

You get more flexibility if you create a completely custom UI using 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-configuration-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.

417.3 Excluding a Configuration UI

If you don't want a UI to be generated for you, you have two options.

- If you don't want a UI generated no matter what, use the `generateUI` property.
- If you only want the UI to render under specific circumstances (defined by logic you'll write yourself), use the configuration visibility SPI.

417.4 Using `generateUI`

To turn off auto-generating at all scopes, include the `ExtendedObjectClassDefinition` annotation property `generateUI` in your configuration interface. The property defaults to `true`; here is an example setting it to `false`:

```
@ExtendedObjectClassDefinition(generateUI=false)
@Meta.OCD(
  id = "com.foo.bar.LowLevelConfiguration",
)
public interface LowLevelConfiguration {

  public String[] foo();
  public String bar();

}
```

Now no UI is auto-generated for this configuration. You can still manage the configuration via a `ConfigurationScreen` implementation, a .config file, or programmatically.

417.5 Using the Configuration Visibility SPI

The configuration visibility SPI involves implementing a single interface, `ConfigurationVisibilityController`. You can see the whole interface here.

To implement the interface, you must identify your configuration interface using an `@Component` property, then write your own logic for the interface's only method, `isVisible`. Here is a sample implementation from Liferay's source code:

```
@Component(
    immediate = true,
    property = "configuration.pid=com.liferay.sharing.internal.configuration.SharingCompanyConfiguration",
    service = ConfigurationVisibilityController.class
)
public class SharingCompanyConfigurationVisibilityController
    implements ConfigurationVisibilityController {
```

```
    @Override
    public boolean isVisible(
        ExtendedObjectClassDefinition.Scope scope, Serializable scopePK) {

        SharingConfiguration systemSharingConfiguration =
            _sharingConfigurationFactory.getSystemSharingConfiguration();

        return systemSharingConfiguration.isEnabled();
    }

    @Reference
    private SharingConfigurationFactory _sharingConfigurationFactory;

}
```

Note that the property `configuration.pid` identifies the configuration interface of the UI to be hidden. In this example, the configuration UI only renders when `systemSharingConfiguration.isEnabled` returns `true`.

CHAPTER 418

CONFIGURATION FORM RENDERER

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 section.

This article 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.

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.

418.1 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});
    }
```

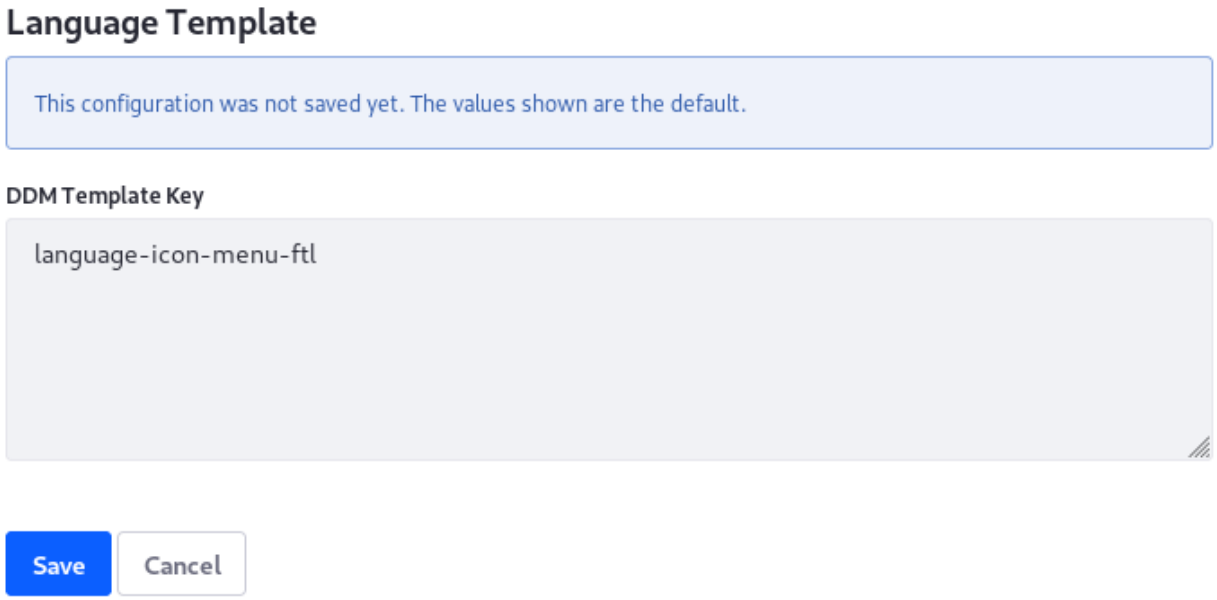

Figure 418.1: The auto-generated UI for the Language Template configuration screen is sub-optimal. A select list with more human readable options is preferable.

```
    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`.

418.2 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 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 = request.getLocale();

    LanguageTemplateConfigurationDisplayContext
        languageTemplateConfigurationDisplayContext =
            new LanguageTemplateConfigurationDisplayContext();

    languageTemplateConfigurationDisplayContext.setCurrentTemplateName(
        _siteNavigationLanguageWebTemplateConfiguration.ddmTemplateKey());

    long groupId = 0;

    long companyId = _portal.getCompanyId(actionRequest);

    Group group = _groupLocalService.fetchCompanyGroup(companyId);

    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.

418.3 Writing the JSP Markup

Now write the JSP:

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

<%
LanguageTemplateConfigurationDisplayContext
    languageTemplateConfigurationDisplayContext = (LanguageTemplateConfigurationDisplayContext)request.getAttribute(LanguageTemplateConfigurationDisp

Admin: Instance Settings    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?

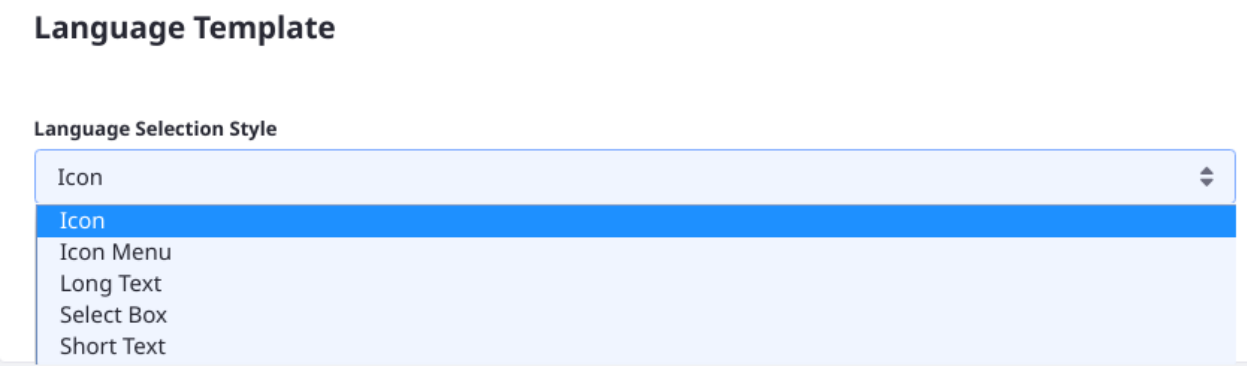

Figure 418.2: A select list provides a more user friendly configuration experience than a text field.

Now, administrators encountering the Language Template entry in System Settings won't be handicapped by not knowing the available DDM Template Keys. Providing the available values in a select field wildly enhances the user experience.

CHAPTER 419

USING DDM FORM ANNOTATIONS IN CONFIGURATION FORMS

The auto-generated configuration form you get by just creating a configuration interface can be too simplistic for some configurations. To enhance it, use the Dynamic Data Mapping (DDM) Form Annotations.

To use DDM Annotations in configuration forms,

1. Configure the module dependencies.
2. Write a `ConfigurationForm` class, including just the fields that you want to leverage the enhanced forms capability. This is similar to the configuration interface, but with field annotations from the Liferay Dynamic Data Mapping API rather than the bndtools metatype specification. The fields here must match fields defined in the configuration interface.
3. Implement a `ConfigurationDDMFormDeclaration` to mark your configuration as having a `ConfigurationForm`.

This article assumes you already have an auto-generated configuration UI.

Note that the example code here splits up the Configuration interface and the Configuration Form interface. If you'd rather mash these together into one class, you can. This approach might make sense if you have a small number of fields in your configuration and a simple form to create. If you have numerous configuration fields and/or a complex form to create, or if you're taking an existing configuration and extending it to use the DDM Form annotations, you can consider separating the classes, as shown here.

419.1 Step 1: Declare the Dependencies

In the `build.gradle` file, add `compileOnly` dependencies on the `dynamic-data-mapping-api` and `configuration-admin-api` module artifacts:

```
compileOnly group: "com.liferay", name: "com.liferay.dynamic.data.mapping.api", version: "5.2.0"
compileOnly group: "com.liferay", name: "com.liferay.configuration.admin.api", version: "2.0.2"
```

419.2 Step 2: Write the Configuration Form

This step requires annotating the class with `@DDMForm` to set up the form, and annotating each method with `@DDMFormField`. Begin by creating the class body, annotating each configuration field (interface method) with `@DDMFormField`:

```
public interface MyFooConfigurationForm {

    @DDMFormField(
        label = "%label-key-for-field-1",
        tip = "%description-key-for-field-1",
        properties = {

            "placeholder=%enter-a-value",
            "tooltip=%some-tooltip-text"
        }
    )
    public String[] textArrayValues();

    @DDMFormField(
        label = "%date",
        tip = "%date-description",
        type = "date")
    public String date();

    @DDMFormField(
        label = "%select",
        optionLabels = {"%foo", "%bar"},
        optionValues = {"foo", "bar"},
        type = "select")
    public String select();

    @DDMFormField(
        label = "%numeric",
        properties = {
            "placeholder=%milliseconds",
            "tooltip=%enter-an-integer-between-1000-and-30000"
        },
        validationErrorMessage = "%please-enter-an-integer-between-1000-and-30000-milliseconds",
        validationExpression = "(numeric ≥ 1000) && (numeric ≤ 30000)",
        type = "numeric")
    public String numeric();

    @DDMFormField(
        label = "%checkbox",
        properties = "showAsSwitcher=true")
    public boolean checkbox();

}
```

Once the field annotations are in place, lay out the form itself, right above the class declaration. This example shows the layout of the `UserFileUploadsConfigurationForm`, so that you can see the resulting form via the below screenshot:

```
@DDMForm
@DDMFormLayout(
    paginationMode = com.liferay.dynamic.data.mapping.model.DDMFormLayout.SINGLE_PAGE_MODE,
    value = {
        @DDMFormLayoutPage(
            {
                @DDMFormLayoutRow(
                    {
                        @DDMFormLayoutColumn(
```

```
                        size = 12,
                        value = {
                            "imageCheckToken", "imageDefaultUseInitials",
                            "imageMaxSize"
                        }
                    )
                }
            ),
            @DDMFormLayoutRow(
                {
                    @DDMFormLayoutColumn(
                        size = 6, value = "imageMaxHeight"
                    ),
                    @DDMFormLayoutColumn(size = 6, value = "imageMaxWidth")
                }
            )
        }
    )
)
public interface MyFooConfigurationForm {
```

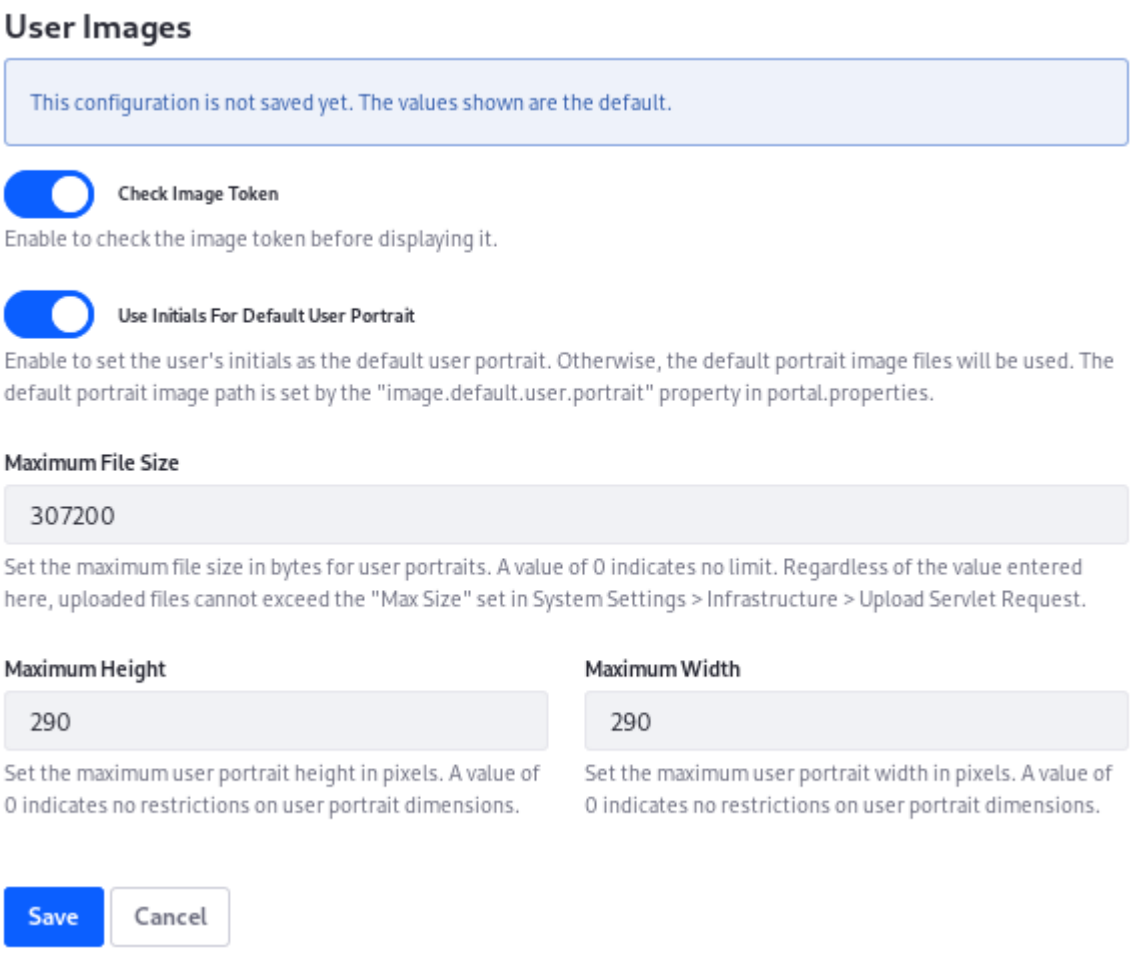

Figure 419.1: The DDM annotations are used to lay out this configuration form.

Next, you must make sure the configuration framework knows about your slick form.

419.3 Step 3: Write the Form Declaration

Create a new implementation of `ConfigurationDDMFormDeclaration` to register your new configuration form class:

```
package com.liferay.docs.my.foo.configuration.definition;

import com.liferay.configuration.admin.definition.ConfigurationDDMFormDeclaration;
import org.osgi.service.component.annotations.Component;
...

@Component(
    immediate = true,
    property = "configurationPid=com.liferay.docs.my.foo.configuration.MyFooConfiguration",
    service = ConfigurationDDMFormDeclaration.class
)
public class MyFooConfigurationDDMFormDeclaration
    implements ConfigurationDDMFormDeclaration {

    @Override
    public Class<?> getDDMFormClass() {
        return MyFooConfigurationForm.class;
    }

}
```

The `configurationPid` must match the fully qualified class name of the configuration interface.

Now your configuration class is backed by the form-building power of Liferay's native Forms application.

To see how this is done for one of Liferay's own configurations, check out all of the configuration classes for the User Images configuration (Control Panel → Configuration → System Settings → User Images):

`UserFileUploadsConfigurationForm`
`UserFileUploadsConfiguration.java`
`UserFileUploadsConfigurationBeanDeclaration.java`
`UserFileUploadsConfigurationDDMFormDeclaration.java`

CHAPTER 420

UPGRADING A LEGACY APP

If you have an app that was made configurable under an earlier version of Liferay DXP, you can upgrade without having to reconfigure any of your app's instances.

If you have an app that was configurable using the mechanisms of Liferay Portal 6.2 and before, refer to Transitioning from Portlet Preferences to the Configuration API.

If you have an app with a configuration interface scoped to anything other than `SYSTEM` and a custom UI for saving configuration values to `PortletPreferences`, you have two options:

- Keep using your custom UI. Deactivate the auto-generated UI in *Instance Settings* by setting the scope in your configuration interface to `SYSTEM`. This is quick and easy, but won't make your code easier to maintain in the long term.

  For other ways to disable the auto-generated UI, see Excluding a Configuration UI

- Write an Upgrade Process to convert your configuration values in `PortletPreferences` to an instance-scoped OSGi configuration, using the `saveCompanyConfiguration` method in the `ConfigurationProvider` interface.

You don't have to use `saveCompanyConfiguration`, but doing so meets all the necessary requirements for an upgrade process: it must be a factory instance with a factory PID of `Unknown macro:[base-pid].scoped`, and it must contain a `companyId` property.

Then remove your custom UI. If you're reading configuration values using `ConfigurationProvider`'s `getCompanyConfiguration` method, the auto-generated UI picks up where you left off, with no need to reconfigure anything.

CHAPTER 421

DYNAMICALLY POPULATING SELECT LIST FIELDS IN THE CONFIGURATION UI

You've always been able to provide a select list for your configuration options by entering each label and value directly in the `@Meta.AD` annotation of the Configuration interface.

```
@Meta.AD(
    deflt = "enabled-with-warning", name = "csv-export",
    optionLabels = {"enabled", "enabled-with-warning", "disabled"},
    optionValues = {"enabled", "enabled-with-warning", "disabled"},
    required = false
)
public String csvExport();
```

Now, thanks to the `ConfigurationFieldOptionsProvider` interface, you can populate select list configurations dynamically, using custom logic.

Follow these steps to populate the select list fields dynamically in your configuration UI:

1. Use an `@Component` annotation to register the `ConfigurationFieldOptionsProvider.class` service and include two properties:

   `configuration.field.name`: The name of the attribute in the configuration interface

   `configuration.pid`: The ID of the corresponding configuration interface (usually the fully qualified class name)

   For example,

   ```
   @Component(
       property = {
           "configuration.field.name=enabledClassNames",
           "configuration.pid=com.liferay.asset.auto.tagger.google.cloud.natural.language.internal.configuration.GCloudNaturalLanguageAssetAutoT
           "configuration.pid=com.liferay.asset.auto.tagger.opennlp.internal.configuration.OpenNLPDocumentAssetAutoTaggerCompanyConfiguration"
       },
       service = ConfigurationFieldOptionsProvider.class
   )
   ```

2. Implement the `ConfigurationFieldOptionsProvider` interface:

```
public class MyConfigurationFieldOptionsProvider implements
ConfigurationFieldOptionsProvider {
    ..
}
```

3. The `getOptions` method returns a list of `Options` consisting of the label and value fields. The labels provided here are translated to `optionLabels`, and the values as `optionValues`, in the configuration interface.

```
public List<Option> getOptions() {
    List<AssetRendererFactory<?>> assetRendererFactories =
        AssetRendererFactoryRegistryUtil.getAssetRendererFactories(
            CompanyThreadLocal.getCompanyId());

    Stream<AssetRendererFactory<?>> stream =
        assetRendererFactories.stream();

    return stream.filter(
        assetRendererFactory -> {
            TextExtractor textExtractor =
                _textExtractorTracker.getTextExtractor(
                    assetRendererFactory.getClassName());

            return textExtractor != null;
        }
    ).map(
        assetRendererFactory -> new Option() {

            @Override
            public String getLabel(Locale locale) {
                return assetRendererFactory.getTypeName(locale);
            }

            @Override
            public String getValue() {
                return assetRendererFactory.getClassName();
            }

        }
    ).collect(
        Collectors.toList()
    );
}
```

   This code gets a list of `AssetRendererFactory` objects and iterates through the list, populating a new list of `Options`, using the asset's type name as the label and the class name as the value. It comes from the `EnabledClassNamesConfigurationFieldOptionsProvider`, which populates the configuration field labeled *Enable Google Cloud Natural Language Text Auto Tagging For* with all the asset types that have registered a `TextExtractor`.

The `ConfigurationFieldOptionsProvider` allows you to populate select lists with configuration options defined by your custom logic.

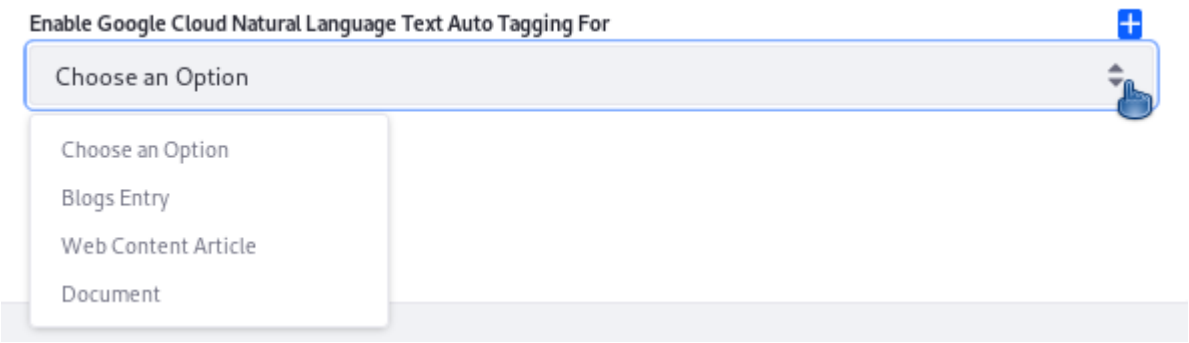

Figure 421.1: The select list in the Google Cloud Natural Language Text Auto Tagging entry is populated programmatically, using the `ConfigurationFieldOptionsProvider`.

CHAPTER 422

CONTENT PUBLICATION MANAGEMENT

Managing content publication is primarily controlled by two frameworks:

- Export/Import
- Staging

These features give you the power to plan page publication and manage content. You can leverage the APIs offered by these frameworks to manage your app's publication process.

The Export/Import and Staging frameworks are closely tied together. They both implement the same interfaces and share the same extension points. By implementing one of these frameworks in your app, you automatically leverage the other. There are a few simple configurations that can be set to customize them separately. You'll learn about this later.

Export/Import can be viewed as the base feature with Staging built on top of it (although they're implemented together). You can visit the Export/Import framework's articles for the base APIs that both it and the Staging frameworks share. You must implement these to implement Staging. Reference the Staging framework's articles for additional configuration pertaining only to it.

Here are a few of the things you can do with the Export/Import and Staging APIs.

422.1 Export/Import

The Export/Import feature adds another dimension to your application by providing a framework for producing reusable content and importing content from other places. By creating LAR files (Liferay ARchive), you can export your data, import it to another system, or even use this feature to back up your content.

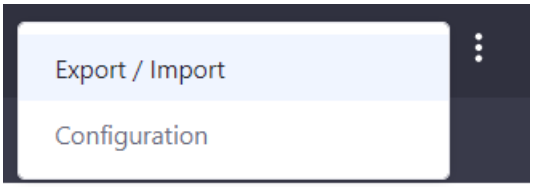

Figure 422.1: Leveraging the Export/Import feature in your app is useful for sharing content.

Export/Import is a default feature for many of Liferay DXP's out-of-the-box apps. It offers an intuitive GUI for managing export/import processes and tracking the history of previous export/imports. You can also easily pick subsets of data to export based on content type, date range, and configurations. Importing content is done using a modern drag-and-drop interface or by selecting the LAR file from your file system.

422.2 Staging

Staging gives you a test environment where you can modify your site and test different configurations without changing your live site. When you implement Staging in your app, its content can be tracked by the staged environment, allowing users to stage your app's data before releasing it to the world.

Here's an example of some functionality you can add to your app with Staging's APIs:

- Local Staging environment tracking
- Remote Staging environment tracking
- Single asset publishing
- Content tracking on page variations

Continue on to learn more about the frameworks that bring content publication management to life!

CHAPTER 423

EXPORT/IMPORT

The Export/Import feature exports content from the portal and imports external content into the portal. Your application is much more site administrator-friendly if users can export/import your application's assets. For example, if you want to export your application's assets to use on another installation or you must 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.

Here's what you'll learn to do with the Export/Import framework:

- Create Staged Models
- Develop Portlet Data Handlers
- Develop Staged Model Data Handlers
- Provide entity-specific local services for Export/Import framework
- Listen to export/import events
- Initiate new export/import processes programmatically

423.1 Staged Models

To track an entity of an application with the Export/Import framework, you must implement the `StagedModel` interface in the app's model classes. It provides the behavior contract for entities during the Export/Import and Staging processes. There are two ways to create staged models for your application's entities:

- Generate them using Service Builder
- Implement the appropriate interfaces manually

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.

See the Developing Staged Models section for more information on the Staged Model architecture.

423.2 Data Handlers

You must implement Data Handlers to use the Export/Import framework to process LAR files in your application. There are two types of data handlers:

- Portlet Data Handlers
- Staged Model Data Handlers

A Portlet Data Handler imports/exports portlet specific data to a LAR file. These classes query and coordinate between staged model data handlers. They also configure the Export/Import and Staging UI options.

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 also exports referenced content.

Visit the Developing Data Handlers section for more information.

423.3 Provide Entity Specific Local Services

When creating data handlers, you must leverage your app's local services to perform Export/Import and Staging related tasks for its entities. When these frameworks operate 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 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.

423.4 Export/Import Event Listeners

The `ExportImportLifecycleListener` framework is for listening for certain staging or export/import events (like export successes and import failures) during the publication process so you can take some action. You can also listen for processes comprised of many events and take action when these processes are initiated. For example, you can listen for when

- Staging has started
- A portlet export has failed
- An entity export has succeeded

After an event is triggered, you can take an action like these:

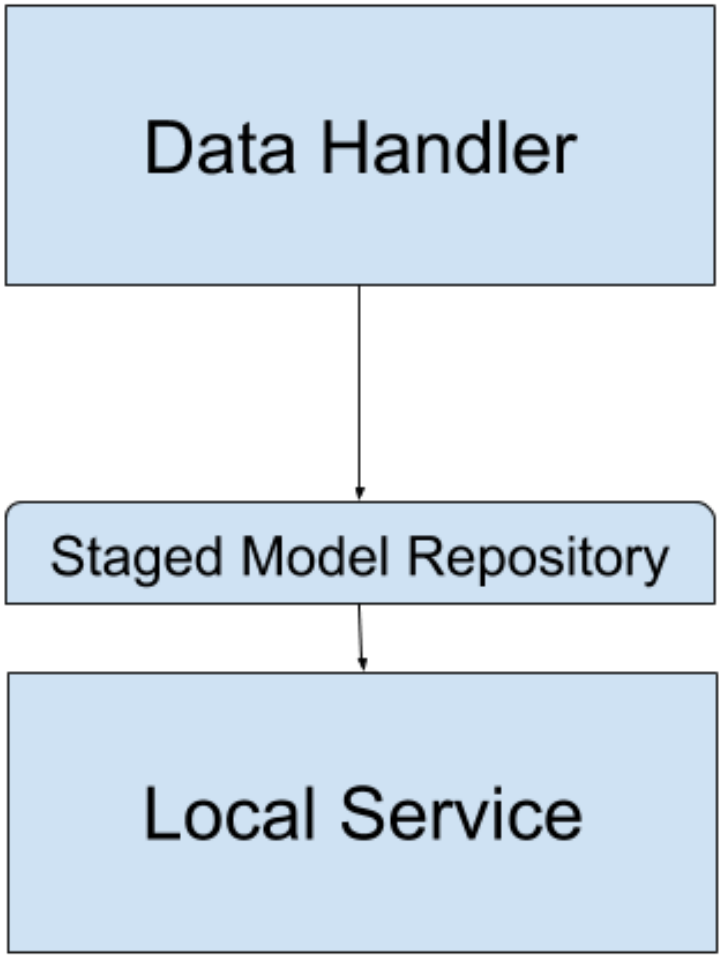

Figure 423.1: Staged Model Repositories provide a Staging-specific layer of functionality for your local services.

- Print information about the event to your console
- When an import process has completed, clear the cache.

For a complete list of events you can listen for, see `ExportImportLifecycleConstants`.
You must extend one of the two Base listener classes:

- `BaseExportImportLifecycleListener`: listens for specific *events* during a lifecycle. For example, if a layout export fails, you might take some action.

- `BaseProcessExportImportLifecycleListener`: listens for *processes* during a lifecycle. A process usually consists of many individual events. For example, if a site publication fails, you might take some action. Methods provided by this base class are only run once when the desired process action occurs.

What's the difference between events and processes?
Events: particular actions that occur during processing (example event listener: `CacheExportImportLifecycleList`
Processes: longer running groups of events (example process listener: `ExportImportProcessCallbackLifecycleLis`
Use the listener type that is most appropriate for your use case.

423.5 Export/Import Processes

You can start the process programmatically instead of through the UI. This lets you provide new interfaces or mimic the functionality of these features in your own application.

To initiate an 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 implementing export, you must call services offered by the `ExportImportService` interface. All the methods in this interface require an `ExportImportConfiguration` object. You can generate these configuration objects, so you can easily pass them in your service methods.

There are three factory classes that are useful to create an `ExportImportConfiguration` object:

- `ExportImportConfigurationSettingsMapFactory`: provides many `build` methods to create settings maps for various scenarios like importing, exporting, and publishing layouts and portlets. For example, you can reference `UserGroupLocalServiceImpl.exportLayouts(...)` and `GroupLocalServiceImpl.addDefaultGuestPublicLayoutsByLAR(...)`.
- `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.

There are two important service interfaces that primarily use `ExportImportConfiguration` objects for exporting, importing, and staging: `ExportImportLocalService` and `StagingLocalService`.

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 also important to know that `ExportImportConfiguration` is a Liferay DXP entity, similar to `User` or `Group`. This means that the `ExportImportConfiguration` framework offers local and remote services, models, persistence classes, and more.

CHAPTER 424

DEVELOPING STAGED MODELS

To track an entity of an application with the Export/Import 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 Developing Data Handlers section 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
- Implementing the required Staging interfaces manually

You can follow step-by-step procedures for creating staged models for your entities by visiting their respective articles.

Continue on to learn more about Staged Models!

424.1 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 for models that function independently 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 {
```

Those are the differences between staged model interfaces.

424.2 Staged Model Attributes

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 differentiates 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. When you go back to modify the entity on your local site and want to publish those changes, the UUID shows that the local and remote entities are the same. The Staging framework can thus 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.

424.3 Adapting Your Business Logic to Build Staged Models

What if you don't want to extend your model with special attributes your business logic doesn't need (removing the ability to leverage Service Builder's auto-generation of staged models)? 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 a completed app and you want 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 instead of the database to access their attributes. Since this kind of app pulls its attributes from a remote system, it is 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 that extends the model-specific interface (e.g., `[Entity]`) and the appropriate staged model interface (e.g., `StagedModel`). This class serves as the Staged Model Adapter.
2. Create a `Staged[Entity]Impl` class that implements the `Staged[Entity]` interface and provides necessary logic for your entity model to be recognized as a staged model.
3. Create a `Staged[Entity]ModelAdapterBuilder` class that implements `ModelAdapterBuilder<[Entity], Staged[Entity]>`. This class adapts the original model to the newly created Staged Model Adapter.
4. Adapt your existing model and call one of the provided APIs to export or import the entity automatically.

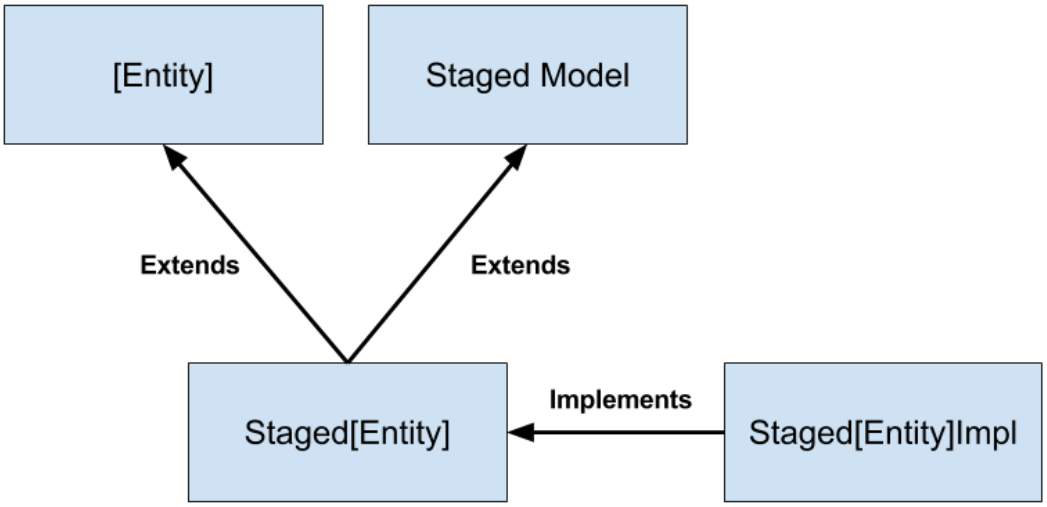

Figure 424.1: The Staged Model Adapter class extends your entity and staged model interfaces.

To step through the process for leveraging the Model Adapter Builder for an existing app, see Creating Staged Models Manually.

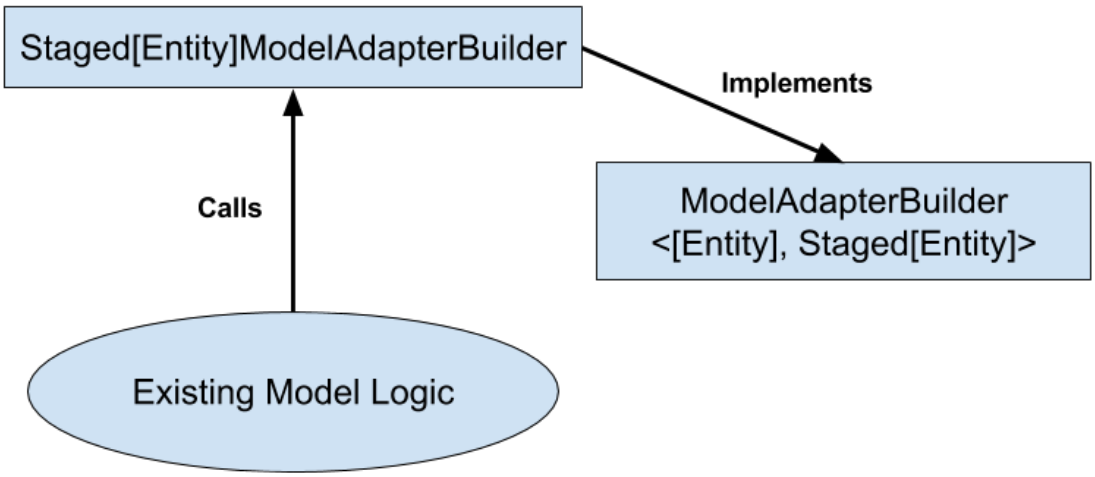

Figure 424.2: The Model Adapter Builder gets an instance of the model and outputs a staged model.

CHAPTER 425

GENERATING STAGED MODELS USING SERVICE BUILDER

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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. If your app doesn't use Liferay's Service Builder, you must configure it in your project. 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 (Gradle-based).

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 service.xml file sets the UUID to true and defines the companyId, createDate, and modifiedDate columns. 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.

CHAPTER 426

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 Export/Import and is referenced solely as an example for how to leverage the ModelAdapterBuilder framework, which lets you adapt your model classes to staged models.

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 {

}
```

```
**Note:** Staged model interfaces typically follow the `Staged[Entity]`
naming convention. The Asset Link framework uses a generic entity called
`AssetLink`.
```

---

2. 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();
```

```
**Note:** 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.
```

```
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:

```
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);
```

Awesome! You've successfully adapted your business logic to build staged models!

CHAPTER 427

DEVELOPING DATA HANDLERS

A common requirement for 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 data handlers as a convenient and reliable way to export/import your data (as a LAR file).

There are two types of data handlers:

- Portlet Data Handlers
- 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.

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, and 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.

Note: Creating data handlers for your app means it's automatically tracked by the Staging framework. You can further customize how Staging handles your app, but creating staged models and data handlers is what registers your app for Staging.

Before implementing data handlers, make sure your application is ready for the Export/Import and Staging frameworks by creating staged models.

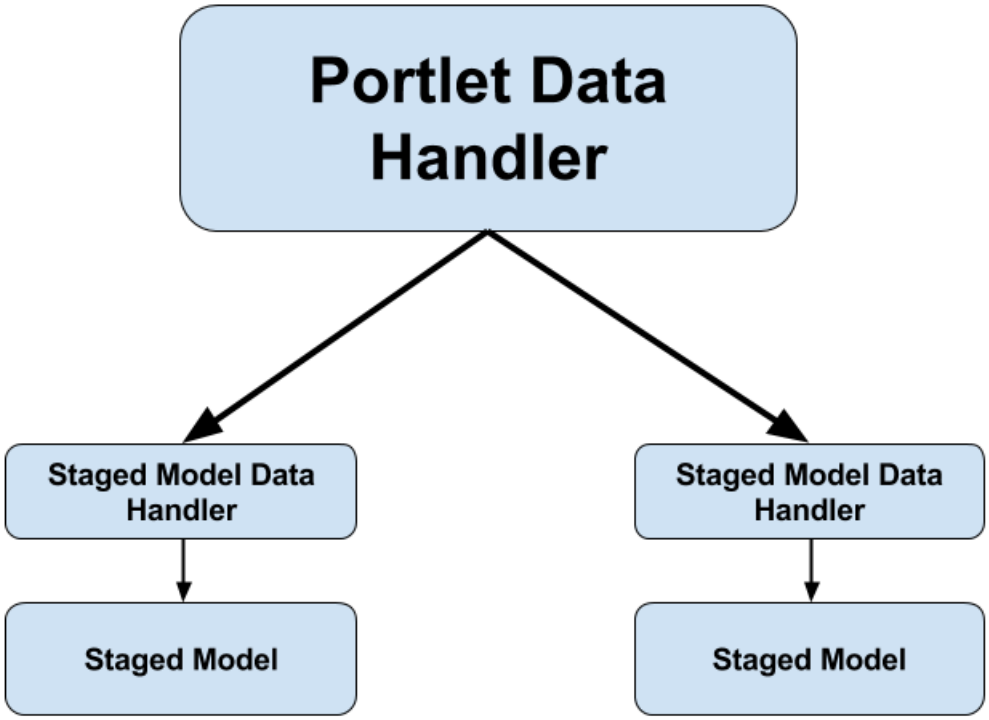

Figure 427.1: The Data Handler framework uses portlet data handlers and staged model data handlers to track and export/import portlet and staged model information, respectively.

427.1 Understanding the `PortletDataHandler` Interface

A Portlet Data Handler imports/exports portlet specific data to a LAR file. These classes query and coordinate between staged model data handlers.

To create a portlet data handler for your staged model, you must implement the `PortletDataHandler` interface by extending the `BasePortletDataHandler` class. Visit the API reference documentation for this interface/class for useful information on the methods provided.

Some guidelines for implementing the `PortletDataHandler` interface are provided below:

The `@Component` annotation section above the implementation class's declaration registers the class as a portlet data handler in the OSGi service registry. There are a few annotation attributes you should set:

`immediate`: activates the component immediately once its provided module has started.

`property`: sets various properties for the component service. You must associate the portlet you wish to handle with this service so they function properly in the export/import environment. You should have one portlet data handler for each portlet (e.g., Bookmarks and Bookmarks Admin).

`service`: points to the `PortletDataHandler.class` interface.

The `activate` method sets what the portlet data handler controls. It also configures the portlet's Export/Import and Staging UI. This method is called during initialization of the component by the `@Activate` annotation; it's invoked after dependencies are set and before services are registered. Five callable `set` methods are described below:

`setDataPortletPreferences`: sets portlet preferences your app should handle.

`setDeletionSystemEventStagedModelTypes`: sets the staged model deletions that the portlet data handler should track. For example, the Bookmarks app tracks Bookmark entries and folders.

`setPublishToLiveByDefault`: controls whether your app is selected to publish on the Publication screen by default.

`setExportControls`: adds fine grained controls over export/import behavior rendered in the Export/Import UI. This also sets the `setImportControls` method. For example, the Bookmarks app adds a checkbox to select Bookmarks content (entries) to export.

`setStagingControls`: adds fine-grained controls over staging behavior rendered in the Staging UI. For example, this enables your app's checkboxes in the Content section displayed during publication.

The `doExportData` method checks if anything should be exported. For example, the Bookmarks app uses this method to check if the user selected Bookmarks entries for export by leveraging the `portletDataContext`. Later, the `ExportImportActionableDynamicQuery` framework runs a query against bookmarks folders and entries to find ones which should be exported to the LAR file.

The `-ActionableDynamicQuery` classes are generated automatically by Service Builder and are available in an app's local services. It queries the database searching for certain Export/Import-specific parameters (e.g., `createDate` and `modifiedDate`), and based on those parameters, finds a list of exportable records from the staged model data handler.

The `doImportData` method queries for entity 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 in the `StagedModelDataHandlerUtil` class. The extracted elements tell Liferay DXP what data to import from the LAR file.

The `doPrepareManifestSummary` method calculates the number of affected entities based on the current export or staging process.

You must retrieve and manage the schema version. This is done with the `getSchemaVersion` and `validateSchemaVersion` methods. 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.getSchemaVersion()` method's Javadoc for more information.

Next you'll learn about the `StagedModelDataHandler` interface.

427.2 Understanding the `StagedModelDataHandler` Interface

A Staged Model Data Handler supplies information about a staged model (entity) to the Export/Import framework, defines a display name for the UI, deletes entities, 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`.

To create a staged model data handler for your staged model, you must implement the `StagedModelDataHandler` interface. This is typically done by extending the `BaseStagedModelDataHandler` class. Visit the API reference documentation for this interface/class for useful information on the methods provided.

Additional implementation details for the `StagedModelDataHandler` interface is provided below:

The `@Component` annotation section above the implementation class's declaration registers the class as a staged model data handler in the OSGi service registry. There are two annotation attributes you should set:

`immediate`: activates the component immediately once its provided module has started.

`service`: points to the `StagedModelDataHandler.class` interface.

The `getClassNames` method provides the class names of the models the data handler tracks. 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.

The `getDisplayName` method retrieves the staged model's display name. This is used in the Export/Import UI.

The `doExportStagedModel` method retrieves your app entity's data element from the `PortletDataContext` and then adds the class model characterized by that data element to the `PortletDataContext`. The `PortletDataContext` data populates the LAR file with your application's data during the export process.

Note: 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. Note that once an entity has been exported, subsequent calls to the export method don't repeat the export process multiple times, ensuring optimal performance.

The `doImportStagedModel` method imports the staged model data. An important feature of the import process is that all exported reference elements are automatically imported when needed. The method must therefore only find the new assigned ID for the folder before importing the entry.

The `PortletDataContext` keeps the data up-to-date during the import process. The old ID and new ID mapping can be reached by using the `portletDataContext.getNewPrimaryKeysMap()` method. This method also checks the import mode (e.g., *Copy As New* or *Mirror*) and, depending on the process configuration and existing environment, adds or updates the entry.

The `doImportMissingReference` 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.

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.

Since you have references from two separate Sites with differing IDs, the system can't match them during publication. 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 consistent 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.

Continue in the section to learn how to develop data handlers for your app.

CHAPTER 428

CREATING PORTLET DATA HANDLERS

In this tutorial, you'll create the `BookmarksPortletModelDataHandler` class used for the Bookmarks application. The Bookmarks application's portlet data handler tracks system events dealing with Bookmarks entities. It also configures the Export/Import and Staging UI options 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:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + BookmarksPortletKeys.BOOKMARKS
    },
    service = PortletDataHandler.class
)
```

4. Set what the portlet data handler controls and the portlet's Export/Import and Staging UIs 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()));
    setStagingControls(getExportControls());
}
```

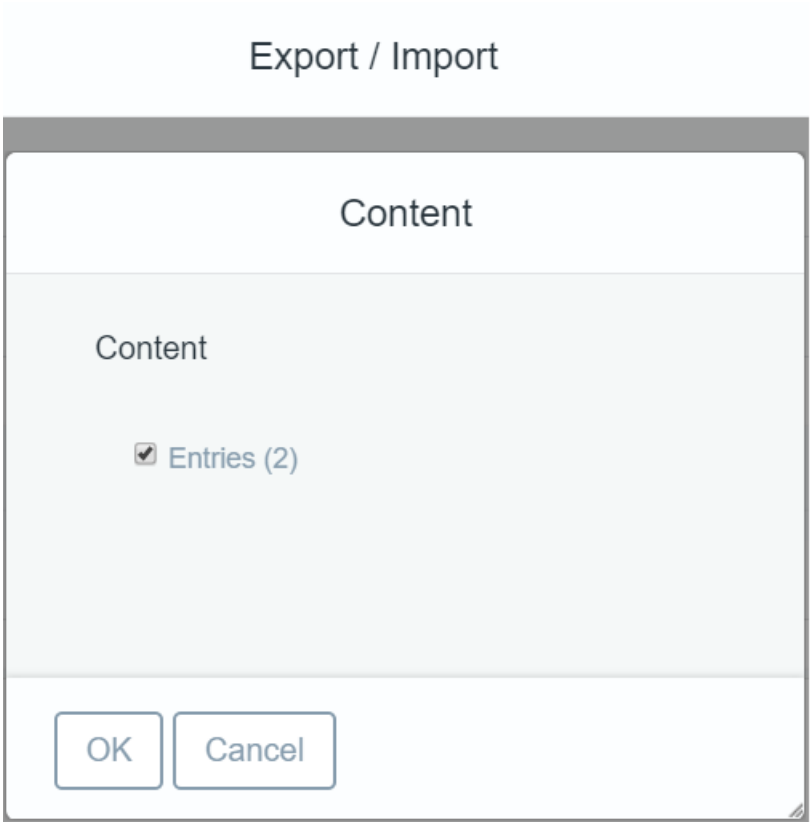

Figure 428.1: You can select the content types you'd like to export/import in the UI.

5. For the Bookmarks portlet data handler to reference its entry and folder staged models successfully, you must set them in your class:

```
@Reference(unbind = "-")
protected void setBookmarksEntryLocalService(
    BookmarksEntryLocalService bookmarksEntryLocalService) {

    _bookmarksEntryLocalService = bookmarksEntryLocalService;
}

@Reference(unbind = "-")
protected void setBookmarksFolderLocalService(
    BookmarksFolderLocalService bookmarksFolderLocalService) {

    _bookmarksFolderLocalService = bookmarksFolderLocalService;
}

private BookmarksEntryLocalService _bookmarksEntryLocalService;
private BookmarksFolderLocalService _bookmarksFolderLocalService;
```

The set methods must be annotated with the @Reference annotation.

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;
}
```

8. 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 {

    if (ExportImportDateUtil.isRangeFromLastPublishDate(
            portletDataContext)) {

        _staging.populateLastPublishDateCounts(
            portletDataContext,
            new StagedModelType[] {
                new StagedModelType(BookmarksEntry.class.getName()),
                new StagedModelType(BookmarksFolder.class.getName())
            });

            return;
        }

    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.

9. Set the XML schema version for the XML files included in your exported LAR file:

```
public static final String SCHEMA_VERSION = "1.0.0";
```

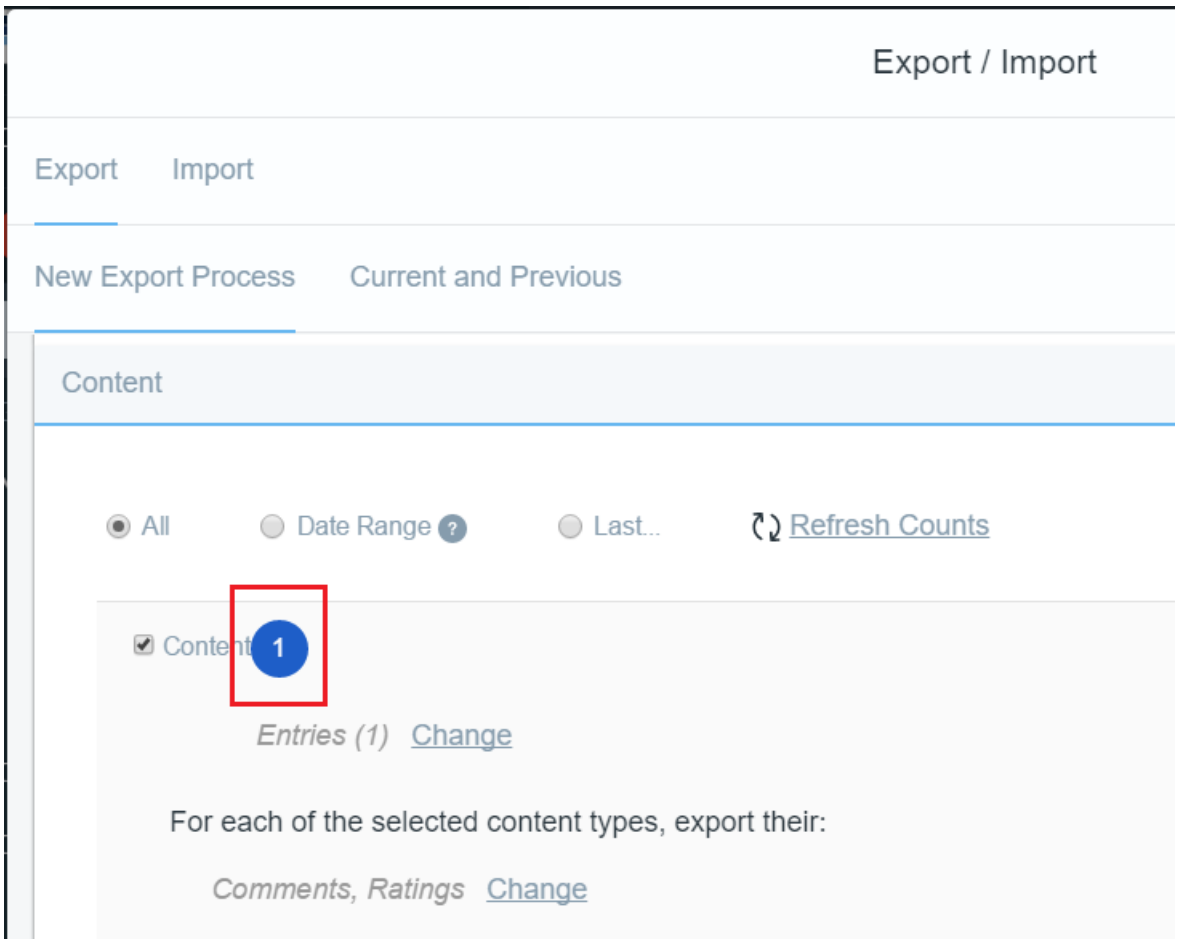

Figure 428.2: The number of modified Bookmarks entities are displayed in the Export UI.

```
@Override
public String getSchemaVersion() {
    return SCHEMA_VERSION;
}

@Override
public boolean validateSchemaVersion(String schemaVersion) {
    return _portletDataHandlerHelper.validateSchemaVersion(
        schemaVersion, getSchemaVersion());
}
```

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 Creating Staged Model Data Handlers for more information.

CHAPTER 429

CREATING STAGED MODEL DATA HANDLERS

In this tutorial, you'll create the `BookmarksStagedModelDataHandler` class used for the Bookmarks application. 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. This tutorial assumes you've already created staged models.

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.

```
@Component(immediate = true, service = StagedModelDataHandler.class)
```

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. 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;
}
```

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.

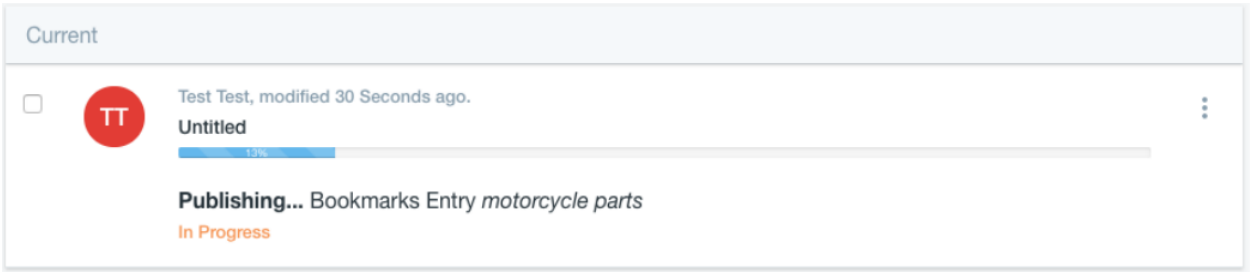

Figure 429.1: Your staged model data handler provides the display name in the Export/Import UI.

7. 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);
    }
```

8. Add the `doImportMissingReference` 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());
    }
```

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 Creating Portlet Data Handlers article to do this for the Bookmarks app.

CHAPTER 430

PROVIDING ENTITY-SPECIFIC LOCAL SERVICES FOR EXPORT/IMPORT

Data handlers must often call your app's local services to perform Export/Import-related tasks for its entities. When the Export/Import framework operates on entities (i.e., staged models), it often cannot manage important information from the entity's local services alone. The *Staged Model Repository* framework removes this barrier by linking an app's staged model to a local service. 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 local services don't expose.

The staged model repository does not avoid using your app's local services. It only provides an additional layer that provides Export/Import-specific functionality. Here's how this works:

- `*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 handle the remainder of the process.

Pretty cool, right? The first thing you must do is implement the `StagedModelRepository` interface. You'll explore this next.

430.1 Understanding the `StagedModelRepository` Interface

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.

The first important task is adding an `@Component` annotation section above the implementation class's declaration. This registers the class as a staged model repository in the OSGi service registry. There are a few annotation attributes you should set:

- `immediate`: activates the component immediately once its provided module has started.
- `property`: 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. This property must set the fully qualified model class name like this: `property = "model.class.name=FULLY_QUALIFIED_MODEL_CLASS"`.
- `service`: points to the `StagedModelRepository.class` interface.

Next, you must implement the `StagedModelRepository` interface. Implementations vary greatly based on the app you're implementing it for. There are two common cases you'll experience when implementing its methods:

- Methods that require additional Export/Import information injected before calling the local service.
- Methods that can call the local service directly.

The `BookmarksEntryStagedModelRepository.addStagedModel(...)` method is an example where only calling the local service would not satisfy the staged model data handler's needs (i.e., its UUID requirement). With the staged model repository layer, however, you can add export/import specific requirements on top of the present local services to serve your data handlers' needs.

The method does this:

- Sets the user ID and service context based on the `PortletDataContext` (used to populate the LAR file with your application's data during the export process).
- Sets the UUID, which is required to differentiate staged content between Sites.
- Calls the entity's local service.

Not every method implementation requires additional export/import information. For example, deleting Bookmarks Entries and deleting Bookmarks Entry staged models are functionally the same, so your staged model repository's method would call the local service directly (e.g., `BookmarksEntryStagedModelRepository.deleteStagedModel(...)`).

Next you'll learn about using a Staged Model Repository.

430.2 Using a Staged Model Repository

You can leverage a staged model repository by

1. Creating a getter and setter method to make a `StagedModelRepository` object available to your entity.

2. Calling the `StagedModelRepository` object to leverage its specialized export/import logic.

The getter and setter methods instantiate a `StagedModelRepository` object that the staged model data handler can use to access your entity's CRUD operations. The setter method should have an `@Reference` annotation listed above its method signature. This injects the component service of the `*StagedModelRepository` into the staged model repository object. The component service was created when you set the `@Component` annotation in the implementation class.

Once you have access to the `StagedModelRepository` object, call it to use its specialized export/import 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. The staged model repository abstracts these requirements away from the data handler.

Continue in the section to learn how to develop staged model repositories for your app.

CHAPTER 431

IMPLEMENTING THE STAGED MODEL REPOSITORY FRAMEWORK

In this article, you'll step through 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 articles for info on generating an app's services. You must have them to leverage most Export/Import and 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=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 set a couple method implementations to get a taste for how it works.

5. Implement the `addStagedModel(...)` method. The Bookmarks entry example 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 provides the UUID for the local service.

6. Not every method implementation requires additional staging information. Implementing the `deleteStagedModels` method calls the local service directly.

```
@Override
public void deleteStagedModels(PortletDataContext portletDataContext)
    throws PortalException {

    _bookmarksEntryLocalService.deleteEntries(
        portletDataContext.getScopeGroupId(),
        BookmarksFolderConstants.DEFAULT_PARENT_FOLDER_ID);
}
```

7. 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 article.

CHAPTER 432

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;
```

2. Call your `_stagedModelRepository` object to leverage its specialized export/import logic. For example,

```
newEntry = _stagedModelRepository.updateStagedModel(portletDataContext, importedEntry);
```

   Without the staged model repository logic, you would've called your local service like this:

```
serviceContext.setUuid(entry.getUuid());

newEntry = _bookmarksEntryLocalService.addEntry(
    userId, portletDataContext.getScopeGroupId(), folderId, entry.getName(), entry.getUrl(), entry.getDescription(), serviceContext);
```

The large number of parameters and UUID setter the local service method requires aren't needed when leveraging the staged model repository.

Great! You've successfully leveraged your staged model repository from a data handler!

CHAPTER 433

USING THE EXPORT/IMPORT LIFECYCLE LISTENER FRAMEWORK

In this tutorial, you'll learn how to use the `ExportImportLifecycleListener` framework to listen for processes/events during the staging and export/import lifecycles.

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 (event listener) or BaseProcessExportImportLifecycleListener (process listener). To choose, you'll need to consider what parts of a lifecycle you want to listen for (event or process).
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. As an example, you'll step through the LoggerExportImportLifecycleListener. This listener extends the `BaseExportImportLifecycleListener`, so you immediately know that it deals with lifecycle events.
6. Add the `getStagedModelLogFragment(...)` method:

   ```
   protected String getStagedModelLogFragment(StagedModel stagedModel) {
       StringBundler sb = new StringBundler(8);
   ```

```
    sb.append(StringPool.OPEN_CURLY_BRACE);
    sb.append("class: ");
    sb.append(ExportImportClassedModelUtil.getClassName(stagedModel));

    if (stagedModel instanceof StagedGroupedModel) {
        StagedGroupedModel stagedGroupedModel =
            (StagedGroupedModel)stagedModel;

        sb.append(", groupId: ");
        sb.append(stagedGroupedModel.getGroupId());
    }

    sb.append(", uuid: ");
    sb.append(stagedModel.getUuid());
    sb.append(StringPool.CLOSE_CURLY_BRACE);

    return sb.toString();
}
```

This retrieves the staged model's log fragment, which is the lifecycle listener's logging information on events.

7. Add the `isParallel()` method:

```
@Override
public boolean isParallel() {
    return false;
}
```

This 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.

8. Add the `onExportImportLifecycleEvent(...)` method:

```
@Override
public void onExportImportLifecycleEvent(
        ExportImportLifecycleEvent exportImportLifecycleEvent)
    throws Exception {

    if (!_log.isDebugEnabled()) {
        return;
    }

    super.onExportImportLifecycleEvent(exportImportLifecycleEvent);
}
```

This consumes the lifecycle event and passes it through the base class's method (as long as Debug mode is not enabled).

9. Each remaining method is called to print logging information for the user. For example, when a layout export fails, logging information directly related to that event is printed:

```
@Override
protected void onLayoutExportFailed(
        PortletDataContext portletDataContext, Throwable throwable)
    throws Exception {

    if (!_log.isDebugEnabled()) {
```

```
            return;
        }

        _log.debug(
            "Layout export failed for group " + portletDataContext.getGroupId(),
            throwable);
    }
```

In summary, the `LoggerExportImportLifecycleListener` uses the lifecycle listener framework to print messages to the log when an export/import event occurs. You can view the other logging methods implemented for this class here.

10. 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.

Terrific! You learned about the Export/Import Lifecycle Listener framework, and you've learned how to create your own listener for events/processes that occur during export/import of your portal's content.

CHAPTER 434

INITIATING NEW EXPORT/IMPORT PROCESSES

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 ExportImportConfigurationSettingsMapFactory class to create a layout export settings map:

```
Map<String, Serializable> exportLayoutSettingsMap =
    ExportImportConfigurationSettingsMapFactory.
        buildExportLayoutSettingsMap(...);
```

2. Create the ExportImportConfiguration object by using an *add* method in the entity's local service. The map created previously is used as a parameter to create the ExportImportConfiguration object.

```
ExportImportConfiguration exportImportConfiguration =
    exportImportConfigurationLocalService.
        addDraftExportImportConfiguration(
            user.getUserId(),
            ExportImportConfigurationConstants.TYPE_EXPORT_LAYOUT,
            exportLayoutSettingsMap);
```

   The ExportImportConfigurationLocalService provides several useful methods to create and modify your custom ExportImportConfiguration.

3. Call the appropriate service using your newly created ExportImportConfiguration object to initiate an export/import or staging process. For example,

```
files[0] = exportImportLocalService.exportLayoutsAsFile(
    exportImportConfiguration);
```

   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.

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 (outlined here). 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 435

STAGING

Staging lets users change a Site without affecting the live Site and then publish all the changes in one fell swoop. If you include staging support in your application, your users can stage its content until it's ready.

For example, if your application uses the Staging framework and provides information intended only during a specific holiday, users can save your application's assets specific for that holiday. They reside in the Staging environment until they're ready for publishing.

Staging and Export/Import share the same base 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. This means that implementing Staging in your app is *almost* the same as implementing the Export/Import framework. You can visit the Export/Import framework's articles for the base APIs that both it and the Staging frameworks share.

If your app supports Export/Import, its entities (staged models) are automatically tracked by Staging with the use of data handlers. There are some Staging-specific configurations you can add that are not shared by Export/Import. Some Staging-specific actions you can complete include

- Control Staging UI settings
- Filter Staging-specific processes
- Check for Staging-specific states

You'll learn about these next.

435.1 Controlling Staging's UI Settings

You can control most of Staging's UI from your portlet data handler. This can be done several ways; first, you can configure predefined setter methods in the portlet data handler's `activate()` method:

- `setStagingControls`: adds fine grained controls over staging behavior that is rendered in the Staging UI. For example, this enables your app's checkboxes in the Content section of the Publication screen. This is usually set like this: `setStagingControls(getExportControls());`. The staging UI typically provides the same content as the export UI (i.e., the Content section for selecting what to publish/export), so it leverages its UI. You can set the Staging UI differently by

configuring the `setStagingControls` method differently. See the `AssetTagsPortletDataHandler` class for an example of not copying the Export UI for the Staging UI.
- `setDataAlwaysStaged`: defines whether you can enable/disable your app's content staging (i.e., selectable from the Publication screen). For example, setting this method to `true` automatically stages your app's content. Users can no longer choose whether its content should be staged.

Other setter methods are available that control both Export/Import and Staging settings. You can reference them by visiting the Understanding the `PortletDataHandler` Interface section.

You can also control whether your app is enabled on the Staged Content screen by adding this method to your portlet data handler:

```
@Override
public boolean isConfigurationEnabled() {
    return false;
}
```

When this is set to `false`, your app is disabled on the Staged Content screen. This is set to `true` by default.

The majority of Staging-specific configurations are completed in a portlet data handler. The staged model data handler does come into play when you want to filter for certain staging processes/states. You'll learn about this next.

435.2 Filtering Staging-Specific Processes and States

You can filter for certain staging-specific processes/states and complete actions based on the returned status. You can do this by leveraging the following classes from a staged model data handler:

- `ExportImportThreadLocal`
- `StagingGroupHelper`

The `ExportImportThreadLocal` class provides boolean methods that return whether a specific process is in progress. Use this to check for events affecting the entire site. For example, you can check if the following processes are in progress:

- Local Staging
- Remote Staging
- Layout Validation
- Portlet Staging
- etc.

The `StagingGroupHelper` interface provides utility methods that return the staging state in your app. This is intended to check for events only affecting your app. For example, you can check if your app is in these states:

- Resides in Local Staging group
- Resides in Remote Live group
- Is a staged portlet

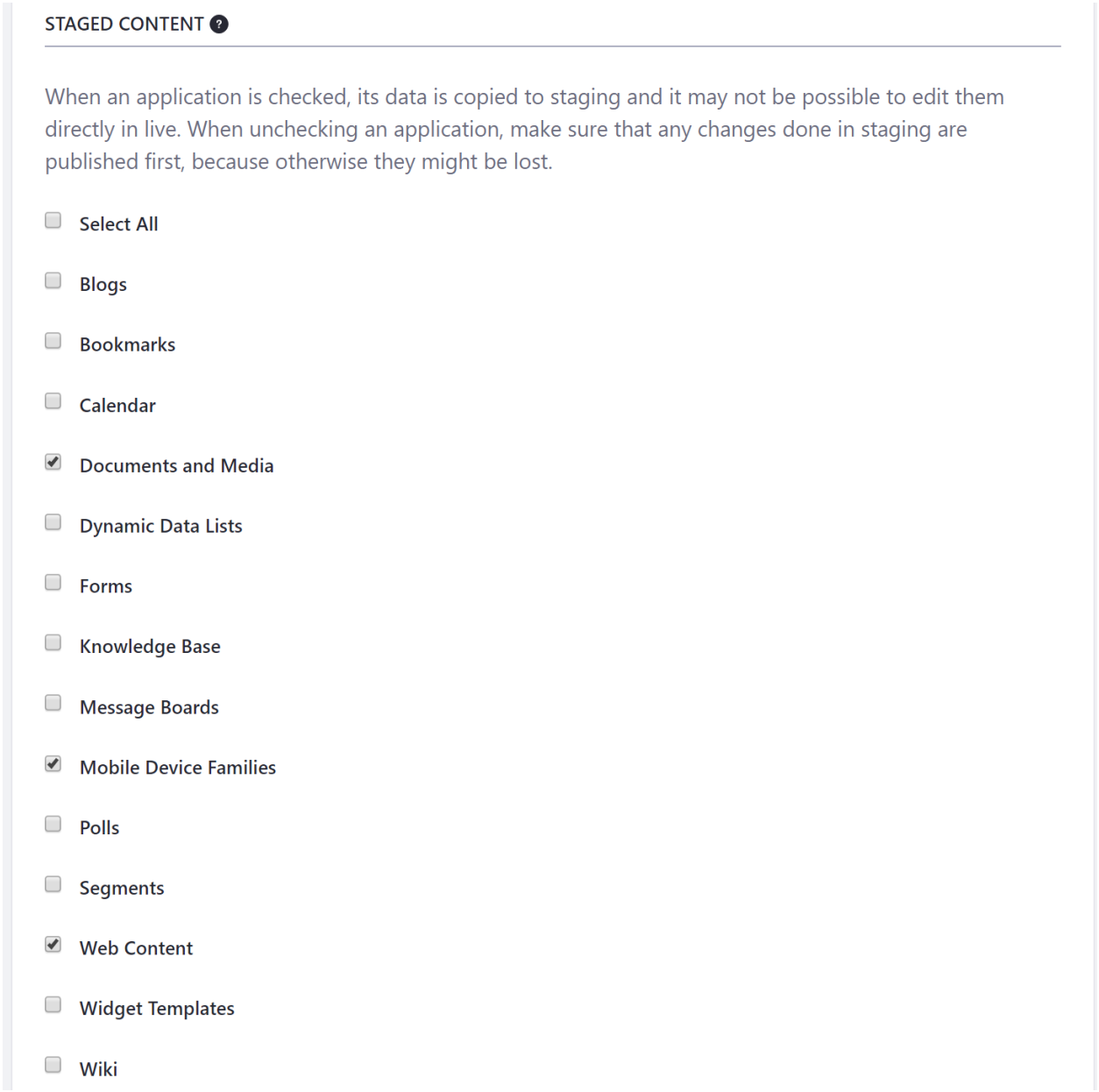

Figure 435.1: There are many apps available to select from the Staged Content screen.

- etc.

A real example filtering for a staging process and state can be found in the `AssetListEntryStagedModelDataHandler` class:

```
if ((assetRendererFactory != null) &&
    ExportImportThreadLocal.isStagingInProcess() &&
    !_stagingGroupHelper.isStagedPortlet(
        assetEntry.getGroupId(),
        assetRendererFactory.getPortletId())) {

    continue;
}
```

The staged model data handler uses the `ExportImportThreadLocal.isStagingInProcess()` method to verify that a staging process is running. It also checks whether the app is staged by executing `!_stagingGroupHelper.isStagedPortlet(...)`.

Excellent! You can now filter for staging-specific processes and states.

CHAPTER 436

DEPENDENCY INJECTION

When you're using a object based on its interface, you don't have to concern yourself with the implementation because it's abstracted from you. At runtime, the implementation used depends on your environment and your app's configuration. Liferay DXP offers several standard ways to register implementations and inject them into your applications.

Contexts and Dependency Injection (CDI): Is the Java EE standard dependency injection mechanism. Liferay DXP's CDI bean container makes an application's concrete classes available as beans. Bean classes can user other beans by way of injecting them into their fields that have the `@Inject` annotation.

OSGi Declarative Services: Liferay DXP's OSGi runtime framework allows components to register as service provides and other components can call on the registry to bind the services to their fields that have the `@Reference` annotation. Liferay DXP's services and services you create using Service Builder are available as OSGi Declarative Services.

Spring DI: The Spring framework includes inversion of control (IoC) and dependency injection. It's available to applications that configure the Spring framework.

As an added bonus, Liferay DXP provides OSGi CDI integration. It lets you publish CDI beans as OSGi services and consume OSGi services in your CDI beans. Which dependency injection mechanism will you use? Read on to learn more about them.

CHAPTER 437

CDI Dependency Injection

Portlet 3.0 (see JSR 362) supports Contexts and Dependency Injection (CDI) so you can create and use injectable classes (CDI beans) in your portlet. It also provides injectable portlet artifacts called Portlet Predefined Beans. They give a portlet's CDI beans access to the portlet configuration, preferences, requests, responses, and more. Here's how to create and use CDI beans and Portlet Predefined Beans:

1. Create a portlet WAR project, if you haven't created one already.

    - Any project that has a class that implements the `javax.portlet.Portlet` interface, either directly or indirectly.

```
**Note:** If you're developing a portlet JAR, such as a
[Liferay MVC Portlet](/docs/7-2/appdev/-/knowledge_base/a/liferay-mvc-portlet),
use CDI via
[OSGi CDI Integration](/docs/7-2/frameworks/-/knowledge_base/f/osgi-cdi-integration).
```

```
**Note:**
Liferay DXP exports the packages provided by the Portlet API and CDI API.
Liferay project templates typically include them as transitive
dependencies. If you must explicitly depend on the portlet API and CDI
artifacts, add them as `compileOnly` (Gradle) or `provided` (Maven)
dependencies.
```

---

2. If your portlet WAR project isn't a Bean Portlet, add this `src/main/webapp/WEB-INF/beans.xml` file to it. This file tells CDI to scan the project for CDI annotations.

    ```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://xmlns.jcp.org/xml/ns/javaee" bean-discovery-mode="all" version="1.2"
           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/beans_1_1.xsd">
        <!-- This file is necessary in order to inform CDI that scanning should occur for CDI annotations. -->
    </beans>
    ```

3. Add the `@ApplicationScoped` annotation to your portlet class.

```
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class MyPortlet ... {
    ...
}
```

4. Make sure all concrete classes you want to make injectable have the default constructor. These classes are now CDI beans.
5. Add a scope to each CDI bean.

```
Bean Scope              | Description      |
----------------------- | ---------------- |
[`@ApplicationScoped`](https://docs.oracle.com/javaee/7/api/javax/enterprise/context/ApplicationScoped.html) | Shares the bean's state across all us
[`@Dependent`](https://docs.oracle.com/javaee/7/api/javax/enterprise/context/Dependent.html) | (default scope) Designates the bean to be for the cli
[`@PortletRequestScoped`](https://docs.liferay.com/portlet-api/3.0/javadocs/javax/portlet/annotations/PortletRequestScoped.html) | Associates the be
[`@PortletSessionScoped`](https://docs.liferay.com/portlet-api/3.0/javadocs/javax/portlet/annotations/PortletSessionScoped.html) | Places the bean i
[`@RenderStateScoped`](https://docs.liferay.com/portlet-api/3.0/javadocs/javax/portlet/annotations/RenderStateScoped.html) | Stores the bean as part
```

---

6. Use the JSR 330 `@Inject` annotation in a CDI bean to inject another CDI bean into it. For example, this code informs Liferay DXP's CDI bean container to inject a `GuestBook` CDI bean into this `guestbook` field.

```
@Inject
private GuestBook guestbook;
```

7. Inject any Portlet Predefined Beans (portlet request scoped or dependent scoped beans) into your `@PortletRequestScoped` CDI beans.

```
@PortletRequestScoped
public class RequestProcessor ... {

    @Inject
    private PortletRequest portletRequest;
    ...
}
```

8. Inject any dependent scoped Portlet Predefined Beans into your `ApplicationScoped` or `@Dependent` scoped CDI beans. For example,

```
@ApplicationScoped
public class MyPortlet ... {

    @Inject
    private PortletConfig portletConfig;
    ...
}
```

9. Use bean EL names to reference any portlet redefined *named beans* in your JSP or JSF pages.
10. Deploy your project.

Congratulations! You have created and used CDI beans and Portlet Predefined Beans in your portlet.

437.1 Related Topics

CDI Portlet Predefined Beans
    Portlets

CHAPTER 438

OSGi CDI Integration

Liferay DXP's runtime environment consists of services (OSGi services). The OSGi service registry and Service Component Runtime (SCR) facilitate providing and consuming services. Contexts and Dependency Injection (CDI) is a Java SE and EE standard for lifecycle events, stateful objects, and dependency injection. OSGi CDI Integration brings features and capabilities of CDI to OSGi and makes OSGi services available to CDI beans. Here you'll learn how to

- Publish CDI beans as OSGi services: Register CDI beans as services you can use to customize Liferay DXP components.
- Use OSGi services in beans: Leverage any OSGi service published on Liferay DXP in any bean.

The following use cases provide more detail.

438.1 Use Case: Registering a CDI bean as an OSGi service

Liferay DXP extension points are implemented as OSGi services. If there's a piece of functionality you must customize, you don't have to learn OSGi to do it: you can write your extension/override as a CDI bean instead and use OSGi CDI integration to publish your bean as an OSGi service.

By implementing the service in your CDI bean class and adding the integration's `@org.osgi.service.cdi.annotations.Service` annotation to it, your bean registers as providing that OSGi service. In this way, service consumers can use your service implementation (i.e., your CDI bean).

For example, the Service Registry in figure 1 shows two implementations of an OSGi service called `S1`:

- `MyBean` is a CDI bean whose service rank is `1000`.
- `MySvcImpl` has a service rank of `0`.

The Service Component Runtime (`SCR`) finds the matching, highest ranked `S1` service provider and binds it to consumer `C1`. The fact that `MyBean` is a CDI bean is transparent to the SCR.

Once a CDI bean is registered as a service, components can use it as they would any other OSGi service.

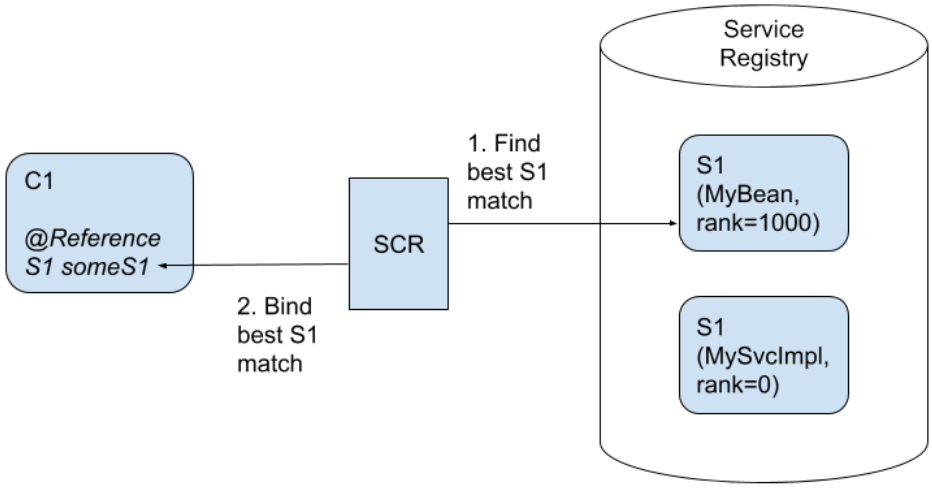

Figure 438.1: OSGi Service Component Runtime (SCR) finds `MyBean` as the best (highest ranked) `S1` service provider and binds it to consumer component `C1`.

438.2 Use Case: Using an OSGi service in a bean

Liferay DXP contains many development frameworks for all of its constructs, such as Users, Sites, Documents, Comments, and the APIs for these assets are all implemented as OSGi services. When developers write new applications using Liferay's development frameworks, new assets become available and integrated with the rest of the system. OSGi CDI integration enables your beans to access these OSGi services.

In figure 2, for example, CDI bean `SomeBean` uses the OSGi CDI integration annotation `@org.osgi.service.cdi.annotations.Reference` (along with CDI annotation `@Inject`) to inject the OSGi service `UserLocalService`. The Service Component Runtime (`SCR`) finds the `UserLocalService` in the Service Registry and binds it to `SomeBean`'s field `userSvc`.

These are the most common use cases, but you might have more. Now you can get started using OSGi CDI integration to publish CDI beans as OSGi services!

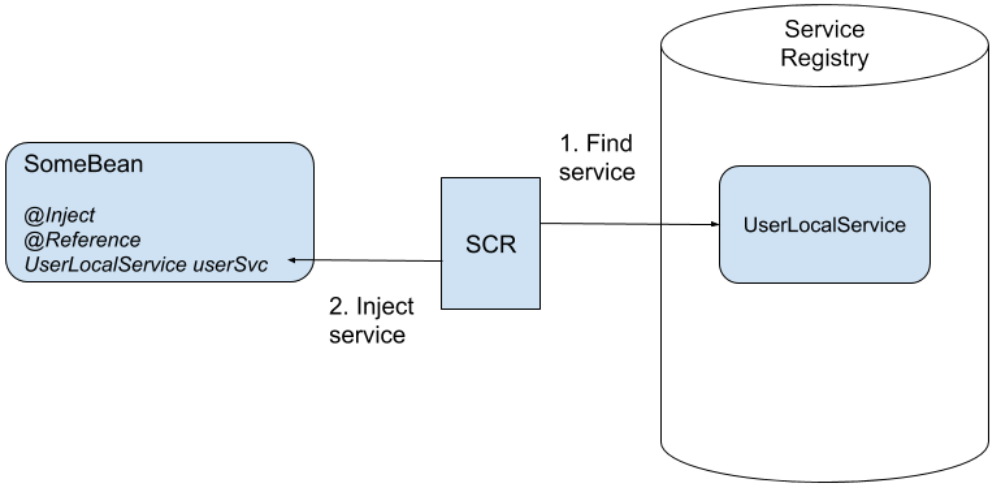

Figure 438.2: Here how Liferay's `UserLocalService` is injected into a bean.

CHAPTER 439

PUBLISHING CDI BEANS AS OSGI SERVICES

You can publish CDI beans as OSGi services, making them accessible via the Liferay's OSGi service registry. Here's how:

1. Add a project dependency on the OSGi CDI Integration artifact. For example, here's the dependency to use in a Maven `pom.xml` file:

```
<dependency>
    <groupId>org.osgi</groupId>
    <artifactId>org.osgi.service.cdi</artifactId>
    <version>1.0.0</version>
</dependency>
```

2. Make your CDI bean implement the service interface you're providing. For example, `ShopImpl` provides the `Shop` service by implementing that interface.

```
package my.package;

public class ShopImpl implements Shop {
    ...
}
```

3. Annotate your CDI bean class with `@org.osgi.service.cdi.annotations.Service`.

```
package my.package;

import org.osgi.service.cdi.annotations.Service;

@Service
public class ShopImpl implements Shop {
    ...
}
```

4. Deploy the API that defines the service interface, if you haven't deployed it already.

5. Build and deploy your service project bundle.

Once your bundle installs and activates, your bean's service implementation is available. You can use Gogo Shell commands can verify that the service registered.

For example, here are steps for verifying that a bundle `com.liferay.portal.samples.cdi.jar.portlet` registers a service called `org.apache.portals.samples.Users`.

1. Navigate to *Control Panel → Configuration → Gogo Shell*.
2. Use the `lb` Gogo command and `grep` (pass in the bundle's symbolic name) to find your bundle (and its ID).

   Example command:

   ```
   g!: lb | grep com.liferay.portal.samples.cdi.jar.portlet
   ```

   Results:

   ```
   924|Active     |   10|com.liferay.portal.samples.cdi.jar.portlet (0.0.1.201901252134)|0.0.1.201901252134
   ```

   The first column contains the bundle ID.
3. Use the `b` Gogo command with your bundle ID to list your bundle's details and verify the bundle includes your service as one of its registered services.

   Example command:

   ```
   g!: b 924
   ```

   Results:

   ```
   com.liferay.portal.samples.cdi.jar.portlet_0.0.1.201901252134 [924]
   Id=924, Status=ACTIVE      Data Root=C:\git\bundles\osgi\state\org.eclipse.osgi\924\data
   "Registered Services"
     ...
     {org.apache.portals.samples.Users}={osgi.command.scope=cdiportlet, service.id=4232, service.bundleid=924, service.scope=singleton, osgi.com
     ...
   ```

Congratulations on publishing your CDI bean as an OSGi service!

CHAPTER 440

USING OSGI SERVICES IN A BEAN

Any bean can use the `@org.osgi.service.cdi.annotations.Reference` annotation to inject OSGi services. It's the easiest way for a bean to access an OSGi service. Here's how:

1. Add a project dependency on the OSGi CDI Integration artifact. For example, here's the dependency to use in a Maven `pom.xml` file:

```
<dependency>
    <groupId>org.osgi</groupId>
    <artifactId>org.osgi.service.cdi</artifactId>
    <version>1.0.0</version>
</dependency>
```

2. Obtain and inject the OSGi service by using the `@org.osgi.service.cdi.annotations.Reference` and `@javax.inject.Inject` annotations respectively. Here's an example of injecting a service of type `ProductStore`.

```
import javax.inject.Inject;

import org.osgi.service.cdi.annotations.Reference;

import package.path.ProductStore;

public class MyBean {

    @Inject
    @Reference
    ProductStore productStore;

    ...
}
```

3. Deploy your bean project to Liferay DXP.

Congratulations on injecting an OSGi service into your bean! Now your bean uses the OSGi service you injected.

CHAPTER 441

DECLARATIVE SERVICES

Liferay DXP's 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 the annotation's `service` attribute to a particular service type allows other components to reference the service component by that 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's field `SomeApi _someApi`.

Note: The `@Reference` annotation can only be used in a class that is annotated with `@Component` (i.e, a Declarative Services component) or a bean class that uses OSGi CDI integration.

Note: 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.

CHAPTER 442

SERVICE TRACKERS FOR OSGI SERVICES

In an OSGi runtime ecosystem, you must consider how your apps can rely on OSGi services in other modules for functionality. It's possible for service implementations to be swapped out or removed entirely, and your app must not just survive but thrive in this environment.

If you want to call OSGi services from an OSGi Declarative Services `@Component` classes, it's easy: you just use a Declarative Services (DS) annotation, `@Reference`, to inject the service. 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 want to call an OSGi service from a bean, use OSGi CDI integration.

DS `@Reference` with `@Component`s and OSGi CDI integration with beans manage much of the complexity of service dynamism for you transparently. If you can use either of them, you should. Otherwise, read implement a Service Tracker to look up services in the service registry.

CHAPTER 443

USING A SERVICE TRACKER

Your non-OSGi and non-bean classes can access any service registered in the OSGi runtime using a Service Tracker. It lets you access any OSGi services including your own Service Builder services and the services published by Liferay's modules (like the popular `UserLocalService`).

You can create a service tracker in two ways:

1. Create a service tracker where you need it.
2. Create a class that extends `org.osgi.util.tracker.ServiceTracker`.
3. Create a service tracker that tracks service events using a callback handler.

Both ways depend on `org.osgi.core`, whose packages Liferay DXP exports by default. Configure it as `compileOnly` (Gradle) or `provided` (Maven). See the Third Party Packages Portal Exports for more information.

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 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.

443.1 Creating a New Service Tracker Where You Need It

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);
```

443.2 Create a Class That Extends ServiceTracker

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. Construct a new instance of your service tracker where you need it. 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. For example, check whether the service is `null`:

```
SomeService someService = someServiceTracker.getService();

if (someService == null) {
    _log.warn("The required service 'SomeService' is not available.");
}
else {
    someService.doSomethingCool();
}
```

Note, 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();
```

If you need to track multiple services or their events, implement a service tracker that uses callback handlers.

443.3 Creating a Service Tracker that Tracks Service Events Using a Callback Handler

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);
```

Using service trackers requires producing some boilerplate code, 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.

CHAPTER 444

FRIENDLY URLS

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.

CHAPTER 445

FRIENDLY URLS

Follow these steps to create friendly URLs:

1. Create friendly URL routes. 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 7.2.0//EN" "http://www.liferay.com/dtd/liferay-friendly-url-routes_7_2_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:

```xml
<pattern>/{userIdAndInstanceId}/render-it</pattern>
```

or

```xml
<pattern>/{instanceId}/render-it</pattern>
```

or

```xml
<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`).

<!--Add link back for 'using an `MVCPortlet` with `MVCRenderCommand` classes'
once mvc-render-command article is available-->

```java
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS, "mvc.command.name=/",
        "mvc.command.name=/blogs/view"
    },
    service = MVCRenderCommand.class
)
```

The [DTD file](https://docs.liferay.com/dxp/portal/7.2-latest/definitions/liferay-friendly-url-routes_7_2_0.dtd.html)
completely defines the `routes.xml` file.
```

3. Provide an implementation of the `FriendlyURLMapper` service. Create a component that specifies a `FriendlyURLMapper` service, with two properties:

    - A `com.liferay.portlet.friendly-url-routes` property sets the path to your `routes.xml` file.
    - 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
)
```

4. 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`). 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.

Great! Now you know how to make your URLS friendly.

445.1 Related Topics

- Dependency Injection
- Localization
- Asset Framework

CHAPTER 446

FRONT-END DEVELOPMENT

You have complete front-end development freedom. You can use Liferay DXP's front-end frameworks, along with the front-end technologies you love the most:

- EcmaScript ES2015+
- React, Angular, Vue, etc.
- Metal.js (developed by Liferay)
- AlloyUI (developed by Liferay)
- jQuery (included)
- Lodash (included, but disabled by default)

To load modules, you must know when they are needed, where they are at build time, whether they should be bundled together or loaded independently, and you must assemble them at runtime. Liferay's Loaders (YUI/AUI, AMD, and npm in AMD format) handle loading for you. All you must do is provide a small bit of information about your module.

The Liferay JS Bundle Toolkit(the JS Portlet Extender, Liferay Bundle Generator, and `liferay-npm-bundler`) has the tools you need to create and develop JavaScript portlets with pure JavaScript tooling. You can use the `liferay-npm-bundler` to bundle npm packages in your applications. It even has 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, you may need to access Liferay DXP-specific information or web services. The `Liferay` global JavaScript Object exposes this information for you, to use in your JavaScript applications.

446.1 Lexicon and Clay

Liferay 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.

446.2 Templates

For templating, you can use Java EE's JSP, FreeMarker, or whatever else you like.

446.3 Themes

Themes use the standard components (CSS, JavaScript, and HTML) along with FreeMarker templates. Although the default themes are nice, you may wish to create your own look and feel for your site. The Liferay JS Theme Toolkit has all the tools you need to create and develop themes, but you can use the tools you prefer.

From the Theme Builder Gradle Plugin, to the Liferay Theme Generator, 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.

446.4 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
- Dynamic Includes

See Theme Components and Understanding the Page Structure for more information.

CHAPTER 447

THEMES

Themes customize the default look and feel of your site. You can inject your own flavor and personality and represent the visual identity of your brand or company.

You'll learn these things:

- Developing Themes: Learn how to use Liferay DXP's tools and features to develop your theme.
- Extending Themes: Learn how to use Liferay DXP's theme extension mechanisms and features to add to your theme.

Themes use the standard components (CSS, JS, and HTML) along with FreeMarker templates for rendering. There are several default FreeMarker templates that each handle a key piece of functionality for the page. There are also theme template utilities that let you use portlets, taglibs, theme objects, and more in your theme templates. CSS extensions and patterns come out-of-the-box, and support SASS, and multiple JavaScript frameworks. Several mechanisms are available for customizing, developing, and extending themes.

CHAPTER 448

THEME WORKFLOW

Themes are built on top of one of the following base themes:

- **Unstyled:** provides basic markup, functions, and images
- **Styled:** inherits from the Unstyled base theme and adds some styling on top

Themes can be built with your choice of tooling. Liferay also offers its own set of tools to get your themes up and running quickly.

The following Liferay tools help you build themes:

- Theme Builder Gradle Plugin
- Liferay Theme Generator
- Blade CLI's Theme Template.

Depending on the tool you choose (Theme Generator, Gradle, Blade CLI, Maven, or Dev Studio), the theme anatomy can be different. The overall development process is the same:

1. Mirror the structure of the files you want to modify. Most of the time, you'll modify these files:

    - `portal_normal.ftl`: main theme markup
    - `_custom.scss`: custom CSS styling
    - `main.js`: the theme's JavaScript

2. Build and deploy the theme.
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, you should enable Developer Mode. This disables the JavaScript minifier and caching for CSS and FreeMarker template files, which makes debugging easier.

If you've built your theme with the Liferay Theme Generator, you can use some helpful Gulp tasks to streamline the process:

- **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.

See Theme Components and Understanding the Page Structure to get a top-level overview of how themes work.

CHAPTER 449

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 covers the available actions that these tasks provide, as well as other information you may find useful while developing your theme.

This section covers these topics:

- Using liferay theme tasks (build, deploy, extend, init, kickstart, status, and watch)
- Using Developer Mode
- Creating thumbnail previews for your theme
- Creating color schemes for your theme
- Making configurable theme settings

While developing your theme, you may notice that your theme's CSS and JS files are minified. This optimizes performance, but it can make debugging difficult during development. Developer Mode, disabled by default, optimizes development instead. For instance, it loads CSS and JS files individually for easier debugging. Also, you don't have to reboot the server as often in Developer Mode. 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. See Using Developer Mode with Themes to learn how to enable Developer Mode in your app server. You can also use the Gulp Watch task to test theme changes on a proxy port before deploying your theme to your server.

CHAPTER 450

USING DEVELOPER MODE WITH THEMES

This article shows how to enable Developer Mode in your app server manually and through Dev Studio DXP, as well as how to configure other settings that may benefit you during development. Each topic is explained in the relevant section below.

450.1 Enabling Developer Mode Manually

Follow these steps to enabled Developer Mode in your app server manually:

1. Create a `portal-ext.properties` file in your server's root folder if it doesn't exist.
2. Add the line below to it:

   ```
   include-and-override=portal-developer.properties
   ```

   Alternatively, add the properties from the portal-developer.properties file to your `portal-ext.properties` file that you want to use.
3. Start your app server to apply the changes.

Read the next section to learn how to enable Developer Mode in Dev Studio DXP.

450.2 Setting Developer Mode in Dev Studio DXP

Follow these steps to enable Developer Mode for your app server in Dev Studio DXP:

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.

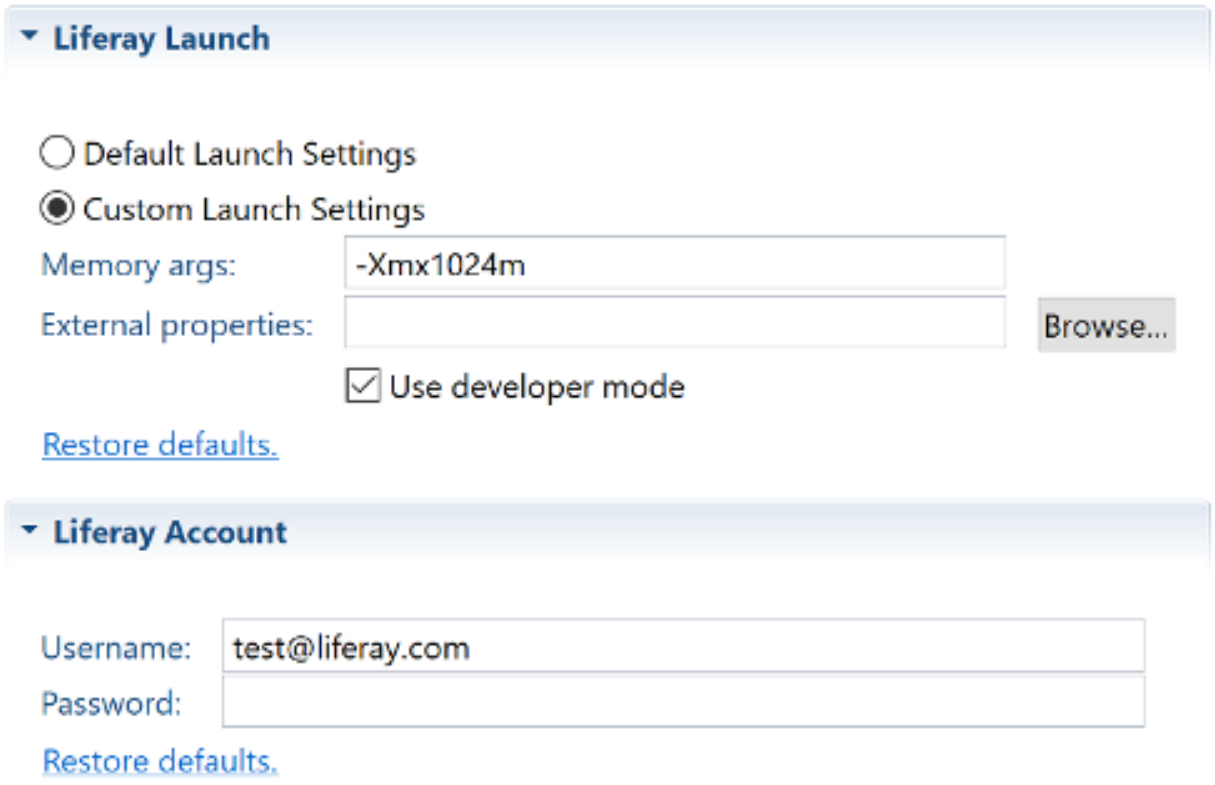

Figure 450.1: The *Use developer mode* option lets you enable Developer Mode for your server in Dev Studio DXP.

Warning: Only change the Server settings from the runtime environment's Liferay Launch section.

450.3 Configuring FreeMarker System Settings

By default, FreeMarker theme templates and web content templates are cached. You can change this behavior through System Settings with the steps below:

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 to improve JavaScript file loading for development.

450.4 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!

450.5 Related Topics

- Generating Layout Templates with the Theme Generator

CHAPTER 451

BUILDING YOUR THEME'S FILES

Follow these steps to build your theme's files with the Build task. Note that this task only works for themes that use the liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator.

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 `gulp build`.
2. The `gulp build` task generates the base theme files (in the `build` folder), compiles Sass into CSS, and compresses all theme files into a `.war` file (in the `dist` folder), that you can deploy to your server. Copy any of these files and folders to your theme's `src` folder to modify them.
3. Deploy the `war` file to your app server to make it available.

```
C:\Users\liferay\Desktop\projects\themes\7-2-themes\my-new72theme-theme>gulp build
[18:28:21] Using gulpfile ~\Desktop\projects\themes\7-2-themes\my-new72theme-theme\gulpfile.js
[18:28:21] Starting 'build'...
[18:28:21] Starting 'build:clean'...
```

Figure 451.1: Run the `gulp build` task to build your theme's files.

451.1 Related Topics

- Automatically Deploying Theme Changes
- Copying an Existing Theme's Files
- Deploying and Applying Themes

CHAPTER 452

DEPLOYING AND APPLYING THEMES

Follow these steps to deploy your theme with the Deploy task. Note that this task only works for themes that use the liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator.

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`.

```
**Note:** If you're running the
[Felix Gogo shell](/docs/7-2/customization/-/knowledge_base/c/using-the-felix-gogo-shell),
you can also deploy your theme using the `gulp deploy:gogo` command.
```

---

2. Your server's log displays that the OSGi bundle is started.

```
2019-04-08 21:15:59.660 INFO  [Refresh Thread: Equinox Container: 05b65d8b-6326-4127-9ffb-d6bea7e
16ac3][ThemeHotDeployListener:108] 1 theme for my-new72theme-theme is available for use
2019-04-08 21:15:59.736 INFO  [Refresh Thread: Equinox Container: 05b65d8b-6326-4127-9ffb-d6bea7e
16ac3][BundleStartStopLogger:39] STARTED my-new72theme-theme_1.0.0 [2120]
```

Figure 452.1: Your server's log notifies you when the theme's bundle has started.

3. Apply your theme through the *Build → 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.

Wonderful! Your theme is deployed to your server and applied to your site.

452.1 Related Topics

- Automatically Deploying Theme Changes
- Copying an Existing Theme's Files
- Creating Themelets with the Theme Generator

CHAPTER 453

UPDATING YOUR THEME'S APP SERVER

Follow these steps to update the configuration for your theme's app server with the Init task. Note that this task only works for themes that use the liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator.

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 `gulp init`.

```
C:\Users\liferay\Desktop\projects\themes\7-2-themes\my-new72theme-theme>gulp init
[12:31:17] Using gulpfile ~\Desktop\projects\themes\7-2-themes\my-new72theme-theme\gulpfile.js
[12:31:17] Starting 'plugin:init'...
? Select your deployment strategy Local App Server
? Enter the path to your app server directory: C:\Users\liferay\opt\Liferay\bundles\liferay-ce-portal-tomcat-7.2.0-b1-2
01903291136\liferay-ce-portal-7.2.0-b1\tomcat-9.0.10
? Enter the url to your production or development site: http://localhost:8080
[12:31:44] Finished 'plugin:init' after 27 s
[12:31:44] Starting 'init'...
[12:31:44] Finished 'init' after 21 µs
```

Figure 453.1: Run the `gulp init` task to update your app server configuration.

2. Enter the updated path to your app server and site.
3. Your theme's `liferay-theme.json` file contains the updated server configuration information:

```
{
  "LiferayTheme": {
    "deploymentStrategy": "LocalAppServer",
    "appServerPath": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\liferay-ce-portal-tomcat-7.2.0\\liferay-ce-portal-
7.2.0\\tomcat-9.0.10",
    "deployPath": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\liferay-ce-portal-tomcat-7.2.0\\liferay-ce-portal-
7.2.0\\deploy",
    "url": "http://localhost:8080",
    "appServerPathPlugin": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\liferay-ce-portal-tomcat-7.2.0\\liferay-ce-portal-
7.2.0\\tomcat-9.0.10\\webapps\\my-new72theme-theme",
```

```
    "deployed": true,
    "pluginName": "my-new72theme-theme"
  }
}
```

Awesome! Now you can deploy your theme to the proper server.

453.1 Related Topics

- Automatically Deploying Theme Changes
- Changing Your Base Theme
- Listing Your Theme's Extensions

CHAPTER 454

AUTOMATICALLY DEPLOYING THEME CHANGES

Follow these steps to automatically preview your theme's changes with the Watch task. Note that this task only works for themes that use the liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator.

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 `gulp watch`. This sets up a proxy for your app server and opens it in a new window in the browser.

```
 **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)
```

2. Make a change to your theme and save the file. The updated files are built, compiled, and copied directly to the proxy port (e.g. `9080`). CSS changes are deployed live, so no page reload is needed.
3. Once you're happy with the changes, re-deploy your theme to apply the changes to your site on your app server.

454.1 Related Topics

- Configuring Your Theme's App Server
- Copying an Existing Theme's Files
- Deploying and Applying Themes

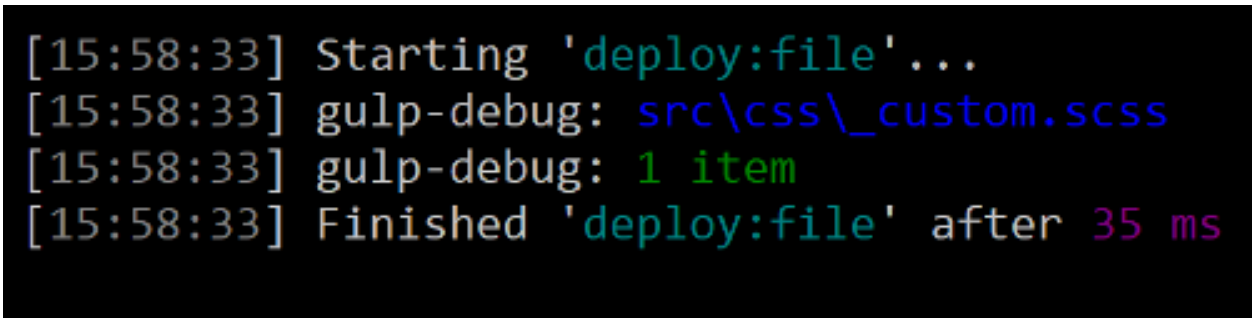

Figure 454.1: The watch task notifies you that the changes are deployed.

CHAPTER 455

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. This is even more important when developing color schemes for a theme, since names are not displayed for color schemes.

This article shows how to create a thumbnail preview for your theme so users can identify it.

Your first step in creating a thumbnail preview for your theme is taking a screenshot of your theme. Once you have a screenshot that you like, follow the steps below to create a thumbnail preview for your theme:

1. Resize the screenshot to 270 pixels high by 480 pixels wide. Your thumbnail *must be* these exact dimensions to display properly.

2. 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).

```
**Note:** The
[Theme Builder Gradle plugin](/docs/7-2/reference/-/knowledge_base/r/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.
```

---

3. Redeploy the theme. The file is displayed as the theme's thumbnail.

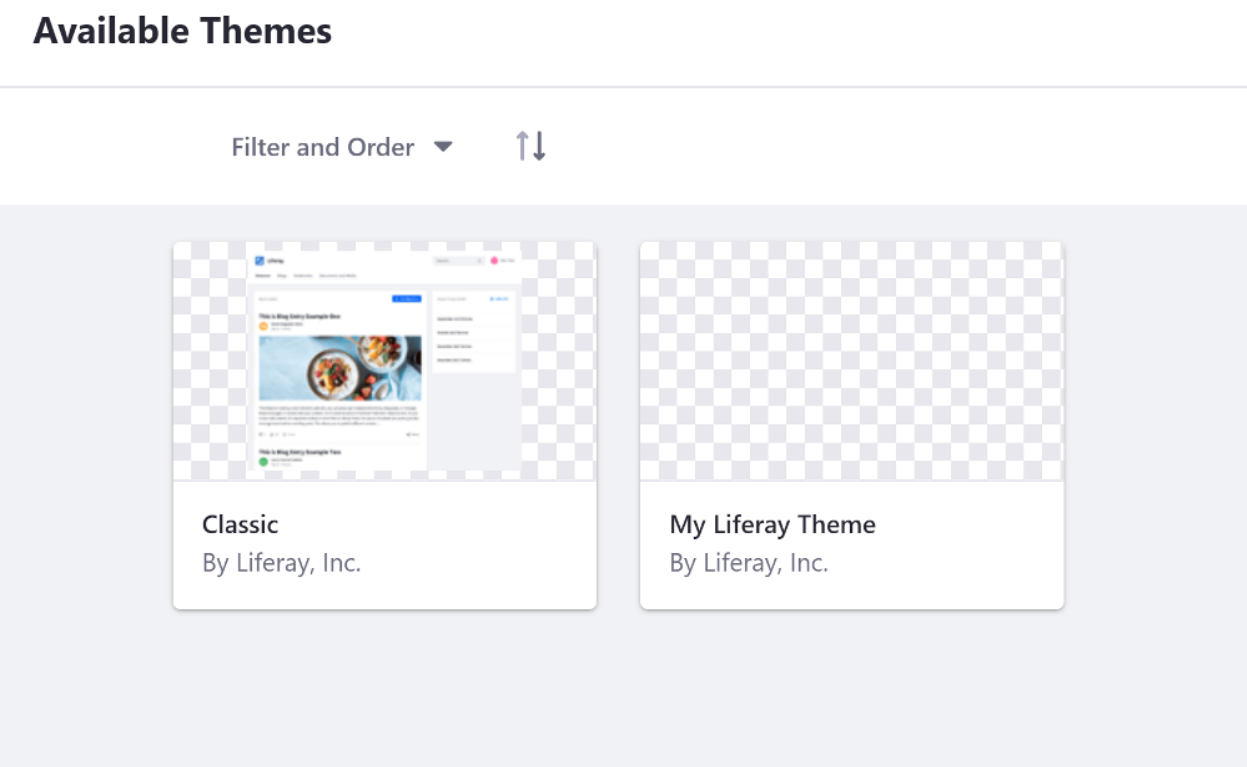

Figure 455.1: Your theme thumbnail is displayed with the rest of the available themes.

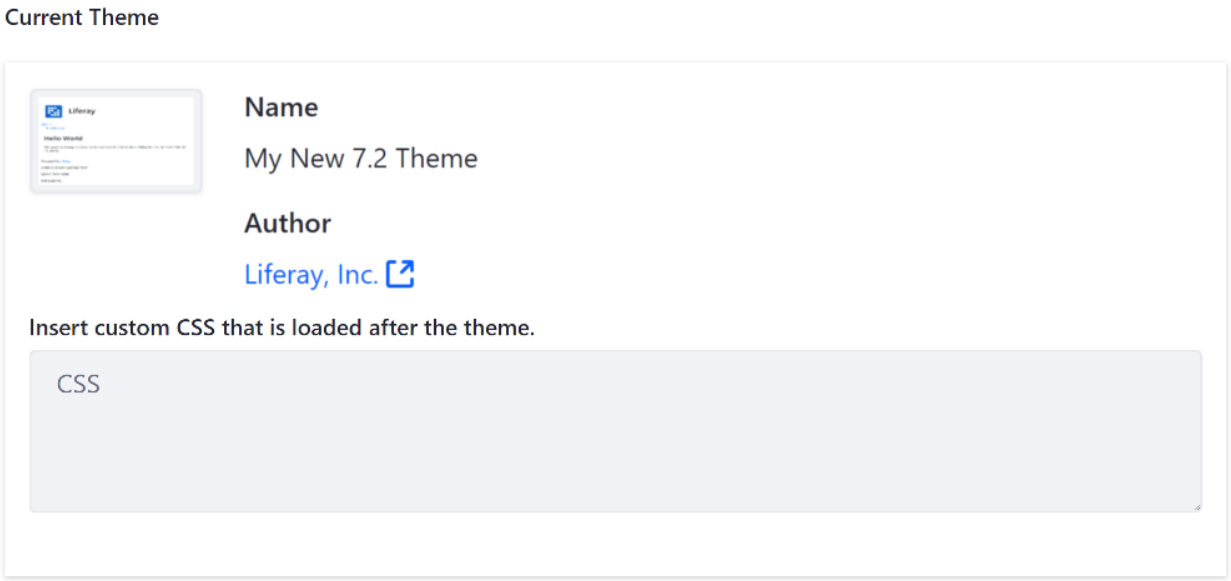

Figure 455.2: Your theme thumbnail is displayed with the rest of the available themes.

455.1 Related Topics

- Installing the Theme Generator and Creating a Theme
- Creating Themelets with the Theme Generator
- Creating Color Schemes for Your Theme

CHAPTER 456

CREATING COLOR SCHEMES FOR YOUR THEME

Color schemes give your theme additional color palettes. With just a small amount of changes to your theme's CSS, you can subtly change the look of your theme, while maintaining the same design and feel to it.

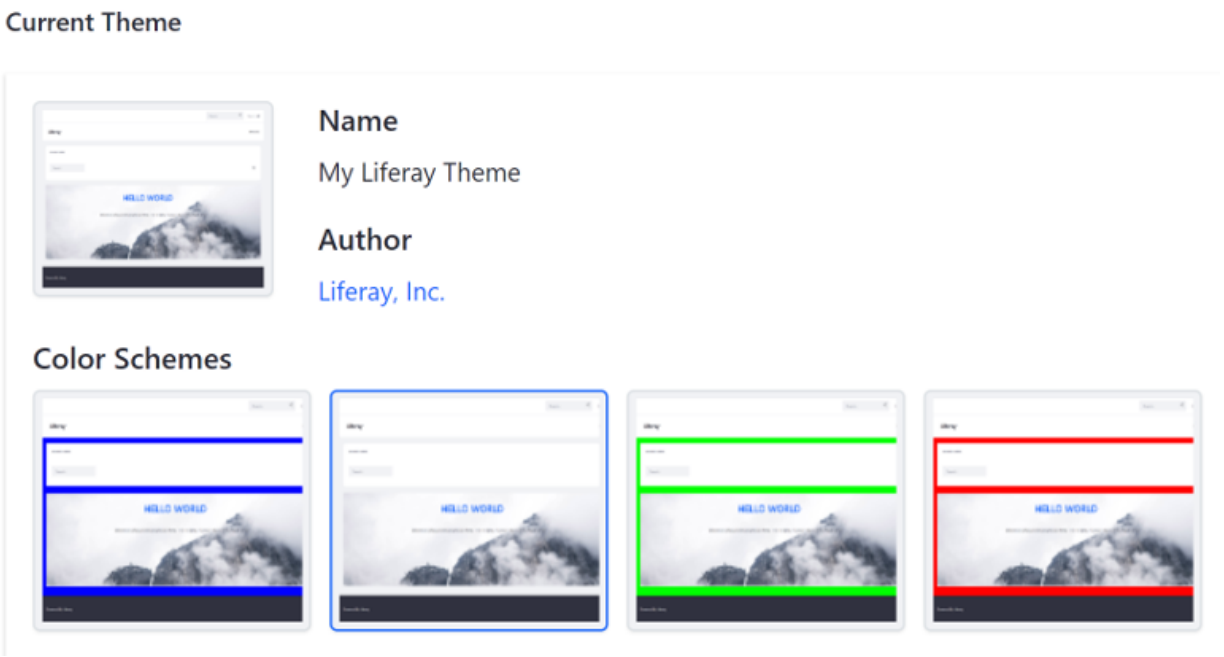

Figure 456.1: Color schemes give administrators some choices for your theme's look.

Follow these steps to create color schemes for your theme:

1. Open the theme's `WEB-INF/liferay-look-and-feel.xml` file and follow the pattern below to add the default color scheme. If your default styles are in `_custom.scss`, use the `default` `<css-class>` as shown in the example below. See the liferay-look-and-feel DTD for an explanation of each of the elements used below:

```
<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.
```

---

2. Add the remaining color schemes below the default color scheme, using the pattern below. Note that the IDs, names, and CSS classes must be unique for each color scheme.

```
<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` configuration is shown below:

```
<look-and-feel>
    <compatibility>
        <version>7.2.0+</version>
    </compatibility>
    <theme id="my-great-theme" name="My Great 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="Dark">
            <css-class>dark</css-class>
        </color-scheme>
        <color-scheme id="03" name="Light">
            <css-class>light</css-class>
        </color-scheme>
        <portlet-decorator ...>
            ...
    </theme>
</look-and-feel>
```

3. Create a folder for your 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, excluding the default color scheme since those styles are included in `_custom.scss`.

4. The color scheme class is added to the theme's `<body>` element when the color scheme is applied, so add the class to the color scheme's styles to target the proper color scheme. The example below specifies styles for a color scheme with the class `day`:

```
body.day { background-color: #DDF; }
.day a { color: #66A; }
```

5. 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";
```

6. Create a folder for each color scheme in your theme's `images` folder, and add a thumbnail preview for them. The folder name *must match* the color scheme's CSS class name.

There you have it. Now you can go color scheme crazy with your themes!

456.1 Related Topics

- Generating Layout Templates
- Creating a Thumbnail Preview for Your Theme

CHAPTER 457

MAKING CONFIGURABLE THEME SETTINGS

If you have an aspect of a theme that you want an Administrator to configure without having to manually update and redeploy the theme, you can create a *theme setting* for it. Theme settings are very versatile and can be customized to meet your needs.

Follow the steps below to create theme settings:

1. Open your theme's `WEB-INF/liferay-look-and-feel.xml` file, and follow the pattern below to nest a `<setting/>` element inside the parent `<settings>` element for each setting you want to add:

```
<look-and-feel>
    <compatibility>
        <version>7.2.0+</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 below adds a text input setting for a custom hex code:

```
<settings>
  <setting configurable="true" key="my-hex-code" type="text" value="blue" />
</settings>
```

   See the `liferay-look-and-feel.xml`'s DTD docs for an explanation of the setting's configuration options.

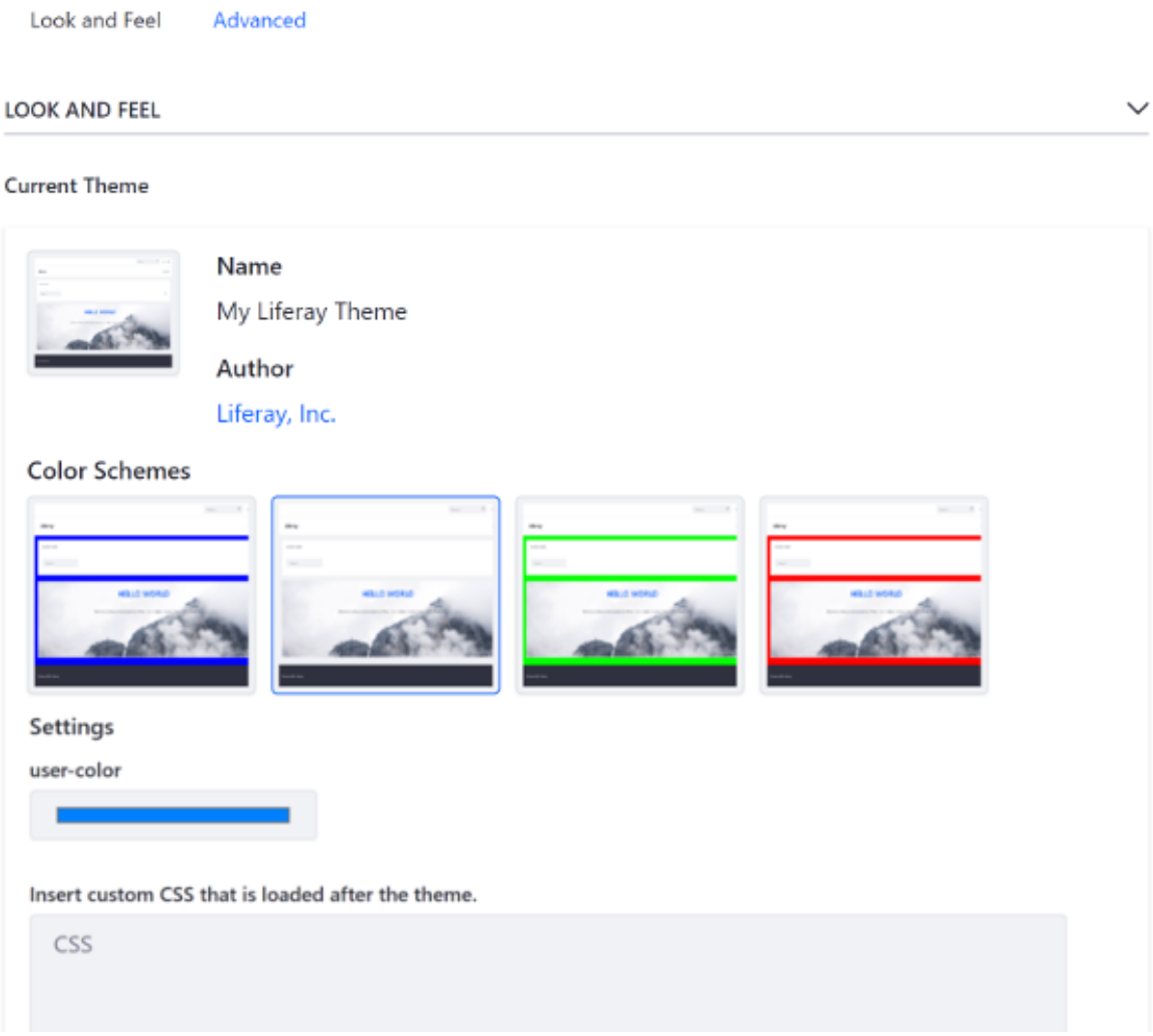

Figure 457.1: Here are examples of configurable settings for the site Admin.

Note: You can modify theme settings with JavaScript to provide a more custom experience. The example below modifies the theme setting, changing its `type` to `color`, to provide a color picker for the user:

```xml
<setting configurable="true" key="user-color"
type="text" value="#993300"
>
<![CDATA[
     AUI().ready('node',function(A) {
          A.one("#[@NAMESPACE@]user-color").setAttribute("type", "color");
          A.one("#[@NAMESPACE@]user-color").setAttribute("style", "height: 35px; width: 200px");
      });
]]>
</setting>
```

2. Create a file called `init_custom.ftl` in your theme's `templates` folder if it doesn't already exist, and follow the patterns in the table below to define your theme setting variables in it:

```
 Return Type | Description | Pattern |
 --- | --- | --- |
 Boolean | a select box with the options `true` and `false` or a checkbox with values `yes` and `no` | `<#assign my_variable_name = getterUtil.getBo
setting-key"))/>` |
 String | a text input or text area input | `<#assign my_variable_name = getterUtil.getString(themeDisplay.getThemeSetting("theme-
setting-key"))/>` |
```

```
The example below adds a custom hex code setting:

    <#assign my_hex_code =
    getterUtil.getString(themeDisplay.getThemeSetting("my-hex-code"))/>
```

3. Add your theme setting variables to the theme template. The example below prints `my_hex-code`'s value as the value of the header's `style` attribute:

   `portal_normal.ftl`:

   ```
   <header style="background-color:${my_hex_code}">
   ```

4. Deploy the theme to apply the changes. To set the theme setting for a Public or Private page set, click the *Gear icon* next to the page set you want to configure and update the setting under the *Look and Feel* tab. Alternatively, you can set the theme setting for an individual page by opening the *Actions menu* next to the page and selecting *Configure* and choosing the *Define a Specific look and feel for this page* option.

Great! You've created configurable settings for your theme.

457.1 Related Topics

- Creating Themelets with the Theme Generator
- Listing Your Theme's Extensions
- Importing Resources with a Theme

CHAPTER 458

USING FONT AWESOME AND GLYPH ICONS IN YOUR THEME

By default, Font Awesome v3.2.1 and Bootstrap 3 Glyphicons are enabled globally in Liferay DXP via a system setting. This means that you can use them in your themes to create social media links, for example. A Site Administrator can disable this to improve performance, if they choose.

458.1 Disabling Enabling Global Font Awesome and Glyphicons in Portal

Since Liferay DXP Fix Pack 2 and Liferay Portal 7.2 CE GA2, Font Awesome is available globally as a system setting, which is enabled by default. You can disable this setting to improve performance. To update the setting, follow these steps:

1. Open the Control Menu and navigate to *Control Panel* → *Configuration* → *System Settings* and select *Third Party* under the *PLATFORM* heading.
2. Select *Font Awesome* under the *System Scope* and check/uncheck the *Enable Font Awesome* checkbox to enable/disable Font Awesome icons and Glyphicons across the site.
3. Click *Save* to save the configuration.

458.2 Including Font Awesome and Glyphicons in Your Theme

As a safeguard, you should include Font Awesome and Glyphicons with your theme if you want to use them. This ensures that your icons won't break if the global system setting is disabled. If you created the theme with the Liferay Theme Generator and answered yes (y) to the Font Awesome prompt, the Font Awesome dependency is added which includes Font Awesome and Glyphicons for you. If you didn't include Font Awesome and Glyphicons when you initially created the theme, follow these steps to include them in your theme now:

1. Use the Font Awesome v3.2.1 or Bootstrap 3 Glyphicons in your theme's template. The example below uses Font Awesome icons:

```
<div id="social-media-links">
  <ul class="nav flex-row mx-auto">
      <li class="mx-2">
          <div id="facebook">
            <a class="icon-facebook icon-3x text-white"
            href="http://www.facebook.com/pages/Liferay/45119213107"
            target="_blank"><span class="hide">Facebook</span>
            </a>
          </div>
      </li>
      <li class="mx-2">
          <div id="twitter">
            <a class="icon-twitter icon-3x text-white"
            href="http://www.twitter.com/liferay"
            target="_blank"><span class="hide">Twitter</span>
            </a>
          </div>
      </li>
      <li class="mx-2">
          <div id="linked-in">
            <a class="icon-linkedin icon-3x text-white"
            href="http://www.linkedin.com/company/83609"
            target="_blank"><span class="hide">LinkedIn</span>
            </a>
          </div>
      </li>
      <li class="mx-2">
          <div id="youtube">
            <a class="icon-youtube icon-3x text-white"
            href="http://www.youtube.com/user/liferayinc"
            target="_blank"><span class="hide">YouTube</span>
            </a>
          </div>
      </li>
      <li class="mx-2">
          <div id="google-plus">
            <a class="icon-google-plus icon-3x text-white"
            href="https://plus.google.com/+liferay/posts"
            target="_blank"><span class="hide">Google</span>
            </a>
          </div>
      </li>
  </ul>
</div>
```

2. Open the theme's `package.json` and include the `liferay-font-awesome` dev dependency:

```
"liferay-font-awesome": "3.4.0"
```

3. Run `gulp deploy` from the theme's root folder to build and deploy the theme's files. This adds a `/css/font/` folder to the theme's `build` folder that contains the Font Awesome and Glyphicon fonts.

458.3 Related Topics

- Installing the Theme Generator and Creating a Theme
- Upgrading Themes

CHAPTER 459

EXTENDING THEMES

Liferay DXP has additional features that you can use to extend your theme and modify your site. This section covers these topics:

- Installing Themelets
- Injecting additional context variables into your theme templates
- Packaging independent UI resources for your site
- Copying an existing theme's files
- Listing your theme's extensions
- Overwriting and extending liferay theme tasks

CHAPTER 460

INSTALLING A THEMELET IN YOUR THEME

After you've created 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 `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.

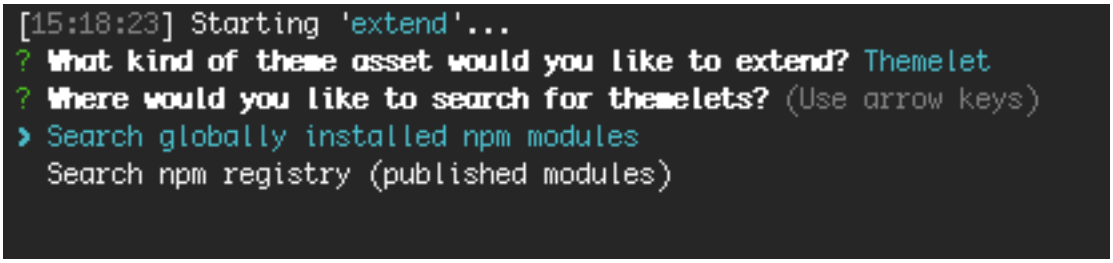

Figure 460.1: You can extend your theme using globally installed npm modules or published npm modules.

```
**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.

Great, now you know how to install a themelet in your theme! The next time you deploy your theme, the themelet will be bundled along with it.

460.1 Related Topics

-Generating Themelets with the Theme Generator -Injecting Additional Context Variables and Functionality into Your Theme Templates -Packaging Independent UI Resources for Your Site

CHAPTER 461

INJECTING ADDITIONAL CONTEXT VARIABLES AND FUNCTIONALITY INTO YOUR THEME TEMPLATES

JSPs are native to Java EE and therefore have access to all the contextual objects inherit to the platform, like the request and session. Through these objects, developers can obtain Liferay DXP-specific context information by accessing container objects like `themeDisplay` or `serviceContext`. This, however, is not the case for FreeMarker templates. To access this information in FreeMarker templates, you must inject it yourself into the template's context. Liferay DXP gives you a head start by injecting several common objects into the template's context and exposing them as FreeMarker macros. To inject other objects into the FreeMarker template's context, you must create a *Context Contributor*.

You can create a Context Contributor to use non-JSP templating languages for themes, widget templates, 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 into your theme's context, and then determine the theme's color based on that information.

Follow the steps below to create a context contributor:

1. Create an OSGi module using your favorite third party tool, or use Blade CLI.
2. Create a component class that implements the `TemplateContextContributor` service, and set the `type` property to the type of context you're injecting into. Set it to `TYPE_THEME` to inject context-specific variables for your theme, or set it to `TYPE_GLOBAL` to inject it into every context execution in Liferay DXP, like themes, widget templates, DDM templates, etc, as defined in TemplateContextContributor. 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`):

```
@Component(
    immediate = true,
    property = {"type=" + TemplateContextContributor.TYPE_THEME},
    service = TemplateContextContributor.class
)
```

3. Implement the TemplateContextContributor interface in your `*TemplateContextContributor` class, and overwrite the `prepare(Map<String,Object>, HttpServletRequest)` method to inject

new or modified variables into the `contextObjects` map. This is your template's context that was described earlier.

The `ProductMenuTemplateContextContributor`'s class is shown as an example below. It overwrites the `prepare(...)` method to inject a modified `bodyCssClass` variable and a new `liferay_product_menu_state` variable into the theme context for the Product Menu. 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:

```
@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);
}
```

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 into any context in Liferay DXP, you're free to fully harness the power of your chosen templating language.

461.1 Related Topics

- Developing Themes
- Theme Contributors

CHAPTER 462

PACKAGING INDEPENDENT UI RESOURCES FOR YOUR SITE

If you want to package UI resources independent of a specific theme and include them on every page, a *Theme Contributor* is your best option. If, instead, you want to include separate UI resources on a page that are attached to a theme, use themelets.

A Theme Contributor is a module that contains CSS and JS resources to apply to the page. The Control Menu, Product Menu, and Simulation Panel are packaged as Theme Contributors.

If you want to edit or style these standard UI components, you must create a Theme Contributor and add your modifications on top. You can also add new UI components to Liferay DXP by creating a Theme Contributor. This article shows how to create a Theme Contributor module.

Follow these steps to create a Theme Contributor:

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. Add the `Liferay-Theme-Contributor-Type` header to your module's `bnd.bnd` file to identify your module as a Theme Contributor, and add the `Web-ContextPath` header to set the context from which the Theme Contributor's resources are hosted. See the Control Menu module's `bnd.bnd` below as an example:

   ```
   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. 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.

3. Add the `Liferay-Theme-Contributor-Weight` to your `bnd.bnd` file to set a priority for your Theme Contributor. To override another Theme Contributor's styles, such as those for the Control Menu, set a higher weight. 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:

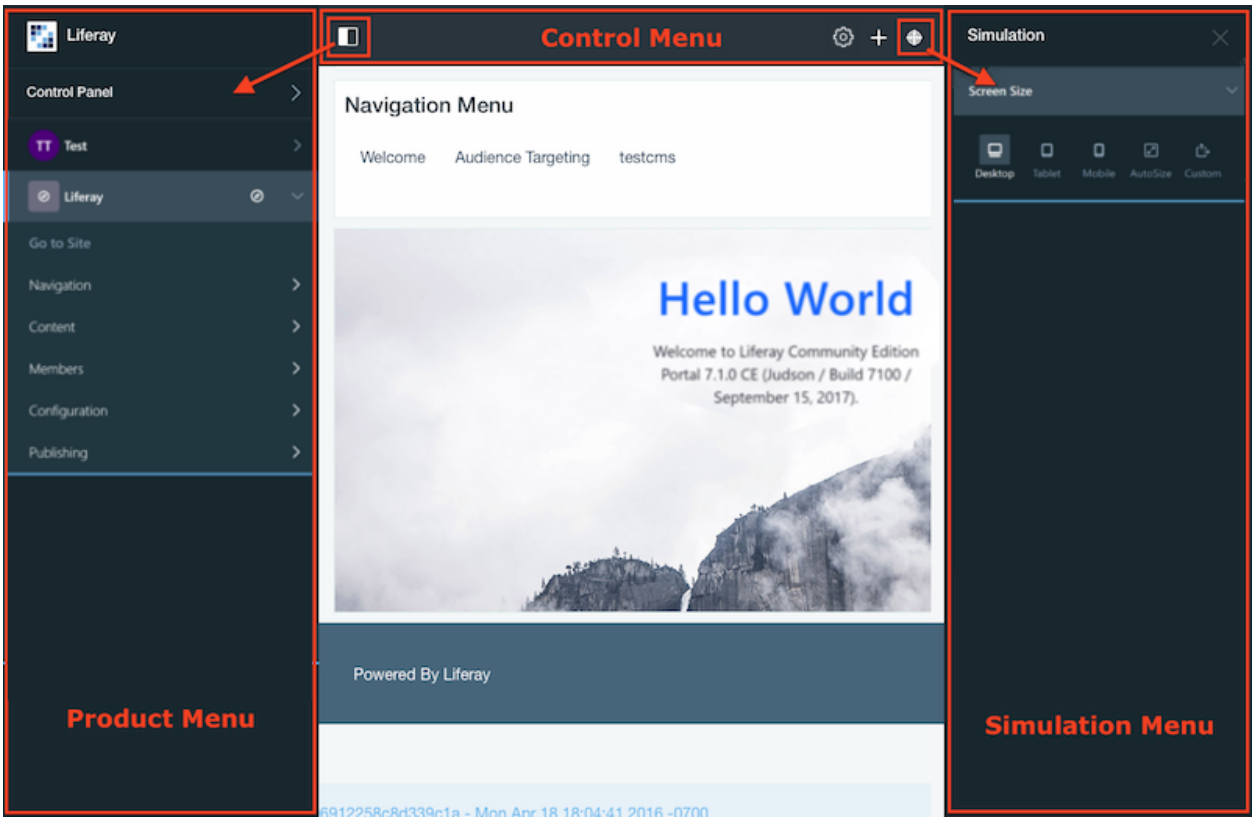

Figure 462.1: The Control Menu, Product Menu, and Simulation Panel are packaged as Theme Contributor modules.

```
Liferay-Theme-Contributor-Weight: [value]
```

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.

462.1 Related Topics

- Developing Themes
- Generating Themelets
- Installing a Themelet

CHAPTER 463

CHANGING YOUR BASE THEME

Follow these steps to change your theme's base theme with the Extend task. Note that this task only works for themes that use the liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator.

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 `gulp extend`.

```
C:\Users\liferay\Desktop\projects\themes\7-2-themes\my-new72theme-theme>gulp extend
[18:52:33] Using gulpfile ~\Desktop\projects\themes\7-2-themes\my-new72theme-theme\gulpfile.js
[18:52:33] Starting 'extend'...
? What kind of theme asset would you like to extend?
  1) Base theme
  2) Themelet
  Answer: 1
```

Figure 463.1: Run the `gulp extend` task to change your base theme.

2. Enter 1 to select a new base theme to extend.
3. 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, a theme published to the npm registry, or you can specify a package
ext-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.
```

Your theme's `package.json` contains the updated base theme configuration:

```
"liferayTheme": {
  "baseTheme": "styled",
  "screenshot": "",
  "templateLanguage": "ftl",
  "version": "7.2"
},
```

Great! You've updated your base theme. When you build your theme's files or deploy it, your theme will inherit the updated base theme's files.

463.1 Related Topics

- Configuring Your Theme's App Server
- Deploying and Applying Themes
- Listing Your Theme's Extensions

CHAPTER 464

COPYING AN EXISTING THEME'S FILES

Follow these steps to copy an existing theme's files with the Kickstart task. Unlike extending a base theme, which is a dynamic inheritance that applies your `src` files on top of the base theme on every build, the Kickstart task is a one time inheritance.

Warning: 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 that this task only works for themes that use the liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator.

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 `gulp kickstart`.

```
C:\Users\liferay\Desktop\projects\themes\7-2-themes\my-new72theme-theme>gulp kickstart
[14:27:12] Using gulpfile ~\Desktop\projects\themes\7-2-themes\my-new72theme-theme\gulpfile.js
[14:27:12] Starting 'kickstart'...
[14:27:12] Warning: the kickstart task will potentially overwrite files in your src directory
? Where would you like to search?
  1) Search globally installed npm modules
  2) Search npm registry (published modules)
  Answer:
```

Figure 464.1: Run the `gulp kickstart` task to copy a theme's files into your own theme.

2. Select where to search for the theme to copy. You can copy files from globally installed themes or themes published on the npm registry.

```
**Note:** **You can't kickstart the Classic Theme.**
```

```
**Note:** To globally install a theme, run the `npm link` command from the
theme's root folder.
```

```
![ You can copy files from  globally installed themes.](./images/theme-ext-kickstarting-themes-global-theme.png)
```

3. The theme's files are copied into your own theme, jump starting development. Add your changes on top of these files.

Congrats! Now you have a head start to developing your theme.

464.1 Related Topics

- Building Your Theme's files
- Generating Themelets with the Theme Generator
- Deploying and Applying Themes

CHAPTER 465

LISTING YOUR THEME'S EXTENSIONS

Do you need to know what base theme/themelet(s) your theme extends? Follow these steps to list your theme's extensions with the Status task. Note that this task only works for themes that use the liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator.

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.
2. Run `gulp status` to print your theme's current extensions to the command line.

Your theme's current extensions are also found under the `baseTheme` and `themeletDependencies` headings in your theme's `package.json`.

```
C:\Users\liferay\Desktop\projects\themes\7-2-themes\my-new72theme-theme>gulp status
[14:45:56] Using gulpfile ~\Desktop\projects\themes\7-2-themes\my-new72theme-theme\gulpfile.js
[14:45:56] Starting 'status'...
Base theme: styled
[14:45:56] Finished 'status' after 4.81 ms

C:\Users\liferay\Desktop\projects\themes\7-2-themes\my-new72theme-theme>
```

Figure 465.1: Run the `gulp status` task to list your theme's current extensions.

465.1 Related Topics

- Changing Your Base Theme
- Configuring Your Theme's App Server
- Generating Themelets with the Theme Generator

CHAPTER 466

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 on your theme's files prior to deploying the theme to the server—such as minifying your JavaScript files. 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:

```
gulp.task('build', function(cb) {
    runSequence(
        'build:clean',
        'build:base',
        'build:src',
        'build:web-inf',
        'build:liferay-look-and-feel',
        'build:hook',
        'build:themelets',
        'build:rename-css-dir',
        'build:compile-css',
        'build:fix-url-functions',
        'build:move-compiled-css',
        'build:remove-old-css-dir',
        'build:fix-at-directives',
        'build:r2',
        'build:war',
        cb
    );
});
```

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!

466.1 Related Topics

- Installing the Theme Generator and Creating a Theme
- Generating Themelets
- Using Developer Mode with Themes

CHAPTER 467

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.

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

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.

This section covers these topics:

- Customizing the Atlas and Clay base themes
- Integrating third party themes with Clay

CHAPTER 468

CUSTOMIZING ATLAS AND CLAY BASE THEMES IN LIFERAY DXP

Whether you're customizing the Atlas or Clay base theme, the process is, for the most part, the same. Follow these steps. If you're customizing the Clay base theme, skip to step 3.

1. By default, Clay base is imported into the theme. If you're overwriting Atlas, add a file named `clay.scss` to your theme's `/src/css/` folder and import `clay/atlas` instead:

   ```
   @import "clay/atlas";
   ```

2. By default, Clay base variables are imported into the theme. If you're overwriting Atlas, add an `_imports.scss` file to your theme's `/src/css/` folder and import Atlas variables instead:

   ```
   @import "bourbon";

   @import "mixins";

   @import "compat/mixins";

   @import "clay/atlas-variables";
   ```

```
**Note:** Bourbon mixins are deprecated as of 7.0 and will be
removed in the next major release. We recommend you use Clay mixins
instead. To use Clay mixins, follow the instructions in
[Using Clay Mixins in Your Theme](/docs/7-2/frameworks/-/knowledge_base/f/using-clay-mixins-in-your-theme)
```

---

3. Add a file named `_clay_variables.scss`. Place your Atlas, Bootstrap, and Clay Base variable modifications in this file.

Great! Now you know how to customize the Atlas and Clay base themes.

468.1 Related Topics

- Integrating Third Party Themes with Clay
- Using Clay Mixins in Your Theme

CHAPTER 469

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, add a file named `_clay_variables.scss`. Place your Atlas, Bootstrap, and Clay Base variable modifications in this file.
3. Create a folder inside `/src/css/` to house your third party theme (e.g. `/src/css/my-third-party-theme/`)
4. Copy the CSS contents of the theme to the folder you just created.
5. In `_clay_variables.scss`, import the file containing the theme variables. For example, `@import "my-third-party-theme/variables.scss";`

```
**Note:** You may omit the leading underscore when importing Sass files.
```

---

6. In `_custom.scss`, import the file containing the CSS. For example, `@import "my-third-party-theme/main.scss";`

Now you know how to integrate third party themes with Clay Base!

469.1 Related Topics

- Customizing Atlas and Clay Base Themes

CHAPTER 470

USING CLAY ICONS IN A THEME

To use Clay icons in your themes, you must use the clay taglib macro. If you want to use Clay icons in your portlets, follow the steps in the Clay taglib icons article. To use Clay icons in your theme, follow these steps:

1. Open the FreeMarker theme template you want to use the Clay icon in.
2. Use the `@clay["icon"]` macro and specify the icon with the `symbol` attribute, as shown in the pattern below:

   ```
   <@clay["icon"] symbol="icon-name" />
   ```

   The full list of icons can be found on ClayUI's site (CSS/Markup tab). Here is an example configuration for a Facebook social media icon:

   ```
   <a class="text-white"
   href="http://www.facebook.com/pages/Liferay/45119213107"
   target="_blank">
     <span class="hide">Facebook</span>
     <@clay["icon"] symbol="social-facebook" />
   </a>
   ```

Great! Now you know how to use Clay icons in your theme.

470.1 Related Topics

- FreeMarker Taglib Macros
- Clay Icons
- Customizing Atlas and Clay Base Themes

CHAPTER 471

USING CLAY MIXINS IN YOUR THEME

Bourbon mixins are deprecated as of 7.0 and will be removed in the next major release. We recommend you use Clay mixins instead. Follow these steps to use Clay mixins in your theme:

1. Add the `clay-css` dependency to the theme's `package.json`:

```
"dependencies":{
  "clay-css": "^2.18.0",
}
```

2. Delete `_imports.scss` if you modified it.
3. Import the library into the theme's `main.scss` file:

```
@import 'node_modules/clay-css/src/scss/atlas-variables'
```

   or import the base-variables if you want to use Clay Base instead:

```
@import 'node_modules/clay-css/src/scss/base-variables'
```

Great! Now you know how to use Clay mixins in your theme!

471.1 Related Topics

- Customizing Atlas and Clay Base Theme

CHAPTER 472

THEMING PORTLETS

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 along 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 7.0's Classic theme.

```
<#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())
/>
```

These variables are described in the table below:

Portlet FTL Variables

Variable	Description
`portletDisplay`	Fetched from the `themeDisplay` object, 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.

472.1 Portlet Decorators

Portlet Decorators modify the style of the application wrapper. Themes come bundled with three default portlet decorators, defined in `liferay-look-and-feel.xml`:

- 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.
- 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 section covers these topics:

- Embedding Portlets in Themes

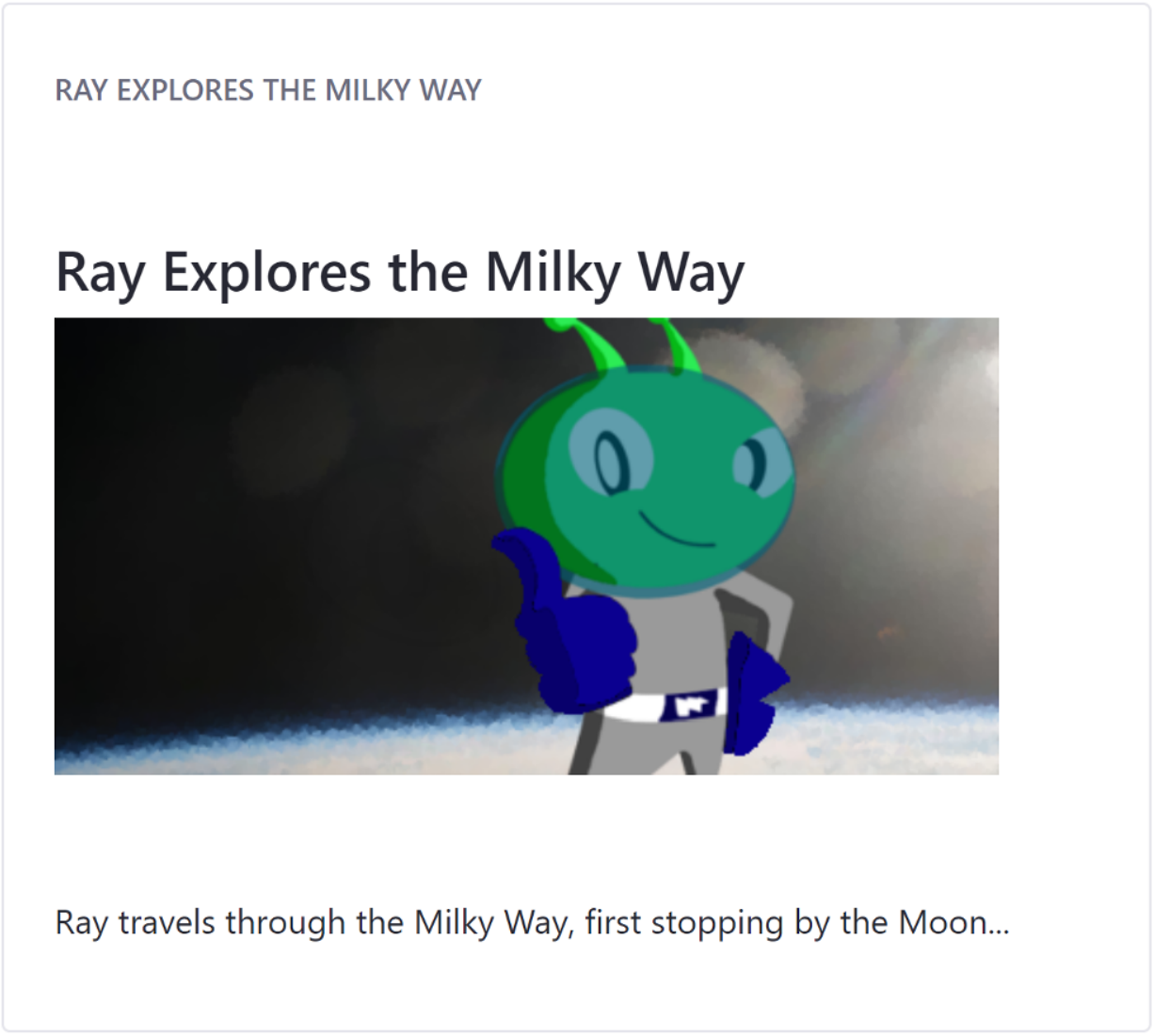

Figure 472.1: The Classic theme's Decorate Application Decorator wraps the portlet in a white box.

RAY EXPLORES THE MILKY WAY

Ray Explores the Milky Way

Ray travels through the Milky Way, first stopping by the Moon...

Figure 472.2: The Classic theme's Borderless Application Decorator displays the application's custom title.

Figure 472.3: The Classic theme's Barebone Application Decorator displays only the application's content.

CHAPTER 473

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.

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 in this section. These topics are covered:

- Embedding a portlet by entity type and action
- Embedding a portlet by instance name and ID

CHAPTER 474

EMBEDDING PORTLETS IN THEMES BY ENTITY TYPE AND ACTION

In this article, 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. Follow these steps:

1. Insert a declaration where you want the portlet embedded. This declaration expects two parameters: the type of action and the class name of the entity type the portlet should handle. An example configuration is shown below:

   ```
   <@liferay_portlet["runtime"]
       portletProviderAction=portletProviderAction.VIEW
       portletProviderClassName="com.liferay.portal.kernel.servlet.taglib.ui.LanguageEntry"
   />
   ```

   This example 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.
```

---

2. 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. Create a module.

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, 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.).

4. 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 must 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`).

5. 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. Below is an example configuration:

```
@Override
public String getPortletName() {
  return SiteNavigationLanguagePortletKeys.SITE_NAVIGATION_LANGUAGE;
}

@Override
public PortletURL getPortletURL(
    HttpServletRequest httpServletRequest, Group group)
  throws PortalException {

  return PortletURLFactoryUtil.create(
    httpServletRequest, getPortletName(), PortletRequest.RENDER_PHASE);
}

/**
 * @deprecated As of Judson (7.1.x)
 */
@Deprecated
@Override
protected long getPlid(ThemeDisplay themeDisplay) throws PortalException {
  return themeDisplay.getPlid();
}
```

   This returns the portlet ID and 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.

6. Generate the module's JAR file and copy it to your app server's `osgi/modules/` folder. 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.

Awesome! 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.

474.1 Related Topics

- Embedding Portlets in Themes
- Portlets
- Service Builder

CHAPTER 475

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. Follow these steps:

1. Open the FreeMarker theme template that you want to embed the portlet in. (`portal_normal.ftl` is a good choice.
2. Add the `liferay_portlet["runtime"]` macro to the template, as shown below. The portlet name **must** be the same as `javax.portlet.name`'s value.

```
<@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`.

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"
/>
```

Great! Now you know how to embed a portlet in your theme's by their name and ID.

475.1 Related Topics

- Embedding Portlets in Themes by Entity Type and Action
- Portlets
- Service Builder

CHAPTER 476

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`:

```
<#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"
/>
```

Below are some additional attributes you can define for embedded portlets:

Preference	Description
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*.

Preference	Description
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 set default preferences for embedded portlets!

476.1 Related Topics

- Embedding Portlets by Name
- Portlets
- Service Builder

CHAPTER 477

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.

Important: The Resources Importer is deprecated as of 7.0 7.1.

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.

477.1 Organizing Your Resources

Add your resources to the theme's `/src/WEB-INF/src/resources-importer` folder as 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`.

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 explains how to use the Resources Importer to import resources with your theme.

CHAPTER 478

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.

Important: The Resources Importer is deprecated as of 7.0 7.1.

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 covers how to create a sitemap for your theme, from defining pages to defining portlets.

CHAPTER 479

DEFINING LAYOUT TEMPLATES AND PAGES IN A SITEMAP

A sitemap defines the layouts—pages—and layout templates that your site or site template uses. Follow these steps to define pages and layout templates in your theme's `sitemap.json`:

1. Define 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"
    }
  ]
}
```

   You can override this by defining a layout template inside a page configuration. 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"
        }
    ]
}
```

```
**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](/docs/7-2/frameworks/-/knowledge_base/f/specifying-where-to-import-your-themes-resources).
```

---

2. Follow the pattern below to specify the public and (optionally) private pages for your theme. 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. See Sitemap Page Configuration Options for a full list of the available options.

```
{
  "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"
}
```

Great! Now you know how to configure pages for the Resources Importer.

479.1 Related Topics

- Preparing and Organizing Web Content for the Resources Importer
- Defining Portlets in a Sitemap
- Specifying Where to Import Your Theme's Resources

CHAPTER 480

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 Setting Default Preference for An Embedded Portlet for more information.

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.

480.1 Related Topics

- Preparing and Organizing Web Content for the Resources Importer
- Defining Layout Templates and Pages in a Sitemap
- Specifying Where to Import Your Theme's Resources

CHAPTER 481

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
   ```

481.1 Related Topics

- Defining Layout Templates and Pages in a Sitemap
- Defining Portlets in a Sitemap
- Preparing and Organizing Web Content for the Resources Importer

```
    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 481.1: Portlet IDs can be found via the Gogo Shell.

CHAPTER 482

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. 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 optionally a 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 if it has one. 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.

482.1 Related Topics

- Creating a Sitemap for the Resources Importer
- Defining Assets for the Resources Importer
- Specifying Where to Import Your Theme's Resources

CHAPTER 483

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.

1. Create the `assets.json` in your theme's `[theme-name]/src/WEB-INF/src/resources-importer` folder.

2. Follow the pattern below to configure your `assets.json` file. 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!

483.1 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

CHAPTER 484

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. These options are covered below. Follow these steps:

1. 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. This is enabled by default for themes generated with the Liferay Theme Generator. Without this property enabled, 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:

```
resources-importer-developer-mode-enabled=true
```

```
**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.**
```

---

2. Specify where to import your resources. By default, resources are imported into a new site template named after the theme. If that's what you want, you can skip this step. If instead 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]
```

Alternatively, you can import resources into an existing site. **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.
```

---

3. Deploy the theme. To view your theme and its resources, 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.

Great! Now you know how to specify where to import your theme's resources.

484.1 Related Topics

- Preparing and Organizing Web Content for the Resources Importer
- Creating a Sitemap for the Resources Importer
- Defining Assets for the Resources Importer

CHAPTER 485

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.

Follow these steps to archive your site's resources:

1. Export the contents of a site using the site scope.
2. Place the `archive.lar` file in your theme's `/src/WEB-INF/src/resources-importer` folder.

Great! Now you know how to archive your site's resources.

485.1 Related Topics

- Preparing and Organizing Web Content for the Resources Importer
- Creating a Sitemap for the Resources Importer
- Defining Assets for the Resources Importer

CHAPTER 486

TROUBLESHOOTING THEMES

These frequently asked questions and answers help you troubleshoot and correct problems in theme development.

Click a question to view the answer.

- 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 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-
2/frameworks/-/knowledge_base/f/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>
```

CHAPTER 487

LAYOUT TEMPLATES

Layout templates dictate where content and apps can be placed on a page. There are several default layout templates for organizing content on your pages:

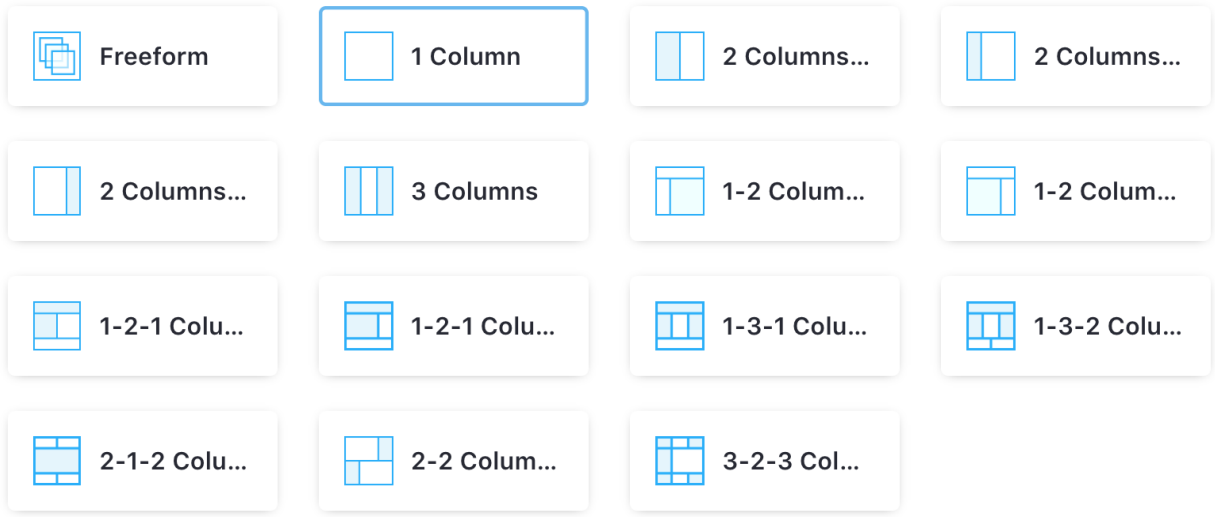

Figure 487.1: There are many default layout templates to choose from.

If the default layouts don't work for your site, you can create your own layout template by following the articles listed below:

- Create the layout template
- Create the layout template thumbnail preview
- Bundle the layout template with a theme

Layout Templates specify a number of rows and columns for the page. The rows and columns dictate where apps (and fragments) can be placed on the page. Layout templates are written in FreeMarker. 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>
```

Columns use the Bootstrap grid system. Every row consists of twelve sections, so columns can range in size 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`), which specifies when the element expands to 100% width. For example, `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, while both `column-1` and `column-2` are 50% width on small sized view ports.

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.

CHAPTER 488

CREATING CUSTOM LAYOUT TEMPLATE THUMBNAIL PREVIEWS

To showcase your layout template properly, you must provide a thumbnail preview for it. Without this, no one will know the design of the layout. Follow these steps to provide a thumbnail preview for your layout template:

1. Navigate to the layout template's `docroot/` folder. If you bundled the layout template with a theme created with the Liferay Theme Generator, the thumbnail is located in your theme's `src/layouttpl/custom/my-layoutttpl` folder.

2. Create a custom thumbnail PNG inside the folder specified in step 1 with the same name as the layout template that is 120 px x 120 px . Delete the temporary thumbnail PNG file if it exist.

Figure 488.1: A thumbnail preview displays the layout's design to the user.

3. 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!

488.1 Related topics

- Layout Templates with the Liferay Theme Generator
- Bundling Layout Templates with a Theme
- Themes

CHAPTER 489

INCLUDING LAYOUT TEMPLATES WITH A THEME

Although you can deploy a layout template by itself, you can also bundle it with a theme. If you generated a layout template with the Layouts sub-generator from inside a generated theme project, the layout template is bundled with the theme automatically. If, however, you generated a layout template and want to bundle it with a theme afterwards, follow these steps to include the layout template with a theme:

1. Copy the layout template's `liferay-layout-templates.xml` file to the theme's `src/WEB-INF/` folder.

2. Create a `layouttpl/custom/my-layouttpl/` folder inside the theme's `src/` folder.

3. Copy the layout template's FreeMarker (.ftl) file, and thumbnail preview (.png) if it exist, over to the `layouttpl/custom/my-layouttpl/` folder.

4. Copy the theme's `liferay-theme.json` file into the `src/layouttpl/custom/my-layouttpl/` folder and rename it `liferay-plugin.json`.

5. Open `liferay-plugin.json`, rename the `LiferayTheme` entry `LiferayPlugin`, and replace the `pluginName` entry's value with the name of the layout template. Below is an example configuration:

```
{
  "LiferayPlugin": {
    "deploymentStrategy": "LocalAppServer",
    "appServerPath": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\liferay-ce-portal-tomcat-7.2.0\\tomcat-9.0.10",
    "deployPath": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\liferay-ce-portal-tomcat-7.2.0\\tomcat-9.0.10\\deploy",
    "url": "http://localhost:8080",
    "appServerPathPlugin": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\liferay-ce-portal-tomcat-7.2.0\\tomcat-9.0.10\\webapps\\my-
layouttpl",
    "deployed": false,
    "pluginName": "my-layouttpl"
  }
}
```

Now you know how to include layout templates with your Liferay DXP themes!

489.1 Related topics

- Layout Templates with the Liferay Theme Generator
- Creating Custom Layout Template Thumbnail Previews
- Themes

CHAPTER 490

CREATING AND BUNDLING JAVASCRIPT WIDGETS WITH JAVASCRIPT TOOLING

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 Liferay JS Generator and liferay-npm-bundler, developers can easily create and develop JavaScript widgets in Liferay DXP using pure JavaScript tooling. The Liferay JS Generator generates JavaScript widgets for Liferay DXP. It is just one of Liferay JS Toolkit's tools.

Note: To use the Liferay JS Generator, you must have the Liferay JS Portlet Extender activated in your Liferay DXP instance. It's activated by default. You can confirm this by opening the Control Menu, navigating to the *App Manager*, and searching for `com.liferay.frontend.js.portlet.extender`.

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.

Once you've installed the Liferay JS Generator and generated a widget, you can configure instance settings, system settings, and even provide localization for your widget. This section explains how to configure these options for your generated JS widget.

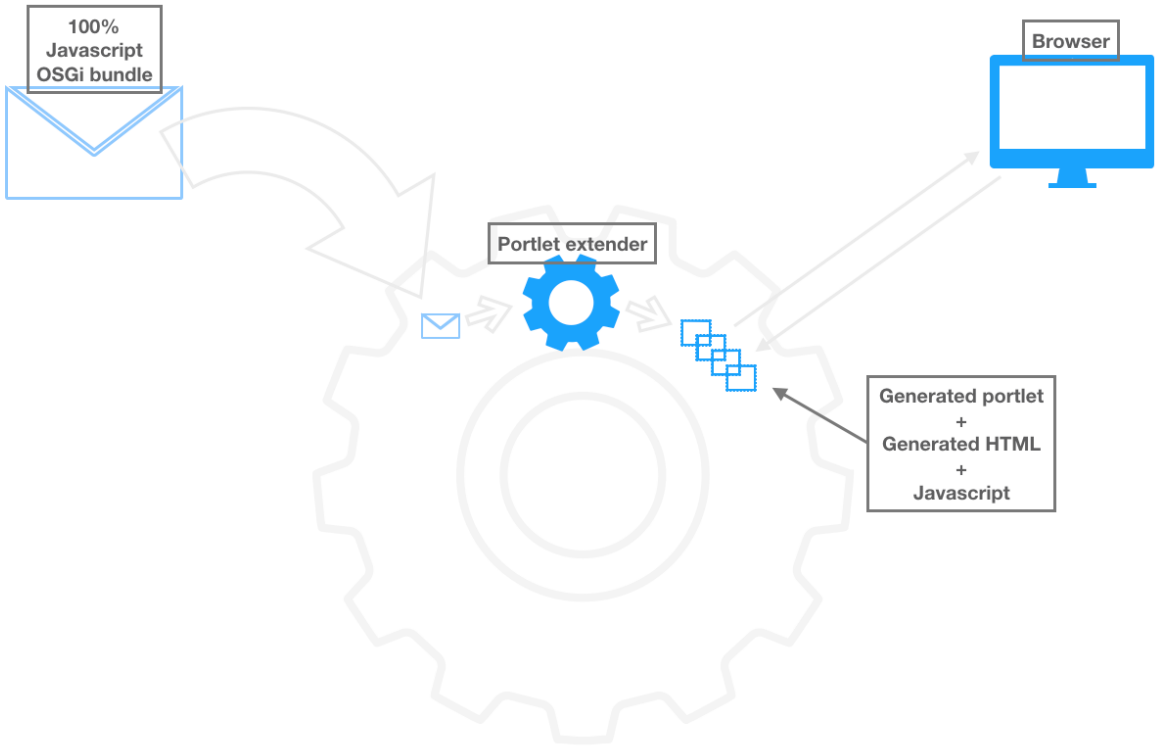

Figure 490.1: The JS Portlet Extender lets you use pure JavaScript tooling to write widgets.

CHAPTER 491

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/7-2/reference/-/knowledge_base/r/understanding-the-npmbundlerrcs-
structure#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 settings 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.

491.1 Related Topics

- Localizing Your Widget
- Using Translation Features in Your JavaScript Widget
- Setting Portlet Properties for Your JavaScript Widget

CHAPTER 492

LOCALIZING YOUR WIDGET

Follow the steps below to learn how to localize your widget:

1. If you chose not to use localization when you generated the bundle, follow this step to enable it in your bundle. 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
portlet-namespace=Portlet 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!

492.1 Related Topics

- Configuring System Settings and Instance Settings for Your JavaScript Widget
- Using Translation Features in Your JavaScript Widget
- Setting Portlet Properties for Your JavaScript Widget

CHAPTER 493

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's 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.

493.1 Related Topics

- Configuring System Settings and Instance Settings for Your JavaScript Widget
- Localizing Your Widget
- Setting Portlet Properties for Your JavaScript Widget

CHAPTER 494

CONFIGURING PORTLET PROPERTIES FOR YOUR WIDGET

Follow these steps to configure your widget'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 widget's properties.

494.1 Related Topics

- Configuring System Settings and Instance Settings for Your JavaScript Widget
- Localizing Your Widget
- Using Translation Features in Your JavaScript Widget

CHAPTER 495

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.

This section shows how to load JavaScript modules in Liferay DXP.

CHAPTER 496

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. The steps below cover 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.

Follow these steps to prepare your module for the Liferay AMD Module Loader:

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 another module 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. You can
configure this value through System Settings. Open the Control Panel and
navigate to *Configuration* &rarr; *System Settings* &rarr; *PLATFORM* &rarr;
*Infrastructure*, and select *JavaScript Loader*. Set the *Module Definition
Timeout* configuration to the time you want and click *Save*.
```

496.1 Related Topics

- Loading Modules with AUI Script
- Using npm in Your Portlets
- Loading Modules with AUI Script

CHAPTER 497

USING EXTERNAL JAVASCRIPT LIBRARIES

You can use external JavaScript libraries in your portlets (i.e., anything but Metal.js or jQuery, which are included by default). If you're the owner or hosting the external library, there are a few more requirements to load them with the JavaScript Loaders.

Follow these steps:

1. 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 {};
}));
```

2. If you're hosting the library (and not loading it from a CDN), you must hide the Liferay AMD Loader to use your Library. Open the Control Panel, navigate to *Configuration → System Settings*.

3. Click *JavaScript Loader* under *Platform → Infrastructure*.

4. 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()`.

Great! Now you know how to adapt external libraries for Liferay's JavaScript Loaders.

497.1 Related Topics

- Liferay AMD Module Loader
- Using ES2015+ Modules in Your Portlet
- Loading Modules with AUI Script

CHAPTER 498

LOADING MODULES WITH AUI SCRIPT

The `aui:script` tag is a JSP tag that loads JavaScript 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 7.2. Eventually, these tags will be replaced with Clay tag counterparts.

The `aui:script` tag supports the following options:

Option	Description
`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.

This section covers how to load ES2015, Metal.js, and AUI modules with the AUI script tag.

CHAPTER 499

LOADING ALLOYUI MODULES WITH AUI SCRIPT

Follow these steps to load modules with <aui:script>:

1. Add the following declaration to your portlet's JSP:

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

2. Add the <aui:script> tag and 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 JavaScript 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>
```

Wonderful! Now you know how to load AUI/YUI modules in AUI scripts.

499.1 Related Topics

- Using External JavaScript Libraries
- Loading AMD Modules
- Loading ES2015 and Metal.js Modules with AUI Script

CHAPTER 500

LOADING ES2015 AND METAL.JS MODULES WITH AUI SCRIPT

Follow these steps to load your ES2015 and Metal.js modules with `<aui:script>`:

1. Add the following declaration to your portlet's JSP:

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

2. Add the `<aui:script>` tag and use the `require` attribute to load ES2015 and Metal.js modules:

```
<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 module 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 JavaScript 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>
```

Great! Now you know how to load ES2015 and Metal.js modules in AUI scripts.

500.1 Related Topics

- Using External JavaScript Libraries
- Loading AMD Modules
- Loading AUI, ES2015, and Metal.js Modules Together with AUI Script

CHAPTER 501

LOADING AUI, ES2015, AND METAL.JS MODULES TOGETHER WITH AUI SCRIPT

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.

Follow these steps to load your ES2015, Metal.js, and AUI modules together with `<aui:script>`:

1. Add the following declaration to your portlet's JSP:

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

2. Add the `<aui:script>` tag and use the `require` attribute to load ES2015 and Metal.js modules, while using the `AUI().use()` function to load AUI modules, as shown in the example below:

```
<aui:script require="path-to/metal/module">
 AUI().use(
    'liferay-aui-module',
    function(A) {
        let var = pathToMetalModule.default;
    }
);
</aui:script>
```

Great! Now you know how to load all your modules with `aui:script`.

501.1 Related Topics

- Using External JavaScript Libraries
- Loading AMD Modules
- Loading ES2015 and Metal.js Modules with AUI Script

CHAPTER 502

LOADING BUNDLED NPM MODULES IN YOUR PORTLETS

Once you've exposed your modules, you can use them in your portlet via the `aui:script` tag's `require` attribute. You can load the npm module in your portlet using the `npmResolvedPackageName` variable, which is available by default since 7.0. You can then create an alias to reference it in your portlet.

Follow these steps:

1. Provide a `Web-ContextPath` in your bundle's `bnd.bnd` file:

   ```
   Web-ContextPath: /my-module-web
   ```

2. Make sure the `<liferay-frontend:defineObjects />` tag is included in your portlet's `init.jsp`. This makes the `npmResolvedPackageName` variable available, setting it to your project module's resolved name. For instance, if your module is called `my-module` and is at version `2.3.0`, the implicit variable `npmResolvedPackageName` is set to `my-module@2.3.0`. This lets you prefix any JS module `require` or soy component rendering with this variable.

3. Use the `npmResolvedPackageName` variable along with the relative path to your JavaScript module file to create an alias in the `<aui:script>`'s `require` attribute. An example configuration is shown below:

   ```
   <aui:script
     require='<%= npmResolvedPackageName +
     "/js/my-module.es as myModule" %>'>
   </aui:script>
   ```

4. Use the alias inside the `aui:script` to refer to your module:

   ```
   <aui:script
     require='<%= npmResolvedPackageName +
     "/js/my-module.es as myModule" %>'>

       myModule.default();
   </aui:script>
   ```

Now you know how to use an npm module's package!

502.1 Related Topics

- Obtaining an OSGi bundle's Dependency npm Package Descriptors
- liferay-npm-bundler
- How liferay-npm-bundler Publishes npm Packages

CHAPTER 503

THE INFO FRAMEWORK

7.0 introduces the Info Framework to provide a greater degree of extensibility for the most common needs of retrieving, processing and displaying any type of information. A key aspect of the Info Framework is that it makes no assumptions about the source of the data or how it is represented in memory (like which Java class the information is from). It can work with information stored in the database, created through some process in memory or retrieved from an external source.

In 7.0, the Info Framework still has limited functionality, but it sets the foundation for obtaining and displaying information from external systems or custom data models in Liferay. It also provides flexibility in customizing how any piece of information is displayed.

The Info Framework is lightweight. By design, it does not require all the information to implement any specific interface. This means that you can use it with any existing Java class, even if you don't have access to modify it. It comprises a collection of loosely coupled micro-frameworks, so that developers can choose which features to use and ignore the others. This lowers the learning curve and minimizes work for developers.

Note: Liferay veterans may notice similarities between the Asset Framework and Info Frameworks. The Info Framework can be considered a generalization of the Asset Framework and its design has considered lots of lessons learned from the Asset Framework. In particular, the Info Framework provides many similar characteristics (such as rendering of information) but with fewer requirements (such as having an `AssetEntry` in the database). We have also made sure the two frameworks play well together so that when the Asset Framework is used, for rendering an asset, the renderer is also available through the Info framework. The Info framework is not meant to be a full replacement for the Asset Framework, but if the Asset Framework seems too complex for your needs, the Info Framework might be a better fit for you.

503.1 Using the Info Framework

The Info Framework helps generalize information handling. Custom applications can use it to make them more generic and extensible.

Some of the out-of-the-box Liferay features use it to achieve that same goal. In particular, Liferay DXP 7.2 uses it in two ways:

- The Asset Publisher can display Assets from Information Lists defined by the Info Framework.
- Display Pages, which previously only worked for an `AssetEntry`, can now leverage the `InfoFramework` to create display pages for any type of information that can be represented by a Java class. Developers can add support for display pages for various entities like Orders, Categories, and Events that are not Assets.

There are currently two tools provided by the Info Framework: Information Item Renderers and Information List Providers. You can create an Information List Provider to obtain a list of information items from a source, or create an Information Item Renderer to provide a custom renderer for any type of information. These two features can be used together or separately.

503.2 List Providers

Information List Providers obtain a list of information items from a source. To do this, a developer must implement the `InfoListProvider` interface and provide the necessary logic for retrieving the information from its source. By providing an implementation of the `InfoListProvider` interface, developers can provide programmatic retrieval of information of any type, as long as it can be represented through a Java class.

`InfoListProvider` has four methods to implement:

`getLabel()` provides the label that is displayed for this provider in the UI of applications like the Asset Publisher.

`getInfoList()` provides the information list. This method has two variants: a plain list or a list with pagination and sorting.

`getInfoListCount()` provides total number of items. This is needed for the paginated variant of `getInfoList`.

For an example of how to create Information List providers, see Creating Information List Providers.

503.3 Item Renderers

Developers can create custom renderers for any type of information. To do this, a developer must provide an implementation of the `InfoItemRenderer` interface to provide programmatic rendering of information. It can be any kind of information as long as it can be represented through a Java class. You can create multiple renderers for a single type of information.

Internally, Liferay's Display Pages use this from the Content component. When it is added to a display page template, this component renders whatever piece of information is shown through that template (whether it is Content in the strict sense or some other entity type). It is rendered by the first `InfoItemRenderer` class registered that entity. Information Item Renderers will be leveraged further in future Liferay versions.

To create an Information Item Renderer you must create a class that implements `InfoItemRenderer` and registers it as a component. Inside that class, you need a `render()` method that contains your logic. To learn about Information Item Renderers, see Creating Information Item Renderers.

CHAPTER 504

CREATING AN INFORMATION LIST PROVIDER

To demonstrate Information List Providers, follow the instructions below to implement an `InfoListProvider` for the most viewed asset entries. In this case the list shows a list of `AssetEntry` instances. Since they already have their own renderer, they can appear in the Asset Publisher with no additional changes. If you create a provider for a custom class, you must also render it.

1. Create a module named asset-entry-info-list-provider.
2. Create a package inside the module named `com.liferay.docs.info.provider`.
3. Inside the package, create a class named `AssetEntryInfoListProvider` that implements `InfoListProvider` and registers it as a component:

```
@Component(service = InfoListProvider.class)
  public class AssetEntryInfoListProvider implements InfoListProvider<AssetEntry> {

}
```

4. Next, add the necessary `@Reference` that you need for the logic of retrieving assets to the bottom of the class.

```
@Reference
AssetEntryLocalService _assetEntryLocalService;
```

5. Then implement `getInfoList` which returns just the list.

```
@Override
public List<AssetEntry> getInfoList(
 InfoListProviderContext infoListProviderContext) {

 return _assetEntryLocalService.getTopViewedEntries(
    new String[0], false, 0, 20);
}
```

   Descending order and a maximum of 20 items to return is hardcoded.

6. Now implement the second method, which provides greater control over how items are returned to the provider.

```
@Override
public List<AssetEntry> getInfoList(
  InfoListProviderContext infoListProviderContext, Pagination pagination,
  Sort sort) {

  return _assetEntryLocalService.getTopViewedEntries(
     new String[0], !sort.isReverse(), pagination.getStart(),
     pagination.getEnd());
}
```

7. Provide a method to get a full count of info list items.

```
@Override
public int getInfoListCount(
InfoListProviderContext infoListProviderContext) {

    Company company = infoListProviderContext.getCompany();

    return _assetEntryLocalService.getCompanyEntriesCount(
    company.getCompanyId());
}
```

8. Finally, add a method that provides a display label for the list.

```
@Override
public String getLabel(Locale locale) {
 return "Most Viewed Content";
 }
```

The completed class should look like this:

```
@Component(service = InfoListProvider.class)
  public class AssetEntryInfoListProvider implements InfoListProvider<AssetEntry> {

          @Override
          public List<AssetEntry> getInfoList(
            InfoListProviderContext infoListProviderContext) {

            return _assetEntryLocalService.getTopViewedEntries(
               new String[0], false, 0, 20);
          }

         @Override
         public List<AssetEntry> getInfoList(
           InfoListProviderContext infoListProviderContext, Pagination pagination,
           Sort sort) {

           return _assetEntryLocalService.getTopViewedEntries(
              new String[0], !sort.isReverse(), pagination.getStart(),
              pagination.getEnd());
         }

         @Override
         public int getInfoListCount(
         InfoListProviderContext infoListProviderContext) {

             Company company = infoListProviderContext.getCompany();

             return _assetEntryLocalService.getCompanyEntriesCount(
             company.getCompanyId());
         }

           @Override
```

```
            public String getLabel(Locale locale) {
              return "Most Viewed Content";
              }

@Reference
AssetEntryLocalService _assetEntryLocalService;
}
```

This class is now ready to go! If you deploy it, it shows the “Most Viewed Content” in any Asset Publisher.

504.1 Next steps

This example is pretty simplistic and probably not useful in real world cases. To begin with, you may want to scope the search to the current site. You can also add more advanced filter criteria or provide a configuration for the provider using Liferay’s configuration framework.

As mentioned, it is also possible to implement providers for custom types. The following code shows a partial example of a provider for a custom `MyOrder` class:

```
@Component(service = InfoListProvider.class)
public class MyOrderProvider implements InfoListProvider<MyOrder> {

    @Override
    public List<MyOrder> getInfoList(
        InfoListProviderContext infoListProviderContext, Pagination pagination,
        Sort sort) {

        return _myOrderLocalService.getOrders(
            [...], !sort.isReverse(), pagination.getStart(),
            pagination.getEnd());
        }

    [..]

    @Reference
    MyOrderLocalService _myOrderLocalService;
```

CHAPTER 505

CUSTOM RENDERING OF INFORMATION WITH InfoItemRenderer

To demonstrate the `InfoItemRenderer`, implement a class that can render information provided through a custom class called `MyOrder`.

1. Create a module named `my-order`.
2. In `my-order`, create a package named `com.liferay.docs.info.myorder`
3. In the package, create a class that implements `InfoItemRenderer` and register it as a component.

```
@Component(service = InfoItemRenderer.class)
public class MyOrderRenderer implements InfoItemRenderer<MyOrder> {
    @Override
    public void render(
       MyOrder myOrder, HttpServletRequest httpServletRequest,
       HttpServletResponse httpServletResponse) {

    }
}
```

4. Next you must add the logic for the `render()` method.

```
@Override
public void render(
   MyOrder myOrder, HttpServletRequest httpServletRequest,
   HttpServletResponse httpServletResponse) {

   StringBundler sb = new StringBundler(3);

   sb.append("<ul>");
   sb.append("<li>By: " + myOrder.getBy());
   sb.append("<li>When: " + myOrder.getWhen());
   sb.append("<li>Items: " + myOrder.getItems());
   sb.append("</ul>");

   try {
      PrintWriter printWriter = httpServletResponse.getWriter();

      printWriter.write(sb.toString());
```

```
        }
        catch (IOException ioe) {
           throw new RuntimeException(ioe);
        }
    }
```

For this example you rendered everything through a `StringBundler`. In more complex cases, you would use JSPs or another templating technology.

The renderer is ready for use! In 7.0, Info Item Renderers are not widely used, but the usages and application will grow in future releases.

CHAPTER 506

USING PROVIDERS WITH CUSTOM APPLICATIONS

Imagine a widget that can display lists of orders. You can use the Info Framework so that it shows any list of orders provided through `InfoListProvider`.

First you must obtain a list of all available providers for the desired type, and then you would obtain a specific provider through that list.

1. The list of all available providers for `MyOrder`, can be obtained done by using the `InfoListProviderTracker`:

   ```
   @Reference
   InfoListProviderTracker _infoListProviderTracker;
   ```

   Once a tracker is available, obtaining the list is as simple as invoking `getInfoListProviders()`:

   ```
   _infoListProviderTracker.getInfoListProviders(MyOrder.class);
   ```

   When the user selects an item from this list, you can store the class's name.

2. When a specific provider is desired it can be obtained through its class name as follows:

   ```
   _infoListProviderTracker.getInfoListProvider(infoListProviderClassName);
   ```

506.1 Leveraging renderers from a custom application

Using renderers from a custom application is almost identical to using providers. Here is the equivalent code to what you've seen previously:

```
_infoItemRendererTracker.getInfoItemRenderers(MyOrder.class.getName());

String infoItemRendererClassName = MyOrderRenderer.class.getName();
_infoItemRendererTracker.getInfoItemRenderer(infoItemRendererClassName);
```

Chapter 507

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.

CHAPTER 508

FORM SERIALIZATION WITH THE DDM IO API

When a form creator saves a form in the Liferay Forms application, the Form object is *serialized* (converted) into JSON for storage in the Liferay DXP database. That's the default behavior; if you need the form in a different format, you must write your own serialization and deserialization code. Why would you want to do that? Maybe you think JSON storage is not secure, or you have another tool that can consume the form if it's in YAML. Whatever your reasons, the form can be stored in any format as long as you write a `DDMFormSerializer` and its corresponding `DDMFormDeserializer` with the proper logic.

First consider what form data looks like by default, in JSON. A simple form, *My Form*, with one text field, *Full Name*, is first created as a `DDMForm` Java object, then *serialized* into JSON for storage in the Liferay DXP database when saved.

```
{
    "availableLanguageIds":["en_US"],
    "successPage":{"body":{},
    "title":{},
    "enabled":false},
    "defaultLanguageId":"en_US",
    "fields":[{
        "autocomplete":false,
        "ddmDataProviderInstanceId":"[]",
        "dataType":"string",
        "predefinedValue":{"en_US":""},
        "tooltip":{"en_US":""},
        "readOnly":false,
        "label":{"en_US":"Full Name"},
        "type":"text",
        "required":false,
        "showLabel":true,
        "displayStyle":"singleline",
        "fieldNamespace":"",
        "indexType":"keyword",
        "visibilityExpression":"",
        "ddmDataProviderInstanceOutput":"[]",
        "repeatable":false,
        "name":"FullName",
        "options":[{"label":{"en_US":"Option"},"value":"Option"}],
        "localizable":true,
        "tip":{"en_US":""},
        "placeholder":{"en_US":""},
        "dataSourceType":"",
        "validation":{"expression":"","errorMessage":""}
    }]
```

```
}
```

From its initial state as a `DDMForm` Java object, the form is *serialized* into JSON format, and upon retrieval from the database, it's *deserialized*: the JSON object representing the form is translated back into a `DDMForm` Java object, with all its requisite fields. For example, the JSON for the above example holds each form field in the `fields` attribute. To translate this back into the necessary `DDMForm` object, first parse the data contained in the JSON object into an actual form field using your deserialization logic. Here's the logic from `DDMFormJsonDeserializer` that parses the JSON `"fields"` element into a list of `DDMFormFields`:

```
protected List<DDMFormField> getDDMFormFields(JSONArray jsonArray)
    throws PortalException {

    List<DDMFormField> ddmFormFields = new ArrayList<>();

    for (int i = 0; i < jsonArray.length(); i++) {
        DDMFormField ddmFormField = getDDMFormField(
            jsonArray.getJSONObject(i));

        ddmFormFields.add(ddmFormField);
    }

    return ddmFormFields;
}
```

Now calling `DDMForm.setDDMFormFields(ddmFormFields)` in the deserializer completes the translation process from the JSON array back to a `DDMFormField` object that the `DDMForm` needs.

If you'd like to store forms in a different format, provide custom *serialization* and *deserialization* functionality.

CHAPTER 509

SERIALIZING FORMS

The DDM IO API serializes and deserializes forms using a request/response structure. The example here creates a serializer for saving form data in YAML format. The same principles shown here apply to writing a deserializer.

To serialize form data into YAML:

1. Create a class that implements `DDMFormSerializer`:

```
@Component(immediate = true, property = "ddm.form.serializer.type=yaml") public
class DDMFormYamlSerializer implements DDMFormSerializer { .....  }
```

   The property `ddm.form.serializer.type=yaml` marks the Component so that `DDMFormSerializerTracker` can find the YAML serializer.

2. Add the serializing logic to the overridden `serialize` method. It takes a `DDMFormSerializerSerializeRequest` and returns a `DDMFormSerializerSerializeResponse` with the serialized string in it.

```
@Override public DDMFormSerializerSerializeResponse serialize(
DDMFormSerializerSerializeRequest ddmFormSerializerSerializeRequest) {

        DDMForm ddmForm = ddmFormSerializerSerializeRequest.getDDMForm();

        ...YOUR CODE FOR BUILDING A YAML OBJECT GOES HERE ...

        DDMFormSerializerSerializeResponse.Builder builder =
            DDMFormSerializerSerializeResponse.Builder.newBuilder(yamlObject.toString());

        return builder.build(); }
```

This is what you need to create your serializer. Of course, `YOUR CODE FOR BUILDING A YAML OBJECT GOES HERE` requires some explanation. While you can do whatever you want here, there are several things you really ought to do:

Add the available Language IDs: Since you have the `DDMForm` object from the request, call `ddmForm.getAvailableLocales()`.

Add the default Language ID: Get this from the `DDMForm` object by calling `ddmForm.getDefaultLocale()`.

Add the Form Fields: Get these from the `DDMForm` object by calling `ddmForm.getDDMFormFields()`.

Add any Form Rules: Get them form the `DDMForm` object with `ddmForm.getDDMFormRules()`.

Add Success Page Settings: Get these from the `DDMForm` with `ddmForm.getDDMFormSuccessPageSettings()`.
All these are done in the default form serializer, `DDMFormJSONSerializer`.
If you have the Liferay DXP source code, you can find the default serializer in

`modules/apps/dynamic-data-mapping/dynamic-data-mapping-io/src/main/java/com/liferay/dynamic/data/mapping/io/internal/DDMFormJSONSerializer.java`

You didn't create serialization code for no reason. You'll want to call it from somewhere.

509.1 Calling the Serializer

To get properly serialized form content, follow these steps:

1. Get the serializer from the `DDMFormSerializerTracker`, passing in the value of the `ddm.form.serializer.type` property.
2. Construct a `DDMFormSerializerSerializeRequest` object using its nested static `Builder` class.
3. Call the `serialize` method you wrote in the last section to create the `DDMFormSerializerSerializeResponse`, passing the `DDMFormSerializerSerializeRequest` object, via a call to the `Builder`'s `build` method.
4. Get the serialized form content from the `DDMFormSerializerSerializeResponse` by calling its `getContent` method.

Here's a code example:

```
DDMFormSerializer ddmFormSerializer =
ddmFormSerializerTracker.getDDMFormSerializer("yaml");

DDMFormSerializerSerializeRequest.Builder builder =
DDMFormSerializerSerializeRequest.Builder.newBuilder(ddmForm);

DDMFormSerializerSerializeResponse ddmFormSerializerSerializeResponse =
ddmFormSerializer.serialize(builder.build());

ddmFormSerializerSerializeResponse.getContent();
```

You can create a serializer for any format that can be saved in the database as a String. Once you create the serializer, make it the default by changing the storage format in the Form's Settings menu.

CHAPTER 510

LOCALIZATION

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 facilitates creating and using message translations and adapting to cultural conventions for user names and initials.

You can leverage Liferay's localization framework or use standard resource bundles to localize your app. The localization framework uses properties files (the same as any resource bundle) but leverages a default properties file called `Language.properties` to propagate messages (language keys) to properties files for all your locales. For example, when you add a new message to the `Language.properties` file and run Language Builder, it propagates the message to your locale files. All you must do is translate the message in each locale file, manually or automatically using Language Builder.

Language Builder integrates the Microsoft Text Translator API to translate each locale file's messages from your default locale to the respective locale. A machine's translation is no substitute for a human's, of course, but the automatic translation gives you a base to work from.

It's common to use the same messages in multiple apps. Liferay DXP provides these message sharing features:

- Liferay DXP's messages (and their translations) are available for all your apps to use. JSP tags such as `<liferay-ui:message ... />` let you use all Liferay DXP messages.
- Language modules are easy to use in all your apps. They're great for centralizing messages in all your locales.

Lastly, Liferay DXP provides settings for adapting your app to cultures:

- Naming conventions for users
- Initial conventions for user avatars
- Text direction settings (left-to-right or right-to-left)

Localization is important to all site users. Liferay helps you get it right! Start with localizing your application using Liferay's localization framework.

CHAPTER 511

LOCALIZING YOUR APPLICATION

Liferay's localization framework helps you create and use localized messages in minutes. You create your messages in a default properties file called `Language.properties` and localize them in properties files that use the convention `Language_xx.properties`, where `xx` is the locale code. After deploying your app, the messages are available to your templates. Liferay's JSP tags, such as `<liferay-ui:message ... />` display them in the user's current locale automatically, without requiring you to access `ResourceBundle` or `Locale` objects explicitly. Here are the steps:

1. Create a default language properties file called `Language.properties` in your project's resource bundle folder.

```
Project Type              | Resource Bundle Folder |
------------------------- | ---------------------- |
Bean Portlet              | `src/main/resources/content/` |
JSF Portlet               | `src/main/resources/` |
Liferay MVC Portlet       | `src/main/resources/content/` |
PortletMVC4Spring Portlet | `src/main/java/resources/content/` |
Angular Widget            | `features/localization/` |
React Widget              | `features/localization/` |
Vue.js Widget             | `features/localization/` |
```

2. Specify your language properties (language keys) using key/value pairs. For example, create a friendly greeting.

   ```
   howdy-partner=Howdy, Partner!
   ```

3. Configure your app's resource bundle and the locales you're supporting. The locales your Liferay DXP instance supports are specified in its portal properties file (here are Liferay DXP's default locales. For example, these configurations support translations for English and Spanish locales:

   `@PortletConfiguration` class annotation: Can be used in Portlet 3.0 portlets such as Bean Portlets.

```
@PortletConfiguration (
    ...
    resourceBundle="content.Language",
    supportedLocales = {"en", "es"}
)
```

Portlet descriptor `portlet.xml`: Can be used in any portlet WAR.

```
<portlet>
...
<supported-locale>en</supported-locale>
<supported-locale>es</supported-locale>
<resource-bundle>content.Language</resource-bundle>
...
</portlet>
```

`@Component` class annotation: Can be used in a portlet module such as a Liferay MVC Portlet.

```
@Component (
    ...
    property = {
        ...
        "javax.portlet.supported-locale=en",
        "javax.portlet.supported-locale=es",
        "javax.portlet.resource-bundle=content.Language"
    }
)
```

4. Create language properties for a locale. For demonstration purposes, create one manually. Automatically generating translations is discussed later.

   For example, create a Spanish translation by copying `Language.properties` to `Language_es.properties` and translating the property values to Spanish.

   ```
   howdy-partner=Hola, Compañero!
   ```

5. In your front-end template code, use the language property. For example, a JSP could use the `howdy-partner` property via the `<liferay-ui:message />` tag.

   ```
   <%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>
   ...
   <liferay-ui:message key="howdy-partner" />
   ...
   ```

```
**Tip:** The
[`liferay-ui`](https://docs.liferay.com/dxp/portal/7.2-latest/taglibs/util-taglib/liferay-ui/tld-summary.html)
tag library has several tags (e.g., `alert`, `error`, and `message`) that
accept language keys.
```

---

6. Deploy your application and view it in different locales. For example, you could view the app locally in Spanish by specifying the `es` locale code in the URL (e.g., `http://localhost:8080/es/...`).

Congratulations on a great start to localizing your application!

Next, you can explore generating translations automatically or create a language module for using language keys across applications.

511.1 Related Topics

Automatically Generating Translations
    Using Language Modules
    Using Liferay DXP's Language Settings

CHAPTER 512

USING LIFERAY'S LOCALIZATION SETTINGS

You can customize a given locale's default language settings by overriding the properties that control those settings. For instructions on this, see Overriding Language Keys. Here, you'll learn which properties correspond to common language settings.

So what all can be customized? This is an excellent question! Consider these examples:

- In the add and edit user forms, you can configure the name fields that are displayed and the field values available in select fields. For example, you can leave out the middle name field or alter the prefix selections.
- You can also control the directionality of content and messages (left to right or right to left).

Language properties exist in `Language_xx.properties` files, where `xx` represents the locale. For example, `Language_es.properties` contains the properties for Spanish, `Language_en.properties` contains the properties for English, and so on.

The default (core) language properties are in `Language.properties`. **Do not edit this file.** You can, however, open it to view the default language settings. There are two ways to do so:

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
   ```

   Open the `content` folder in the JAR to find the language files.

The first section in the core `Language.properties` file is labeled *Language Settings* and contains language properties that begin with `lang`:

```
##
## 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
```

To use the language settings mentioned here, you must have a module. See the articles 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_xx.properties` file for the locale whose properties you want to override.

512.1 Localizing User Names

Naming conventions can differ between locales. For example, users in some locales have more than one last name. You can therefore change the user name properties to fit the given locale. The properties for changing user name settings begin with `lang.user.name`.

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
  ```

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.

For most of the locales enabled by default, the user name properties are tailored to each locale. The default locales are specified by the `locales.enabled` property:

```
locales.enabled=ar_SA,ca_ES,zh_CN,nl_NL,en_US,fi_FI,fr_FR,de_DE,hu_HU,ja_JP,pt_BR,es_ES,sv_SE
```

For example, here are the English (`Language_en.properties`) properties for setting user name fields:

```
lang.user.name.field.names=prefix,first-name,middle-name,last-name,suffix
lang.user.name.prefix.values=Dr,Mr,Ms,Mrs
lang.user.name.required.field.names=last-name
lang.user.name.suffix.values=II,III,IV,Jr,Phd,Sr
```

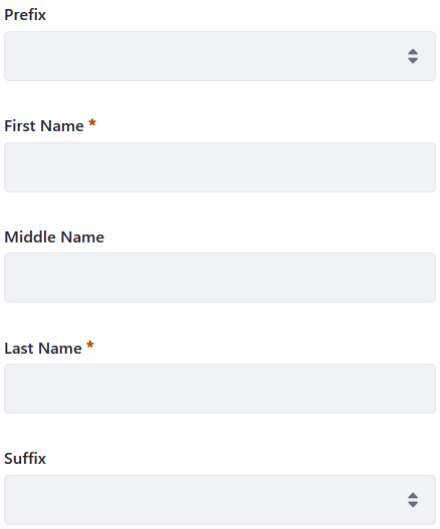

Figure 512.1: The user name settings impact the appearance of user information and forms.

Compare those to the Spanish (`Language_es.properties`) settings:

```
lang.user.name.field.names=prefix,first-name,last-name
lang.user.name.prefix.values=Sr,Sra,Sta,Dr,Dra
lang.user.name.required.field.names=last-name
```

The biggest difference between the English and Spanish form fields is that the middle name and suffix fields are omitted in the Spanish configuration. Other differences include the specific prefix values. ¡Muy excelente!

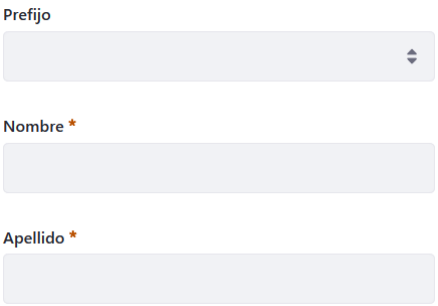

Figure 512.2: The Spanish user name settings omit the suffix and middle name fields.

512.2 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 appropriate `Language_xx.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 512.3: The user's initials are displayed for their avatar by default.

If you're leveraging the user's initials for the default avatar, you can use the `lang.user.initials.field.names` property to organize how the initials are displayed. Valid values for this property include `first-name`, `middle-name`, and `last-name`, in any order.

512.3 Right to Left or Left to Right?

The first three properties in the core `Language.properties` file's Language Settings section change the display direction of a language's characters. Most languages read from left to right, but some languages read from right to left (e.g., Arabic, Hebrew, and Persian).

Here are the relevant language properties for a right-to-left language:

```
lang.dir=rtl
lang.line.begin=right
lang.line.end=left
```

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.

512.4 Related Topics

Overriding Language Keys
Localizing Your Application
Creating a Language Module
Using a Language Module

CHAPTER 513

CREATING A LANGUAGE MODULE

You might have an application with multiple modules that provide the view layer. These modules are often called web modules. For example, this application contains three such 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, likely resulting in 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 should create a single module—a language module—for housing all your app's language keys.

Follow these steps to create a language module:

1. 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.

2. In the language module, create a `src/main/resources/content` folder. In this folder, create a `Language.properties` file for your app's default language keys. For example, such a file might look like this:

   ```
   my-app-title=My Application
   add-entity=Add Entity
   ```

3. Create any translations you want in additional language properties files, appending the locale's ID to the file name. For example, a file `Language_es.properties` holds Spanish (`es`) translations and could contain something like this:

   ```
   my-app-title=Mi Aplicación
   add-entity=Añadir Entity
   ```

   Here's the folder structure of an example language module called `my-application-lang`. This module contains the app's default language keys (`Language.properties`) and a Spanish translation (`Language_es.properties`):

```
my-application-lang/
    bnd.bnd
    src/
        main/
            resources/
                content/
                    Language.properties
                    Language_es.properties
                    ...
```

On building the language module, Liferay DXP's `ResourceBundleLoaderAnalyzerPlugin` detects the module's `Language.properties` file and adds a resource bundle capability to the module. A capability is a contract a module declares to the 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.

513.1 Related Topics

Overriding Language Keys
    Localizing Your Application
    Using Liferay's Localization Settings
    Using a Language Module

CHAPTER 514

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 here.

514.1 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 (including their language keys) along with your own.

Here's how to use this bnd instruction:

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 you wish 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.

Note: 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.

514.2 Using a Language Module from a Traditional Plugin

To use a language module from a traditional Liferay plugin you must specify the language module via the `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`, 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 the 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.

514.3 Related Topics

Localizing Your Application
    Overriding Language Keys

CHAPTER 515

AUTOMATICALLY GENERATING TRANSLATIONS

If your app uses a language module or `Language.properties` file for its user interface messages, you're in the right place. Language Builder provides these localization capabilities:

- Generating language files for each supported locale with a single command. It also propagates new default language file keys to all language files, while keeping their translated values intact.
- Generating translations automatically using Microsoft's Translator Text API. This gives you a jump start on creating translations.

Note: Language Builder is available as a plugin for projects that use Gradle or Maven.

Start with Configuring the Language Builder plugin.

515.1 Configuring the Language Builder Plugin

Configure the Language Builder plugin for Gradle or Maven.

Gradle:

```
buildscript {
    dependencies {
        classpath 'com.liferay:com.liferay.gradle.plugins.lang.builder:latest.release'
    }

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

apply plugin: "com.liferay.lang.builder"

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

```
    }
}
```

Maven:

```
<project>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>com.liferay</groupId>
                <artifactId>com.liferay.lang.builder</artifactId>
                <version>1.0.30</version>
                <configuration>
                    <langDirName>.</langDirName>
                    <translateClientId>${microsoft.translator.client.id}</translateClientId>
                    <translateClientSecret>${microsoft.translator.client.secret}</translateClientSecret>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

Now you can invoke Language Builder in your project.

515.2 Running Language Builder

When you run Language Builder, it generates properties files for all your locales and propagates all new language properties from `Language.properties` to your locale language files (newly generated and existing). Additionally if you configured the Microsoft Translator Text API (discussed next), Language Builder translates your locale language properties.

Here's the command:

Gradle:

```
gradlew buildLang
```

Maven:

```
mvn lang-builder:build
```

Tip: Run Language Builder to update your locale files each time you change your `Language.properties` file.

Note, until you configure translation credentials (discussed next), Language Builder prints this message:

```
Translation is disabled because credentials are not specified
```

If you want to configure your app to generate automatic translations using the Microsoft Translator Text API, keep reading.

515.3 Translating Language Keys Automatically

If you've configured the Language Builder plugin (above) in your project, you're well on your way to translating language keys automatically. Now you have to configure Microsoft's Translator Text API so you can translate language keys automatically.

Important: Lang Builder does not translate language keys containing HTML (e.g., `<em>`, `<b>`, `<code>`, etc.). Default language keys that contain HTML are only *copied* to your locale 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.

Here's how to set up the translator and generate translations.

1. Generate a translation subscription key for the Microsoft Translator Text API. Follow the instructions here.

2. Add your client credentials to the Language Builder plugin configuration. For security reasons, pass the credentials to a property that's stored in your local build environment (e.g., see the Gradle environment guide).

   Gradle:

   Make sure the `buildLang` task knows to use your subscription key for translation by setting the `translateSubscriptionKey` property:

   ```
   buildLang {
      translateSubscriptionKey = langTranslateSubscriptionKey
   }
   ```

   Here's the entire `build.gradle` example code,

   ```
   buildscript {
       dependencies {
           classpath 'com.liferay:com.liferay.gradle.plugins.lang.builder:latest.release'
       }

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

   apply plugin: "com.liferay.lang.builder"

   buildLang {
      translateSubscriptionKey = langTranslateSubscriptionKey
   }

   repositories {
       maven {
   ```

```
        url "https://repository.liferay.com/nexus/content/repositories/liferay-public-releases/"
    }
}
```

Maven:

Set the following Language Builder plugin `<translateClientId />` and `<translateClientSecret />` configuration elements using Maven build environment properties:

```
<configuration>
    <langDirName>.</langDirName>
    <translateClientId>${microsoft.translator.client.id}</translateClientId>
    <translateClientSecret>${microsoft.translator.client.secret}</translateClientSecret>
</configuration>
...
```

Here's the entire `pom.xml` example code,

```
<project>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>com.liferay</groupId>
                <artifactId>com.liferay.lang.builder</artifactId>
                <version>1.0.30</version>
                <configuration>
                    <langDirName>.</langDirName>
                    <translateClientId>${microsoft.translator.client.id}</translateClientId>
                    <translateClientSecret>${microsoft.translator.client.secret}</translateClientSecret>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

3. Run Language Builder.

   Gradle:

   ```
   gradlew buildLang
   ```

   Maven:

   ```
   mvn lang-builder:build
   ```

Great! You can now generate language files and provide automatic translations of your language keys.

515.4 Related Topics

Localizing Your Application
    Using Language Modules
    Gradle Language Builder Plugin
    Maven Language Builder Plugin
    Liferay Workspace
    Tooling

CHAPTER 516

PORTLETS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Liferay DXP started off as a portal server, designed to serve Java-based web applications called *portlets* (see JSR 168, JSR-286, and JSR-362). Portlets process requests and generate responses like any other web application. One key difference, however, 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 need only 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. Portlets run only in a portal server. They 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.

Many portlets can be placed on a single page by users (if they have permission) or portal administrators. 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. You can drag and drop to reposition and resize portlets on a page without altering any portlet code. Alternatively, a single portlet can take up an entire page if it's the only app you need on that page. For example, message boards or Wikis with complex user interfaces are best suited on their own pages. In short, portlets alleviate many of the traditional pain points associated with developing Java-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 content based on its current state. When this phase runs on one portlet, it also runs on all other portlets on the page. The Render phase runs when any portlets on the page complete the Action or Event phases.
- **Action:** In response to a user action, the Action phase performs 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-portlet communication (IPC). Once the portlet processes all events, the portal calls the Render phase on all portlets on the page.

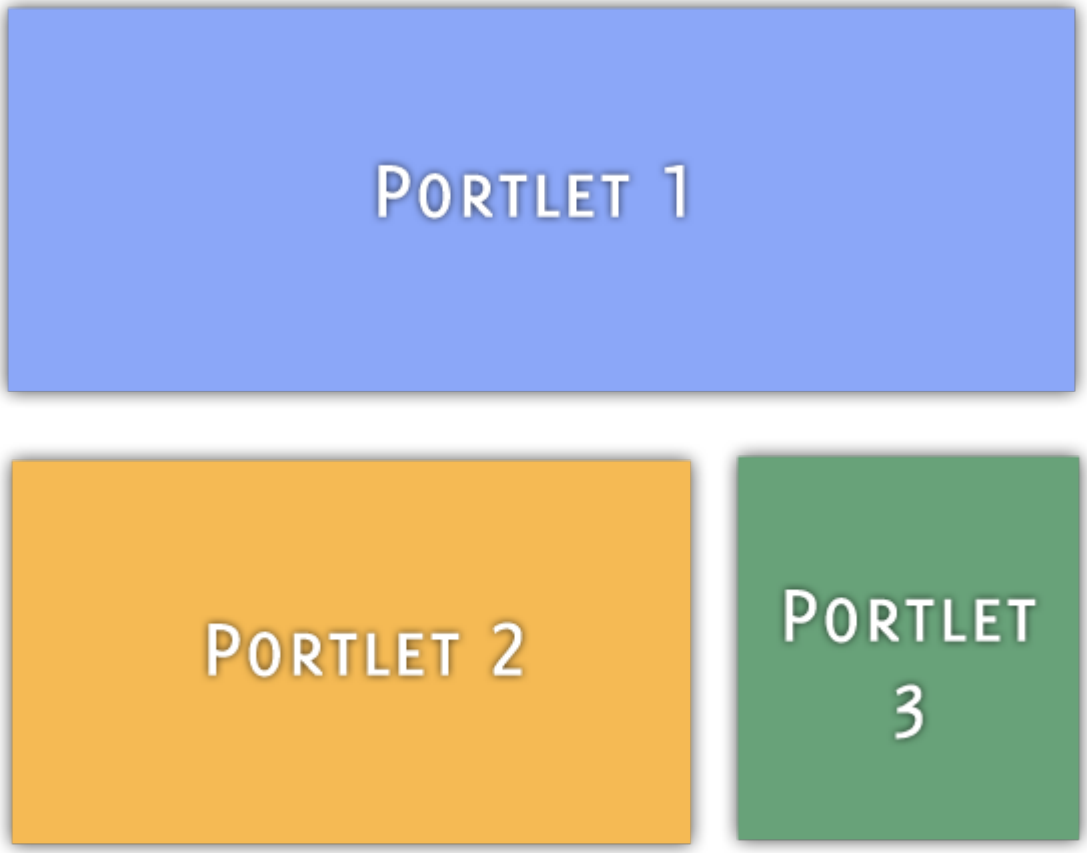

Figure 516.1: You can place multiple portlets on a single page.

- **Resource-serving:** Serves a resource independently 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.
- **Header:** Lets you specify resource dependencies, such as CSS, prior to the Render phase.

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 provides a mechanism for a portlet to read only 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 might 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 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.

All of the ways to develop web front-ends on Liferay DXP involve portlets. The JavaScript-based widgets use Liferay's JS Portlet Extender behind the scenes and the Java-based web front-ends are explicitly portlets. All of the web front-end types vary in their support of Portlet 3.0, dependency injection (DI), Model View Controller (MVC), and modularity, giving you plenty of good options for developing portlets.

516.1 Related Topics

Spring Portlet MVC: PortletMVC4Spring
    Liferay MVC Portlet
    JSF Portlet
    Portlet 3.0 API Opt In

CHAPTER 517

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?

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. In this section, you'll learn how to prepare your JavaScript files to leverage ECMAScript and npm features in your portlets.

CHAPTER 518

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.

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!

518.1 Related Topics

- Customizing JSPs
- Web Services

CHAPTER 519

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.

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 to achieve dependency isolation. The bundler injects the dependencies in the `package.json` pertaining to the module to make them available at runtime.

This section covers how to set up npm-based portlet projects.

CHAPTER 520

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 article shows how to use the liferay-npm-bundler to set up npm-based portlet projects.

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 this 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"
   }
   ```

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). See the liferay-npm-bundler's `.npmbundlerrc` structure reference for more information on the available options. Specify the loaders and rules to use for your project's source files. The example below processes the JavaScript files in the project's `/src/` and `/assets/` folders with Babel via the `babel-loader`:

```
{
  "sources": ["src", "assets"],
  "rules": [
    {
      "test": "\\.js$",
      "exclude": "node_modules",
      "use": [
        {
          "loader": "babel-loader",
          "options": {
            "presets": ["env"]
          }
        }
      ]
    }
  ]
}
```

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
```

Note: By default, the AMD Loader times out in seven seconds. 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*.

Great! Now you know how to use the liferay-npm-bundler to bundle your npm-based portlets for the Liferay AMD Loader.

520.1 Related Topics

- Preparing Your JavaScript Files for ES2015+
- Understanding liferay-npm-bundler's Loaders and Rules

CHAPTER 521

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.

CHAPTER 522

MIGRATING A PLAIN JAVASCRIPT, BILLBOARD JS, JQUERY, METAL JS, REACT, OR VUE JS PROJECT TO USE BUNDLER 2.X

After following the steps covered in the intro to this section, 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`.

Awesome! Your project is migrated to use the new version of the liferay-npm-bundler.

522.1 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

CHAPTER 523

MIGRATING AN ANGULAR PROJECT TO USE BUNDLER 2.X

After following the steps covered in the intro to this section, 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"
```

Great! Your project is migrated to use the new version of the liferay-npm-bundler.

523.1 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

CHAPTER 524

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 runs the whole process, like webpack, and is 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 use only 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 AMD loader, they are not compatible.
```

524.1 Related Topics

- Default liferay-npm-bundler Loaders
- Understanding liferay-npm-bundler's Loaders

CHAPTER 525

CREATING CUSTOM LOADERS FOR THE LIFERAY-NPM-BUNDLER

Since webpack creates JavaScript 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 containing 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 that 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 and 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"]
          }
        }
      ]
    }
  ],
  ...
}
```

525.1 Related Topics

- Default liferay-npm-bundler Loaders
- Understanding liferay-npm-bundler's Loaders

CHAPTER 526

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, read the topics in this section to learn how to retrieve your OSGi bundle's npm package and module information.

CHAPTER 527

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!

527.1 Related Topics

- Obtaining an OSGi bundle's Dependency npm Package Descriptors
- liferay-npm-bundler
- How Liferay DXP Publishes npm Packages

CHAPTER 528

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 article 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!

528.1 Related Topics

- Referencing an npm Module's Package
- The Structure of OSGi Bundles Containing npm Packages
- How Liferay DXP Publishes npm Packages

CHAPTER 529

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!** If Spa is disabled, ensure that the `com.liferay.frontend.js.spa.web-[version]` module is deployed and active.

529.1 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.

529.2 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. SennaJS exposes lifecycle events that represent state changes in the application. See Available SPA Lifecycle Events for more information.

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.

This section covers these topics:

- Configuring SPA System Settings
- Disabling SPA
- Specifying how resources are loaded during SPA navigation
- Detaching Global Listeners

CHAPTER 530

CONFIGURING SPA SYSTEM 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, see Disabling SPA.

Follow these steps to configure system settings for SPA:

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.

Great! Now you know how to configure system settings for SPA.

530.1 Related Topics

- Disabling SPA
- Specifying How Resources Are Loaded During SPA Navigation
- Detaching Global Listeners

CHAPTER 531

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. You can disable SPA at these scopes:

- disable SPA across an entire Liferay DXP instance
- disable SPA in a portlet
- Disable SPA in individual elements

Follow the steps in the corresponding section to disable SPA for that scope.

531.1 Disabling SPA across an Instance

To disable SPA across an entire Liferay DXP instance, add the following line to your `portal-ext.properties`:

```
javascript.single.page.application.enabled = false
```

531.2 Disabling SPA for a Portlet

To disable SPA for a portlet, you must blacklist it. To blacklist a portlet from SPA, follow these steps:

1. Open your portlet class.
2. Set the `com.liferay.portlet.single-page-application` property to false:

   ```
   com.liferay.portlet.single-page-application=false
   ```

   If you prefer, you can set this property to false in your `liferay-portlet.xml` instead by adding the following property to the `<portlet>` section:

   ```
   <single-page-application>false</single-page-application>
   ```

3. Alternatively, you can override the `isSinglePageApplication` method of the portlet to return `false`.

531.3 Disabling SPA for an Individual Element

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`. See the example below:

```
<a data-senna-off="true" href="/pages/page2.html">Page 2</a>
```

Nice! Now you know how to disable SPA in your app.

531.4 Related Topics

- Configuring SPA System Settings
- Specifying How Resources Are Loaded During SPA Navigation
- Detaching Global Listeners

CHAPTER 532

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" />
```

Great! Now you know how to specify how resources are loaded during SPA navigation.

532.1 Related Topics

- Configuring SPA System Settings
- Disabling SPA
- Detaching Global Listeners

CHAPTER 533

Detaching Global Listeners

SPA provides several improvements that highly benefit your site and users, but with great power comes great re-SPA-nsibility. I apologize for that last sentence, but the fact remains that there is potentially some additional maintenance as a consequence of SPA. 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.

Follow these steps to prevent potential problems:

1. Listen to the navigation event in order to detach your listeners. For example, you would use the following code to listen to a global `category` event and `destroyPortlet` event:

```
Liferay.on('category', function(event){...});
Liferay.on('destroyPortlet', function(event){...});
```

2. Detach the event listeners when the portlet is destroyed, as shown in the example below:

```
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);
```

Great! Now you know how to properly maintain your global listeners for SPA.

533.1 Related Topics

- Available SPA Lifecycle Events
- Disabling SPA
- Configuring SPA System Settings

Chapter 534

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. This section explains how to apply Clay's design patterns to achieve the same look and feel as Portal's built-in apps.

This section covers these topics:

- Applying Clay to navigation
- Implementing the Management Toolbar

CHAPTER 535

APPLYING CLAY PATTERNS TO NAVIGATION

This article 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:

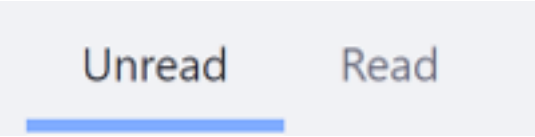

Figure 535.1: The navigation bar should be light for apps on the live site.

Admin app navigation bar:

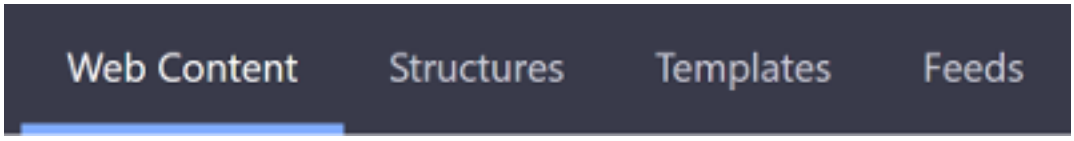

Figure 535.2: The navigation bar should be dark (inverted) in admin apps.

Sweet! Now you know how to style a navigation bar with Clay.

535.1 Related topics

Implementing the Management Toolbar

Chapter 536

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 taglib reference covers how to use the Clay taglibs to create the Management Toolbar. This section covers how to create the features below for the Management Toolbar:

- Implementing View Types
- Sorting and Filtering Items

CHAPTER 537

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.

- **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 %>"
    viewTypeItems="<%=
        new JSPViewTypeItemList(pageContext, baseURL, selectedType) {
            {
                addCardViewTypeItem(
                    viewTypeItem -> {
                        viewTypeItem.setActive(true);
                        viewTypeItem.setLabel("Card");
                    });

                addListViewTypeItem(
                    viewTypeItem -> {
                        viewTypeItem.setLabel("List");
                    });

                addTableViewTypeItem(
                    viewTypeItem -> {
                        viewTypeItem.setLabel("Table");
                    });
            }
        }
    %>"
/>
```

    `viewTypeItems`: The available view types

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 follow the instructions in the rest of this section to configure them.

CHAPTER 538

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 reference for examples and use cases of each card.

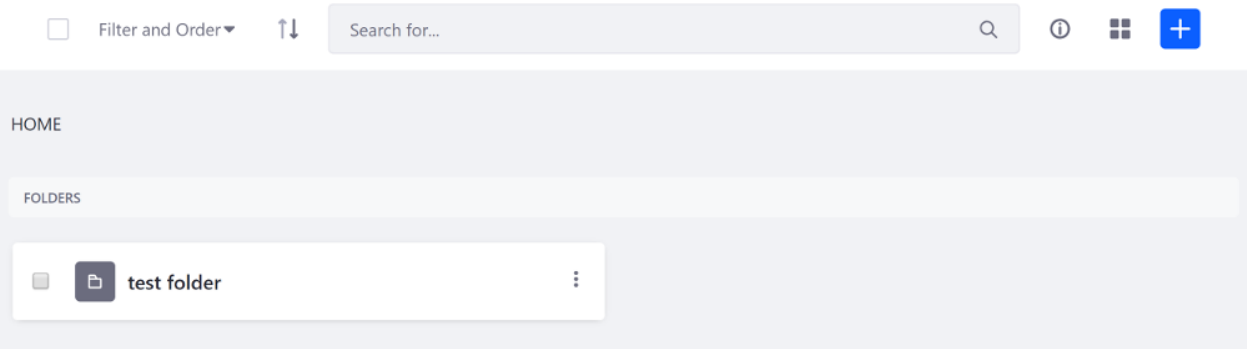

Figure 538.1: The Management Toolbar's card view gives a quick summary of the content's description and status.

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 %>"
/>
```

Great! Now you know how to implement the Card view!

538.1 Related Topics

- Configuring the Clay Management Toolbar Taglib
- Filtering and Sorting Items with the Management Toolbar

CHAPTER 539

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.

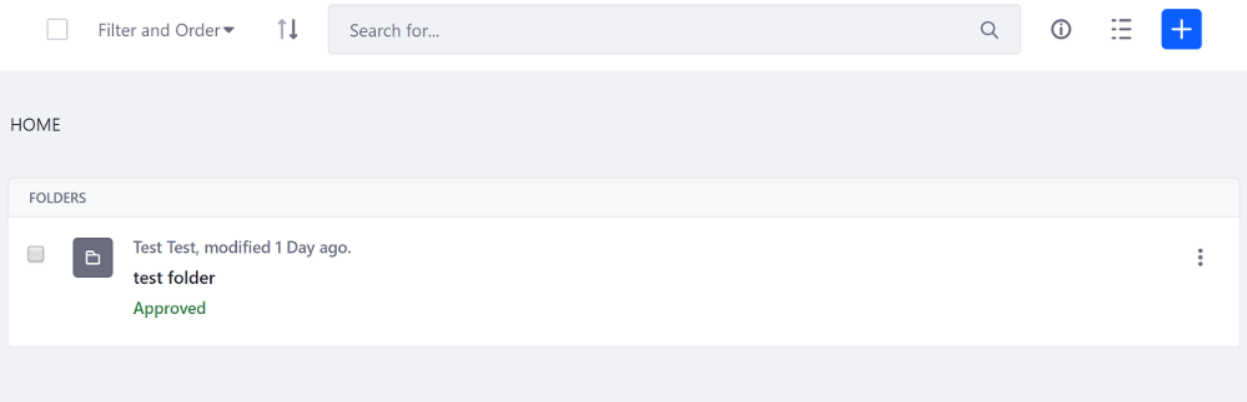

Figure 539.1: The Management Toolbar's list view gives the content's full description.

Follow these steps to configure the List view:

1. 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>
```

2. Follow the example below to add the three columns to the conditional block:

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"/>

Great! Now you know how to implement the List view!

539.1 Related Topics

- Configuring the Clay Management Toolbar Taglib
- Filtering and Sorting Items with the Management Toolbar

CHAPTER 540

IMPLEMENTING THE TABLE VIEW

The table view list the search container columns from left to right.

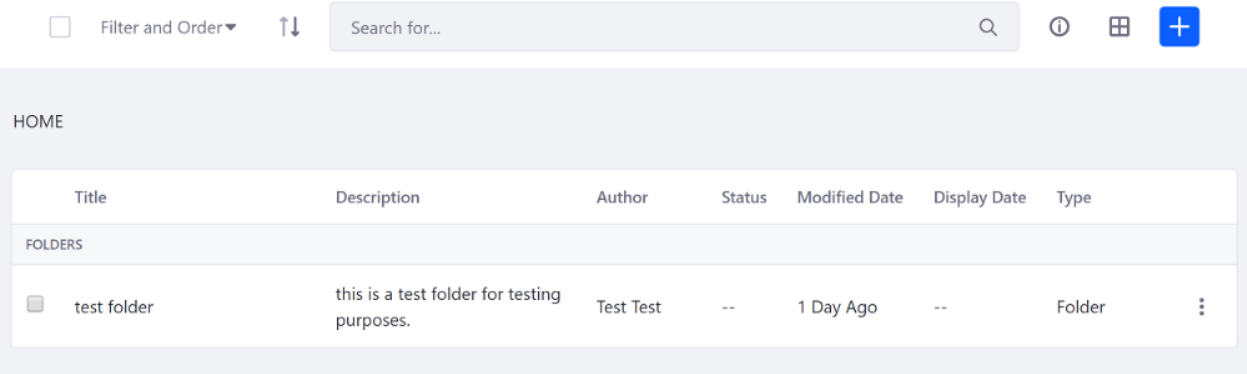

Figure 540.1: The Management Toolbar's table view list the content's information in individual columns.

Follow these steps to configure the Table view:

1. 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>
```

2. Follow the example below to add the required columns to the conditional block:

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" />

Great! Now you know how to implement the Table view!

540.1 Related Topics

- Configuring the Clay Management Toolbar Taglib
- Filtering and Sorting Items with the Management Toolbar

CHAPTER 541

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. Follow these steps:

1. 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.

2. Save the changes.

Great! You've updated your search iterator to use your display styles.

541.1 Related Topics

- Configuring the Clay Management Toolbar Taglib
- Filtering and Sorting Items with the Management Toolbar

CHAPTER 542

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.

542.1 Related Topics

- Configuring Filtering and Sorting Management Toolbar Attributes
- Implementing the View Types

CHAPTER 543

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 543.1: 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.

543.1 Related Topics

- Applying Clay Patterns to Your Navigation Bar
- Setting Empty Results Messages

CHAPTER 544

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:

```
<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.

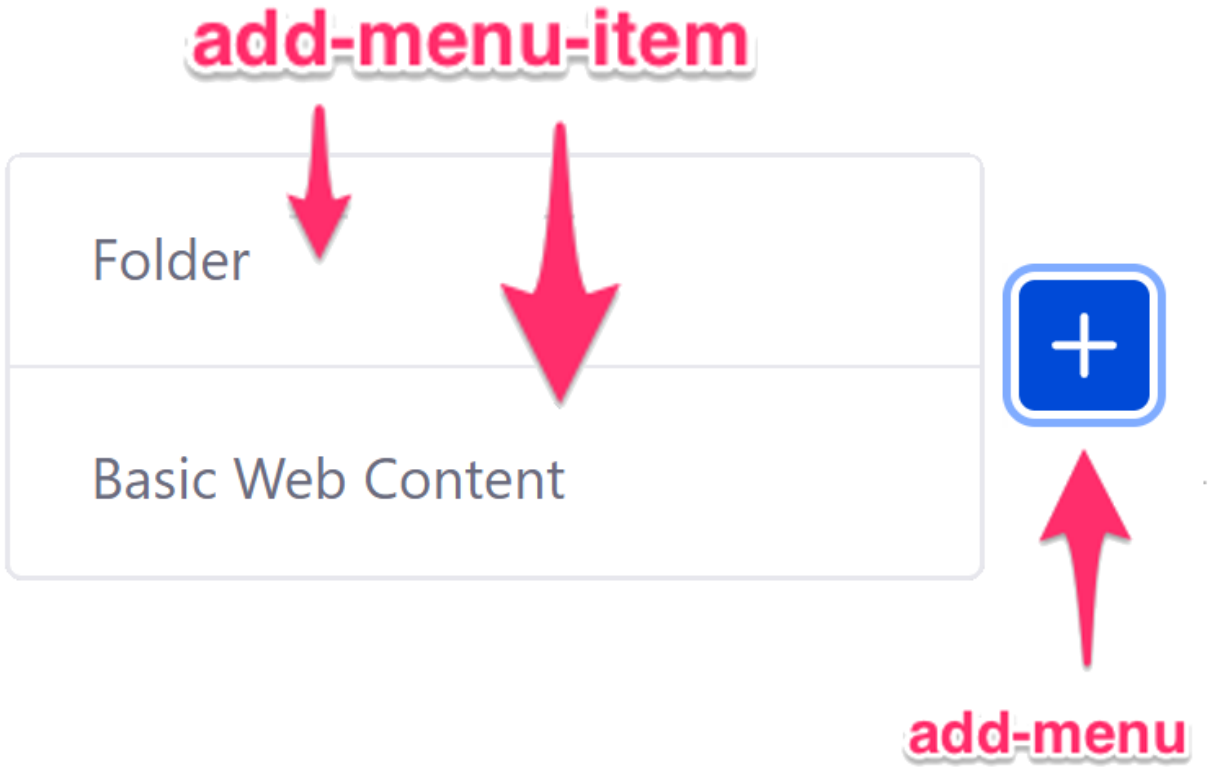

Figure 544.1: The add button pattern consists of an `add-menu` tag and at least one `add-menu-item` tag.

544.1 Related Topics

- Setting Empty Results Messages
- Implementing the Management Toolbar

CHAPTER 545

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:

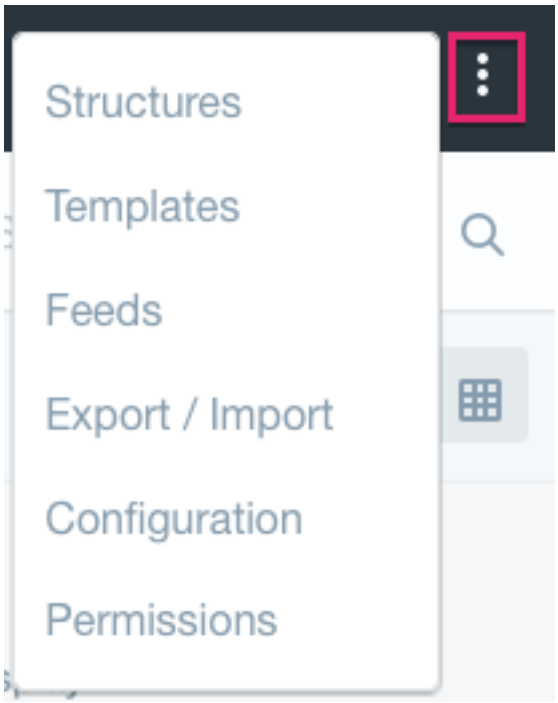

Figure 545.1: The upper right ellipsis menu contains most of the actions for the app.

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.

545.1 Related Topics

- Applying Clay Patterns to Your Navigation Bar
- Configuring Your Application's Title and Back Link

CHAPTER 546

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.

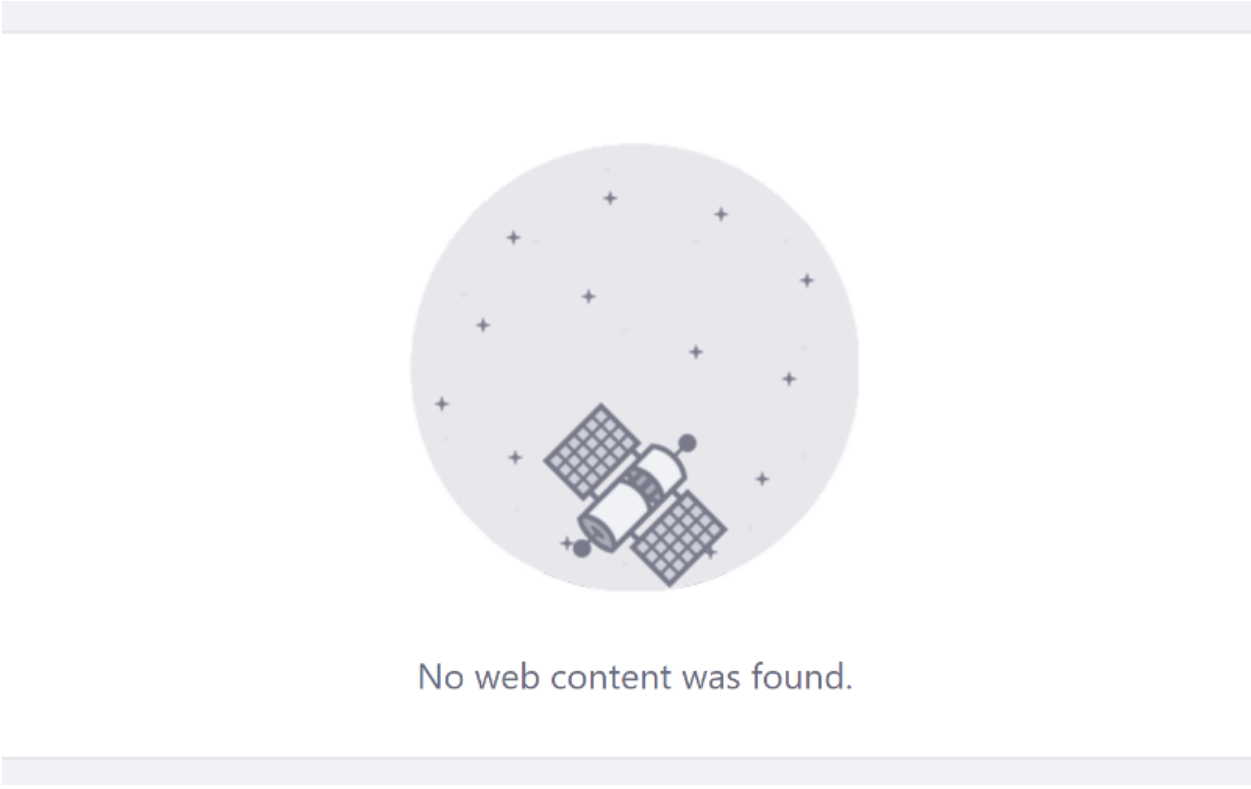

Figure 546.1: This is a still frame from the Web Content portlet's empty results animation.

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:

```markup
<liferay-frontend:empty-result-message
    actionDropdownItems="<%= (availableSegmentsEntries.size() > 0) ?
    editAssetListDisplayContext.getAssetListEntryVariationActionDropdownItems() : null %>"
    animationType="<%= EmptyResultMessageKeys.AnimationType.NONE %>"
    componentId='<%= renderResponse.getNamespace() + "emptyResultMessageComponent" %>'
    description='<%= LanguageUtil.get(request, "no-personalized-variations-were-found") %>'
    elementType='<%= LanguageUtil.get(request, "personalized-variations") %>'
/>
```
```

```
 **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/7.2.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.
```

---

empty_state.gif:

Figure 546.2: If you can use the add button to add entities to the app, use the empty state animation.

search_state.gif:

Figure 546.3: If you can use the add button to add entities to the app, use the search state animation.

success_state.gif:

Figure 546.4: If you can use the add button to add entities to the app, use the success state animation.

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.

546.1 Related Topics

- Using the Liferay Front-End Taglib
- Applying the Add Button Pattern

CHAPTER 547

SEARCH

Liferay DXP contains a search engine based on Elasticsearch. You can extend it, implement search for your own applications, and it's highly configurable.

547.1 Basic Search Concepts

Indexing: During indexing, a document is sent to the search engine. This document contains fields of various types (string, etc.). The search engine processes each field and determines whether to store the field or analyze it. 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. Then it passes each token through a series of analyzers, which 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 might contain both queries and filters (more on this later). Queries and filters specify a field to search within and the value to match against. The search engine iterates through each field within the nested queries and filters and may perform special analysis prior to executing the query (search time analysis). Search time analysis can be configured for each field (see Mapping Definitions).

547.2 Mapping Definitions

Mappings control how a search engine processes a given field. For instance, if a field name ends in "es_ES", it should be processed as Spanish, removing any common Spanish words like "si".

In Elasticsearch and Solr, the two supported search engines for Liferay DXP, mappings are defined in `liferay-type-mappings.json` and `schema.xml`, respectively.

The Elasticsearch mapping JSON file is 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` is 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.

You can customize these mappings to fit your needs. For example, you might want to use a special analyzer for a custom inventory number field.

547.3 Liferay Search Infrastructure

Search engines already provide native APIs, but Liferay wraps the engine with a search infrastructure that 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. Summarize returned results.

That's just a taste of Liferay's Search Infrastructure. Continue reading to learn more.

CHAPTER 548

AGGREGATIONS

Aggregations take the results of a query and group the data into logical sets. Aggregations can be composed to provide complex data summaries.

7.0 has a new API that exposes Elasticsearch's native Aggregation functionality.

Currently, these aggregation types are supported:

- Bucketing aggregations create buckets of documents based on some criterion. They support sub-aggregations.
  - Supported bucket aggregations include children aggregations, date histogram aggregations, date range aggregations, diversified sampler aggregations, filter aggregations, filters aggregations, geo distance aggregations, geo hash grid aggregations, global aggregations, histogram aggregations, missing aggregations, nested aggregations, range aggregations, reverse nested aggregations, sample aggregations, significant terms aggregations, significant text aggregations, and terms aggregations.
- Metrics aggregations compute metrics over a set of documents.
  - Supported metrics aggregations include average aggregations, cardinality aggregations, extended stats aggregations, geo bounds aggregations, geo centroid aggregations, max aggregations, min aggregations, percentile ranks aggregations, percentiles aggregations, scripted metric aggregations, stats aggregations, sum aggregations, top hits aggregations, value count aggregations, and weighted average aggregations.
- Pipeline aggregations aggregate the output of other aggregations and their associated metrics.
  - Supported pipeline aggregations include average bucket pipeline aggregations, bucket metrics pipeline aggregations, bucket script pipeline aggregations, bucket selector pipeline aggregations, bucket sort pipeline aggregations, cumulative sum pipeline aggregations, derivative pipeline aggregations, extended stats bucket pipeline aggregations, max bucket pipeline aggregations, min bucket pipeline aggregations, moving function pipeline aggregations, percentiles bucket pipeline aggregations, pipeline aggregations, serial diff pipeline aggregations, stats bucket pipeline aggregations, and sum bucket pipeline aggregations.

All the supported aggregations are found in the `portal-search-api` module's `com.liferay.portal.search.aggregati` package.

In addition to these aggregations, other aggregation-like features are present in the Liferay DXP search API:

Group By collects search results (documents) based on a particular field. For example, if you want to group the search results based on the asset type (e.g., web content article, document, blog post, etc.), you can create a search query that contains a com.liferay.portal.kernel.search.GroupBy aggregation with the field `entryClassName`.

You can specify these attributes for returned groups:

- The maximum number of results in each group
- Special sorting for the grouped results

Facets act like bucket aggregations, holding results that share a certain characteristic.

548.1 Using Aggregations

The generalized approach for using aggregations in your own search code is like this:

1. Instantiate and construct the aggregation object
2. Add the aggregation information to the search request
3. Process the search response

These steps are covered in more detail (with examples) here.

548.2 External References

- https://www.elastic.co/guide/en/elasticsearch/reference/7.x/search-aggregations.html
- https://www.elastic.co/guide/en/elasticsearch/reference/7.x/search-aggregations.html#_structuring_aggregations

548.3 Search Engine Connector Support

- Elasticsearch 6: Yes
- Solr 7: No

548.4 New/Related APIs

API (FQCN)	Provided by Artifact	Notes
`com.liferay.portal.search.aggregation.*`	com.liferay.portal.search.api	The whole “aggregation” package is new as of 7.0

CHAPTER 549

CREATING AGGREGATIONS

Each aggregation has a different purpose and should be processed differently once returned from the search engine, but you add the aggregation information to the search request in a similar way between all aggregations.

549.1 Instantiate and Construct the Aggregation

1. Use the `com.liferay.portal.search.aggregation.Aggregations` to instantiate the aggregation you'll construct. For example,

   ```
   PercentilesAggregation percentilesAggregation =
       aggregations.percentiles("percentiles", Field.PRIORITY);
   ```

   To discover what fields each aggregation must have (e.g., `Sting name, String field` in the case of the above `PercentilesAggregation`), see the `Aggregations` interface.

2. Build out the aggregation to get the desired response. This looks different for each aggregation type, but Elasticsearch's documentation on the aggregation type explains it well (such as Percentiles Aggregations) combined with the setters in Liferay's corresponding interface.

   For example, use the `setPercentilesMethod(PercentilesMethod.HDR)` method to use the High Dynamic Range Histogram for calculating the percentiles.

   ```
   percentilesAggregation.setPercentilesMethod(PercentilesMethod.HDR);
   ```

Once the aggregation itself is in good shape, feed it to the search query.

549.2 Build the Search Query

1. Get an instance of `com.liferay.portal.search.searcher.SearchRequestBuilder` from the `SearchRequestBuilderFactory` service:

```
SearchRequestBuilder searchRequestBuilder =
    searchRequestBuilderFactory.builder();
```

2. Get a `com.liferay.portal.search.searcher.SearchRequest` instance from the builder, then add the aggregation to it and run its `build` method:

```
SearchRequest searchRequest =
    searchRequestBuilder.addAggregation(percentilesAggregation).build();
```

549.3 Execute the Search Query

1. Perform a search using the `Searcher` service and the `SearchRequest` to get a `com.liferay.portal.search.searcher.`

```
SearchResponse searcher.search(searchRequest);
```

2. To satisfy the dependencies of the example code here, get a reference to `com.liferay.portal.search.searcher.S` and `com.liferay.portal.search.searcher.Searcher`:

```
@Reference
protected Searcher searcher;

@Reference
SearchRequestBuilderFactory searchRequestBuilderFactory;
```

549.4 Process the response

What you'll do with the `SearchResponse` returned by the `searcher.search` call depends on the type of aggregation and your specific use case. A separate article will be written to demonstrate how to process the response.

CHAPTER 550

STATISTICAL AGGREGATIONS

Support for GroupBy and Stats aggregations were introduced in 7.0.

Cardinality Aggregations extend Liferay DXP's metrics aggregation capabilities, providing an approximate (i.e., statistical) count of distinct values returned by a search query. For example, you could compute a count of distinct values of the *tag* field. Refer to the Elasticsearch documentation for more details.

While this functionality was available in the past directly in the portal kernel code, it's been extracted and re-implemented in `StatsRequest` and `StatsResponse`, both introduced in the `com.liferay.portal.search.api` module to avoid modifying `portal-kernel`. `StatsRequest` provides the same statistical features that the legacy `com.liferay.portal.kernel.search.Stats` does, and adds the new cardinality option.

550.1 StatsRequest

The `StatsRequest` Provides a map of field names and the metric aggregations that are to be computed for each field.

1. Get a reference to `com.liferay.portal.search.searcher.SearchRequestBuilderFactory`:

```
@Reference
SearchRequestBuilderFactory searchRequestBuilderFactory;
```

2. Get an instance of `com.liferay.portal.search.searcher.SearchRequestBuilder`:

```
SearchRequestBuilder searchRequestBuilder = searchRequestBuilderFactory.builder();
```

3. Get a `com.liferay.portal.search.searcher.SearchRequest` instance from the builder:

```
SearchRequest searchRequest = searchRequestBuilder.build();
```

4. Get a reference to `com.liferay.portal.search.stats.StatsRequestBuilderFactory`:

```
@Reference
StatsRequestBuilderFactory statsRequestBuilderFactory;
```

5. Get a `com.liferay.portal.search.stats.StatsRequestBuilder` instance and build `com.liferay.portal.search.sta` with the desired metrics:

```
StatsRequestBuilder statsRequestBuilder =
    statsRequestBuilderFactory.getStatsRequestBuilder();
StatsRequest statsRequest = statsRequestBuilder
    .cardinality(true)
    .count(true)
    .field(field)
    .max(true)
    .mean(true)
    .min(true)
    .missing(true)
    .sum(true)
    .sumOfSquares(true)
    .build();
```

6. Set `StatsRequest` on the `SearchRequest`:

```
searchRequest.statsRequests(statsRequest);
```

7. Get a reference to `com.liferay.portal.search.searcher.Searcher`:

```
@Reference
protected Searcher searcher;
```

8. Perform a search using `Searcher` and `SearchRequest` to get `com.liferay.portal.search.searcher.SearchResponse`:

```
SearchResponse searcher.search(searchRequest);
```

Click here to see an example from Liferay's codebase

550.2 StatsResponse

The stats response contains the metrics aggregations computed by the search engine for a given field.

1. Get the map containing the metrics aggregations computed by the search engine:

```
Map<String, StatsResponse> map = searchResponse.getStatsResponseMap();
```

2. Get the `StatsResponse` for a given field:

```
StatsResponse statsResponse = map.get(field);
```

3. Get the desired metric, for example *cardinality*:

```
statsResponse.getCardinality();
```

Click here to see an example from Liferay's codebase

550.3 Using the Legacy `Stats` Object

The legacy `com.liferay.portal.kernel.search.Stats` object continues to be fully supported:

1. Create a `Stats` instance with the desired metrics:

```
Stats stats = new Stats() {
    {
        setCount(true);
        setField(field);
        setMax(true);
        setMean(true);
        setMin(true);
        setSum(true);
        setSumOfSquares(true);
    }
};
```

2. Set `Stats` on the `SearchContext`:

```
searchRequestBuilder.withSearchContext(searchContext -> searchContext.addStats(stats));
```

Click here to see an example from Liferay's codebase

550.4 External References

- https://www.elastic.co/guide/en/elasticsearch/reference/7.x/search-aggregations-metrics.html
- https://www.elastic.co/guide/en/elasticsearch/reference/7.x/search-aggregations-metrics-cardinality-aggregation.html
- https://lucene.apache.org/solr/guide/7_5/the-stats-component.html

550.5 Search Engine Connector Support

- Elasticsearch 6: Yes
- Solr 7: Yes

550.6 New/Related APIs

These are the relevant APIs for building Statistics Aggregations:

API (FQCN) | Provided by Artifact | `com.liferay.portal.search.searcher.SearchRequestBuilder#statsRequests(Sta
statsRequests)` | `com.liferay.portal.search.api` `com.liferay.portal.search.searcher.SearchResponse#getStatsRespon
| `com.liferay.portal.search.api` **`com.liferay.portal.search.stats.StatsRequest`** | `com.liferay.portal.search.api`
`com.liferay.portal.search.stats.StatsRequestBuilder` | `com.liferay.portal.search.api` `com.liferay.portal.search.s
| `com.liferay.portal.search.api` **`com.liferay.portal.search.stats.StatsResponse`** | `com.liferay.portal.search.api`
`com.liferay.portal.kernel.search.Stats` | `portal-kernel`

550.7 Deprecated APIs

- SearchSearchRequest#getStats()
- SearchSearchRequest#setStats(Map<String, Stats> stats)

CHAPTER 551

MODEL ENTITY 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. This paradigm applies to Liferay's own developers and anyone developing model entities for use with Liferay DXP.

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>`. Liferay DXP version 7.1 introduced a new pattern that relies on composition instead of inheritance. That said, if you want to use the old approach, feel free to extend `BaseIndexer`. It's still supported.

551.1 Search and Indexing Overview

Starting with the 7.0 version of Liferay DXP, the Search API has become less tied to Lucene. 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:

- Ensure all indexed documents include some required fields (e.g., `entryClassName`, `entryClassPK`, `assetTagNames`, `assetCategories`, `companyId`, `groupId`, staging status).
- Ensure the scope of returned search results is appropriate by applying the right filters to search requests.
- Provide permission checking and hit summaries for display in the built-in search application.

551.2 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	BlogsEntrySearchRegistrar
setDefaultSelectedFieldNames	SearchRegistrar.activate	BlogsEntrySearchRegistrar
setDefaultSelectedLocalizedFieldNames	SearchRegistrar.activate	BlogsEntrySearchRegistrar
setPermissionAware	ModelResourcePermissionRegistrar	DLFileEntryModelResourcePermissionR
setFilterSearch	ModelResourcePermissionRegistrar	DLFileEntryModelResourcePermissionR
getDocument/doGetDocument	ModelDocumentContributor	BlogsEntryModelDocumentContributor
reindex/doReindex	ModelIndexerWriterContributor	BlogsEntryModelIndexerWriterContrib
addRelatedEntryFields	RelatedEntryIndexer	DLFileEntryRelatedEntryIndexer
postProcessContextBooleanFilter/PostProcessContextQuery	ModelPreFilterContributor	BlogsEntryModelPreFilterContributor
postProcessSearchQuery	KeywordQueryContributor	BlogsEntryKeywordQueryContributor
getFullQuery	SearchContextContributor	DLFileEntryModelSearchContextContri
isVisible/isVisibleRelatedEntry	ModelVisibilityContributor	BlogsEntryModelVisibilityContributo
getSummary/createSummary/doGetSummary	ModelSummaryContributor	BlogsEntryModelSummaryContributor
Indexer.search/searchCount	No change	BlogEntriesDisplayContext
Indexer.delete/doDelete	No change	MBMessageLocalServiceImpl.deleteMes

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.

551.3 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.

551.4 Annotating Service Methods to Trigger Indexing

Having objects 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 in the service layer. For Service Builder entities, this occurs in the LocalServiceImpl classes. An annotation 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 AOP infrastructure, so if you have a method with that annotation, you must call it using a service instance variable injected by your dependency injector, and not using the `this` keyword. Whether using OSGi's Declarative Services (DS) or Spring for dependency injection, there's a protected variable declared in the superclass (`*LocalServiceBaseImpl`) that can be used in the `*LocalServiceImpl`, like this.

```
blogsEntryLocalService.deleteEntry(entry);
```

Since you're using the injected service variable, that means you must not call

```
this.deleteEntry(...)
```

in your `*LocalServiceImpl` methods. The annotation won't be executed and you'll be left with a state mismatch between the search engine document and the database column.

551.5 Search and Localization: a Cheat Sheet

Localization is important. Search and localization can play nicely together, if you take some precautions:

- For each field that should be localized (e.g., `content`), index a separate field for each of the site's languages (e.g., `content_en_US`, `content_ja_JP`, `content_es_ES`, ...).
- Search the localized fields. Whatever you index, that's what you should be querying for.
- Don't index content in plain (unlocalized) fields if you expect the content to be present in multiple locales.
- Don't index both the plain and the localized field.

The indexing and searching articles included in this section demonstrate how to handle localized fields in the search code properly.

CHAPTER 552

INDEXING MODEL ENTITIES

Model entities are searchable when their data fields are indexed by the search engine. Search and indexing code relies on Search APIs and SPIs.

The extension points (i.e., the interfaces to implement) in this section are provided by the `com.liferay.portal.search.spi` bundle. Calls are also made to the `com.liferay.portal.search.api` bundle's methods.

Here are the Gradle dependencies for Liferay DXP 7.2.0 GA1:

```
dependencies {
    compileOnly group: "com.liferay", name: "com.liferay.portal.search.spi", version: "3.2.1"
    compileOnly group: "com.liferay", name: "com.liferay.portal.search.api", version: "3.7.0"
}
```

APIs and SPIs: SPIs are a special type of API. Generally, code inside a SPI module (e.g., `portal-search-spi`) is used to customize existing behavior, while API modules contain behavior you want to use. Put simply, implement interfaces from an SPI, and consume the code from the API.

SPI example: `ModelDocumentContributor` lives in an SPI module because you're supposed to implement it directly, defining your own indexing behavior.

API example: `SearchRequest` lives in an API module because its behavior is leveraged inside your code to build a search request.

552.1 Contributing Model Entity Fields to the Index

Write a `ModelDocumentContributor` class to control how model entity fields are indexed in search engine documents.

Extension Point (SPI): `com.liferay.portal.search.spi.model.index.contributor.ModelDocumentContributor<T>`

Declare the Component's `indexer.class.name` and its service type as a `ModelDocumentContributor` class:

```
@Component(
    immediate = true,
    property = "indexer.class.name=com.liferay.foo.model.FooEntry",
    service = ModelDocumentContributor.class
```

```
)
public class FooEntryModelDocumentContributor
    implements ModelDocumentContributor<FooEntry> {
```

Implement the `contribute` method, calling the `com.liferay.portal.kernel.Document.add()` method appropriate for the data type the index should use (e.g., `addText` for fields that should be searched using a full text search strategy).

```
@Override
public void contribute(Document document, FooEntry fooEntry) {

    document.addDate(Field.DISPLAY_DATE, fooEntry.getDisplayDate());
    document.addDate(Field.MODIFIED_DATE, fooEntry.getModifiedDate());
    document.addText(Field.SUBTITLE, fooEntry.getSubtitle());
```

For fields that should be localized, index a field for each locale in the Site. Many times you'll want to localize the title and content fields, for example:

```
for (Locale locale :
        LanguageUtil.getAvailableLocales(fooEntry.getGroupId())) {

    String languageId = LocaleUtil.toLanguageId(locale);

    document.addText(
        LocalizationUtil.getLocalizedName(Field.CONTENT, languageId),
        content);
    document.addText(
        LocalizationUtil.getLocalizedName(Field.TITLE, languageId),
        fooEntry.getTitle());
}
```

The above strategy is a good one: loop through the available locales in the site, then use `com.liferay.portal.kernel.util.LocalizationUtil` to add the localized field name to the document.

The `contribute` method is called each time the `add` and `update` methods in the entity's service layer are called.

552.2 Configure Re-Indexing and Batch Indexing Behavior

`ModelIndexerWriterContributor` classes configure the re-indexing and batch re-indexing behavior for the model entity. This class's `customize` 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 the contributor.

Extension Point (SPI): `com.liferay.portal.search.spi.model.index.contributor.ModelIndexerWriterContributor`

The bulk of the work is in the `customize` method. This code uses the actionable dynamic query helper method to retrieve all existing Foo entities in the virtual instance (identified by the Company ID). If you're using Service Builder, this query method was generated for you when you built the services. Each Foo Entry document is then retrieved and added to a collection.

1. First set up the component and class declarations:

    ```
    @Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.foo.model.FooEntry",
        service = ModelIndexerWriterContributor.class
    )
    public class FooEntryModelIndexerWriterContributor
        implements ModelIndexerWriterContributor<FooEntry> {
    ```

2. Write the `customize` method:

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

    batchIndexingActionable.setAddCriteriaMethod(
        dynamicQuery -> {
            Property displayDateProperty = PropertyFactoryUtil.forName(
                "displayDate");

            dynamicQuery.add(displayDateProperty.lt(new Date()));

            Property statusProperty = PropertyFactoryUtil.forName("status");

            Integer[] statuses = {
                WorkflowConstants.STATUS_APPROVED,
                WorkflowConstants.STATUS_IN_TRASH
            };

            dynamicQuery.add(statusProperty.in(statuses));
        });
    batchIndexingActionable.setPerformActionMethod(
        (FooEntry fooEntry) -> {
            Document document =
                modelIndexerWriterDocumentHelper.getDocument(fooEntry);

            batchIndexingActionable.addDocuments(document);
        });
}
```

3. Override `getBatchIndexingActionable`:

```
@Override
public BatchIndexingActionable getBatchIndexingActionable() {
    return _dynamicQueryBatchIndexingActionableFactory.
        getBatchIndexingActionable(
            _fooEntryLocalService.getIndexableActionableDynamicQuery());
}
```

4. Override `getcompanyId`, getting the ID from your entity:

```
@Override
public long getCompanyId(FooEntry fooEntry) {
    return fooEntry.getCompanyId();
}
```

5. Override `getIndexerWriterMode`:

```
@Override
public IndexerWriterMode getIndexerWriterMode(FooEntry fooEntry) {
    int status = fooEntry.getStatus();

    if ((status == WorkflowConstants.STATUS_APPROVED) ||
        (status == WorkflowConstants.STATUS_IN_TRASH) ||
        (status == WorkflowConstants.STATUS_DRAFT)) {

        return IndexerWriterMode.UPDATE;
    }

    return IndexerWriterMode.DELETE;
}
```

`com.liferay.portal.search.spi.model.index.contributor.helper.IndexerWriterMode` defines the following index-writing options:

- `IndexerWriterMode.DELETE`: instructs the search framework to delete the given document in the search index without re-creating it
- `IndexerWriterMode.PARTIAL_UPDATE`, `IndexerWriterMode.UPDATE`: instructs the search framework to update the given document in the search index.
- `IndexerWriterMode.SKIP`: instructs the search framework to not write anything to the search index.

The default is `IndexerWriterMode.UPDATE`.

6. Add the services referenced:

```
@Reference
private FooEntryLocalService _fooEntryLocalService;

@Reference
private DynamicQueryBatchIndexingActionableFactory
    _dynamicQueryBatchIndexingActionableFactory;
```

552.3 Contribute Fields to Every Document

`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` contains logic to contribute `GROUP_ID` and `SCOPE_GROUP_ID` fields for all documents with a backing entity that's also a `GroupedModel`.

CHAPTER 553

SEARCHING THE INDEX FOR MODEL ENTITIES

The heart of searching for your model entity documents is querying for what you indexed. To do this, contribute search terms to the Liferay DXP search query.

The extension points (i.e., the interfaces to implement) on this page are provided by the `com.liferay.portal.search.spi` bundle.

Here's the Gradle dependency for Liferay DXP 7.2.0 GA1:

```
dependencies {
    compileOnly group: "com.liferay", name: "com.liferay.portal.search.spi", version: "3.2.1"
}
```

553.1 Adding your Model Entity's Terms to the Query

`KeywordQueryContributor` classes contribute clauses to the ongoing search query, to control the way model entities are searched. If you're storing localized fields in the index (a good idea, as covered in the example code for your `ModelDocumentContributor`), query the localized fields at search time.

Extension Point (SPI): com.liferay.portal.search.spi.model.query.contributor.KeywordQueryContributor

```
@Component(
    immediate = true,
    property = "indexer.class.name=com.liferay.foo.model.FooEntry",
    service = KeywordQueryContributor.class
)
public class FooEntryKeywordQueryContributor
    implements KeywordQueryContributor {

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

        SearchContext searchContext =
            keywordQueryContributorHelper.getSearchContext();

        queryHelper.addSearchLocalizedTerm(
            booleanQuery, searchContext, Field.CONTENT, false);
        queryHelper.addSearchTerm(
            booleanQuery, searchContext, Field.SUBTITLE, false);
```

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

    @Reference
    protected QueryHelper queryHelper;

}
```

The entity in this example has a title, subtitle, and content field. The subtitle field wasn't stored as a localized field, so it's not searched that way, either.

553.2 Contributing Query Clauses to Every Search

It's a less common need, but 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`.

It includes a String array that includes the fields that are always searched:

```
private static final String[] _ALWAYS_PRESENT_FIELDS = {
    Field.COMMENTS, Field.CONTENT, Field.DESCRIPTION, Field.PROPERTIES,
    Field.TITLE, Field.URL, Field.USER_NAME
};
```

553.3 Pre-Filtering

*`PreFilterContributor` classes 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. They're constructed each time a query for the model entity is made.

Extension Pointi (SPI): `ModelPreFilterContributors`

To contribute filter clauses specific to a model entity, use a `ModelPreFilterContributor`. This one adds a filter for workflow status:

```
@Component(
    immediate = true,
    property = "indexer.class.name=com.liferay.foo.model.FooEntry",
    service = ModelPreFilterContributor.class
)
public class FooEntryModelPreFilterContributor
    implements ModelPreFilterContributor {

    @Override
    public void contribute(
        BooleanFilter booleanFilter, ModelSearchSettings modelSearchSettings,
        SearchContext searchContext) {

        addWorkflowStatusFilter(
            booleanFilter, modelSearchSettings, searchContext);
    }

    protected void addWorkflowStatusFilter(
        BooleanFilter booleanFilter, ModelSearchSettings modelSearchSettings,
        SearchContext searchContext) {

        workflowStatusModelPreFilterContributor.contribute(
```

```
            booleanFilter, modelSearchSettings, searchContext);
    }

    @Reference(target = "(model.pre.filter.contributor.id=WorkflowStatus)")
    protected ModelPreFilterContributor workflowStatusModelPreFilterContributor;

}
```

Extension Point (SPI): `com.liferay.portal.search.spi.model.query.contributor.QueryPreFilterContributor`

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 sub-filter.

CHAPTER 554

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. You can exert control over your model entity's summary.

554.1 Creating a Results Summary

`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.

Extension Pointi (SPI): `com.liferay.portal.search.spi.model.result.contributor.ModelSummaryContributor`

```
@Component(
    immediate = true,
    property = "indexer.class.name=com.liferay.foo.model.FooEntry",
    service = ModelSummaryContributor.class
)
public class FooEntryModelSummaryContributor
    implements ModelSummaryContributor {

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

        String languageId = LocaleUtil.toLanguageId(locale);

        return _createSummary(
            document,
            LocalizationUtil.getLocalizedName(Field.CONTENT, languageId),
            LocalizationUtil.getLocalizedName(Field.TITLE, languageId));
    }

    private Summary _createSummary(
        Document document, String contentField, String titleField) {

        String prefix = Field.SNIPPET + StringPool.UNDERLINE;

        Summary summary = new Summary(
            document.get(prefix + titleField, titleField),
            document.get(prefix + contentField, contentField));

        summary.setMaxContentLength(200);

        return summary;
```

```
    }

}
```

554.2 Controlling the Visibility of Model Entities

`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.

Extension Point (SPI): `com.liferay.portal.search.spi.model.result.contributor.ModelVisibilityContributor`

```
@Component(
    immediate = true,
    property = "indexer.class.name=com.liferay.foo.model.FooEntry",
    service = ModelVisibilityContributor.class
)
public class FooEntryModelVisibilityContributor
    implements ModelVisibilityContributor {

    @Override
    public boolean isVisible(long classPK, int status) {
        try {
            FooEntry entry = fooEntryLocalService.getEntry(classPK);

            return isVisible(entry.getStatus(), status);
        }
        catch (PortalException pe) {
            if (_log.isWarnEnabled()) {
                _log.warn("Unable to check visibility for foo entry ", pe);
            }
        }

        return false;
    }

    protected boolean isVisible(int entryStatus, int queryStatus) {
        if (((queryStatus != WorkflowConstants.STATUS_ANY) &&
             (entryStatus == queryStatus)) ||
            (entryStatus != WorkflowConstants.STATUS_IN_TRASH)) {

            return true;
        }

        return false;
    }

    @Reference
    protected FooEntryLocalService fooEntryLocalService;

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

}
```

Once you index model entities, add their terms to the Liferay DXP search query, and get the summary just right, your model entity is ready to be searched.

CHAPTER 555

SEARCH SERVICE REGISTRATION

The search framework must know about your entity and how to handle it during a search request. To register model entities with Liferay's search framework, `SearchRegistrar`s use the search framework's registry to define certain things about your model entity's `ModelSearchDefinition`: the default fields used to retrieve documents from the search engine, and the optional search services registered for your entity (for example, the `ModelIndexWriterContributor` for you entity). Registration occurs as soon as the Component is activated (during portal startup or deployment of the bundle).

A Registrar is required so the container knows about your implementation.

1. First, declare the class a component and create the class declaration:

```
@Component(immediate = true, service = {})
public class FooEntrySearchRegistrar {
```

2. Next write the `activate` method, annotated with the OSGi annotation `@Activate`. On activation of this component, call the `ModelSearchRegistrarHelper.register` method and use the call to build out a `ModelSearchDefinition`:

```
@Activate
protected void activate(BundleContext bundleContext) {
    _serviceRegistration = modelSearchRegistrarHelper.register(
        FooEntry.class, bundleContext,
        modelSearchDefinition -> {
            modelSearchDefinition.setDefaultSelectedFieldNames(
                Field.ASSET_TAG_NAMES, Field.COMPANY_ID,
                Field.ENTRY_CLASS_NAME, Field.ENTRY_CLASS_PK,
                Field.GROUP_ID, Field.MODIFIED_DATE, Field.SCOPE_GROUP_ID,
                Field.UID);
            modelSearchDefinition.setDefaultSelectedLocalizedFieldNames(
                Field.CONTENT, Field.TITLE);
            modelSearchDefinition.setModelIndexWriteContributor(
                modelIndexWriterContributor);
            modelSearchDefinition.setModelSummaryContributor(
                modelSummaryContributor);
            modelSearchDefinition.setModelVisibilityContributor(
                modelVisibilityContributor);
        });
}
```

On activation of the Component, a chain of search and indexing classes is registered for the Foo entity. In addition, 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.

In addition to the `ModelSearchDefinition` setter methods used in the above code, there's another to be aware of:

To select all locales all the time when searching for your model entity, pass `true` to `setSelectAllLocales`:

```
modelSearchDefinition.setSelectAllLocales(true);
```

Technically, there's another setter in `ModelSearchDefinition` that takes a boolean, `setSearchResultPermissionFilterSuppressed`. However, this is intended for internal consumption.

3. Write a corresponding `deactivate` method:

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

4. Get references to the services needed in the class. For the search services you're providing, specify them by entering the FQCN of your model entity in the reference target's `indexer.class.name` property:

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

@Reference
protected ModelSearchRegistrarHelper modelSearchRegistrarHelper;

@Reference(
    target = "(indexer.class.name=com.liferay.foo.model.FooEntry)"
)
protected ModelSummaryContributor modelSummaryContributor;

@Reference(
    target = "(indexer.class.name=com.liferay.foo.model.FooEntry)"
)
protected ModelVisibilityContributor modelVisibilityContributor;

private ServiceRegistration<?> _serviceRegistration;
```

It's quite possible you'll want to write this class after first getting all the search and indexing logic into place. How can you register a `ModelIndexerWriterContributor` if you haven't written one yet?

CHAPTER 556

SEARCH QUERIES AND FILTERS

To get sensible results from the search engine, you must provide a sensible query.

556.1 Queries and Filters in Liferay's Search API

Elasticsearch and Solr do not make API level distinctions between queries and filters. In the past, Liferay's API explicitly provided two sets of APIs, one for queries and one for filters. Both APIs lived in `portal-kernel` (the 7.1 source code for filters is here).

In 7.0, there's a new way of querying and filtering via Liferay's Search API, and the APIs for it live in the `portal-search-api` module. Instead of calling specific filter APIs, you'll now construct a query and add it to the search request, specifying it as a filter using the `SearchRequestBuilder.postFilterQuery(Query)` method. See the 7.2 Query APIs.

Note: Support for the legacy `com.liferay.portal.kernel.search.Query.getPreBooleanFilter()` is only present in the new search request builder and assembler implementation to allow for backwards compatibility with the `Indexer` framework's handling of queries. The older approach encourages some practices that are not ideal:

- Wrapping a `BooleanQuery` with another `BooleanQuery`.
- Some queries shouldn't have filters according to Elasticsearch's API.

Despite the more unified filtering and querying code, you should understand the functional difference between filtering and querying:

Filters ask a yes or no question for every document. A filter might ask *is the status field equal to staging or live?*

Queries ask the same yes or no question AND how well a document matches the specified criteria. This is the concept of relevance scoring. A query might ask *Does the document's content field field contain the words "Liferay", "Content", or "Management", and how relevant is the content of the document to the searched keywords?*

Hint: Filtering is faster than querying, since the documents matching a filter can be cached. Queries not only match documents but also calculate scores. We recommend using filtering and querying together: filters to reduce the number of matched documents, queries for the final examination.

556.2 Supported Query Types

Liferay's Search API supports the following types of queries:

556.3 Full Text Queries

Full text queries are high-level queries usually used for querying full text fields like the `content` field of a Blogs Entry. How terms are matched depends on the query type.

Supported Full Text Queries

```
CommonTermsQuery
MatchPhraseQuery
MatchPhrasePrefixQuery
MatchQuery
MultiMatchQuery
SimpleStringQuery
StringQuery
```

Here are some common full text queries:

- Match Query: A full text query, scored by relevance.
- Multi Match Query: Execute a `MatchQuery` over several fields.
- String Query: Use Lucene query syntax.

556.4 Term Queries

Term queries look for exact matching on keyword fields and indexed terms.

```
ExistsQuery
FuzzyQuery
IdsQuery
PrefixQuery
RangeQuery
RegexpQuery
TermQuery
TermsQuery
TermRangeQuery
TermsSetQuery
WildcardQuery
```

Here are some common term queries:

- Wildcard Query: Wildcard (* and ?) matching on keyword fields and indexed terms
- Fuzzy Query: Scrambles characters in input before matching

556.5 Compound Queries

Compound queries are often used to wrap other queries. They're commonly used to switch from query to filter context.

```
BooleanQuery
BoostingQuery
ConstantScoreQuery
DisMaxQuery
FunctionScoreQuery
```

Here are some common compound queries:

- Boolean Query: Allows a combo of several query types. Individual queries are as clauses with `SHOULD` | `MUST` | `MUST_NOT` | `FILTER`
- Constant Score Query: Wraps another query, switches it to filter mode, and gives all returned documents a constant relevance score.

556.6 Joining Queries

The concept of a join doesn't work well in a distributed index. Joining queries allow similar behavior in the search index, such as using the `nested` datatype to index an array of objects that can be queried independently, using the `NestedQuery`.

```
NestedQuery
```

Nested Query: Query nested objects as if they each had a separate document in the index.

556.7 Geo Queries

In Elasticsearch, you can index latitude/longitude pairs and geographic shapes. Geo queries let you query for these points and shapes.

```
GeoBoundingBoxQuery
GeoDistanceQuery
GeoDistanceRangeQuery
GeoPolygonQuery
GeoShapeQuery
```

A common Geo Query is the `GeoDistanceQuery`, used to find documents within a certain distance of a geographic point (latitude/longitude).

556.8 Specialized Queries

These queries don't fit into another group:

```
MoreLikeThisQuery
PercolateQuery
ScriptQuery
```

- **More Like This:** Use a document to query for similar documents.
- **Percolate:** Match individual documents against indexed queries (for alerting to new documents of interest, or automatically categorizing documents).
- **Script:** Filter based on a script.

556.9 Other Queries

`MatchAllQuery` Matches all documents in the index.

The proper search query is entirely context- and search engine-specific, so you should read the Query documentation straight from Elasticsearch or Solr to determine which queries are available and what they do.

All the recommended and supported queries and filters are found in the `portal-search-api` module's `com.liferay.portal.search.query` and `com.liferay.portal.search.filter` packages.

Legacy queries and filters, which are still supported but moving towards deprecation, are found in the `com.liferay.portal.kernel.search.*` packages provided by `portal-kernel`.

556.10 Using Queries

Here's the generalized approach for querying and filtering search documents in your own search code:

1. Instantiate and construct the query object.
2. Add the query to the search request—the method you use determines whether the context is filtering or querying (or both in the same request).
3. Execute the search request.
4. Process the search response.

These steps are covered in more detail (with examples) in the next article.

556.11 Search Queries in Liferay's Code

The APIs for creating queries are best exemplified in Liferay's own test cases. For example, BaseTermsQueryTestCase constructs a search request containing a `TermsQuery` using the `Queries` API:

```
TermsQuery termsQuery = queries.terms(Field.USER_NAME);
```

This code executes the search request with the terms query constructed above in a query context.

Other query test cases are also available to reference in the `portal-search` module's source code.

556.12 External References

- https://www.elastic.co/guide/en/elasticsearch/reference/7.x/query-dsl.html
- https://lucene.apache.org/solr/guide/7_1/query-syntax-and-parsing.html

556.13 Search Engine Connector Support

- Elasticsearch 6: Yes
- Solr 7: No* (Only the "legacy" query types from `com.liferay.portal.kernel.search.*` are supported as of Liferay DXP Beta 2.)

556.14 New/Related APIs

Package | Provided by Artifact | Notes | `com.liferay.portal.search.query.*` | com.liferay.portal.search.api | Most of the provided query types are new as of 7.2 `com.liferay.portal.search.filter.*` | com.liferay.portal.search.api | Some of the provided filter types are new as of 7.2

CHAPTER 557

BUILDING SEARCH QUERIES AND FILTERS

Each filter and query has a different purpose, but the way you'll add the information to the search request is similar between all queries and filters.

557.1 Queries

A mostly-complete code snippet for building Queries is provided for your copying and pasting convenience below.

557.2 Declare Gradle Dependencies

Add the following to your `build.gradle` file:

```
dependencies {
    compileOnly group: "biz.aQute.bnd", name: "biz.aQute.bndlib", version: "3.5.0"
    compileOnly group: "com.liferay.portal", name: "release.portal.api", version: "7.2.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

With this you can import all the Search APIs.

557.3 Reference the Search Services

To satisfy the dependencies of the example code presented here, get references to

- `com.liferay.portal.search.searcher.SearchRequestBuilderFactory`
- `com.liferay.portal.search.searcher.Searcher`
- `com.liferay.portal.search.query.Queries`

```
@Reference
protected Queries queries;

@Reference
```

```
protected Searcher searcher;

@Reference
protected SearchRequestBuilderFactory searchRequestBuilderFactory;
```

557.4 Build the Search Query

1. Use the `com.liferay.portal.search.query.Queries` interface to instantiate the queries you'll construct. For example,

   ```
   TermsQuery termsQuery = queries.terms("fieldName");
   MatchQuery matchQuery = queries.match("fieldName", "value");
   BooleanQuery booleanQuery = queries.booleanQuery();
   ```

   To discover what parameters each query must have (e.g., `String field` in the case of the above `com.liferay.portal.search.query.TermsQuery`), see the `Queries` interface.

2. Build out the queries to get the desired response. This looks different for each query type, as explained in Elasticsearch's Query documentation.

   ```
   termsQuery.addValues("value1", "value2");
   ```

3. You might want to wrap queries. For example, use the queries constructed above as MUST clauses in a `BooleanQuery` wrapper:

   ```
   booleanQuery.addMustQueryClauses(termsQuery, matchQuery);
   ```

Once the query itself is in good shape, feed it to the search request.

557.5 Build the Search Request

1. Use `com.liferay.portal.search.searcher.SearchRequestBuilderFactory` to get an instance of `com.liferay.portal.search.searcher.SearchRequestBuilder`:

   ```
   SearchRequestBuilder searchRequestBuilder =
       searchRequestBuilderFactory.builder();
   ```

   If not setting search keywords into the `SearchContext` (covered below), make sure the request builder enables empty search.

   ```
   searchRequestBuilder.emptySearchEnabled(true);
   ```

   Set the `long companyId` and, optionally, `String keywords` into the `com.liferay.portal.kernel.search.SearchConte`

   ```
   searchRequestBuilder.withSearchContext(
       searchContext -> {
           searchContext.setCompanyId(companyId);
           searchContext.setKeywords(keywords);
       });
   ```

Setting the Company ID into the `SearchContext` is required to ensure the correct index is searched.

Setting "keywords" on the `SearchContext` is necessary if you want to search via user input. For example, if providing a Search bar in an application's view layer, pass its input into the search context. Liferay's search framework adds the user input keywords and any other data in the `SearchContext` object to its own queries, searching the appropriate fields of each indexed entity, as defined by its `KeywordQueryContributor` or by the `postProcessSearchQuery` method of its `Indexer`.

2. To execute the query, get a `com.liferay.portal.search.searcher.SearchRequest` instance from the builder by adding the query to it and running its `build` method:

```
SearchRequest searchRequest =
    searchRequestBuilder.query(booleanQuery).build();
```

3. To use a constructed query in a filter context, call the `postFilterQuery` method while building the request:

```
SearchRequest searchRequest =
    searchRequestBuilder.postFilterQuery(termsQuery).build();
```

4. When constructing a search request, you'll often find it necessary to chain the builder methods together:

```
SearchRequest searchRequest =
    searchRequestBuilder.postFilterQuery(myQuery1).query(myQuery2).build();
```

   Chaining allows you to add filters and queries (and anything else from the builder API) to the same request in one fell swoop.

557.6 Execute the Search Request

Perform a search using the `com.liferay.portal.search.searcher.Searcher` service and the `SearchRequest` to get a `com.liferay.portal.search.searcher.SearchResponse`:

```
SearchResponse searchResponse = searcher.search(searchRequest);
```

557.7 Process the Search Response

What you'll do with the `SearchResponse` returned by the `searcher.search` call is dependent on the type of query and your specific use case. Much of the time you'll want to loop through the `com.liferay.portal.search.hits.SearchHit` and `com.liferay.portal.search.document.Document` objects, so that's what's shown here, with a simple message printed in the log for each one.

1. Get the `SearchHits` from the response:

```
SearchHits searchHits = searchResponse.getSearchHits();
```

2. Get a List of the `SearchHit` objects:

```
List<SearchHit> searchHitsList = searchHits.getSearchHits();
```

3. Loop through the `SearchHit` objects in the List, get the `Document` associated with each one, printing its score and UID to the console:

```
searchHitsList.forEach(
    searchHit -> {
        float hitScore = searchHit.getScore();

        Document doc = searchHit.getDocument();

        String uid = doc.getString(Field.UID);

        System.out.println(
            StringBundler.concat(
                "Document ", uid, " had a score of ", hitScore));
    });
```

557.8 Search Insights: Request and Response Strings

When building a search application, it can be useful to inspect the request string (translated into the search engine's dialect), and subsequently see the response string returned by the search server.

Retrieve these from the `SearchResponse` as

```
searchResponse.getRequestString();
searchResponse.getResponseString();
```

The format depends on your search engine: with Elasticsearch, both are JSON.

Note: The JSON returned as a request string is pruned from several Elasticsearch query defaults for clarity. To see the full request JSON that Elasticsearch processed, adjust the Elasticsearch server's logging.

Inspecting the request string produced by the code example included here reveals two main `"bool":"must"` query clauses in the JSON being sent to the search engine:

1. The `BooleanQuery` explicitly declared in the code example.

2. A (very long) query determined by the logic embedded in the `SearcherImpl#doSearch` method.

How you construct the `SearchRequest` determines how the `Searcher` API processes it, which in turn influences the request String sent to Elasticsearch. For example, sending `keywords` into the `SearchContext` object passed to the `SearchRequest` ensures that queries for certain fields are executed on all searchable documents.

557.9 Queries Example

The code below performs a `MatchQuery` on the `title_en_US` field for the value provided via the `keywords` String. In addition, a `TermsQuery` on the `folderId` field is executed to match a value of `0` (root `JournalFolders` are identified by `JournalFolderConstants.DEFAULT_PARENT_FOLDER_ID`, which evaluates to `0`). Both queries are wrapped in a `BooleanQuery` must clause.

Because this example passes `keywords` to the `SearchContext`, `emptySearchEnabled(true)` is not called on the `SearchRequestBuilder`. The `keywords` variable is not explicitly declared because this should come from user input. Therefore the example `search` method receives `keywords` as a parameter, along with the `companyId`:

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

import com.liferay.petra.string.StringBundler;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.search.Field;
import com.liferay.portal.kernel.util.LocaleUtil;
import com.liferay.portal.search.document.Document;
import com.liferay.portal.search.hits.SearchHit;
import com.liferay.portal.search.hits.SearchHits;
import com.liferay.portal.search.query.BooleanQuery;
import com.liferay.portal.search.query.MatchQuery;
import com.liferay.portal.search.query.Queries;
import com.liferay.portal.search.query.TermsQuery;
import com.liferay.portal.search.searcher.SearchRequest;
import com.liferay.portal.search.searcher.SearchRequestBuilder;
import com.liferay.portal.search.searcher.SearchRequestBuilderFactory;
import com.liferay.portal.search.searcher.SearchResponse;
import com.liferay.portal.search.searcher.Searcher;

import java.util.ArrayList;
import java.util.List;

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

@Component(
    service = MySearchComponent.class
)
public class MySearchComponent {

    public List<String> search(long companyId, String keywords)
        throws PortalException {

        MatchQuery titleQuery = queries.match(
            Field.getLocalizedName(LocaleUtil.US, Field.TITLE), keywords);

        TermsQuery rootFolderQuery = queries.terms(Field.FOLDER_ID);

        rootFolderQuery.addValues(String.valueOf(
                JournalFolderConstants.DEFAULT_PARENT_FOLDER_ID));

        BooleanQuery booleanQuery = queries.booleanQuery();

        booleanQuery.addMustQueryClauses(rootFolderQuery, titleQuery);

        SearchRequestBuilder searchRequestBuilder =
            searchRequestBuilderFactory.builder();

        // Uncomment this line below if you aren't setting "keywords"
        // on the SearchContext
        //      searchRequestBuilder.emptySearchEnabled(true);
```

```
        searchRequestBuilder.withSearchContext(
            searchContext -> {
                searchContext.setCompanyId(companyId);
                searchContext.setKeywords(keywords);
            });

        SearchRequest searchRequest = searchRequestBuilder.query(
            booleanQuery
        ).build();

        SearchResponse searchResponse = searcher.search(searchRequest);

        SearchHits searchHits = searchResponse.getSearchHits();

        List<SearchHit> searchHitsList = searchHits.getSearchHits();

        List<String> resultsList = new ArrayList<>(searchHitsList.size());

        searchHitsList.forEach(
            searchHit -> {
                float hitScore = searchHit.getScore();

                Document doc = searchHit.getDocument();

                String uid = doc.getString(Field.UID);

                System.out.println(
                    StringBundler.concat(
                        "Document ", uid, " had a score of ", hitScore));

                resultsList.add(uid);
            });

        System.out.println(StringPool.EIGHT_STARS);

        /*
         *  // Uncomment to inspect the Request and Response Strings
         * System.out.println( "Request String:\n" + searchResponse.getRequestString() +
         * "\n" + StringPool.EIGHT_STARS);
         * System.out.println( "Response String:\n" +
         * searchResponse.getResponseString() + "\n" + StringPool.EIGHT_STARS);
         */

        return resultsList;
    }

    @Reference
    protected Queries queries;

    @Reference
    protected Searcher searcher;

    @Reference
    protected SearchRequestBuilderFactory searchRequestBuilderFactory;

}
```

557.10 Filters

Filters as a distinct API-level object in Liferay DXP are going away. It's best to mirror the APIs of the search engine, and neither supported search engine makes an API level distinction between queries and filters. In recognition of this, there's a new way to perform post-filtering, which is filtering the returned search documents at the end of the search request (after calculating any

aggregations). Add the filter to the query using the `postFilterQuery` method in the request builder:

```
SearchRequestBuilder.postFilterQuery(Query);
```

As you can see, this takes a `Query` object, not a `Filter`. Therefore, construct the `Query` as in the previous section, and specify it as a post-filter while building the request. All of the legacy `Filter` objects from `portal-kernel` can now be constructed as queries, per the above query-building documentation.

557.11 Legacy Filters in Legacy Search Calls

Constructing the filters found in `portal-kernel`'s `com.liferay.portal.kernel.search.filter` package is demonstrated by this `new` term filter:

```
TermFilter termFilter = new TermFilter("fieldName", "filterValue");
```

Filters are added in legacy search calls by going through the `Indexer` framework's `postProcessContextBooleanFilter` method, which is invoked while the search framework is constructing the main search query. See the `UserIndexer`'s `addContextQueryParams` method, which is called in the overridden `postProcessContextBooleanFilter` to add the filter logic.

557.12 Discovering Indexed Fields

To find the fields to use in your Queries, navigate to *Control Panel → Configuration → Search* in a running portal. From there, open the Field Mappings tab and browse the mappings for the `liferay-[companyId]` index. Scroll to the `properties` section of the mapping.

A summary of the text fields that are localized can be found here.

CHAPTER 558

SEGMENTATION AND PERSONALIZATION

Segments are groups of users that are defined by a specific criteria. You can use the metadata from the user or organization profile, context information derived from the user's behavior, or some combination of the two to define that criteria. Alternatively, segments can be a static set of manually selected members.

In 7.0, the creation of user segments and experience personalization are now part of the product's core functionality. Up to Liferay DXP 7.1, this functionality was provided through the Audience Targeting application. In addition to the administration features of Segmentation and Personalization, developers can integrate and extend it.

558.1 Managing segments

The API to manage segments is provided by the `com.liferay.segments.api module`. The `SegmentsEntryService` provides the methods to perform permission aware operations on segments. You can use the provided tools to assign members to segments and to extend segment criteria.

The `segmentsEntry` criteria field determines the conditions that a user must meet to be assigned to the segment. A condition represents a combination of properties, operations, target values, and conjunctions. For example, a condition identifying Liferay Engineers for a Segment might look like this:

```
organization name EQUALS Liferay AND Job Title EQUALS Engineer
```

In the Segments UI, the segments criteria is built using the Segments Editor. The available properties are grouped by topic (e.g. User, Organization, Session). Technically, they are called a `SegmentsCriteriaContributor`, because they *contribute* conditions to the segments criteria.

You can see a number of common Segment management operations with example code in Segment Management.

558.2 Extending Segment Criteria

The default segment capabilities are robust enough to cover most use cases, and many types of third party integration can be performed without developing a code extension. Some cases, like retrieving an external segment or list can be be handled by using the REST API.

On the other hand, if you want to segment users based on a field provided from an external source (for example, the number of followers a user has on Instagram), you can contribute an indexable custom field to the User entity and query the value using the Expando API. Your new field is automatically available for its use as a profile-based criteria.

You must only develop a code extension if you must:

- Add a custom session property. This is done through the `RequestContextContributor`.
- Extend the criteria query with new queries, based either on existing model entities or in custom model entities. This is done through the `SegmentsCriteriaContributor`.

558.3 RequestContextContributor

User and Organization properties are model-based properties. That means that the available criteria for users and organizations are based on the attributes for users and organizations defined by the entities `model`. Criteria for model-based entities can be extended by creating a Custom Field for the corresponding model. Session properties are context-based properties and can't be extended through custom fields. To allow for user segmentation based on new context-based properties, like custom HTTP headers or attributes, you must develop an extension and deploy it in your Liferay DXP instance.

The default fields available for context-based segmentation can be found in the Context interface. Liferay generates a context instance with real-time information for every request. These are mostly obtained from the `HttpServletRequest`. The `RequestContextContributor` interface provides an extension point for adding a new context-based property to the Session panel in the Segments criteria editor and populating the segmentation context with the right value for that field.

The following service properties define a `RequestContextContributor`:

`request.context.contributor.key`: the unique key of the contributed field.

`request.context.contributor.type`: the contributor field type (boolean, date, double, integer, or string[default]).

The `contribute` method of the `RequestContextContributor` adds the custom field key-value pair to the context.

To create a `RequestContextContributor` through the step by step process, see Creating a Request Context Contributor.

558.4 SegmentsCriteriaContributor

The `SegmentsCriteriaContributor` interface provides a mechanism to extend the segment criteria query. Each `SegmentsCriteriaContributor` contributes a sub-query (or criterion) and the conjunction (AND, OR) to build the complete criteria query that defines the segment. They also provide a list of Field elements to be shown in the Segment Editor UI, as depicted in the figure:

The following service properties describe a `SegmentsCriteriaContributor`:

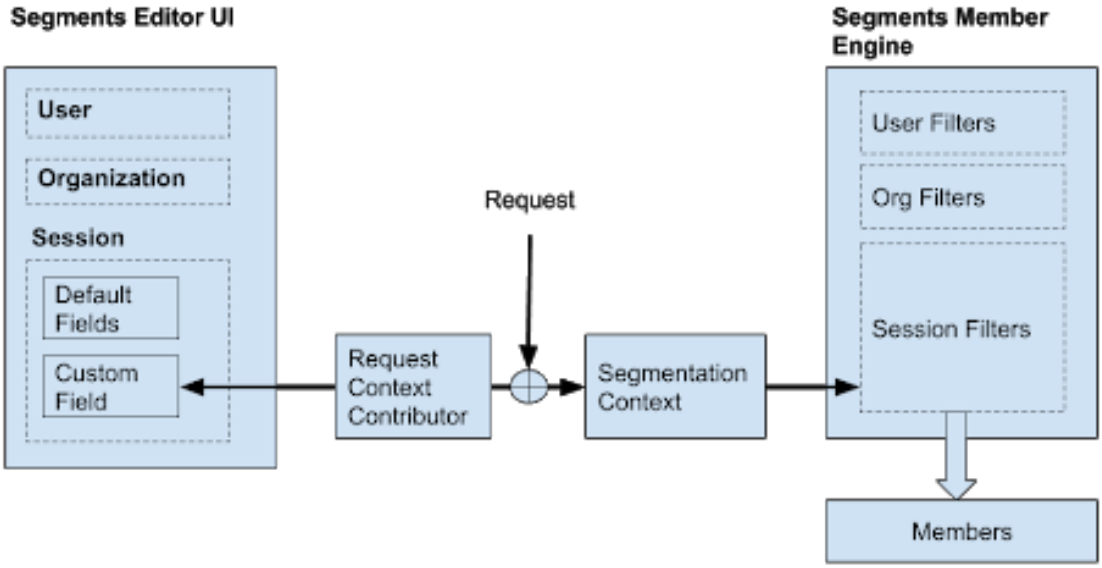

Figure 558.1: Learn more about a `RequestContextContributor` by viewing how it's used.

`segments.criteria.contributor.key`: the unique key that identifies the contributor.

`segments.criteria.contributor.model.class.name`: the entity type the contributor targets.

`segments.criteria.contributor.priority`: the order in which the fields and queries are contributed.

The `UserOrganizationSegmentsCriteriaContributor` is a good example of how a `SegmentsCriteriaContributor` works. It contributes new organization-related fields (*Organization Properties*) to the segments criteria editor, executes a query on the Organization based model, and finally contributes a subquery to the global user query (AND/OR the user belongs to the organizations found in the Organization model query). In summary, you can filter users based on aspects of a different but related entity, such as the organization.

To create a `SegmentsCriteriaContributor` through the step by step process, see Creating Segment Criteria Contributors.

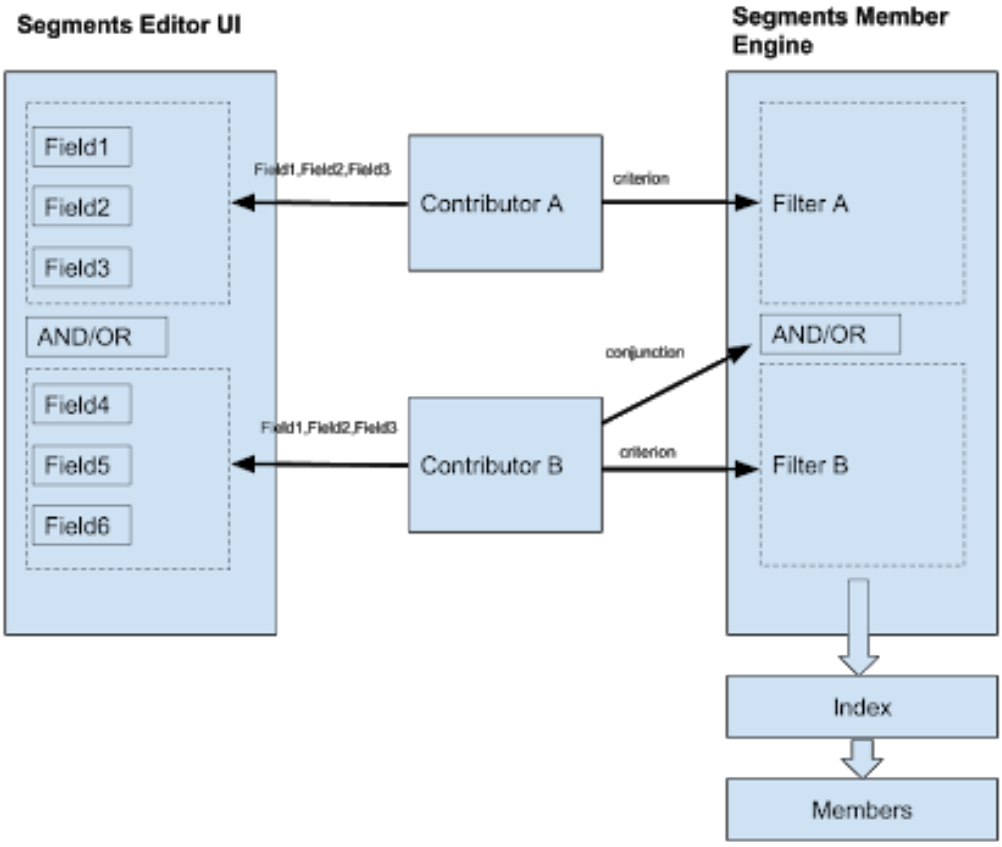

Figure 558.2: Learn more about a `SegmentsCriteriaContributor` by viewing how it's used.

CHAPTER 559

SEGMENT MANAGEMENT

There are a broad array of uses for Segments and ways that you can integrate them with your application. You'll learn more about how to manage segments next.

559.1 Defining a Segment

This snippet defines a segment by retrieving `@Reference` objects from `SegmentsCriteriaContributor` and instantiating a new `Criteria` object. It then adds user criteria using the `segmentsEntryService`:

- the user's `jobTitle` is `Developer` **AND**
- the user belongs to an Organization with a name that contains `America`

```
private void addSegmentWithCriteria() {
    Criteria criteria = new Criteria();

    _userSegmentsCriteriaContributor.contribute(
        criteria, "(jobTitle eq 'Developer')", Criteria.Conjunction.AND);
    _organizationCriteriaContributor.contribute(
        criteria, "contains(name,'America')", Criteria.Conjunction.OR);

    segmentsEntryService.addSegmentsEntry(
        "segment-key", nameMap, descriptionMap, true, CriteriaSerializer.serialize(criteria),
                    User.class.getName(), serviceContext);
}

@Reference(target = "(segments.criteria.contributor.key=organization)")
private SegmentsCriteriaContributor _organizationSegmentsCriteriaContributor;

@Reference(target = "(segments.criteria.contributor.key=user)")
private SegmentsCriteriaContributor _userSegmentsCriteriaContributor;
```

559.2 Manual Segment Member Assignments

To define manual user-segment member assignments with the `SegmentsEntryRelService`, use a snippet like this:

```
segmentsEntryRelService.addSegmentsEntryRel(
    segmentsEntryId, _portal.getClassNameId(User.class), userId, serviceContext)
```

This assigns a user identified by a `userId` to a segment identified by a `segmentsEntryId`:

559.3 Retrieving Segments

Segments and Segment Members can be retrieved programmatically. The code snippet below retrieves an ordered range of active segments for the `User`, within a site identified by a `groupId`.

```
List<SegmentsEntry> segmentsEntries = segmentsEntryService.getSegmentsEntries(groupId, true, User.class.getName(), 0, 10, orderByComparator);
```

559.4 Retrieving segment members

The local API to query computed segment-member associations is available in the `com.liferay.segments.api` module. The `SegmentsEntryProvider` service provides methods to obtain the entities associated to a segment, and the segments associated to an entity.

This snippet retrieves a range of primary keys of the entities associated to a segment identified by a `segmentsEntryId`:

```
long[] segmentsEntryClassPKs = segmentsEntryProvider.getSegmentsEntryClassPKs(segmentsEntryId, 0, 10);
```

To obtain the total count of entities associated to a segment, use the `getSegmentsEntryClassPKsCount` method, as shown in the following snippet:

```
int segmentsEntryClassPksCount =
    segmentsEntryProvider.getSegmentsEntryClassPKsCount(segmentsEntryId);
```

The method `getSegmentsEntryIds` obtains the reverse association — the segments associated to a specific entity. For example, this snippet returns the segments associated to a user identified by a `userId`:

```
int segmentsEntryClassPksCount =
    segmentsEntryProvider.getSegmentsEntryIds(User.class.getName(), userId);
```

Great! You now know how to manage segments!

CHAPTER 560

CREATING A REQUEST CONTEXT CONTRIBUTOR

To better understand the Request Context Contributor, you'll explore how to create one. First, you'll create the `SampleRequestContentContributor` class file, which contains the `contribute` method that contributes a new field to the context with a custom attribute. You can view the full project on Github.

1. Create a new module.
2. Inside the module, create a package named `com.liferay.segments.context.extension.sample.internal.context`
3. Create a Java class named `SampleRequestContentContributor`.
4. Inside the file, insert the `@Component` declaration:

```
@Component(
    immediate = true,
    property = {
        "request.context.contributor.key=" + SampleRequestContextContributor.KEY,
        "request.context.contributor.type=boolean"
    },
    service = RequestContextContributor.class
)
```

5. Add the class declaration:

```
public class SampleRequestContextContributor
    implements RequestContextContributor {

}
```

6. Create the attribute that you're adding. In this case, it's just a static string.

```
public static final String KEY = "sample";
```

7. Create the `contribute` method:

```
@Override
public void contribute(
    Context context, HttpServletRequest httpServletRequest) {

        context.put(KEY,
        GetterUtil.getBoolean(httpServletRequest.getAttribute("sample.attribute")));
    }
```

To customize your field label or add a set of selectable options, you can add an optional `SegmentsFieldCustomizer` service associated to your contributed field by its key. Create one now.

1. Inside the module, create a package named `com.liferay.context.extension.sample.internal.field.customizer`
2. Create a Java class named `SampleSegmentsFieldCustomizer`.
3. Inside the file, insert the `@Component` declaration:

```
@Component(
    immediate = true,
    property = {
        "segments.field.customizer.entity.name=Context",
        "segments.field.customizer.key=" + SampleSegmentsFieldCustomizer.KEY,
        "segments.field.customizer.priority:Integer=50"
    },
    service = SegmentsFieldCustomizer.class
)
```

4. Create the class declaration:

```
public class SampleSegmentsFieldCustomizer implements SegmentsFieldCustomizer {

}
```

5. Create the `KEY` value:

```
public static final String KEY = "sample";
```

6. Create the methods to provide a list of fields to be displayed when configuring the criteria.

```
@Override
public List<String> getFieldNames() {
    return _fieldNames;
}

@Override
public String getKey() {
    return KEY;
}

@Override
public String getLabel(String fieldName, Locale locale) {
        ResourceBundle resourceBundle = ResourceBundleUtil.getBundle(
        "content.Language", locale, getClass());

        return LanguageUtil.get(resourceBundle, "sample-field-label");
}

private static final List<String> _fieldNames = ListUtil.fromArray(
        new String[] {"sample"});
```

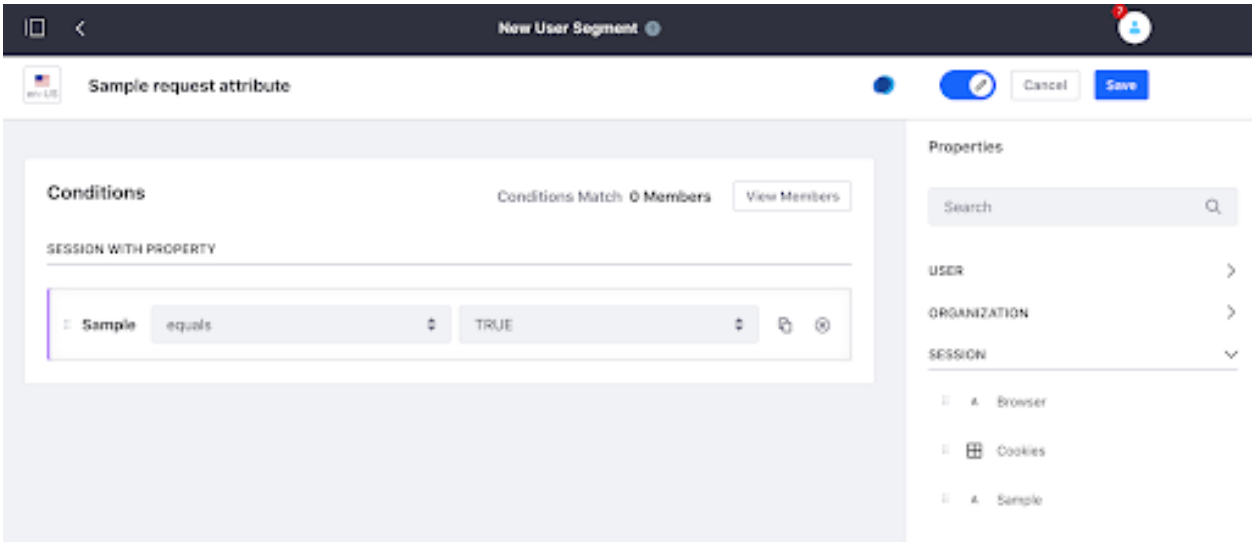

Figure 560.1: The sample field appears.

Once you deploy your extensions, the session section of the segment criteria editor includes your new context-based field.

Great! You've created a Request Context Contributor!

CHAPTER 561

CREATING A SEGMENT CRITERIA CONTRIBUTOR

To demonstrate the Segment Criteria Contributor, you'll create a contributor that segments users based on the title of Knowledge Base articles they have authored.

The first step is to make your related entity searchable through OData queries. For this purpose, you must have classes:

- `EntityModel`: represents your associated entity (in this case, the `KBArticle`) with its fields of interest.
- `ODataRetriever`: obtains the `KBArticles` that match a given OData query.

You can view the full project on Github.
Follow the instructions below to get started.

1. Create a module.
2. Create the following packages within the module:

    - `com.liferay.segments.criteria.extension.sample.internal.odata.retreiver`
    - `com.liferay.segments.criteria.extension.sample.internal.odata.entity`
    - `com.liferay.segments.criteria.extension.sample.internal.criteria.contributor`

Excellent! You have your module ready. Next, you'll create the entity model.

561.1 Creating the Entity Model

First, create the Entity Model for `KBArticle`.

1. Inside the `...internal.odata.entity` package, create the `KBArticleEntityModel` class which implements `EntityModel`:

```
public class KBArticleEntityModel implements EntityModel {

}
```

2. Create the key for the entity name:

```
public static final String NAME = "KBArticle";
```

3. Create the variable for the entity field map:

```
private final Map<String, EntityField> _entityFieldsMap;
```

4. Create the methods to retrieve the `KBArticleEntity`, entity map, and entity name key:

```
public KBArticleEntityModel() {
    _entityFieldsMap = Stream.of(
        new StringEntityField("title", locale -> "titleKeyword")
        ).collect(
            Collectors.toMap(EntityField::getName, Function.identity())
        );
}

@Override
public Map<String, EntityField> getEntityFieldsMap() {
    return _entityFieldsMap;
}

@Override
public String getName() {
    return NAME;
}
```

Next, you'll create the OData Retriever.

561.2 Creating the `ODataRetriever`

Next, create the `ODataRetriever` which gets the data using the relevant filter.

1. Inside the `...internal.odata.retreiver` package, create `KBArticleODataRetriever.java` which implements `ODataRetriever`:

```
public class KBArticleODataRetriever implements ODataRetriever<KBArticle> {

}
```

2. Add the `@Component` declaration above the class declaration:

```
@Component(
    immediate = true,
    property = "model.class.name=com.liferay.knowledge.base.model.KBArticle",
    service = ODataRetriever.class
)
```

3. Create the `@Reference` objects that you need for the Filter Parser, Knowledge Base Article Service, and OData Search Adapter:

```
@Reference
private FilterParserProvider _filterParserProvider;

 @Reference
private KBArticleLocalService _kbArticleLocalService;

@Reference
private ODataSearchAdapter _oDataSearchAdapter;
```

4. Create and instantiate the `_entityModel` object for the `KBArticle` model:

```
private static final EntityModel _entityModel = new KBArticleEntityModel();
```

5. Create the public methods to retrieve the results and the results count from the OData filter:

```
@Override
public List<KBArticle> getResults(
        long companyId, String filterString, Locale locale, int start,  int end)
    throws PortalException {

    Hits hits = _oDataSearchAdapter.search(
        companyId, filterString, KBArticle.class.getName(), _entityModel,
            _getFilterParser(), locale, start, end);

    return _getKBArticles(hits);
}

@Override
public int getResultsCount(
        long companyId, String filterString, Locale locale)
    throws PortalException {

    return _oDataSearchAdapter.searchCount(
        companyId, filterString, KBArticle.class.getName(), _entityModel,
            _getFilterParser(), locale);
}
```

6. Create the private methods for instantiating the `FilterParser` and retrieving the Knowledge Base article(s) that meet the criteria:

```
private FilterParser _getFilterParser() {
    return _filterParserProvider.provide(_entityModel);
}

private KBArticle _getKBArticle(Document document) throws PortalException {
    long resourcePrimKey = GetterUtil.getLong(
        document.get(Field.ENTRY_CLASS_PK));

    return _kbArticleLocalService.getLatestKBArticle(resourcePrimKey, 0);
}

private List<KBArticle> _getKBArticles(Hits hits) throws PortalException {
    Document[] documents = hits.getDocs();

    List<KBArticle> kbArticles = new ArrayList<>(documents.length);

    for (Document document : documents) {
        kbArticles.add(_getKBArticle(document));
    }

    return kbArticles;
}
```

You're all set to create the Segments Criteria Contributor!

561.3 Creating the `SegmentsCriteriaContributor`

Now create the `SegmentsCriteriaContributor` class that consumes the previous classes to retrieve the articles that match the query generated by the criteria editor, and contributes a query to filter users based on the articles they authored.

1. In the `...internal.criteria.contributor` package, create a `UserKBArticleSegmentCritieriaContributor` class that implements `SegmentsCriteriaContributor`.

```
public class UserKBArticleSegmentsCriteriaContributor
    implements SegmentsCriteriaContributor {

}
```

2. Create the `@Component` declaration to set properties and declare the service class.

```
@Component(
    immediate = true,
    property = {
        "segments.criteria.contributor.key=" + UserKBArticleSegmentsCriteriaContributor.KEY,
        "segments.criteria.contributor.model.class.name=com.liferay.portal.kernel.model.User",
        "segments.criteria.contributor.priority:Integer=70"
    },
    service = SegmentsCriteriaContributor.class
)
```

3. Create the variables to enable logging, retrieve the entity model, and entity key.

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

    private static final EntityModel _entityModel = new KBArticleEntityModel();

    public static final String KEY = "user-kb-article";
```

4. Create the reference variables for the OData retriever and Portal instance.

```
@Reference(
    target = "(model.class.name=com.liferay.knowledge.base.model.KBArticle)"
)

private ODataRetriever<KBArticle> _oDataRetriever;

@Reference
private Portal _portal;
```

5. Create the methods to define the implementation of `SegmentsCriteriaContributor`.

```
@Override
public void contribute(
    Criteria criteria, String filterString,
        Criteria.Conjunction conjunction) {

    criteria.addCriterion(getKey(), getType(), filterString, conjunction);

    long companyId = CompanyThreadLocal.getCompanyId();
```

```
    String newFilterString = null;

    try {
        StringBundler sb = new StringBundler();

        List<KBArticle> kbArticles = _oDataRetriever.getResults(
            companyId, filterString, LocaleUtil.getDefault(),
            QueryUtil.ALL_POS, QueryUtil.ALL_POS);

        for (int i = 0; i < kbArticles.size(); i++) {
            KBArticle kbArticle = kbArticles.get(i);

            sb.append("(userId eq '");
            sb.append(kbArticle.getUserId());
            sb.append("')");

            if (i < (kbArticles.size() - 1)) {
                sb.append(" or ");
            }
        }

        newFilterString = sb.toString();
    }
    catch (PortalException pe) {
        _log.error(
            com.liferay.petra.string.StringBundler.concat(
                "Unable to evaluate criteria ", criteria, " with filter ",
                filterString, " and conjunction ", conjunction.getValue()),
            pe);
    }

    if (Validator.isNull(newFilterString)) {
        newFilterString = "(userId eq '0')";
    }

    criteria.addFilter(getType(), newFilterString, conjunction);
}

@Override
public EntityModel getEntityModel() {
    return _entityModel;
}

@Override
public String getEntityName() {
    return KBArticleEntityModel.NAME;
}

@Override
public List<Field> getFields(PortletRequest portletRequest) {
    return Collections.singletonList(
        new Field(
            "title",
            LanguageUtil.get(_portal.getLocale(portletRequest), "title"),
            "string"));
}

@Override
public String getKey() {
    return KEY;
}

@Override
public Criteria.Type getType() {
    return Criteria.Type.MODEL;
}
```

6. Deploy your module.

After deploying your extension, the segment criteria editor includes a new section containing Knowledge Base properties. Notice that the section's UI, the properties, and their associated input fields and operations have been automatically generated based on the information provided by the extension services. For instance, the Knowledge Base article title supports *equals*, *not equals*, *contains*, and *not contains* operations because it was defined as a `StringEntityField`.

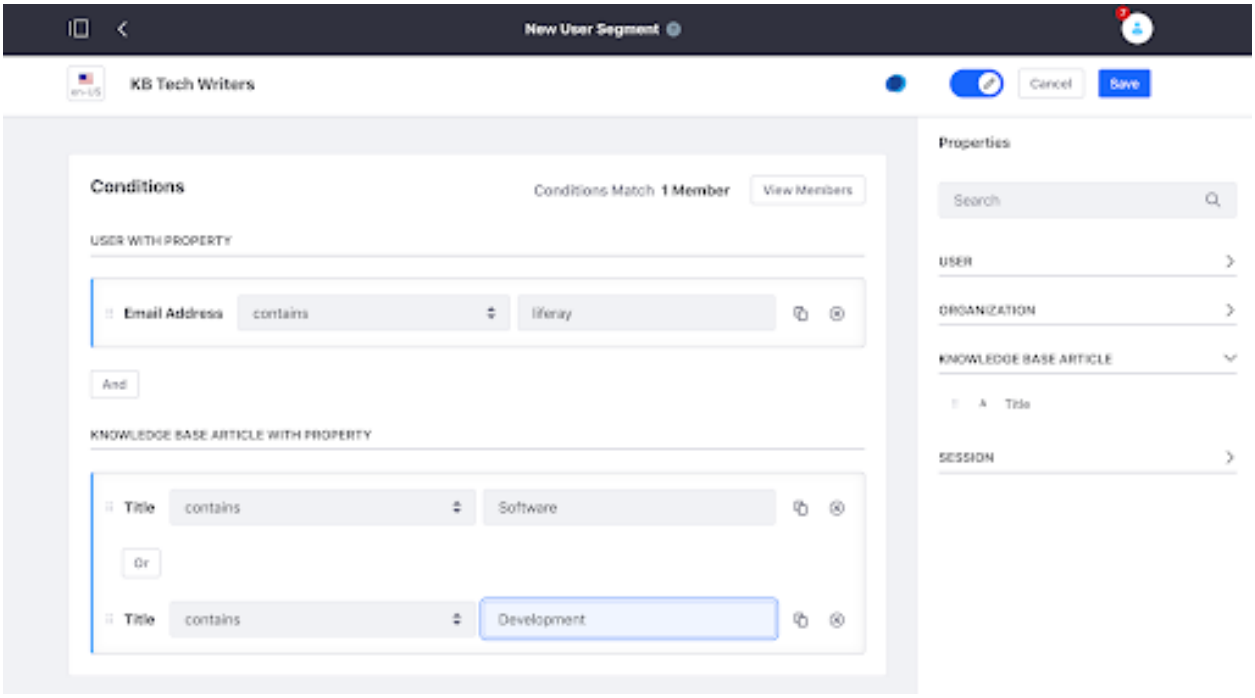

Figure 561.1: The sample field appears.

Awesome! You've created a Segment Criteria Contributor!

CHAPTER 562

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 article 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.

562.1 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.

562.2 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.

562.3 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: 20101,
        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.

Next, you'll learn how to access information from a `ServiceContext` object.

562.4 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:

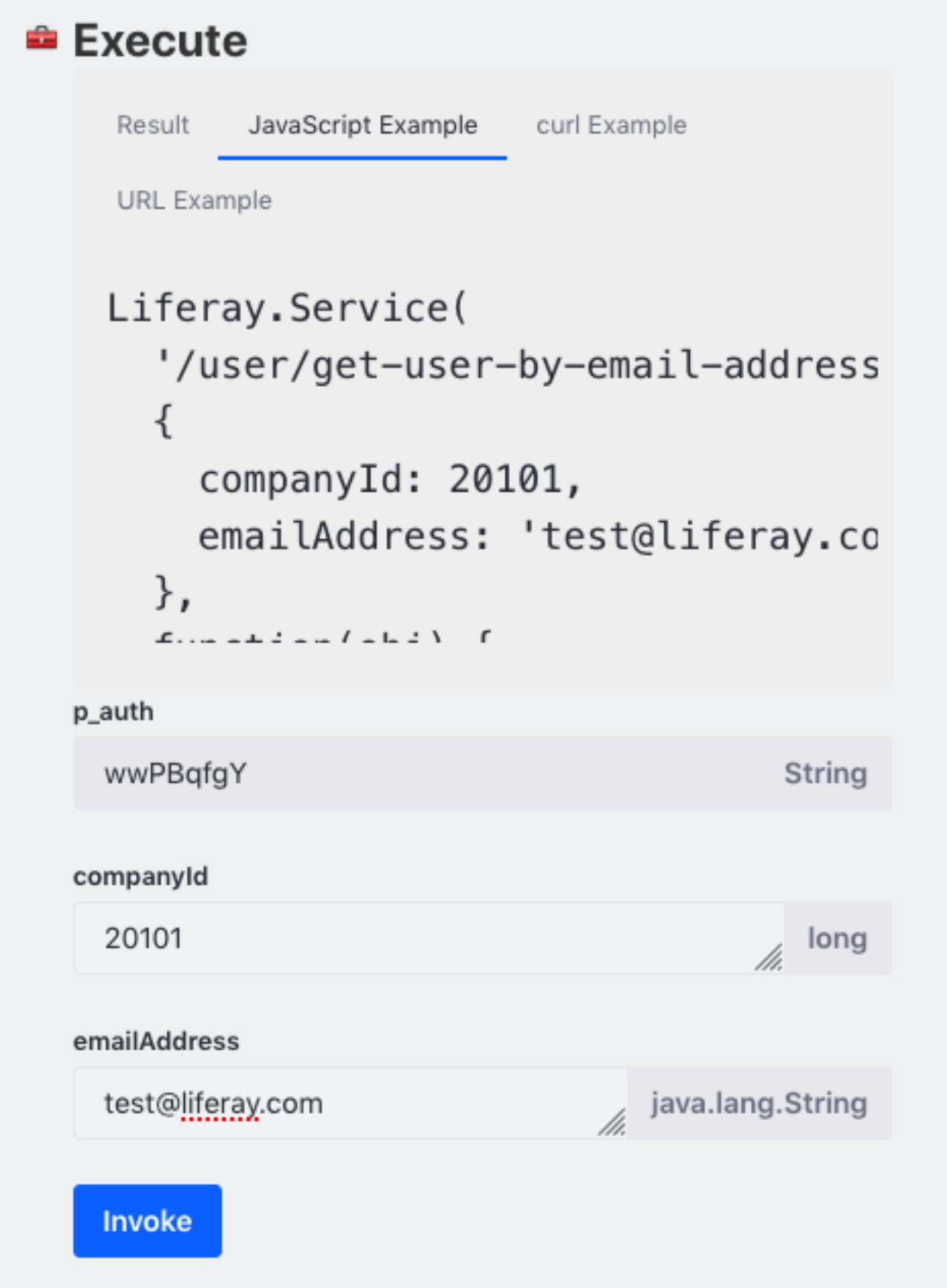

Figure 562.1: When you invoke a service from Liferay's JSON web services page, you can view the result of your service invocation as well as example code for invoking the service via JavaScript, curl, or URL.

```
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.

562.5 Related Topics

Business Logic with Service Builder
    Invoking Local Services
    Web Services

CHAPTER 563

INJECTING SERVICE COMPONENTS INTO INTEGRATION TESTS

Test driven development plays a key role in quality assurance. Liferay's tooling and integration with standard test frameworks support test driven development and help you reach quality milestones. 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.

Follow these steps 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.

   `@Inject` uses reflection to inject a field with a service component object matching the field's interface. Test rule `LiferayIntegrationTestRule` provides the annotation.

4. Optionally add a `filter` string or `type` parameter to further specify the service component object to inject. They can be used 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.

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.

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;

}
```

Great! Now you can inject service components into your tests.

563.1 Related Topics

- Service Trackers

CHAPTER 564

Upgrade Processes

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.

Here's what's involved in creating an upgrade process for your app:

- Specifying the schema version
- Declaring dependencies
- Writing upgrade steps
- Writing the registrator
- Waiting for upgrade completion

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. Meaningful schema versioning is important to clearly communicate the updates to your users.

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. 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 framework executes the upgrade steps to update the current module data to the latest schema. 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.

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**

A registration's upgrade step list can consist of as many upgrade steps as needed. 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.

The diagram below illustrates the relationship between the registrator and the upgrade steps.

This section covers these topics:

- Creating an upgrade process for your app
- Creating upgrade processes for former service builder plugins
- Upgrading data schemas in development

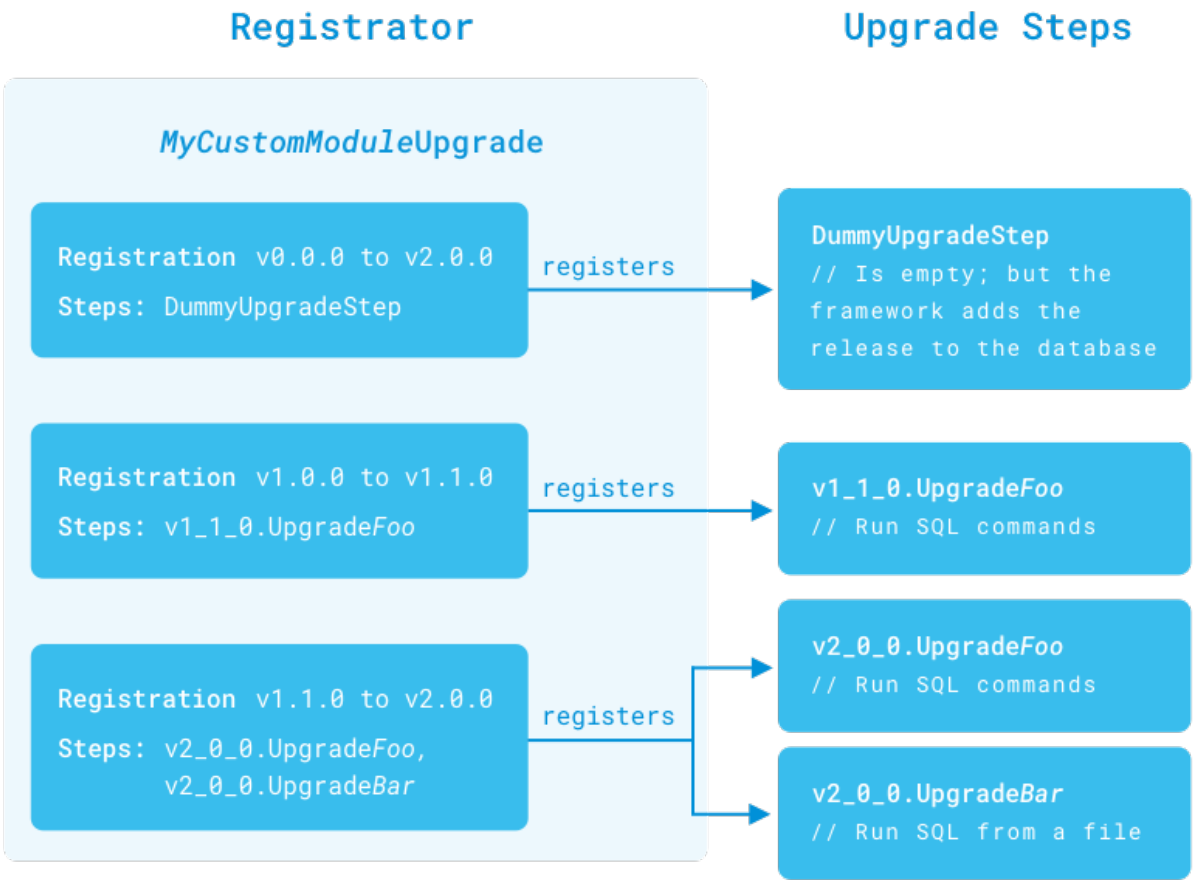

Figure 564.1: In a registrator class, the developer specifies a registration for each schema version upgrade. The upgrade steps handle the database updates.

CHAPTER 565

CREATING UPGRADE PROCESSES FOR MODULES

Follow these steps to create an upgrade process for your module:

1. Open your module's `bnd.bnd` file, and 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
   ```

```
**Important**: If no `Liferay-Require-SchemaVersion` header is specified,
Liferay DXP considers the `Bundle-Version` header value to be the database
schema version.
```

---

2. Add a dependency on the `com.liferay.portal.upgrade` module, along with any other modules your upgrade process requires, in your your module's dependency management file (e.g., Maven POM, Gradle build file, or Ivy `ivy.xml` file). An example configuration for a `build.gradle` file is shown below:

   ```
   compile group: "com.liferay", name: "com.liferay.portal.upgrade.api", version: "2.0.3"
   ```

3. Create an `UpgradeProcess` class that extends the `UpgradeProcess` base class (which implements the `UpgradeStep` interface):

   ```
   public class MyUpgradeSchemaClass extends UpgradeProcess {
     ...
   }
   ```

4. Override the `UpgradeProcess` class's `doUpgrade()` method with instructions for modifying the database. Use the `runSQL` and `runSQLTemplate*` methods (inherited from the `BaseDBProcess` class) 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, as shown in journal-service module's `UpgradeSchema` upgrade step class below which uses the `runSQLTemplateString` method to execute ANSI SQL DDL from an SQL file and the `alter` method and `UpgradeProcess`'s `UpgradeProcess.AlterColumnName` and `UpgradeProcess.AlterColumnType` inner classes as token classes to modify column names and column types:

```
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"));
  }
}
```

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"));
        }

}
```

5. If your application was modularized from a former traditional Liferay plugin application (application WAR) and it uses Service Builder, create and register a Bundle Activator to register it in Liferay DXP's `Release_` table.

6. Create an `UpgradeStepRegistrator` OSGi Component class of service type `UpgradeStepRegistrator.class` that implements the `UpgradeStepRegistrator` interface:

```
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 {


}
```

7. Override the `register` method to implement the module's upgrade registrations—abstractions for the upgrade steps required to update the database from one schema version to the next. For example, the upgrade step registrator class `MyCustomModuleUpgrade` (below) registers three upgrade steps incrementally for each schema version (past and present, `0.0.0` to `2.0.0`, `1.0.0` to `1.1.0`, and `1.1.0` to `2.0.0`).

   The first registration is applied if the module hasn't been installed previously. It contains only one empty upgrade step: `new DummyUpgradeStep()`. This registration records the module's latest schema version (i.e., `2.0.0`) in Liferay DXP's `Release_` table. Note that if the same class name is used in multiple packages, you must provide the fully qualified class name for the class, as shown in the second registration (`1.0.0` to `1.1.0`) below for the `UpgradeFoo` class:

```
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 UpgradeBar());
    }

}
```

```
**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.
```

---

8. If your upgrade step uses an OSGi service, **your upgrade must wait for that service's availability**. Use the `@Reference` annotation to declare any classes that the registrator class depends on. For example, the `WikiServiceUpgrade` registrator class below references the `SettingsFactory` class for the `UpgradePortletSettings` upgrade step. 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:

```
@Component(immediate = true, service = UpgradeStepRegistrator.class)
public class WikiServiceUpgrade implements UpgradeStepRegistrator {

    @Override
    public void register(Registry registry) {
        registry.register("0.0.1", "0.0.2", new UpgradeSchema());

        registry.register("0.0.2", "0.0.3", new UpgradeKernelPackage());

        registry.register(
            "0.0.3", "1.0.0", new UpgradeCompanyId(),
            new UpgradeLastPublishDate(), new UpgradePortletPreferences(),
            new UpgradePortletSettings(_settingsFactory), new UpgradeWikiPage(),
            new UpgradeWikiPageResource());

        registry.register("1.0.0", "1.1.0", new UpgradeWikiNode());

        registry.register(
            "1.1.0", "1.1.1",
            new UpgradeDiscussionSubscriptionClassName(
                _subscriptionLocalService, WikiPage.class.getName(),
                UpgradeDiscussionSubscriptionClassName.DeletionMode.ADD_NEW));

        registry.register(
            "1.1.1", "2.0.0",
            new BaseUpgradeSQLServerDatetime(
                new Class<?>[] {WikiNodeTable.class, WikiPageTable.class}));
    }

    @Reference
    private SettingsFactory _settingsFactory;

    @Reference
    private SubscriptionLocalService _subscriptionLocalService;

}
```

9. Upgrade the database to the latest database schema version before making its services available. To do this, configure the Bnd header `Liferay-Require-SchemaVersion` to the latest schema version for Service Builder services. For all other services, specify 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 upgrades to schema version `2.0.0` by defining the reference below:

```
@Reference(
    target = "(&(release.bundle.symbolic.name=com.liferay.comment.page.comments.web)(release.schema.version=2.0.0))"
)
private 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 to schema versions. Upgrade processes are associated
with schema versions but `VerifyProcess` instances are not.
```

Great! Now you know how to create data upgrades for all your modules.

565.1 Related Topics

- Upgrade Processes for Former Service Builder Plugins
- Upgrading Code to 7.0
- Configurable Applications

CHAPTER 566

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.

Important: The steps covered in this article only apply to modular applications that use Service Builder and were modularized from traditional Liferay plugin applications. They do not apply to you if your application doesn't 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) {
    _serviceTracker.close();
}

private ServiceTracker<Object, Object> _serviceTracker;

}
```

Follow these steps to create a Bundle Activator:

1. Create a class that implements the `org.osgi.framework.BundleActivator` interface:

   ```
   public class KnowledgeBaseServiceBundleActivator implements 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 listen 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.

Wonderful! You've set your service module's data upgrade process.

566.1 Related Topics

- Creating Upgrade Processes for Modules
- Upgrading Code to 7.0

CHAPTER 567

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.

Follow these steps to use the Build Auto Upgrade feature to test schema changes in development:

1. Create a `portal-ext.properties` file in your app server's `[Liferay_Home]/` folder if it doesn't already exist.

2. Enable Developer Mode in your app server by adding the following line to the properties file:

   ```
   include-and-override=portal-developer.properties;
   ```

   The Build Auto Upgrade feature is a global property `schema.module.build.auto.upgrade` in the file `[Liferay_Home]/portal-developer.properties`, so enabling Developer Mode automatically enables this property as well.

   Alternatively, if you prefer not to enable all the other properties included in Developer Mode, you can just add the `schema.module.build.auto.upgrade` property to your `portal-ext.properties` file and set it to `true`:

   ```
   schema.module.build.auto.upgrade = true;
   ```

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 table below 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.

567.1 Related Topics

Creating Data Upgrade Process for Modules

CHAPTER 568

MANAGING USER-ASSOCIATED DATA STORED BY CUSTOM APPLICATIONS

Administrators can delete or anonymize User Associated Data (UAD) using management tools that aid compliance efforts with the EU's General Data Protection Regulation (GDPR). Out of the box, the UAD management tool supports Liferay DXP applications (Blogs, Documents and Media, etc.), and you can also anonymize data stored by your custom applications.

This task is made easier for Service Builder applications. At the core of the anonymization effort, you must identify the model entity's fields to anonymize. With Service Builder, attach anonymization attributes to elements in the `-service` module's `service.xml` file. For the entire DTD for Service Builder, see here. These two are the most important attributes for the UAD framework:

- The `uad-anonymize-field-name=fieldName` attribute indicates a field whose value is replaced by that of the anonymous user in the UAD deletion process.
- The `uad-nonanonymizable=true` attribute indicates data that cannot be anonymized automatically and must be reviewed by an administrator.

Once your application uses the UAD framework to manage User data, there are more features in 7.0 that make searching and deleting User Associated Data even easier.

CHAPTER 569

ADDING THE UAD FRAMEWORK TO A SERVICE BUILDER APPLICATION

You'll touch two modules of a Service Builder application to implement the UAD features: the `-service` module and a brand new module that Service Builder generates for you, the `-uad` module.

569.1 Update the Service Module

Before you specify your model entity's fields to manage with the UAD framework, make sure you have the right dependencies.

569.2 Include Dependencies

To compile the code that Service Builder generates, you need dependencies on `com.liferay.petra.String` and `com.liferay.portal kernel`. Make sure your service module's `build.gradle` includes both:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "4.4.0"
    compileOnly group: "com.liferay", name: "com.liferay.petra.string", version: "3.0.0"
    ...
}
```

569.3 Choose the Fields to Anonymize

In the `service.xml` file, choose the fields to be handled by the framework by adding a `uad-anonymize-field-name="[fieldName]"` or `uad-nonanonymizable=true`.

For example, to replace the `userName` field of your custom entity with the `fullName` of the anonymous user,

```
<column name="userName" type="String" uad-anonymize-field-name="fullName" />
```

This declaration specifies that the content field cannot be auto-anonymized by the framework but should be reviewed manually.

```
<column name="content" type="String" uad-nonanonymizable="true" />
```

Run Service Builder. A new module is generated alongside your other modules, called my-app-uad. It requires a little massaging.

569.4 Update the UAD Module

First, include your dependencies, and then provide your application's name to the user interface.

569.5 Include Dependencies

The new module is generated without a build script, so you must provide one. It should include dependencies on osgi.service.component.annotations, com.liferay.portal.kernel, com.liferay.petra.string, the com.liferay.user.associated.data.api, and your own application's -api module:

```
dependencies {
  compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "4.4.0"
  compileOnly group: "com.liferay", name: "com.liferay.user.associated.data.api", version: "4.1.1"
  compileOnly group: "com.liferay", name: "com.liferay.petra.string", version: "3.0.0"
  compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
  compileOnly project(":modules:custom:custom-api")
  }
```

569.6 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]=Custom App
```

The bracketed text is the Bundle-SymbolicName from your -uad module's bnd.bnd file.

While this approach is recommended, it has one downside: multiple language keys are used to label a single application. Liferay DXP applications use the com.liferay.lang.merger plugin to avoid this. Here's the relevant part of the Blogs application's blogs-uad/build.gradle:

```
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"]
}
```

CHAPTER 570

ENHANCING THE DATA ERASURE UI

As of 7.0, there are new features that enhance an administrator's experience finding data in the Personal Data Erasure UI:

Filtering User content can now be viewed and acted upon based on whether it is part of the User's personal Site, some other Site, or the overall company.

Search A search bar now allows for filtering content based on a search term.

Hierarchy Display If there are multiple types of content that are related in a hierarchy, you can define that relationship using a `UADHierarchyDeclaration`, and the user interface shows that hierarchy (e.g. files and folders).

570.1 Filtering and Searching in the Data Erasure UI

To support filtering and searching in your custom entities, implement three methods in the `UADDisplay` class (found in your `-uad` module):

- `isSiteScoped`
- `search`
- `searchCount`

The `isSiteScoped` method returns a boolean, determining if the entity can be associated with a particular Site. This is used to determine which filter they are associated with ("instance", "personal-site", or "regular-sites").

The `search` method takes the following parameters:

`userId`: The `userId` of the selected User.

`groupIds`: An array of `groupIds` used to filter which data is shown by which groups it is associated with. In the case that no `groupIds` are given (it can be null), the search method should return data that is not scoped to any given group.

`keywords`: The contents of the search bar. The search method should filter by whatever fields are relevant for the given entity.

`orderByField`: The name of the field used to sort the results. This is one of the names returned by `getSortingFieldNames`.

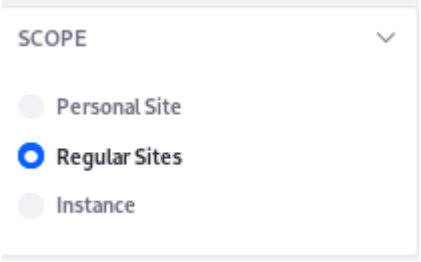

Figure 570.1: Items in the Personal Data Erasure screen can be filtered by scope.

`orderByType`: Sort the results in ascending order or descending order (`asc` or `desc`), for pagination.
`start`: The starting index of the result set. For pagination.
`end`: The ending index of the result set. For pagination.
The `searchCount` method takes the following parameters, which are treated identically to the ones in `search`:

- `userId`
- `groupIds`
- `keywords`

Read here for instructions on how to implement search and filtering for your entity.

570.2 Hierarchy Display

Hierarchical UAD display optionally shows entities with a natural parent-child relationship (for example, Document Library Folders and Document Library File Entries). Viewing these entities in a hierarchy helps administrators make sense of the data they're reviewing for possible erasure.
To implement a hierarchy display, you must do two things:

1. Implement a `UADHierarchyDeclaration` class.
2. Add a method to the `*UADDisplay` class for each type involved in the hierarchy.

Once implemented, a hierarchy view is displayed for any of the types returned in the `UADHierarchyDeclaration`. For container entities, a count of all child entities is calculated and displayed using the hierarchy-related methods in `UADDisplay`.

570.3 UAD Hierarchy Declaration

The `UADHierarchyDeclaration` defines the types in the hierarchical relationship. There are two classifications for a type in a hierarchy: a *container*, and a *non-container*. These are defined by `getContainerUADDisplays` and `getNoncontainerUADDisplays`.
Each returns an array of one or more `UADDisplay` classes. Containers can be parents or children in the hierarchy. An example is a folder in a file system, which can contain both files and other

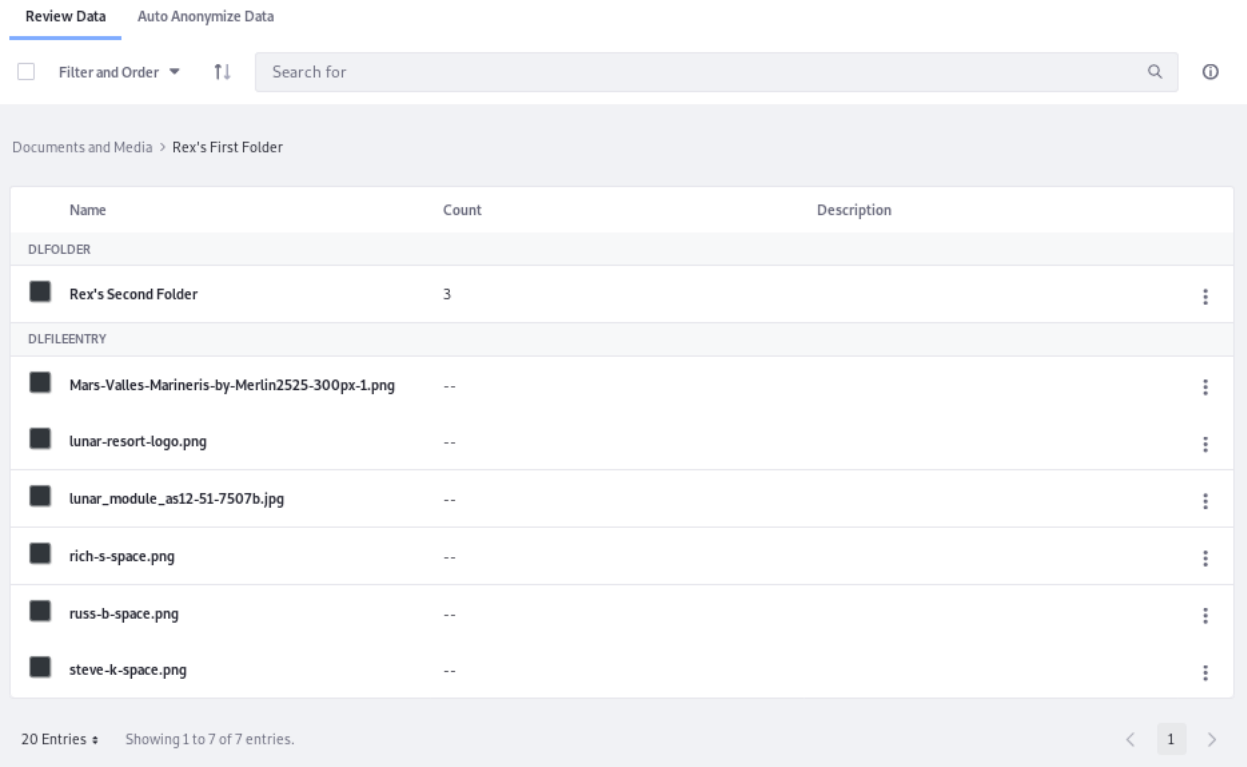

Figure 570.2: Hierarchical representation of nested entities is useful for administrators reviewing User data for possible deletion.

folders. The non-container entities can only be children in the hierarchy. An example is a file in a file system.

The `UADHierarchyDeclaration` provides some other methods for display purposes.

- A label for the hierarchy is provided through the `getEntitiesTypeLabel` method.
- If any additional information is to be displayed in the table for any of the entity types in addition to the count, those column names should be returned by `getExtraColumnNames`. This is optional.

See `DLUADHierarchyDeclaration` for an example.

570.4 Add methods to UADDisplay

Each type involved in the hierarchy should implement some additional methods in `UADDisplay`.

These methods must be implemented by containers and non-containers alike, in the `*UADDisplay` class:

`getName` returns a display name for the given entity.

`getParentContainerClass` returns the class of the type that contains this type. It can return itself (for example, a folder can contain a folder).

`getParentContainerId` returns the primary key of the container that contains the entity passed to this method.

`isUserOwned` returns whether or not the given entity is owned by the user.

Additionally, implement `getTopLevelContainer` in the `*UADDisplay` class for all types classified as *containers*. It's used to derive the count of how many user-owned entities are contained inside a given container's tree. It answers the question "which type `T` ancestor of `childObject` is an immediate child of the container identified by `parentContainerClass` and `parentContainerId`?" The method may return `null` if `childObject` is not a child of the parent container. This method is the most complicated to implement and requires some consideration for each case. Refer to the test case for examples of the requirements used for `DLFolder`: DLFolderUADDisplayTest#testGetTopLevelContainer

See the actual implementation for `DLFolder` in DLFolderUADDisplay#getTopLevelContainer.

The method returns either `null` or the container object of type T that is the top level container of the `childObject` (which could be any type of object that is a part of the hierarchy). This container does not necessarily have to be owned by the user, but is understood to contain data related to the user. This information is used to count how much user data is inside the container designated by `parentContainerClass` and `parentContainerId`.

```
@Override
public DLFolder getTopLevelContainer(
    Class<?> parentContainerClass, Serializable parentContainerId,
    Object childObject) {

    try {
        DLFolder childFolder = null;

        if (childObject instanceof DLFileEntry) {
            DLFileEntry dlFileEntry = (DLFileEntry)childObject;

            childFolder = dlFileEntry.getFolder();
        }
        else {
            childFolder = (DLFolder)childObject;
        }

        long parentFolderId = (long)parentContainerId;

        if ((childFolder.getFolderId() == parentFolderId) ||
            ((parentFolderId ≠
                DLFolderConstants.DEFAULT_PARENT_FOLDER_ID) &&
             !StringUtil.contains(
                 childFolder.getTreePath(), String.valueOf(parentFolderId),
                 "/"))) {

            return null;
        }

        if (childFolder.getParentFolderId() == parentFolderId) {
            return childFolder;
        }

        List<Long> ancestorFolderIds = childFolder.getAncestorFolderIds();

        if (parentFolderId == DLFolderConstants.DEFAULT_PARENT_FOLDER_ID) {
            return get(ancestorFolderIds.get(ancestorFolderIds.size() - 1));
        }

        if (ancestorFolderIds.contains(parentFolderId)) {
            return get(
                ancestorFolderIds.get(
                    ancestorFolderIds.indexOf(parentFolderId) - 1));
        }
    }
    catch (PortalException pe) {
        _log.error(pe, pe);
```

```
    }

    return null;
}
```

The exact implementation details vary for each entity type.

CHAPTER 571

FILTERING AND SEARCHING UAD-MARKED ENTITIES

In the data erasure UI, it's important that administrators can find what they're looking for. The native Liferay DXP entities support filtering and search, and when you follow the steps here, your entities will, too.

To add filtering and searching for your custom entities, implement three methods in the `UADDisplay` class (in your application's `-uad` module):

571.1 Filtering

The `isSiteScoped` method returns a boolean denoting if the entities can be associated with a particular Site: `false` if not, and `true` if the entities are scoped to a Site. This determines which filter they are associated with ("instance", "personal-site", or "regular-sites").

```
@Override
public boolean isSiteScoped() {

    return false;
}
```

571.2 Search

Implement the `search` and `searchCount` methods to enable search in the UAD interface:

1. The `search` method must return a `List` of entities associated with the `userId`. For example, you could search the database for records associated with the `userId`:

    ```
    @Override
    public List<T> search(
        long userId, long[] groupIds, String keywords, String orderByField,
        String orderByType, int start, int end) {

        FooService<T> fooService = getFooService();

        return dummyService.getEntities(userId);
    }
    ```

But if you've gone through the trouble of indexing your model entity's fields in a search engine, it's more likely you'll want to do the initial search, querying for documents matching the `userId`, at the search engine level. After the search, retrieve the matching entities from the database.

```
@Override
public List<T> search(
    long userId, long[] groupIds, String keywords, String orderByField,
    String orderByType, int start, int end) {

    SearchContext searchContext = new SearchContext();

    searchContext.setStart(start);
    searchContext.setEnd(end);
    searchContext.setGroupIds(groupIds);
    searchContext.setKeywords(keywords);

    BooleanQuery booleanQuery = BooleanQueryFactoryUtil.create(
        searchContext);

    booleanQuery.addExactTerm("userId", userId);

    BooleanClause booleanClause = BooleanClauseFactoryUtil.create(
        booleanQuery, BooleanClauseOccur.MUST.getName());

    searchContext.setBooleanClauses(new BooleanClause[] {booleanClause});

    Indexer indexer = IndexerRegistryUtil.getIndexer(FooEntry.class);

    Hits hits = indexer.search(searchContext);

    List<FooEntry> fooEntries = new ArrayList<FooEntry>();

    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 = _fooEntryLocalService.getFooEntry(fooEntryId);
            } catch (PortalException pe) {
                    _log.error(pe.getLocalizedMessage());
            } catch (SystemException se) {
                    _log.error(se.getLocalizedMessage());
            }

            fooEntries.add(fooEntry);
    }

    return fooEntries;
}
```

It largely boils down to instantiating and populating the search context, which gets passed to the `indexer.search` call to retrieve the `Hits`. Subsequently, populate the `List` by iterating through the `Hits`, using each one's `ENTRY_CLASS_PK` field as the primary key of the entity in the call to the entity's getter. The `BooleanClause` construction and inclusion in the search context ensures that all the results returned correspond to the `userId` that's passed to this method.

2. The `searchCount` method returns a long of the result `List`'s `size` method. You could just invoke the class's `search` method, then call the `List` object's `size` method.

```
@Override
public long searchCount(long userId, long[] groupIds, String keywords) {
    List<T> results = search(
        userId, groupIds, keywords, null, null, QueryUtil.ALL_POS,
        QueryUtil.ALL_POS);

    return results.size();
}
```

But, again, if the model entity is being indexed in a search engine, you can use it to get a count without ever hitting the database. Using the `Hits` object returned from a search (see the code from step 1, but don't include `start` and `end` parameters in the `SearchContext`), call `hits.getLegnth()` and you get the count, as an `int`.

Now administrators responsible for complying with GDPR or other data erasure concerns can search and filter your entity from the Liferay DXP UAD interface.

CHAPTER 572

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 flex their muscles in content creation. These articles 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
- Supporting Custom Content Types
- Screen Navigation

For more information on applying these frameworks for users, see the Web Experience Management user articles.

CHAPTER 573

PAGE FRAGMENTS

You can use Page Fragments to take your design vision and accurately realize it on a web page. You start with a "blank slate." You then have three tools at your disposal to accomplish your vision:

HTML: The markup of the fragment. Fragments use standard HTML with special tags to add dynamic behavior.

CSS: Styles and positions the fragment's markup.

JavaScript: Provides dynamic behavior to the fragment.

The HTML, CSS, and JavaScript are all completely standard, but can be enhanced with Liferay-specific features. You can specify text, images, and links as editable and provide for "rich" text with formatting.

You can also access the FreeMarker templates engine from your HTML using the alternative (square bracket) syntax. Learn more about available FreeMarker objects in Front-end Reference.

Liferay portlets can also be embedded in Fragments as widgets, making pages with Fragments more dynamic than regular web content.

Now you'll step through some Page Fragment basics.

573.1 Developing Page Fragments

There are two types of Page Fragments: *Sections* and *Components*. A Section defines columns, padding, and spacing on the page. A Component contains content that is added to a Section.

Fragments are created inside of Collections. Collections provide an easy way to manage and share groups of related Fragments. Users navigate Collections when selecting Fragments to add to a page. To see examples, the admin page shows all the out-of-the-box Fragments (and their code).

You can create and manage Fragments and Collections without using any external tools, but you can also use your preferred web development tools. For an explanation of Fragment creation using Liferay's built in tools, see Creating a Fragment.

573.2 Making a Fragment Configurable

Note: Defining configurations for Page Fragments is available in Liferay DXP 7.2 SP1+ and Liferay Portal GA2+.

Page Fragments are also configurable: defining configuration options for your fragment eliminates having to maintain multiple other fragments similar in style. For example, if you want a dark background banner and a light background banner, you can create one banner with a configuration option for background type.

The following field types are supported for Fragment configurations:

- `checkbox`
- `colorPalette`
- `itemSelector`
- `select`
- `text`

This is available for all Fragment types (e.g., Fragment Renderer, etc.).

For more information, see Making a Fragment Configurable.

573.3 Fragments CLI

To streamline fragment development, 7.0 provides command line tools for generating, importing, and exporting fragments and fragment collections. For more information about the CLI, see the official Liferay Fragments CLI project reference. Using this CLI is also covered in Developing a Fragment using Desktop Tools.

573.4 Contributed Collections

Most of the time, Page Fragments are created and imported through the Page Fragments interface or created directly using the built-in tools. Any user with the right permissions can update or edit Page Fragments created like this. You may have certain situations, however, where you want 100% static fragments that cannot be modified. In this case you can create a Contributed Fragment Collection.

Contributed Fragment Collections are deployable modules containing Page Fragments. Those fragments can be used just like regular fragments, but are not contained in the database, and cannot be modified except by updating the module they came from. Use the Creating Contributed Collections guide to learn to create your own Contributed Collections.

573.5 Fragment Specific Tags

In addition to standard HTML, CSS, and JavaScript you can use Liferay-specific tags to make editable sections or embed widgets in your Fragment.

Editable elements can be modified before publication. This means that web developers can create simple, reusable fragments that have identical formatting, but contain elements that are adaptable to the specific context.

You can make text, images, and links in a fragment editable by using an `<lfr-editable>` tag. The `<lfr-editable>` tag requires a unique `id`, a type, and some content of the specified type inside.

The following four `type` options are available in an `lfr-editable` tag:

`text`: Creates a space for editable plain text.

`image`: Must contain a valid `<img>` tag which can then be replaced with an image before publishing—including those from Documents and Media.

`rich-text`: Provides rich text formatting, such as bold, italics, underline, links, and predefined styles.

`link`: Must contain a valid anchor tag for which the style, target URL, and link text can be edited before publishing.

The text or images you provide here are the default values for the fields. You may want to display them in the final version of the page, or you may want filler text that should be replaced before the page is published.

All of these work together to help you create dynamic, reusable elements for building a site. For example, if you need a small text box with an image and link to provide a product description, you can create a fragment containing editable filler text, space for an editable image, the appropriate formatting, and an editable link. That fragment can be added to multiple pages, and marketers can define the image, text, and link for each product they need to describe.

You can make a Fragment even more dynamic by including a widget. Currently, portlets are the only embeddable types of widgets, but other options are planned.

You can find a complete list and usage examples of these in the Page Fragments Reference.

573.6 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.

573.7 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.

573.8 JavaScript

Avoid adding a lot of JavaScript code, since it isn't easily reusable. Instead, reference external JS libraries.

CHAPTER 574

DEVELOPING FRAGMENTS

This tutorial assumes you understand Fragments and Collections. If you don't, read the Creating Page Fragments article first. Once, you're ready, start by creating a Collection:

1. From the menu for your selected site, click *Site Builder* → *Page Fragments*.
2. Create a new Collection named *Developing Fragments*.

First, you'll create a *Section*.

574.1 Creating a Section

The list of Collections appears on the left in the Page Fragments page.

1. Ensure that you are in the *Developing Fragments* collection.
2. Click the *New* button  and select *Section*.
3. Name your Section *Basic Section*

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. The Fragment Editor even comes with autocomplete functionality!

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 pane goes to `index.js`. The preview pane renders everything as it looks on the page.

Warning: Including images inline in base64 in your Page Fragments can increase publishing, import, and export times for pages using those Fragments. Use references to resources in your Page Fragments instead.

A Section defines a work space. Now create a section with an editable rich text area where content can be entered:

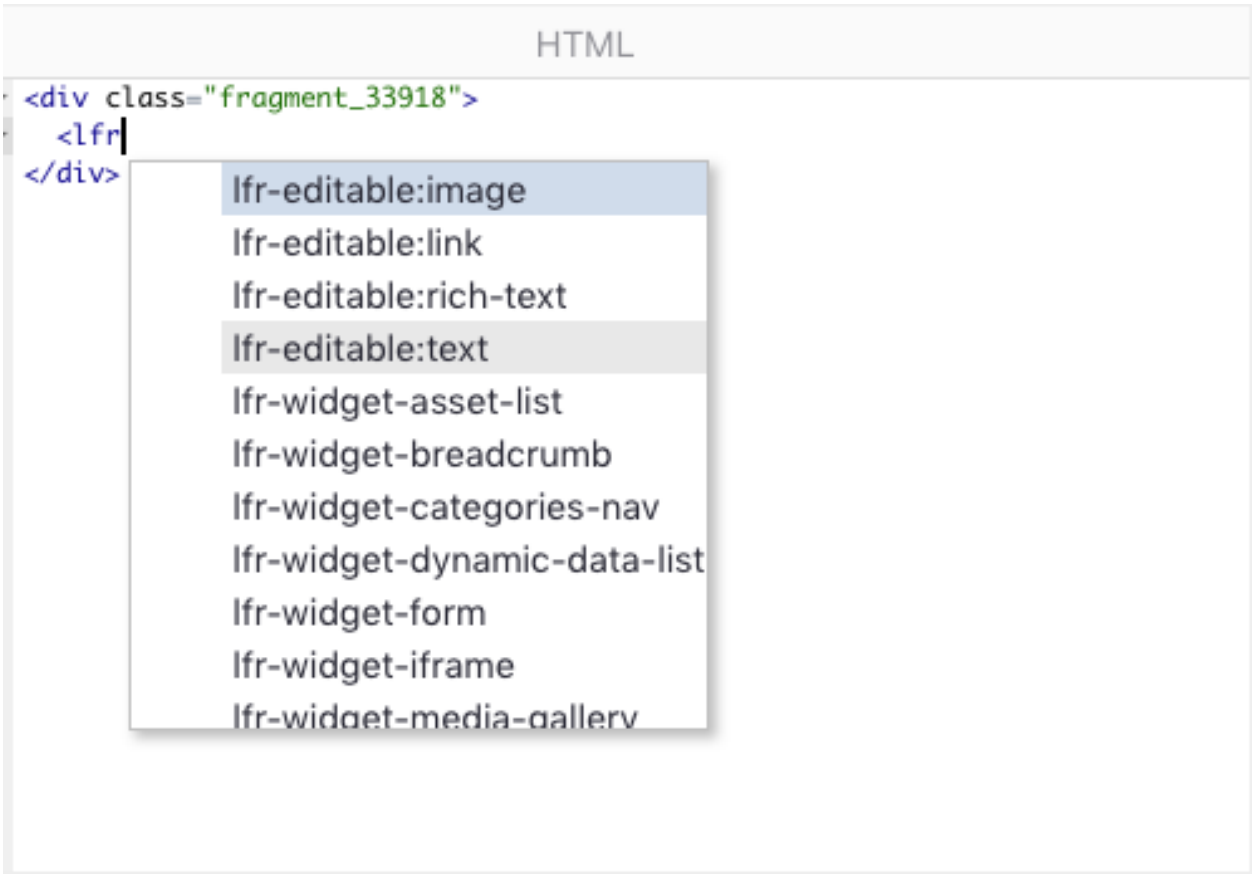

Figure 574.1: The Fragment editor provides autocomplete for Liferay Fragment specific tags.

1. Add the following code inside the HTML pane:

```
<div class="banner py-6 py-md-8 text-white" data-lfr-background-image-id="banner">
    <div class="container my-lg-6">
        <div class="row">
            <div class="col-12 col-md-8 col-xl-6">
                <h1>
                    <lfr-editable id="01-title" type="rich-text">
                        Banner Title Example
                    </lfr-editable>
                </h1>

                <div class="mb-4 lead">
                    <p>
                        <lfr-editable id="02-subtitle" type="rich-text">
                            This is a simple banner component that you can use must provide extra information.
                        </lfr-editable>
                    </p>
                </div>

                <lfr-editable id="03-link" type="link">
                    <a href="#" class="btn btn-primary">Go Somewhere</a>
                </lfr-editable>
            </div>
        </div>
    </div>
</div>
```

2. Replace the code in the CSS pane with the following:

```
.banner {
    background-color:#415fa9;
    background-position: center;
    background-size: cover;
}
```

3. Click *Publish* to save your work and make it available to add to a content page.

Note: When you start typing the name of a tag, the HTML editor provides auto-completion for `lfr` tags like editable elements and embeddable widgets.

As you work, you can observe your changes in the preview pane.

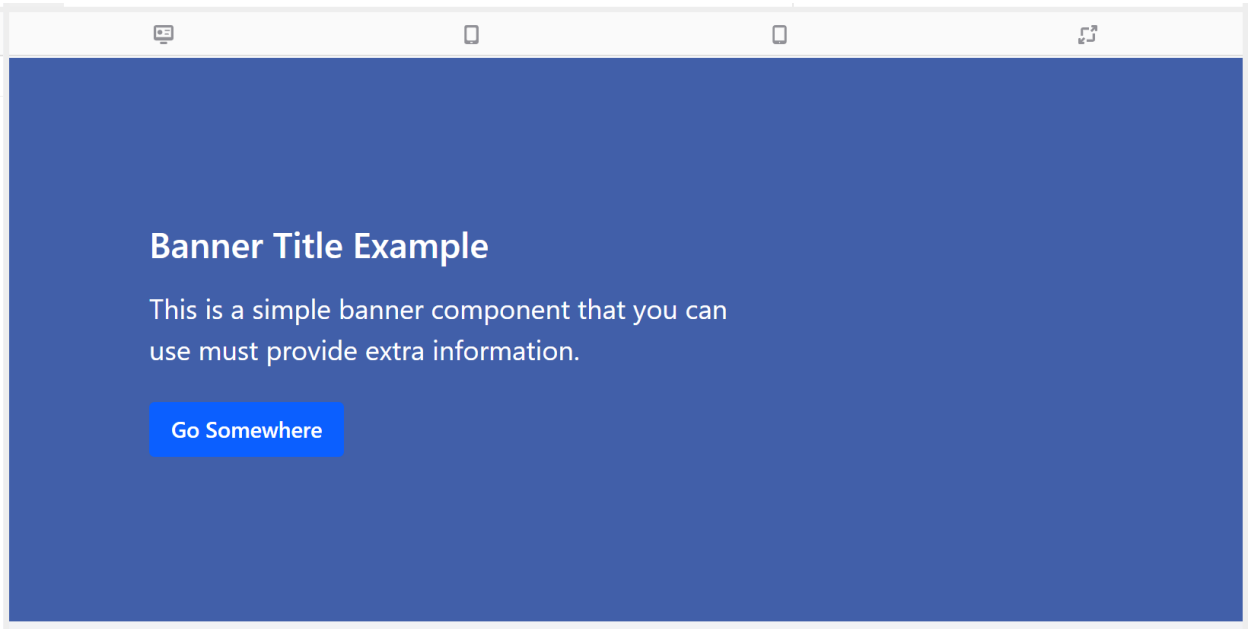

Figure 574.2: The Fragment editor with HTML and CSS code and a live preview.

574.2 Creating a Component

Components are simple, reusable elements for building parts of a page. Next create a button with a link as a Component:

1. Go back to the *Site Builder* → *Page Fragments* page and select the *Developing Fragments* Collection.
2. Click the *New* button and select *Component*.
3. Name it *Basic Component*.

4. Back in the editor, add the following code inside the HTML pane:

```
<div class="basic-link-button">
    <lfr-editable id="btn00" type="link">
        <a href="#" class="btn btn-primary">Read More</a>
    </lfr-editable>
</div>
```

5. Click *Publish* to save your work and make it available to add to a content page.

This fragment did not require any CSS. For the button link, no target is provided by default, so the link must be configured when it is added to the page.

From here, the Fragment can be added to a Page. To see this process in action, see the Building Content Pages article.

CHAPTER 575

MAKING A FRAGMENT CONFIGURABLE

Note: Defining configurations for Page Fragments is available in Liferay DXP 7.2 SP1+ and Liferay Portal GA2+.

Defining configuration options for a Fragment gives it more flexibility, reducing the number of Fragments you must maintain. To make a Fragment configurable,

1. Navigate to the *Site Builder* → *Page Fragments* page.
2. Click the *Actions* button (⋮) → *Edit* for the Fragment (Section or Component) you want to make configurable.
3. Select the *Configuration* tab at the top of the page.

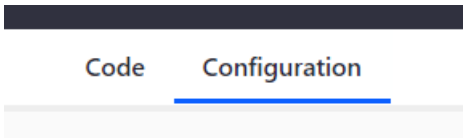

Figure 575.1: Switch from the Code tab to the Configuration tab to create your configuration logic.

4. In the editor, add your JSON code. This code is added to your fragment's `index.json` file. For example, the code below provides the `select` option to choose *dark* or *light* for a Fragment's heading:

```
{
    "fieldSets": [
        {
            "label": "heading",
            "fields": [
                {
                    "name": "headingAppliedStyle",
                    "label": "applied-style",
                    "description": "this-is-the-style-that-will-be-applied",
                    "type": "select",
                    "dataType": "string",
                    "typeOptions": {
                        "validValues": [
```

```
                            {
                                "value": "dark"
                            },
                            {
                                "value": "light"
                            }
                        ]
                    },
                    "defaultValue": "light"
                }
            ]
        }
    ]
}
```

```
**Note:** The `label` property is optional. If it's left out, your
configuration option has no title.
```

```
**Note:** If your configuration is invalid, you can't save the
code. Be sure to always have a valid JSON configuration before previewing
or saving it.
```

```
The configuration values selected by the user are made available to the
Fragment developer through the FreeMarker context. A configuration value can
be referenced using the notation `${configuration.[fieldName]}`. For the
example snippet above, `${configuration.headingAppliedStyle}` returns
`dark` or `light` depending on the configuration value selected by the user.
```

5. You can refer to your Fragment's configuration values in its HTML file (e.g., `index.html`). Navigate back to the *Code* tab at the top of the page and add your HTML. For example,

   ```
   [#if configuration.headingAppliedStyle == 'dark']
   ...
   [#else]
   ...
   [/#if]
   ```

   Configuration values inserted into the FreeMarker context honor the defined `datatype` value specified in the JSON file. Therefore, for this example, `configuration.headingAppliedStyle?is_string` is `true`.

6. Click *Publish* to save your work and make it available to add to a Content Page.

Note: You can also make a Fragment configurable by leveraging the Fragments Toolkit. You can create/modify the Fragment's configuration JSON file and then reimport the Fragment to your Liferay instance. For more information, see Page Fragment Desktop Tools.

Although this example highlights accessing configuration values in HTML via the FreeMarker context, you can also access these values via JavaScript. JavaScript configuration objects are named the same as their FreeMarker counterparts.

For example, a configuration object could be built like this:

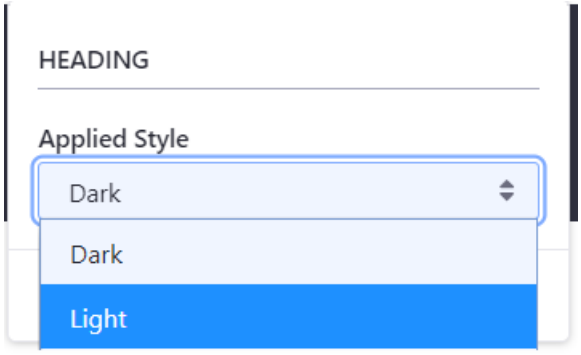

Figure 575.2: You can click your Fragment to view its configuration options.

```
var configuration = {
    field1: value1,
    field2: value2
}
```

Another example of setting the configuration object and using it is shown below:

```
const configurationValue = configuration.field1

console.log(configurationValue);
```

For more examples of Fragment configuration, see Fragment Configuration Types. Awesome! You now have a configurable Fragment!

CHAPTER 576

MANAGING FRAGMENTS AND COLLECTIONS

After you create Collections and Fragments, you have a handful of options for managing them. You'll explore several management options next.

576.1 Collections Management Menu

To access the collections management menu,

1. Select the Collection you want to manage from the *Collections* list.
2. Click on the ⋮ menu next to the collection name.
3. Select whether you want to *Edit*, *Export*, *Import*, or *Delete* the collection.
   Edit: change the name or description for the collection.
   Export: download a `.zip` file containing the full collection.
   Import: select a `.zip` file to upload with additional Fragments.
   Delete: remove the current collection and all of its contents.

Next, you'll learn about the Fragment Management Menu.

576.2 Fragment Management Menu

To access the fragment management menu,

1. Select the Collection containing the Fragment you want to manage from the *Collections* list.
2. Click on the ⋮ menu next to the Fragment name.
3. Select whether you want to *Edit*, *Rename*, *Move*, *Make a Copy*, *Change Thumbnail Export*, or *Delete*.

Did you know you can enable automatic propagation for Fragments? You'll do this next.

576.3 Propagating Fragment Changes Automatically

Note: Propagating Fragment changes is available in Liferay DXP 7.2 SP1+ and Liferay Portal GA2+.

By default, when a Fragment developer makes a change to an existing fragment, the change is not automatically propagated to the pages that were using it. This gives marketers and page authors more control over the pages they own, avoiding unexpected changes. For example, if three pages were using the same Fragment, an update to the Fragment could introduce unintended changes to some of the pages using it. While this is a safeguard for the production environment, developers must manually propagate Fragment changes during testing, which can be tedious. To give developers more freedom, you can enable automatic propagation for Fragment changes:

1. Navigate to the Control Panel → *Configuration* → *System Settings* → *Page Fragments*.
2. Enable the checkbox *Propagate Fragment Changes Automatically*.
3. Click *Save*.

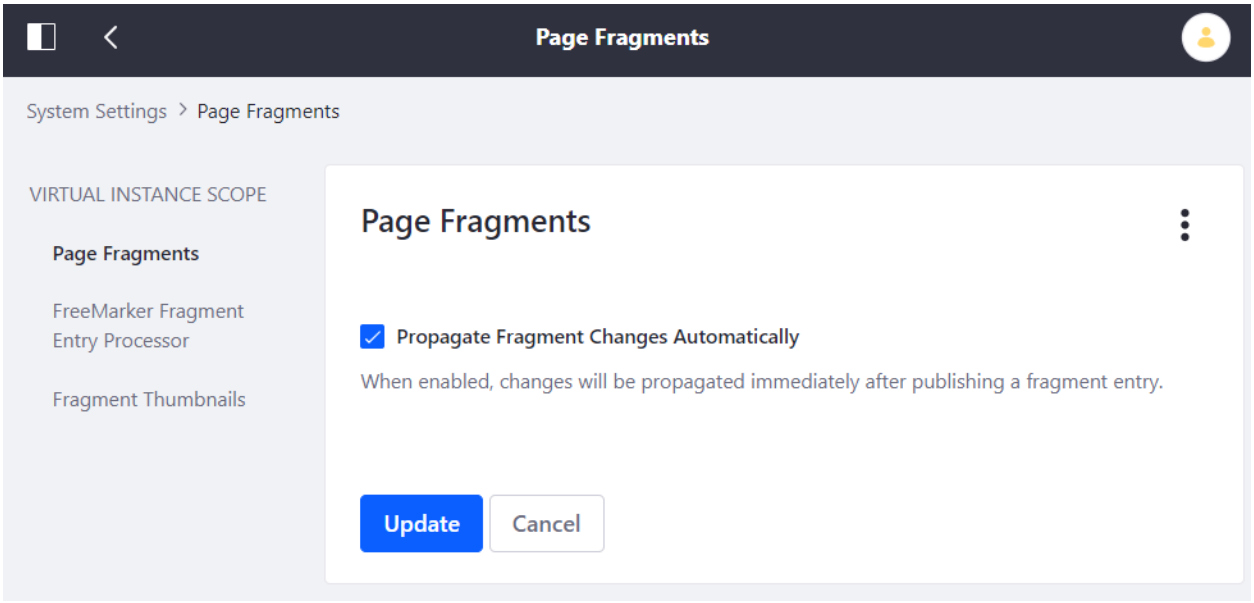

Figure 576.1: Once Fragment propagation is enabled, developers can automatically propagate Fragment changes to all pages using them.

Great! You've enabled Fragment propagation system wide! Now when a developer publishes a Fragment, the changes apply immediately to all Content Pages, Content Page Templates, and Display Page Templates using it, overwriting existing Fragment code. Automatic propagation works only for HTML, CSS, and JS Fragment code, not the editable values.

Note: It's recommended to only leverage this functionality during testing, as automatic propagation on the production environment can cause unintended consequences.

When using the Fragment Editor, you're now notified that automatic Fragment propagation is enabled.

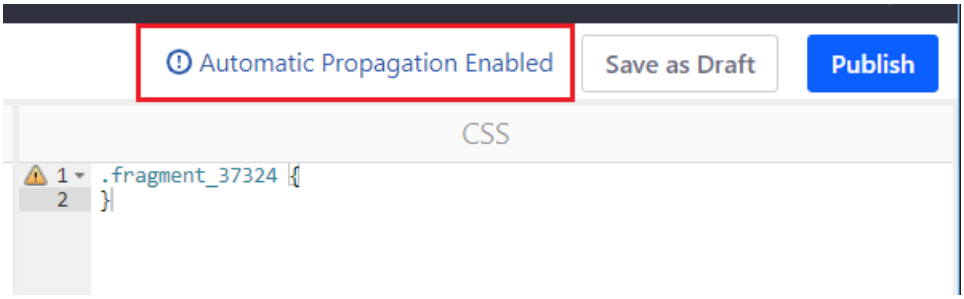

Figure 576.2: You're notified when automatic propagation is enabled.

Now that you've seen how to use Liferay's built-in tools to manage Fragments, you can see how to do it using your own tools of choice and the Fragments Toolkit.

CHAPTER 577

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.

577.1 Collection Format

To import a Collection into Liferay DXP, 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>"}`.
- `language.properties`: the language keys defined for the collection.
    - `[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

```
{
    "cssPath": "index.css",
    "configurationPath": "index.json",
    "htmlPath": "index.html",
    "jsPath": "index.js",
    "name": "<fragment-name>",
    "type": "<fragment-type>"
 }
```

          Update the *Path properties in your `fragment.json` file if you change the names of your `index.*` files.

        * `index.css`: the CSS source for the fragment.
        * `index.html`: the HTML source for the fragment.
        * `index.json`: a JSON file that defines the fragment's configuration.

    * `index.js`: the JavaScript source for the fragment.

    * `thumbnail.png`: the thumbnail that will display when the Fragment is displayed in a list.

  - `[resources]/`: a folder containing any additional images or other external files needed for the fragment.

A collection can contain any number of fragments, so you can have multiple subfolders in the collection. This format is the same as what's exported from within Liferay. If you import a `.zip` file that is not organized like this, any fragments that are found will be imported into special collection called *Imported* which is created for orphaned fragments.

577.2 Fragment CLI

You can manage Fragment creation and deployment manually, or you can use Liferay's Fragment CLI:

1. Follow the project instructions to set up the Fragments Toolkit.
2. Run `yo liferay-fragments`.
3. Follow the prompts to create a fragment.

Now you will have the basic structure created, but there's still more that the Fragments Toolkit can help you with.

Note: You can see all of the available tasks inside the `scripts` section in the Fragment CLI `package.json`.

577.3 Creating Collections

Before you can create any Page Fragments, you must create a Collection. You can learn more about Collections in the Creating Page Fragments section. Creating a Collection will create the base folder structure and some information about your Collection. To do this,

1. From inside of your project, run `npm run add-collection`.
2. Follow the prompts to name your Collection.

You can now create Page Fragments inside of this Collection.

577.4 Creating Fragments

A Page Fragment is made up of three primary files, `index.html`, `index.css`, and `index.js`. However, the files must be properly arranged in the folder structure and have the appropriate metadata to be imported onto your server. The Fragments Toolkit will create the files in the correct hierarchy with all of the necessary information.

1. From inside of the Collection you created, run `npm run add-fragment`.
2. Follow the prompts to add the necessary information about your Page Fragment.

Now the files are all created and you can edit them using your editor of choice.

577.5 Importing and Exporting Fragments

The Fragments Toolkit can connect to your currently running Liferay DXP to import and export fragments. You can even have fragments that you create with the toolkit imported into Liferay DXP automatically.

- To get collections and fragments from a running server, run `npm run export`.
- To send the collections and fragments from your current project to a running server, run `npm run import`. If your Fragment's configuration JSON (if available) is invalid, the import fails and provides an error message.
- To have collections and fragments automatically imported into Liferay DXP as they are created or modified, run `npm run import:watch`.
- To preview how a fragment will look when it's imported, run `npm run preview`. This renders a fragment on a specified Liferay server without importing it. When changes are made to the fragment while it's previewed, changes are auto reloaded to rapidly display updates. Note, this is available for Liferay DXP 7.2 SP1+ and Liferay Portal 7.2 GA2+. You must install the OAuth 2 plugin in your portal instance for this command to work properly.
- To create a `.zip` file that can be manually imported into Liferay DXP, run `npm run compress`.

With these tools at your disposal, you can more efficiently manage creating and editing Page Fragments with whatever tools and environments work best for you.

CHAPTER 578

CREATING A CONTRIBUTED FRAGMENT COLLECTION

This document has been updated and ported to Liferay Learn and is no longer maintained here.

To create a Contributed Fragment Collection, a developer must,

1. Create a module which will contain the necessary logic and the fragments.
2. Extend the class `BaseFragmentCollectionContributor` with all the logic for reading the contributed fragments.
3. Add fragments as resources in the module.

Once you deploy the module, any fragments contained in it will be available for use.
To better understand Contributed Fragment Collections, create one called `DemoFragmentCollectionContributor`.

578.1 Create a Module

First you must create the module in your development environment. Follow the instructions in Creating a Project.

578.2 Create the Java Class

Next, you must create the Java package and class to handle the logic for the contributed collection:

1. Create a package in your module named `com.liferay.fragment.collection.contributor.demo`
2. Inside of that package, create a Java class named `DemoFragmentCollectionContributor` that extends `BaseFragmentCollectionContributor`.
3. Above the class declaration, add the `@Component` annotation to set the service class:

```
@Component(service = FragmentCollectionContributor.class)
```

4. Create the variable for the servlet context:

```
private ServletContext _servletContext;
```

5. Define the `getFragmentCollectionKey()` and `getServletContext()` methods:

```
@Override
public String getFragmentCollectionKey() {
    return "DEMO";
}

@Override
public ServletContext getServletContext() {
    return _servletContext;
}
```

6. Below that use the `@Reference` annotation to define your module's symbolic name:

```
@Reference(
    target = "(osgi.web.symbolicname=com.liferay.fragment.collection.contributor.demo)"
)
```

```
**Note:** `osgi.web.symbolicname` must match `Bundle-SymbolicName` from `bnd.bnd`
```

---

7. Organize your imports and save.

578.3 Create the Resources

Next you need to include the fragments that you want to contribute in your module:

1. In your module's `resources/` folder, create the folder structure `/com/liferay/fragment/collection/contributor/`

```
**Note:** The class package name and resources package name must match
(e.g. `[my.class.package.structure].dependencies`).
```

---

2. Copy the Fragments you want to distribute into the folder. You can learn how to create a Fragment in the Creating Fragments section.
3. Create a file named `collection.json` in the same folder with this format:

```
{
    "fragments": [
        "[fragment-1]",
        "[fragment-2]",
        "[fragment-3]",
            ...
    ],
    "name": "[collection-name]"
}
```

If a fragment is not listed in `collection.json`, it will not be available in the Contributed Collection, even if the files are included in the module.

Next, you'll configure the module's metadata so the fragments are imported.

578.4 Configuring the Metadata

Follow these steps:

1. Open your bundle's `bnd.bnd` file and add the `Web-ContextPath` header to point to your bundle's folder so the fragment resources are loaded properly:

   ```
   Web-ContextPath: /my-fragment-collection-contributor
   ```

2. Add the `-dsannotations-options` instruction and set it to use the `inherit` option. This specifies to use DS annotations found in the class hierarchy of the component class:

   ```
   -dsannotations-options: inherit
   ```

Next, you'll dive into providing thumbnail images and language keys/translations.

578.5 Providing Thumbnail Images

You can also provide thumbnail images for reference for your fragments:

1. Under `resources/META-INF/resources` create a folder named `thumbnails`.
2. Copy thumbnail images into the folder with the format `[fragment-name].png` for each fragment.

Note: All fragments added through a Contributed Fragment Collection will be available globally to all Sites.

578.6 Providing Language Keys

Providing language keys in your Fragment gives you the option for translating the text you display. Here's how to do it:

1. You must define your language keys in the Fragment's collection folder. Create the `[COLLECTION]/src/main/resources/content/Language.properties` file.
2. Add your language keys. For example,

   ```
   applied-style=Applied Style
   this-is-the-style-that-will-be-applied=This is the style that will be applied.
   dark=Dark
   light=Light
   ```

You can learn more about providing translations in the Localizing Your Application article.

578.7 Deploy the Contributed Fragment Collection

Now that you have created the necessary pieces of the module, you can build it and deploy it to Liferay DXP. After it's deployed, the Fragments will be available for use. This can also be done by using the Fragments Toolkit. Contributed Fragments cannot be edited with Liferay, and can only be updated by editing the fragments in your module and the building and redeploying them.

CHAPTER 579

INCLUDING DEFAULT RESOURCES IN FRAGMENTS

When creating Page Fragments, you can upload resources (e.g., images, documents, etc.) to your Fragment Collection to make them always available from the Collection, rather than relying on resources uploaded in other areas of your Site (e.g., Documents and Media). For more information on how to include resources in your Fragment Collection from the Page Fragments interface, see Creating Page Fragments.

Once you've uploaded your resource to a Fragment Collection, you can specify it in your Fragment:

1. Navigate to the Fragment editing page by clicking your Fragment's *Actions* → *Edit* button.
2. Specify the image by using this syntax: `[resources:IMAGE_NAME]`. For example, you could include an image `building.png` within an HTML image tag like this:

   ```
   <img src="[resources:building.png]">
   ```

   You can view a full example snippet below:

   ```
   <div class="fragment_38314">
       <lfr-editable id="img" type="image">
           <img src="[resources:building.png]">
       </lfr-editable>
   </div>
   ```

3. Add any additional HTML, CSS, or JavaScript to your Fragment and then click *Publish*.

```
**Note:** You can also reference your Fragment Collection's resources
in your CSS code too. It follows the same syntax as its HTML.
```

Great! You've successfully referenced a default resource from your Fragment Collection!

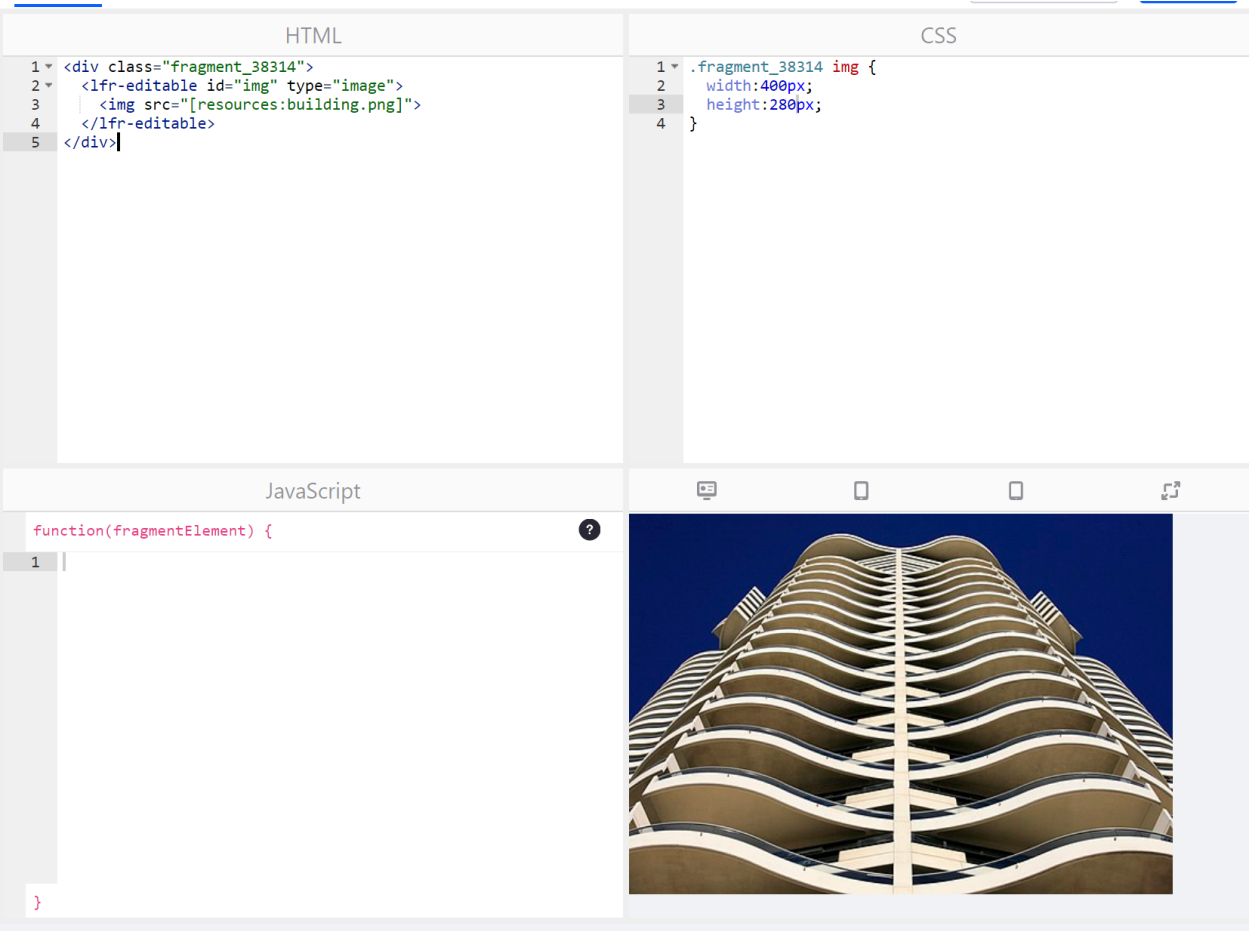

Figure 579.1: Any Fragment from the Fragment Collection has access to the uploaded resources.

CHAPTER 580

SUPPORTING CUSTOM CONTENT TYPES IN CONTENT AND DISPLAY PAGES

Content Pages and Display Page Templates can display several types of content out-of-the-box:

- Web Content Article
- Document
- Blogs Entry

You can publish these content types in highly customizable ways using Page Fragments. You can use these page types to map fields of certain content (e.g., Web Content) to fields defined in a Page Fragment. Then you can publish the content on a page using the Page Fragment as a template. To see an example of how Display Page Templates work, see the Display Page Template Example. For more info on creating Content Pages, see the Building Content Pages article.

If you want to extend the Content Page or Display Page Template framework to support other content types, you must use the Info framework.

You must complete the following steps:

1. Provide basic information about your custom content type.
2. Provide your content type's fields so they're configurable in the Page Editor.
3. Provide friendly URLs for your page type.
4. Handle the information that the user is requesting.

As an example, you'll step through how to provide this information to the Content Page and Display Page Template frameworks.

Note: This section assumes you're customizing Display Page Templates' available content types. The same process outlined in these articles also applies to Content Pages, although it's not explicitly stated.

Continue on to begin!

CHAPTER 581

MAPPING A CONTENT TYPE TO A PAGE

You must allow the mapping of your custom content type to the page type. To do this, implement the `InfoDisplayContributor` interface. Follow the steps below to complete this for the custom User content type.

1. Inside your custom model project, create a new Java package and add a class named `UserInfoDisplayContributor`.

2. Implement the `InfoDisplayContributor` interface and pass the `User` model as the type parameter. Then add the `@Component` annotation:

   ```
   @Component(immediate = true, service = InfoDisplayContributor.class)
   public class UserInfoDisplayContributor
       implements InfoDisplayContributor<User> {
   }
   ```

   The `@Component` annotation registers the class as an info display contributor in the OSGi service registry. Set the `service` property to the interface you're implementing.

3. Implement the methods. For the example User content type, three methods are crucial to mapping its model to the Display Page Template framework:

   ```
   @Override
   public String getClassName() {
       return User.class.getName();
   }

   @Override
   public String getInfoURLSeparator() {
       return "/user/";
   }

   @Override
   public String getLabel(Locale locale) {
       return "Users";
   }
   ```

   - The class name is used to link the Display Page Template to the User model.
   - The URL separator is used to generate friendly URLs for the Display Page Template.

- The label is the display name for the new content type.

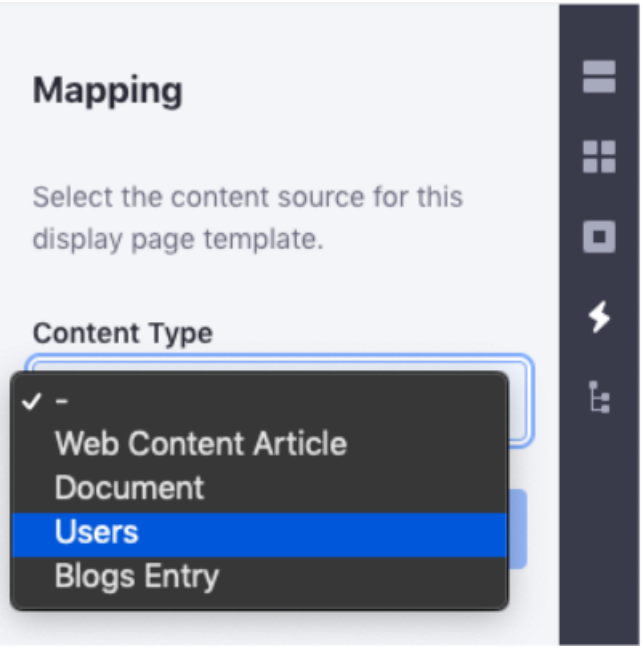

Figure 581.1: After creating the `*InfoDisplayContributor` class, you can create Display Page Templates and map them to your custom model.

Great! You've mapped your custom content type to the Display Page Template framework. Next, you'll provide your content type's fields.

CHAPTER 582

SPECIFYING THE FIELDS OF A CUSTOM CONTENT TYPE

Now that your custom content type is selectable for a Display Page Template, you must specify the fields you want the user to map to the fragment's editable fields in the Display Page Template. To do this, implement the `InfoDisplayContributorField` interface.

Follow the steps below to create a user name field for the User content type:

1. Inside your custom model project, add a class named `UserNameInfoDisplayContributorField`.
2. Implement the `InfoDisplayContributorField` interface and pass the `User` model as the type parameter. Then add the `@Component` annotation:

```
@Component(
    property = "model.class.name=com.liferay.portal.kernel.model.User",
    service = InfoDisplayContributorField.class
)
public class UserNameInfoDisplayContributorField
    implements InfoDisplayContributorField<User> {
}
```

   The `@Component` annotation declares the class as an info display contributor field in the OSGi service registry. You also set the property `model.class.name`, which associates the content type you wish to configure with this service.

3. Implement the methods.

```
@Override
public String getKey() {
    return "userName";
}

@Override
public String getLabel(Locale locale) {
    return "User Name";
}

@Override
public InfoDisplayContributorFieldType getType() {
```

```
    return InfoDisplayContributorFieldType.TEXT;
}

@Override
public String getValue(User user, Locale locale) {
    return user.getFullName();
}
```

The above methods

- set the content type field key to `username`.
- set the field label to `User Name`.
- set the field type to text.
- set the field value to the user's full name.

4. Now you must override the `getInfoDisplayFields` method in your `*DisplayContributor` class, so the mappable fields are displayed. Open the `UserInfoDisplayContributor` class and add the following method:

```
@Override
public Set<InfoDisplayField> getInfoDisplayFields(
        long classTypeId, Locale locale)
    throws PortalException {

    Set<InfoDisplayField> infoDisplayFields = new LinkedHashSet<>();

    List<InfoDisplayContributorField> infoDisplayContributorFields =
        _infoDisplayContributorFieldTracker.getInfoDisplayContributorFields(
        getClassName());

    for (InfoDisplayContributorField infoDisplayContributorField :
    infoDisplayContributorFields) {

        InfoDisplayContributorFieldType infoDisplayContributorFieldType =
            infoDisplayContributorField.getType();

        infoDisplayFields.add(
            new InfoDisplayField(
            infoDisplayContributorField.getKey(),
            infoDisplayContributorField.getLabel(locale),
            infoDisplayContributorFieldType.getValue()));
    }

    return infoDisplayFields;
}

@Reference
private InfoDisplayContributorFieldTracker _infoDisplayContributorFieldTracker;
```

This method references your new `*InfoDisplayContributorField` class to specify your content type's fields.

Awesome! You've mapped the content type's fields to the editable fields of the provided fragments. Next, you'll provide the friendly URLs for the Display Page Template.

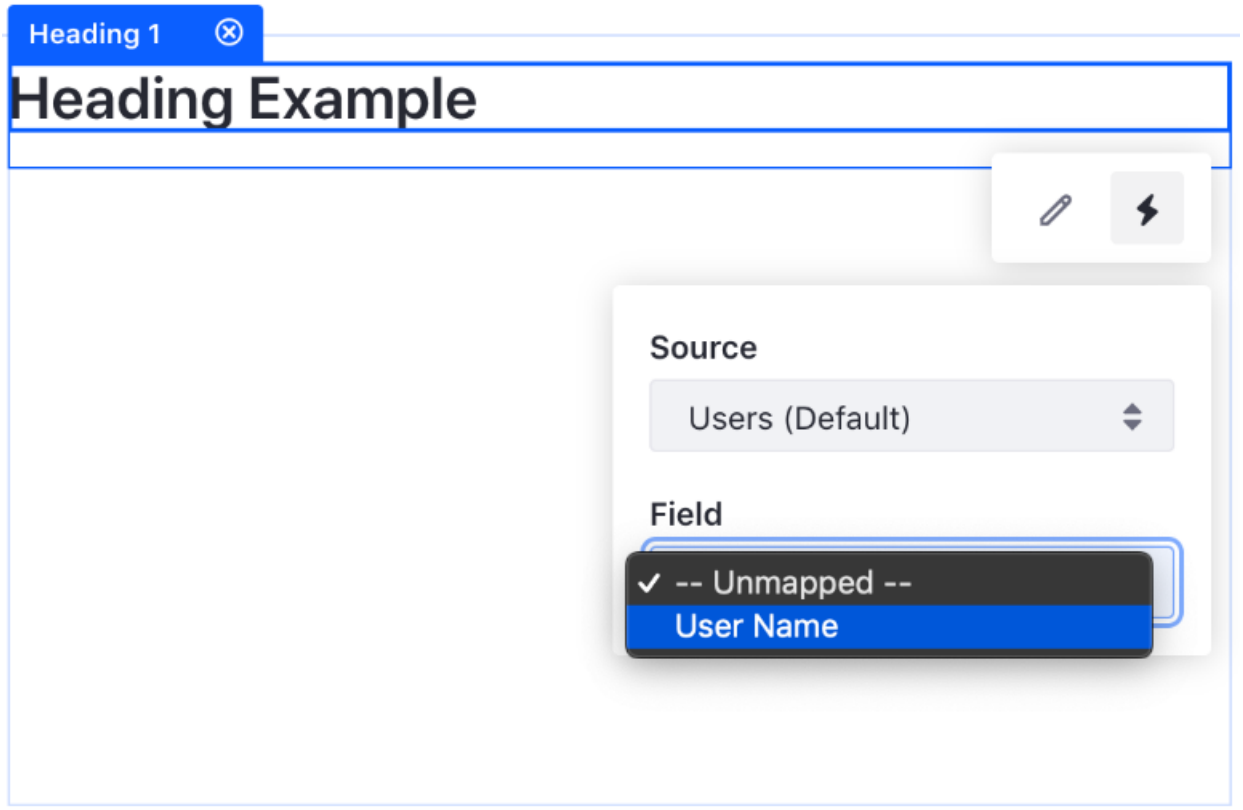

Figure 582.1: After creating the *`InfoDisplayContributorField` class, your custom content type has a new field to map.

CHAPTER 583

PROVIDING FRIENDLY URLS FOR A CUSTOM CONTENT TYPE

To provide a friendly URL for your custom content type, you must implement a friendly URL resolver to retrieve the desired information. For the User content type example, you'll want to retrieve profiles by a user's screen name.

To do this, follow the steps below.

1. Open the *`InfoDisplayContributor` class you created previously (covered here). Implement the `getInfoDisplayObjectProvider` method. For the User example, it looks like this:

```
@Override
public InfoDisplayObjectProvider<User> getInfoDisplayObjectProvider(
        long groupId, String urlTitle)
    throws PortalException {

    Group group = _groupLocalService.getGroup(groupId);

    User user = _userLocalService.getUserByScreenName(
        group.getCompanyId(), urlTitle);

    return new InfoDisplayObjectProvider<User>() {

        @Override
        public long getClassNameId() {
            return _classNameLocalService.getClassNameId(getClassName());
        }

        @Override
        public long getClassPK() {
            return user.getUserId();
        }

        @Override
        public long getClassTypeId() {
            return 0;
        }

        @Override
        public String getDescription(Locale locale) {
            return StringPool.BLANK;
        }
```

```
        @Override
        public User getDisplayObject() {
            return user;
        }

        @Override
        public long getGroupId() {
            return groupId;
        }

        @Override
        public String getKeywords(Locale locale) {
            return StringPool.BLANK;
        }

        @Override
        public String getTitle(Locale locale) {
            return user.getFullName();
        }

        @Override
        public String getURLTitle(Locale locale) {
            return user.getScreenName();
        }

    };
}
```

This method returns a new `InfoDisplayObjectProvider` for the User type. This is the specific model instance used to retrieve the mapped values and check for the display page. This is required by the friendly URL resolver. Now you'll implement the friendly URL resolver for the User content type.

2. Inside your custom model project, add a class named `UserDisplayPageFriendlyURLResolver`.
3. Implement the `FriendlyURLResolver` interface. Then add the `@Component` annotation:

```
@Component(service = FriendlyURLResolver.class)
public class UserDisplayPageFriendlyURLResolver implements FriendlyURLResolver {

}
```

The `@Component` annotation declares the class as a friendly URL resolver in the OSGi service registry.

4. Implement the methods. The User type's implementation looks like this:

```
@Override
public String getActualURL(
    long companyId, long groupId, boolean privateLayout,
      String mainPath, String friendlyURL, Map<String, String[]> params,
    Map<String, Object> requestContext)
throws PortalException {

    HttpServletRequest request = (HttpServletRequest)requestContext.get(
        "request");

    InfoDisplayContributor infoDisplayContributor =
        _getInfoDisplayContributor();

    List<String> paths = StringUtil.split(friendlyURL, CharPool.SLASH);
```

```
    InfoDisplayObjectProvider infoDisplayObjectProvider =
        infoDisplayContributor.getInfoDisplayObjectProvider(
            groupId, paths.get(1));

    request.setAttribute(
        AssetDisplayPageWebKeys.INFO_DISPLAY_OBJECT_PROVIDER,
            infoDisplayObjectProvider);

    request.setAttribute(
        InfoDisplayWebKeys.INFO_DISPLAY_CONTRIBUTOR,
            infoDisplayContributor);

    Layout layout = _getInfoDisplayObjectProviderLayout(
        infoDisplayObjectProvider);

    return _portal.getLayoutActualURL(layout, mainPath);
}

@Override
public LayoutFriendlyURLComposite getLayoutFriendlyURLComposite(
    long companyId, long groupId, boolean privateLayout,
    String friendlyURL, Map<String, String[]> params,
    Map<String, Object> requestContext)
throws PortalException {

    Layout layout = _getInfoDisplayObjectProviderLayout(
        _getInfoDisplayObjectProvider(groupId, friendlyURL));

    return new LayoutFriendlyURLComposite(layout, friendlyURL);
}

@Override
public String getURLSeparator() {
    return "/user/";
}

private InfoDisplayContributor _getInfoDisplayContributor()
    throws PortalException {

    InfoDisplayContributor infoDisplayContributor =
        _infoDisplayContributorTracker.
            getInfoDisplayContributorByURLSeparator(getURLSeparator());

    if (infoDisplayContributor == null) {
        throw new PortalException(
            "Info display contributor is not available for " +
            getURLSeparator());
    }

    return infoDisplayContributor;
}

private InfoDisplayObjectProvider _getInfoDisplayObjectProvider(
        long groupId, String friendlyURL)
    throws PortalException {

    List<String> paths = StringUtil.split(friendlyURL, CharPool.SLASH);

    InfoDisplayContributor infoDisplayContributor =
        _infoDisplayContributorTracker.getInfoDisplayContributor(
            User.class.getName());

    return infoDisplayContributor.getInfoDisplayObjectProvider(
        groupId, paths.get(1));
}

private Layout _getInfoDisplayObjectProviderLayout(
    InfoDisplayObjectProvider infoDisplayObjectProvider) {
```

```
    LayoutPageTemplateEntry layoutPageTemplateEntry =
        _layoutPageTemplateEntryService.fetchDefaultLayoutPageTemplateEntry(
            infoDisplayObjectProvider.getGroupId(),
            infoDisplayObjectProvider.getClassNameId(),
            infoDisplayObjectProvider.getClassTypeId());

    if (layoutPageTemplateEntry != null) {
        return _layoutLocalService.fetchLayout(
            layoutPageTemplateEntry.getPlid());
    }

    return null;
}

@Reference
private InfoDisplayContributorTracker _infoDisplayContributorTracker;

@Reference
private LayoutLocalService _layoutLocalService;

@Reference
private LayoutPageTemplateEntryService _layoutPageTemplateEntryService;

@Reference
private Portal _portal;
```

Notice you're finding the `InfoDisplayObjectProvider` corresponding to the current user. This serves as the representation/descriptor of the mapped object.

```
**Note:** This `FriendlyURLResolver` implementation uses the default
display page template for the User model.
```

```
When this implementation is deployed, you'll receive an empty page when
calling the URL `[host]/web/guest/user/[screenName]`. You must return the
values from the users that are mapped to the display page. You'll do this
next.
```

5. Implement the `getInfoDisplayFieldsValue` method in the previously created `*InfoDisplayContributor` class.

```
@Override
public Map<String, Object> getInfoDisplayFieldsValues(
        User user, Locale locale)
    throws PortalException {

    Map<String, Object> infoDisplayFieldsValues = new HashMap<>();

    List<InfoDisplayContributorField> infoDisplayContributorFields =
        _infoDisplayContributorFieldTracker.getInfoDisplayContributorFields(
            getClassName());

    for (InfoDisplayContributorField infoDisplayContributorField :
        infoDisplayContributorFields) {

        Object fieldValue = infoDisplayContributorField.getValue(
            user, locale);

        infoDisplayFieldsValues.putIfAbsent(
```

```
                infoDisplayContributorField.getKey(), fieldValue);
        }

        return infoDisplayFieldsValues;
    }
```

Great! Now you have a friendly URL that maps to your display page template's custom content type.

CHAPTER 584

INTEGRATING DISPLAY PAGES INTO CONTENT CREATION

After you add support for Display Pages in your custom entities, you can integrate display page configuration into your entity's creation form.

584.1 Display Page Taglib Example

To provide the Display Page selector for the User type after you created fields for it,

1. Open your JSP used for displaying the editing interface (e.g., `.../META-INF/resources/.../edit_entry.jsp`).
2. Add this code in the appropriate place in the layout to add the Display Page selector:

```
<liferay-asset:select-asset-display-page
    classNameId="<%= PortalUtil.getClassNameId(User.class) %>"
    classPK="<%= userId %>"
    groupId="<%= scopeGroupId %>"
/>
```

Now, a selector is available to define a default Display Page when editing or creating a User.

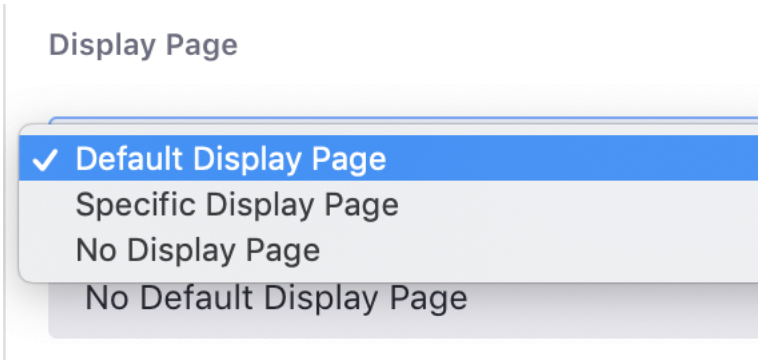

Figure 584.1: You need to add the Display Page selection to your content type's create/edit page to define the Display Page for each instance of that asset.

Awesome! Your custom content type is now available for Content Pages and/or Display Page Templates.

CHAPTER 585

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.

The framework uses 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.

585.1 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. To use Screen Navigation for your application, first you'll create the necessary Java classes and then add the front-end support through JSPs.

Your `ScreenNavigationCategory` class must be a a component that implements the `ScreenNavigationCategory` interface with these methods:

`getCategoryKey()`: returns the category's primary key.

`getLabel(Locale locale)`: returns the label of the key.

`getScreenNavigationKey()`: returns the navigation key that the category belongs in, as defined in your application.

Your `ScreenNavigationEntry` class, similarly must be a component which implements `ScreenNavigationEntry` with the following methods:

`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.

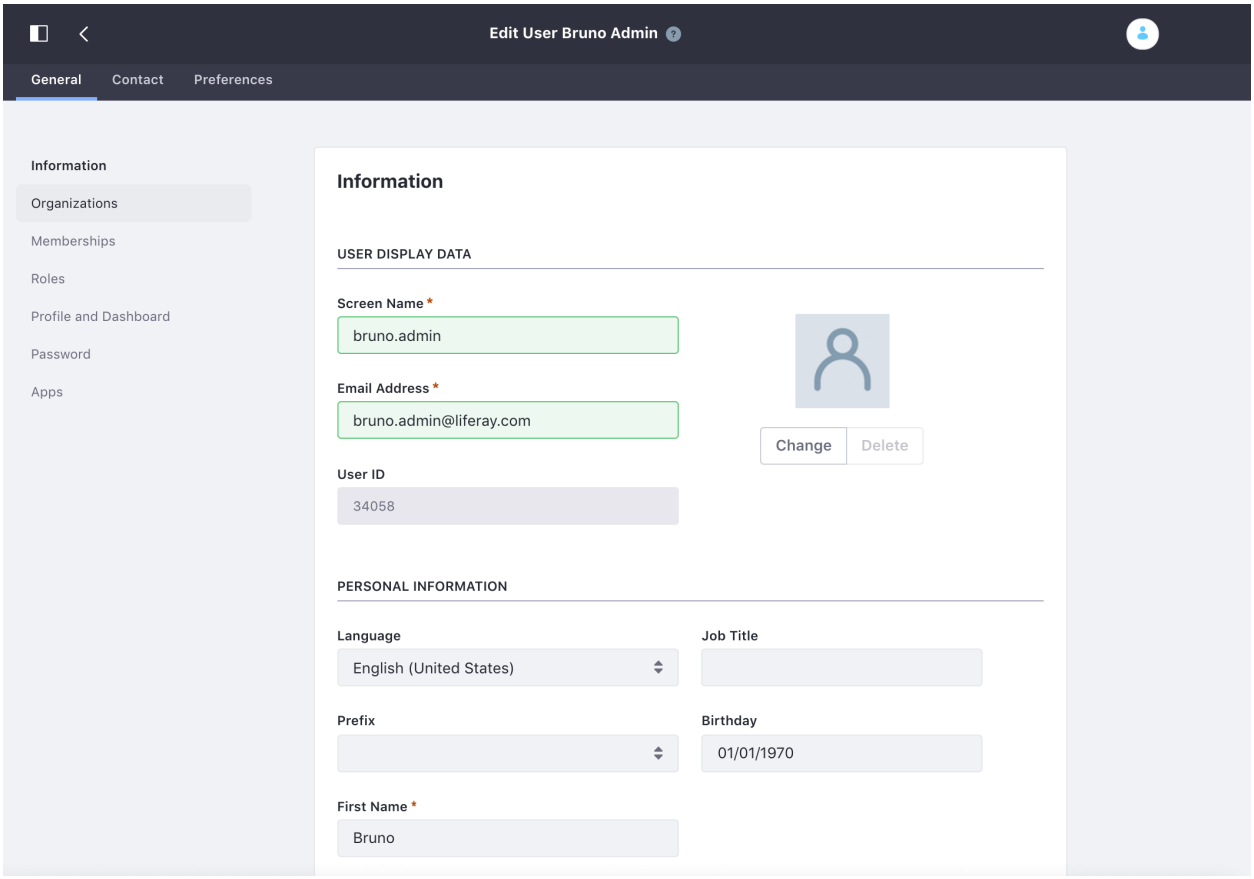

Figure 585.1: The User Management application has three Screen Navigation Categories: General, Contact, and Preference; and each of those have a number of Screen Navigation Entries

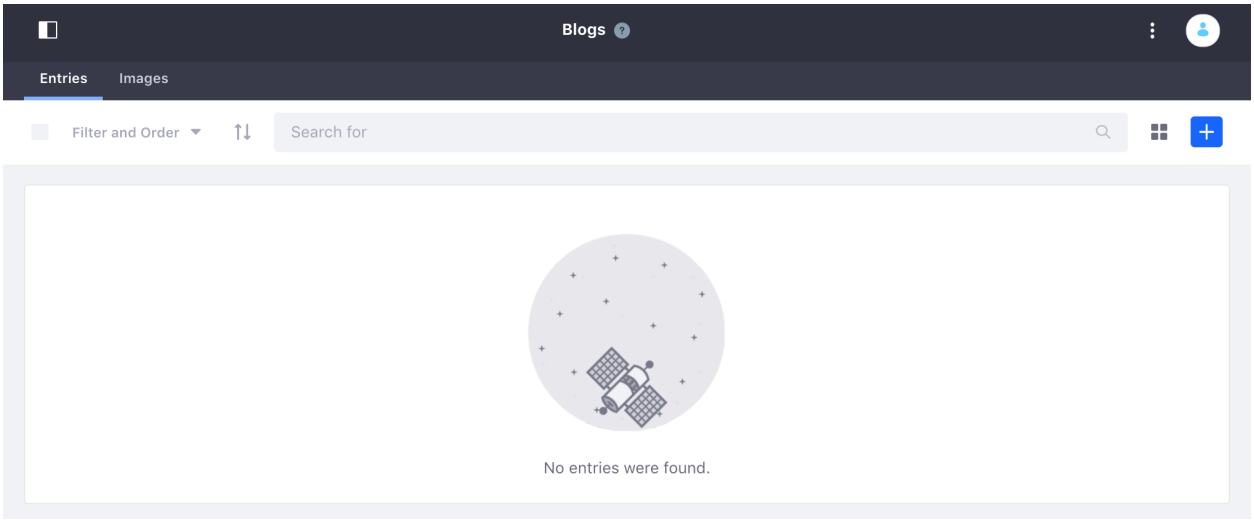

Figure 585.2: Many application only use Screen Navigation Categories for their functionality, and don't have Screen Navigation Entries. For Blogs, Entries and Images are Categories with no Entries.

`render(HttpServletRequest request, HttpServletResponse response)`: renders the entry. The `render` method is also where any logic for creating the configuration goes.

585.2 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 and added to a Liferay application, integrating them seamlessly with the original application.

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 addition to these parameters, you must build the page that users will use for configuration, and connect it to your business logic through the render command.

CHAPTER 586

USING SCREEN NAVIGATION FOR YOUR APPLICATION

To use the Screen Navigation framework with your application you must create a Screen Navigation Category and Entry, and then create the JSP to provide the layout for the Screen Navigation Entry.

586.1 Adding Screens to Your Application's Back-end

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. To demonstrate this, follow the instructions for integrating Screen Navigation for a sample application.

First, create the Navigation Category:

1. In your existing application, create a component named `SampleScreenNavigationCategory` that implements the `ScreenNavigationCategory` interface.
2. In the `@Component` declaration, set your priority property which determines what order items appear in in the side navigation:

```
property = "screen.navigation.category.order:Integer=10",
```

3. Add the following methods to the class body:

```
@Override
public String getCategoryKey() {
    return SampleScreenNavigationConstants.
        CATEGORY_KEY_SAMPLE_CONFIGURATION;
}

@Override
public String getLabel(Locale locale) {
    return LanguageUtil.get(locale, "general");
}

@Override
public String getScreenNavigationKey() {
```

```
    return SampleScreenNavigationConstants.
        SAMPLE_KEY_METHOD;
}
```

When you're finished, your `ScreenNavigationCategory` class looks like this:

```
@Component(
property = "screen.navigation.category.order:Integer=10",
service = ScreenNavigationCategory.class
)
public class SampleScreenNavigationCategory
  implements ScreenNavigationCategory {

  @Override
  public String getCategoryKey() {
    return SampleScreenNavigationConstants.
        CATEGORY_KEY_SAMPLE_CONFIGURATION;
  }

  @Override
  public String getLabel(Locale locale) {
    return LanguageUtil.get(locale, "general");
  }

  @Override
  public String getScreenNavigationKey() {
    return SampleScreenNavigationConstants.
        SAMPLE_KEY_METHOD;
  }
}
```

Next, add a Navigation Entry:

1. Create a component named `SampleScreenNavigationEntry` which implements `ScreenNavigationEntry`.
2. Create the the `@Reference` variables you need for the render logic:

```
@Reference
private ConfigurationProvider _configurationProvider;

@Reference
private JSPRenderer _jspRenderer;

@Reference
private Portal _portal;

@Reference(
    target = "(osgi.web.symbolicname=com.liferay.commerce.payment.method.sample)"
)
private ServletContext _servletContext;
```

3. Implement the following methods in your component:

```
@Override
public String getCategoryKey() {
    return SampleScreenNavigationConstants.
        CATEGORY_KEY_SAMPLE_CONFIGURATION;
}

@Override
public String getEntryKey() {
    return SampleScreenNavigationConstants.
```

```
        ENTRY_KEY_SAMPLE_CONFIGURATION;
}

@Override
public String getLabel(Locale locale) {
    return LanguageUtil.get(
        locale,
        SampleScreenNavigationConstants.
            CATEGORY_KEY_SAMPLE_CONFIGURATION);
}

@Override
public String getScreenNavigationKey() {
    return SpaceShipScreenNavigationConstants.
        SAMPLE_KEY_METHOD;
}

@Override
public boolean isVisible(
    User user, SamplePermissions spaceShipPermissions) {

    if (samplePermissions.criteriaMethod())
    {

        return true;
    }

    return false;
}

@Override
public void render(HttpServletRequest request, HttpServletResponse response)
throws IOException {

    _jspRenderer.renderJSP(request, response, "/my-category/view-category.jsp");
}
```

Here is what the `SampleScreenNavigationEntry` class looks like:

```
@Component(
property = "screen.navigation.entry.order:Integer=20",
service = ScreenNavigationEntry.class
)
public class
    SampleScreenNavigationEntry
    implements ScreenNavigationEntry<SampleApplication> {

    public static final String
        ENTRY_KEY_SAMPLE_CONFIGURATION =
            "sample-configuration";

    @Override
    public String getCategoryKey() {
        return SpaceShipScreenNavigationConstants.
            CATEGORY_KEY_SAMPLE_CONFIGURATION;
    }

    @Override
    public String getEntryKey() {
        return ENTRY_KEY_SAMPLE_CONFIGURATION;
    }

    @Override
    public String getLabel(Locale locale) {
        return LanguageUtil.get(
            locale,
```

```
        SpaceShipScreenNavigationConstants.
            CATEGORY_KEY_SAMPLE_CONFIGURATION);
}

@Override
public String getScreenNavigationKey() {
    return SpaceShipScreenNavigationConstants.
        SAMPLE_KEY_METHOD;
}

@Override
public boolean isVisible(
    User user, SamplePermissions spaceShipPermissions) {

    if (samplePermissions.criteriaMethod())
    {

        return true;
    }

        return false;
    }

@Override
public void render(HttpServletRequest request, HttpServletResponse response)
throws IOException {

    _jspRenderer.renderJSP(request, response, "/my-category/view-category.jsp");
}

@Reference
private JSPRenderer _jspRenderer;

@Reference(
    target = "(osgi.web.symbolicname=com.liferay.commerce.payment.method.sample);

}
```

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.

586.2 Adding Screens to Your Application's Front-end

The `render` method that you created in your last step references `/my-category/view-category.jsp`. Create the JSP now:

1. In `/src/resources/META-INF/resources` create the `my-category` folder.
2. Inside of that folder, create `view-category.jsp`.
3. Inside the JSP add the `liferay-frontend:screen-navigation` taglib with the required parameters:

```
<liferay-frontend:screen-navigation key="<%= AssetCategoriesConstants.CATEGORY_KEY_GENERAL %>"
    modelBean="<%= category %>"
    portletURL="<%= portletURL %>"
/>
```

After that tag, add the rest of the content of the JSP file to handle user interactions and communication with the back-end for configuration.

CHAPTER 587

EXTENDING 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 scriptlet 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 588

DEVELOPING A FRAGMENT RENDERER

When creating Fragments through Liferay DXP's provided UI, you're given three front-end languages to leverage: CSS, HTML, and JavaScript. Although you can harness a lot of power with these languages alone, they do not provide an easy way to retrieve and process information from the database or third party systems. A common solution for this issue is creating a full-fledged portlet to complete common back-end necessities, but this is sometimes overkill for what you need.

For a lightweight alternative, you can develop a *Fragment Renderer* to use Liferay's provided Java APIs for back-end tasks related to your Fragment. To do this, you must implement the `FragmentRenderer` interface.

Optionally, you can

- Leverage the `FragmentRendererContext`.
- Use JSPs for your Fragment's display.
- Choose when to display the component.
- Translate the Collection language key.

You'll explore each step next.

588.1 Implementing the FragmentRenderer Interface

The `FragmentRenderer` interface requires the implementation of two methods:

`getCollectionKey`: returns the unique key for the component's Collection. Define this key in several components to group them under a collapsible panel in the Page Editor.

`getLabel`: provides the Fragment name.

The remaining methods are optional, but can be useful in many scenarios:

`getImagePreviewURL`: returns the URL for previewing the Fragment's image.

`getKey`: returns the Fragment's key.

`getType`: returns the Fragment's type. Type values include `FragmentConstants.TYPE_COMPONENT` and `FragmentConstants.TYPE_SECTION`.

`isSelectable`: defines whether page authors can select the Fragment Renderer. You'll learn more about this in the Choosing When to Display a Component section.

`render` (highly recommended): defines how to render the Fragment Renderer (e.g., JSP or FreeMarker). You can leverage the `FragmentRendererContext` in this method to facilitate the rendering process.

Next, you'll learn about leveraging the `FragmentRendererContext`.

588.2 Leveraging the FragmentRendererContext

The `render` method receives a read-only instance of the interface `FragmentRendererContext`. This provides information about the context in which the Fragment is being rendered. The fields of information that are accessible through it include

Fragment Entry Link: The specific instance of the Fragment being rendered. This information can be used to identify the specific site or page to which the Fragment was added, when it was added, the user who added it, etc.

Locale: The current locale to be used for any multi-locale text.

Mode: There are three available modes:

- **VIEW**: The component is being rendered within a page being viewed (not edited).
- **ASSET_DISPLAY_PAGE**: The component is being edited on a Display Page.
- **EDIT**: The component is being edited on a Content Page.

There are other fields which should only be necessary for advanced use cases:

Preview Class PK: If the Fragment supports displaying content, this field supports previewing an *In progress* version of the content before it's ready to publish. In this case, the `render` method returns the content's primary key.

Preview Type: Represents the preview type you want to show. The accepted values include

- **TYPE_LATEST_APPROVED**: The latest approved version of the content.
- **TYPE_LATEST**: The latest version of the content.

Field Values: Fragments can have editable elements through `<lfr-editable>` tags; this also applies to those created with `FragmentRenderer`. The `getFieldValuesOptional()` method retrieves the field values the user may have introduced in them. This only applies in the context of a Display Page with the values of the mapped structure.

Segment Experience IDs: A list of identifiers for experiences that have been configured for the current page.

588.3 Rendering JSPs

Usually you'll want to avoid writing HTML in your Java code. Fortunately, you can use the `render` method to use any templating mechanism of your choice. JSP integration is provided out-of-the box.

For example, rendering a JSP for your Fragment Renderer would look like this:

```
@Override
public void render(
    FragmentRendererContext fragmentRendererContext,
    HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse) throws IOException {
```

```
    httpServletRequest.setAttribute(
        "fragmentRendererContext", fragmentRendererContext);

    _jspRenderer.renderJSP(
        httpServletRequest, httpServletResponse, "/my-component.jsp");
}

@Reference
private JSPRenderer _jspRenderer;

@Reference(
    target = "(osgi.web.symbolicname=com.liferay.fragment.renderer.docs)",
    unbind = "-"
)
private ServletContext _servletContext;
```

This sets the `FragmentRendererContext` in the HTTP servlet request, which is then used to render the included JSP file (e.g., `my-component.jsp`).

To leverage JSPs, you must specify the servlet context for the JSP files. Since your Fragment Renderer is an OSGi module, your `bnd.bnd` file must define a web context path:

```
Bundle-SymbolicName: com.liferay.fragment.renderer.docs
Web-ContextPath: /my-fragment-renderer
```

Then you must reference the Servlet context using the symbolic name of your module, as was shown above:

```
@Reference(
    target = "(osgi.web.symbolicname=com.liferay.fragment.renderer.docs)",
    unbind = "-"
)
private ServletContext _servletContext;
```

Note: To use the JSP Renderer, your module must set the `com.liferay.frontend.taglib` dependency in its build file.

Next, you'll learn about controlling when your Fragment Renderer is displayed.

588.4 Choosing When to Display a Component

Sometimes offering Fragment components only makes sense in specific cases. You can implement the `isSelectable(...)` method to specify under which conditions the Fragment Renderer is available to page authors.

For example, if you wanted to make your Fragment Renderer only available in Display Pages, you could implement the `isSelectable` method like this:

```
@Override
public boolean isSelectable(HttpServletRequest httpServletRequest) {
    Layout layout = (Layout)httpServletRequest.getAttribute(WebKeys.LAYOUT);

    if (Objects.equals(
        layout.getType(), LayoutConstants.TYPE_ASSET_DISPLAY)) {

        return true;
    }
```

```
    return false;
}
```

This determines the Fragment Renderer's page type and returns `true` when the page type is a Display Page or `false` if it's not.

588.5 Translating the Collection Language Key

When setting your Fragment Renderer's collection name via the `getCollectionKey` method, you should specify it as a language key and then define it in a resource bundle.

For example, a `getCollectionKey` method could look like this:

```
@Override
public String getCollectionKey() {
    return "sample-components";
}
```

To specify `sample-components` in a resource bundle, create the `src/main/resources/content/Language.properties` file within the Fragment Renderer module and define it using the language key `fragment.collection.label.{collecti key}`. For example,

```
fragment.collection.label.sample-components=Sample Components
```

To learn more about resource bundles, see the Localization section.

Next, you'll step through creating a Fragment Renderer.

CHAPTER 589

CREATING A FRAGMENT RENDERER

Creating a Fragment Renderer lets you call Liferay's provided Java APIs for back-end tasks related to your Fragment. In this article, you'll create a sample Fragment Renderer that displays values stored in the current Liferay DXP instance's database.

1. Create a default module project in your development environment.
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 `FragmentRenderer` (e.g., `ShowContextFragmentRenderer`).
3. Configure your new class to implement the `FragmentRenderer` interface:

```
public class ShowContextFragmentRenderer implements FragmentRenderer {
}
```

4. Insert the following `@Component` annotation above the class declaration:

```
@Component(service = FragmentRenderer.class)
```

   This sets the OSGi service type to `FragmentRenderer`.

5. Implement the two required `FragmentRenderer` methods:

```
@Override
public String getCollectionKey() {
    return "sample-components";
}

@Override
public String getLabel(Locale locale) {
    return "Show Context Component";
}
```

   The `getCollectionKey()` method returns a language key, which you'll define later. The name displayed for this Fragment is defined as *Show Context Component*.

6. Implement the `render` method:

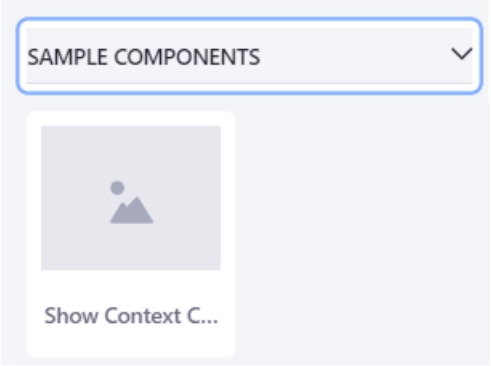

Figure 589.1: The new Fragment Renderer appears in its defined component collection.

```
@Override
public void render(
    FragmentRendererContext fragmentRendererContext,
    HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse) throws IOException {

        PrintWriter printWriter = httpServletResponse.getWriter();

        printWriter.write("<h3>Context</h3>");
        printWriter.write("<ul>");

        FragmentEntryLink fragmentEntryLink =
        fragmentRendererContext.getFragmentEntryLink();

        printWriter.write("<li>Added by: " + fragmentEntryLink.getUserName());
        printWriter.write("<li>Added in: " + fragmentEntryLink.getCreateDate());

        printWriter.write("<li>Locale: " + fragmentRendererContext.getLocale());
        printWriter.write("<li>Mode: " + fragmentRendererContext.getMode());
        printWriter.write("<li>PreviewClassPK: " + fragmentRendererContext.getPreviewClassPK());
        printWriter.write("<li>PreviewType: " + fragmentRendererContext.getPreviewType());
        printWriter.write("<li>Segment experiences: " + StringUtil.merge(fragmentRendererContext.getSegmentsExperienceIds(), ", "));
        printWriter.write("</ul>");
}
```

This method leverages the `FragmentRendererContext`, which provides the Fragment's context information stored in the database. This information is displayed in the Fragment Renderer when it's placed on a page.

7. Define the language key `sample-components` that you used in the `getCollectionKey()` method. To do this, create the `src/main/resources/content/Language.properties` file and add the following language key:

```
fragment.collection.label.sample-components=Sample Components
```

8. Provide the appropriate dependencies to compile your Fragment Renderer project. For example, the following dependencies are defined for the Show Context Component Fragment Renderer sample (Gradle build) deployed to Liferay Portal 7.2 GA1:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "4.13.0"
    compileOnly group: "com.liferay", name: "com.liferay.fragment.api", version: "2.7.2"
    compileOnly group: "com.liferay", name: "com.liferay.fragment.service", version: "2.0.10"
```

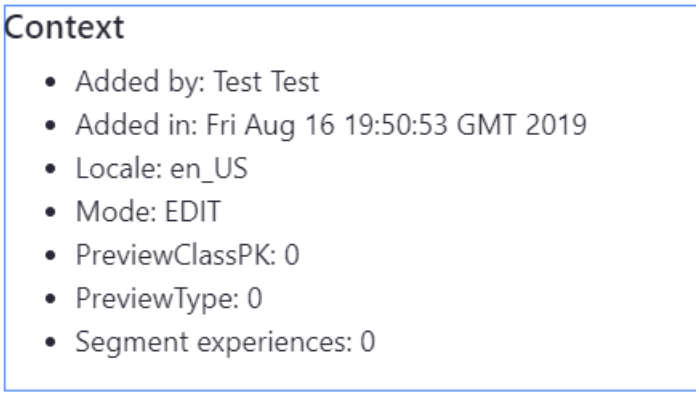

Figure 589.2: When adding the new Fragment Renderer to a page, the context information is displayed.

```
    compileOnly group: "com.liferay", name: "com.liferay.frontend.taglib", version: "4.0.15"
    compileOnly group: "com.liferay", name: "com.liferay.petra.string", 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: "jstl", name: "jstl", version: "1.2"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

To stay in sync with the appropriate versions of your project's dependencies, consider using the Target Platform framework.

That's it! You can compile the sample *Show Context Component* Fragment Renderer and deploy it! It'll be available to add for a Fragment-enabled page under the *Sample Components* collection.

CHAPTER 590

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 three 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.

GraphQL: All the power of doing multiple queries in a unique request following GraphQL specification.

Plain Web/REST Services: This is the old way to build and consume web services in Liferay DXP, but is still supported.

You can also create your own Headless REST and GraphQL APIs through the **REST builder**.

CHAPTER 591

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.

591.1 OpenAPI

OpenAPI (originally called Swagger) is a Linux Foundation project specification that defines machine-readable files that describe REST APIs and how to consume them.

OpenAPI has become a widely adopted standard for defining REST APIs and is supported by major players in the API ecosystem such as Google, Amazon, and Microsoft. As a spec, it is language-agnostic, and many libraries implement it or provide code generation to help validate, consume, or produce APIs.

Liferay DXP leverages existing knowledge of OpenAPI to define, create and consume REST APIs.

591.2 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.

CHAPTER 592

GET STARTED: FIND 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.

592.1 Related Topics

Get Started: Invoke a Service

CHAPTER 593

How To 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.

593.1 Related Topics

Get Started: Discover the API

API Formats and Content Negotiation

CHAPTER 594

MAKING AUTHENTICATED REQUESTS

To make an authenticated request, you must authenticate as a specific User.

There are three 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.

Cookie/Session authentication: From inside the portal you can make direct requests to the APIs by sending the session token.

First, you'll learn how send requests with basic authentication.

594.1 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. 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
}
```


594.2 OAuth 2.0 Authorization

7.0 supports authorization via OAuth 2.0, which is a token-based authorization 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.

594.3 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.

594.4 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.

594.5 Using Cookie Authentication or Making Requests from the UI

You can call the REST APIs using the existing session from outside Liferay DXP by passing the session identifier (the cookie reference) and the Liferay Auth Token (a Cross-Site Request Forgery—CSRF—token).

To do a request from outside Liferay DXP you must provide the `Cookie` identifier in the header. In CURL, pass the `-H` parameter:

```
-H 'Cookie: JSESSIONID=27D7C95648D7CDBE3347601FC4543F5D'
```

You must also provide the CSRF token by passing it in the `p_auth` query parameter, or by adding the URL to the whitelist of CSRF allowed URLs or disabling CSRF checks altogether with the `auth.verifier.auth.verifier.PortalSessionAuthVerifier.check.csrf.token` property (application level).

Here's a sample CURL request with the cookie and CSRF token:

```
curl -H 'Cookie: JSESSIONID=27D7C95648D7CDBE3347601FC4543F5D' http://localhost:8080/o/headless-delivery/v1.0/sites/{siteId}/blog-postings/?p_auth=O4dCU1Mj
```

To do a cookie request from inside Liferay DXP, from JavaScript code or a Java method, the session identifier is not needed and you must only provide the CRSF token or add the API to the whitelist of CSRF allowed URLs.

594.6 Making Unauthenticated Requests

Unauthenticated requests are disabled by default in Liferay DXP's headless REST APIs. You can, however, enable them manually by defining an exception in the Service Access Policy to allow unauthenticated requests.

1. Go to Control Panel → Configuration → Service Access Policy.
2. Add a new Service Access Policy.
3. Enable both *Enabled* and *Default* options.
4. Use `com.liferay.headless.delivery.internal.resource.v1_0.OpenAPIResourceImpl` for the Service Class and `getOpenAPI` for the Method Name (or the method/class you want to expose).
5. 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.

594.7 Cross-Origin Resource Sharing (CORS)

Modern web browsers block access to content from domains other than the one currently being visited. For example, browsers block fetch/ajax requests from a local JavaScript application (being executed in localhost:4000) that tries to access a Tomcat server (running in localhost:8080).

Cross Origin Resource Sharing allows the configuration of safe resource sharing between sites. A web application using APIs can only request endpoints that have the same origin/domain unless some special CORS headers are set that explicitly allow querying from different domains.

For development purposes, it's common to enable CORS headers to allow scripts to call APIs served by a different server.

Follow these instructions to configure Cross-Origin Resource Sharing (CORS) in Liferay DXP.

594.8 Related Topics

Get Started: Invoke a Service

Working with Collections of Data

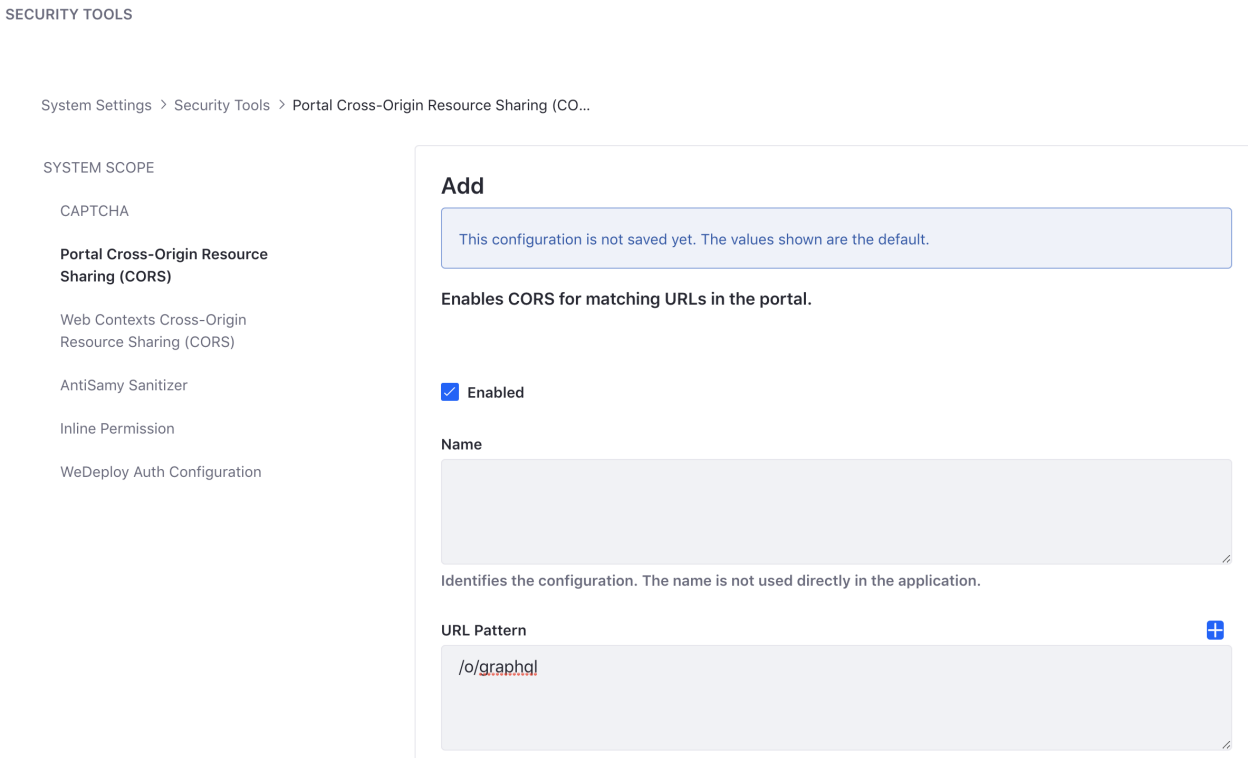

Figure 594.1: Configure Cross-Origin Resource Sharing in Liferay

CHAPTER 595

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.

595.1 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.

CHAPTER 596

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
}
```

596.1 Related Topics

Pagination
    Making Authenticated Requests
    API Formats and Content Negotiation

CHAPTER 597

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`).

597.1 Related Topics

Working with Collections of Data
    Making Authenticated Requests

CHAPTER 598

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'
```

598.1 Related Topics

Getting Collections
    Pagination
    Making Authenticated Requests

CHAPTER 599

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
```

599.1 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"
}
```


599.2 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 adds that translation. PATCH requests return an error (you are expected to update, not create, in a PATCH request).

599.3 Related Topics

Get Started: Discover the API

Get Started: Invoke a Service

CHAPTER 600

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.

600.1 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`

API	Internal Model
StructuredContent	JournalArticle
StructuredContentFolder	JournalFolder

600.2 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.

600.3 Related Topics

API Formats and Content Negotiation

CHAPTER 601

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.

601.1 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.

601.2 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 | `startswith(addressLocality, 'Lond')` |

601.3 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.

601.4 Grouping Operators

Operator | Description | Example | `( )` | Precedence grouping | `(price eq 5) or (addressLocality eq 'London')` |

601.5 String Functions

Function	Description	Example
`contains`	Contains	`contains(title,'edmon')`

601.6 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.

601.7 Operator combinations and OData syntax

Syntax examples and other operator combinations are covered in the OData standard reference.

601.8 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
}
```

601.9 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"
```

601.10 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
}
```


601.11 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
```

601.12 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.

601.13 Related Topics

Making Authenticated Requests
    API Formats and Content Negotiation
    Working with Collections of Data

CHAPTER 602

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
}
```

602.1 Related Topics

Making Authenticated Requests
API Formats and Content Negotiation
Working with Collections of Data

CHAPTER 603

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"
}
```

603.1 Related Topics

Making Authenticated Requests
    API Formats and Content Negotiation
    Working with Collections of Data

CHAPTER 604

How to get siteId

Several APIs (generally all collection APIs) need the `siteId` parameter to to execute requests. The `siteId` is the internal identifier of the Site where that content was created.

604.1 Using siteId or siteKey

In all the APIs available from 7.2 GA2+, the `siteKey` is also accepted as a valid parameter. The `siteKey` is the external name of the Site (for example `Guest`).

In the REST APIs, you can use `siteKey` in all the places that expect a `siteId`; in GraphQL APIs, there are two different parameters: `siteId` and `siteKey`.

Using the `siteKey` is recommended over `siteId` in all situations because it's more recognizable, doesn't expose an internal parameter, and doesn't change in import/export processes.

604.2 Obtain siteId

There are several ways to retrieve the `siteId`:

- Use the Site API to query it by name, friendly URL or in the list of a User's Sites.
- From Liferay DXP's UI in the Site Administration menu (not recommended).
- From the `Group` table in the database (not recommended).
- From the `ThemeDisplay` object in JavaScript or Java:

```
themeDisplay.getSiteGroupId()
```

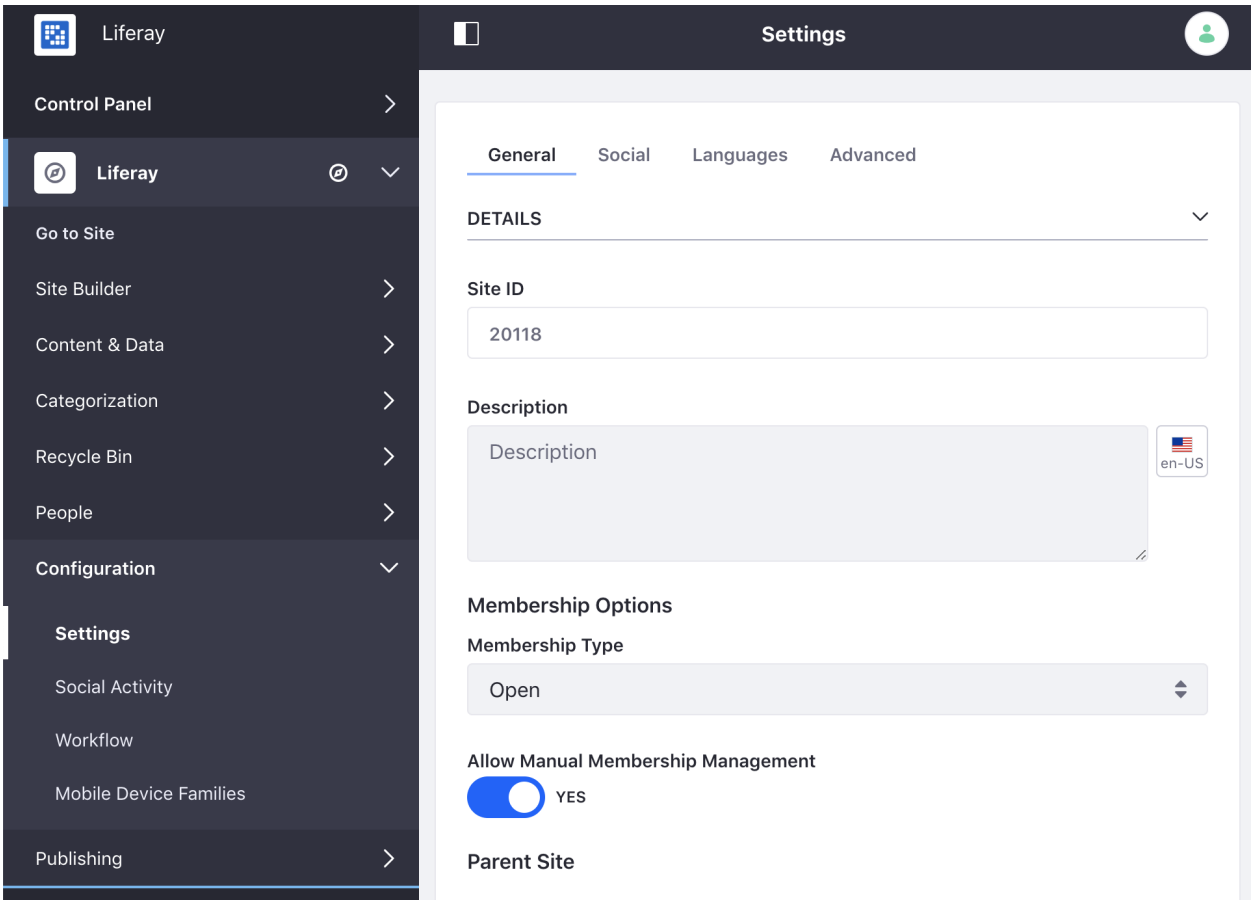

Figure 604.1: GraphQL BlogPostings definition

CHAPTER 605

FILTERABLE PROPERTIES

Some APIs return data that can be filtered and sorted by several properties. This is a non-comprehensive list of the properties that can be used to filter or sort.

605.1 Headless Delivery API

605.2 BlogPosting

Key | Type | Example | taxonomyCategoryIds | list | `taxonomyCategoryIds/any(t:t eq 1)` | keywords | list | `keywords/any(k:contains(k,'substring1'))`| customFields | complex | `customFields/Name eq 'Article1'` | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z` | creatorId | integer | `creatorId eq 1` | headline | string | `contains(headline,'substring1')` |

605.3 BlogPostingImage

Key | Type | Example | encodingFormat | id | `encodingFormat eq 1` | sizeInBytes | integer | `sizeInBytes eq 1` | fileExtension | string | `contains(fileExtension,'substring1')` | title | string | `contains(title,'substring1')` |

605.4 Comment

Key | Type | Example | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z`| creatorId | integer | `creatorId eq 1` |

605.5 ContentStructure

Key | Type | Example | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z` | name | string | `contains(name,'substring1')` |

605.6 Document

Key | Type | Example | taxonomyCategoryIds | list | `taxonomyCategoryIds/any(t:t eq 1)` | keywords | list | `keywords/any(k:contains(k,'substring1'))`| customFields | complex | `customFields/Name eq 'Article1'` | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z` | encodingFormat | id | `encodingFormat eq 1` | creatorId | integer | `creatorId eq 1` | sizeInBytes | integer | `sizeInBytes eq 1` | fileExtension | string | `contains(fileExtension,'substring1')` | title | string | `contains(title,'substring1')` |

605.7 DocumentFolder

Key | Type | Example | customFields | complex | `customFields/Name eq 'Article1'` | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z` | creatorId | integer | `creatorId eq 1` | name | string | `contains(name,'substring1')` |

605.8 KnowledgeBaseArticle

Key | Type | Example | taxonomyCategoryIds | list | `taxonomyCategoryIds/any(t:t eq 1)` | keywords | list | `keywords/any(k:contains(k,'substring1'))`| customFields | complex | `customFields/Name eq 'Article1'` | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z` | title | string | `contains(title,'substring1')` |

605.9 MessageBoardMessage

Key | Type | Example | showAsAnswer | boolean | `showAsAnswer eq true` | showAsQuestion | boolean | `showAsQuestion eq true` | taxonomyCategoryIds | list | `taxonomyCategoryIds/any(t:t`

eq 1) | keywords | list | keywords/any(k:contains(k,'substring1'))| customFields | complex | customFields/Name eq 'Article1' | dateCreated | date | dateCreated lt 2018-02-13T12:33:12Z | dateModified | date | dateModified lt 2018-02-13T12:33:12Z | creatorId | integer | creatorId eq 1 | messageBoardSectionId | integer | messageBoardSectionId eq 1 | headline | string | contains(headline,'substring1') |

605.10 MessageBoardSection

Key | Type | Example | customFields | complex | customFields/Name eq 'Article1' | dateCreated | date | dateCreated lt 2018-02-13T12:33:12Z | dateModified | date | dateModified lt 2018-02-13T12:33:12Z | creatorId | integer | creatorId eq 1 | title | string | contains(title,'substring1') |

605.11 StructuredContent

Key | Type | Example | taxonomyCategoryIds | list | taxonomyCategoryIds/any(t:t eq 1) | keywords | list | keywords/any(k:contains(k,'substring1'))| contentFields | complex | contentFields/Name eq 'Article1' | customFields | complex | customFields/Name eq 'Article1' | dateCreated | date | dateCreated lt 2018-02-13T12:33:12Z | dateModified | date | dateModified lt 2018-02-13T12:33:12Z | datePublished | date | datePublished lt 2018-02-13T12:33:12Z | contentStructureId | integer | contentStructureId eq 1 | title | string | contains(title,'substring1') |

605.12 StructuredContentFolder

Key | Type | Example | customFields | complex | customFields/Name eq 'Article1' | dateCreated | date | dateCreated lt 2018-02-13T12:33:12Z | dateModified | date | dateModified lt 2018-02-13T12:33:12Z | creatorId | integer | creatorId eq 1 | name | string | contains(name,'substring1') |

605.13 WikiNode

Key | Type | Example | dateCreated | date | dateCreated lt 2018-02-13T12:33:12Z | dateModified | date | dateModified lt 2018-02-13T12:33:12Z | creatorId | integer | creatorId eq 1 | name | string | contains(name,'substring1') |

605.14 WikiPage

Key | Type | Example | customFields | complex | `customFields/Name eq 'Article1'` | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z` | headline | string | `contains(headline,'substring1')` |

605.15 Headless Admin User API

605.16 Organization

Key | Type | Example | keywords | list | `keywords/any(k:contains(k,'substring1'))`| dateModified | date | `dateModified lt 2018-02-13T12:33:12Z` | parentOrganizationId | id | `parentOrganizationId eq 1` | name | string | `contains(name,'category')` |

605.17 User

Key | Type | Example | keywords | list | `keywords/any(k:contains(k,'substring1'))`| dateModified | date | `dateModified lt 2018-02-13T12:33:12Z` | id | id | `id eq 1` | organizationIds | id | `organizationIds eq 1` | roleIds | id | `roleIds eq 1` | userGroupIds | id | `userGroupIds eq 1` | alternateName | string | `contains(alternateName,'substring1')` | emailAddress | string | `contains(emailAddress,'substring1')` | familyName | string | `contains(familyName,'substring1')` | givenName | string | `contains(givenName,'substring1')` | jobTitle | string | `contains(jobTitle,'substring1')` |

605.18 Headless Admin Taxonomy API

605.19 Category

Key | Type | Example | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z`| name | string | `contains(name,'category')` |

605.20 Keyword

Key | Type | Example | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z`| name | string | `contains(name,'category')` |

605.21 Vocabulary

Key | Type | Example | dateCreated | date | `dateCreated lt 2018-02-13T12:33:12Z` | dateModified | date | `dateModified lt 2018-02-13T12:33:12Z`| name | string | `contains(name,'category')` |

CHAPTER 606

USING REST APIS

Liferay DXP's headless REST APIs can be used with any REST client you prefer. The only usual requirements are setting up the `Authentication` header (either OAuth, Cookie, Basic...) and the `Content-Type` header if you are creating content.

Our recommendation for JavaScript applications is to use `fetch` directly, like this:

```
fetch(`http://localhost:8080/o/headless-delivery/v1.0/sites/${SITE_ID}/structured-contents/'`,
    {
        method: 'GET',
        headers: {
            'Authorization': `Basic ${BASIC_AUTH}`
        }
    }
);
```

Or for a `POST` request:

```
fetch(`http://localhost:8080/o/headless-delivery/v1.0/sites/${SITE_ID}/structured-contents/`,
    {
        method: 'POST',
        headers: {
            'Authorization': `Basic ${BASIC_AUTH}`,
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(
            {
                "title": "New appointment",
                "contentStructureId": STRUCTURE_ID,
                "contentFields": [
                    {
                        "name": "User",
                        "value": {
                            "data": USER,
                        }
                    },
                ]
            }
        )
    }
)
```

Here are two examples of JavaScript applications using the Headless REST APIs:

- Alexa skill using Headless REST APIs with node-fetch.

- Example API from scratch using REST Builder.

CHAPTER 607

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 makes this easy by providing a template.

Create a project and choose the *rest* template.

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 three things:

1. The endpoint for the service
2. The service name as it appears in the OAuth 2.0 configuration
3. (Optional) Properties you may want to set for further 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.

607.1 Authenticating to JAX-RS Web Services

Authentication during development can be done through Basic Authentication or portal sessions, but you don't want to leave that enabled for production. For production, you want OAuth 2.0. Here's how to configure JAX-RS authentication.

607.2 During Development: Basic Auth

When you deploy a JAX-RS application, an Auth Verifier filter is registered for it. You can set its properties in your `@Component` annotation by prefixing the properties with `auth.verifier`. For example, to disable guest access to the service, configure it like this:

```
@Component(
        property = {
            JaxrsWhiteboardConstants.JAX_RS_APPLICATION_BASE + "=/greetings",
            JaxrsWhiteboardConstants.JAX_RS_NAME + "=Greetings.Rest",
            "auth.verifier.guest.allowed=false"
        },
        service = Application.class)
```

Basic Auth is great during development, but credentials passed on the URL appear in server logs, so when you're done developing, you should disable Basic Auth and use OAuth2 instead. To disable Basic Auth, create and deploy a configuration file called `com.liferay.portal.security.auth.verifier.internal.tracker.AuthVerifierFilterTracker.config` that contains this property:

```
default.registration.property=["filter.init.auth.verifier.OAuth2RESTAuthVerifier.urls.includes=*","filter.init.auth.verifier.PortalSessionAuthVerifie
```

This disables Basic Auth for all JAX-RS applications, but keeps Portal Session and OAuth2 enabled.

607.3 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*.

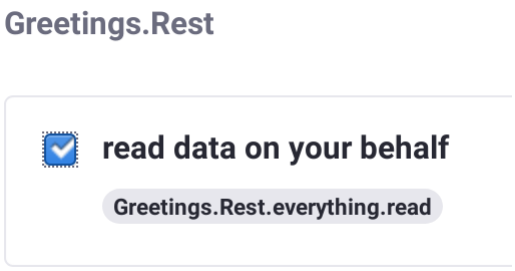

Figure 607.1: Enable the scope to grant access to the service.

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.

OAuth2 Scopes

Without any special Liferay OAuth2 annotations or properties, a standard OSGi JAX-RS application is inspected by the Liferay OAuth2 runtime, and scopes are derived by default based on the HTTP verbs supported by the application.

When developers want more control, they can use the property `oauth2.scopechecker.type=annotations` and the annotation `com.liferay.oauth2.provider.scope.RequiresScope` exported from the `Liferay OAuth2 Provider Scope API` bundle to annotate endpoint resource methods or whole classes like this:

```
@RequiresScope("scopeName")
```

Once deployed, this becomes a scope in the OAuth 2.0 configuration. You can disable scope checking (not recommended) by setting the scope checker to a non-existent type:

```
oauth2.scope.checker.type=none
```

Requiring OAuth2

You can specify OAuth2 authorization as required for your JAX-RS application by using this property:

```
osgi.jaxrs.extension.select=(osgi.jaxrs.name=Liferay.OAuth2)
```

607.4 JAX-RS and Service Access Policies

When authenticating via Basic Auth, the Service Access Policy `SYSTEM_USER_PASSWORD` is enforced. When authenticating via OAuth 2.0, the `AUTHORIZED_OAUTH2_SAP` policy is enforced. Configure them appropriately for your environment, as by default, they allow invoking all remote services. To disable Service Access Policy enforcement for JAX-RS endpoints (not recommended), set this property:

```
liferay.access.control.disable=true
```

With this configured, guests can call these endpoints without administrators having to define a default Service Access Policy.

607.5 Public JAX-RS Services

To create a public endpoint for development purposes, all you must do is set two properties:

```
@Component(
    property={
        "auth.verifier.guest.allowed=true",
        "liferay.access.control.disable=true"
    },
    service = Application.class
)
```

Don't keep this configuration for production. For public services, it's best to leave the security in place and whitelist the particular endpoints you're making public. See Service Access Policies for further information.

607.6 Using JAX-RS with CORS

If you foresee that JavaScript in a browser might access your JAX-RS web service from a different domain, you might want to use the CORS annotation. You can use the `@CORS` annotation to define CORS policies on your deployed JAX-RS applications. Note that the annotations can be overridden by an administrator. It only takes three steps:

1. Add the Portal Remote CORS API dependency to your module:

```
compileOnly project(":apps:portal-remote:portal-remote-cors-api")
```

2. Activate the CORS annotation feature in your application properties:

```
@Component(
    property = {
        "osgi.jaxrs.application.base=/my-application",
        "osgi.jaxrs.name=My.Application.Name",
        "liferay.cors.annotation=true"
    },
    service = Application.class
    )
    public class MyApplication extends Application {
    ...
    }
```

3. Use the `@CORS` annotation throughout your application globally or by method.

Globally:

```
@Component(
    property = {
        "osgi.jaxrs.application.base=/my-application",
        "osgi.jaxrs.name=My.Application.Name",
        "liferay.cors.annotation=true"
    },
    service = Application.class
)
@CORS(allowMethods="GET")
public class MyApplication extends Application {
...
}
```

By method:

```
@CORS
    @GET
    @Path("/users")
    public List<User> getUserList() throws Exception {
        return _users;
    }
```

You can use the annotation to provide a configuration for any of the CORS headers. Here are some examples:

| Header | Annotation Example |

Access-Control-Allow-Credentials|`@CORS(allowCredentials = false)`| Access-Control-Allow-Headers|`@CORS(allowHeaders = "X-PINGOTHER")`| Access-Control-Allow-Methods|`@CORS(allowMethods = "OPTIONS,POST")`| Access-Control-Allow-Origin|`@CORS(allowOrigin = "http://www.liferay.com")`|

If for some reason you want to disable the `@CORS` annotations in your application, you can do it globally by disabling it in your `@Component` annotation:

```
@Component(
    property = {
        "osgi.jaxrs.application.base=/no-cors-application",
        "osgi.jaxrs.name=NoCors.Application.Name",
        "liferay.cors.annotation=false"
    },
    service = Application.class
)
```

Great! Now you know how to create, deploy, and invoke JAX-RS web services on Liferay DXP's platform!

607.7 Related Topics

REST Builder

CHAPTER 608

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

608.1 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.

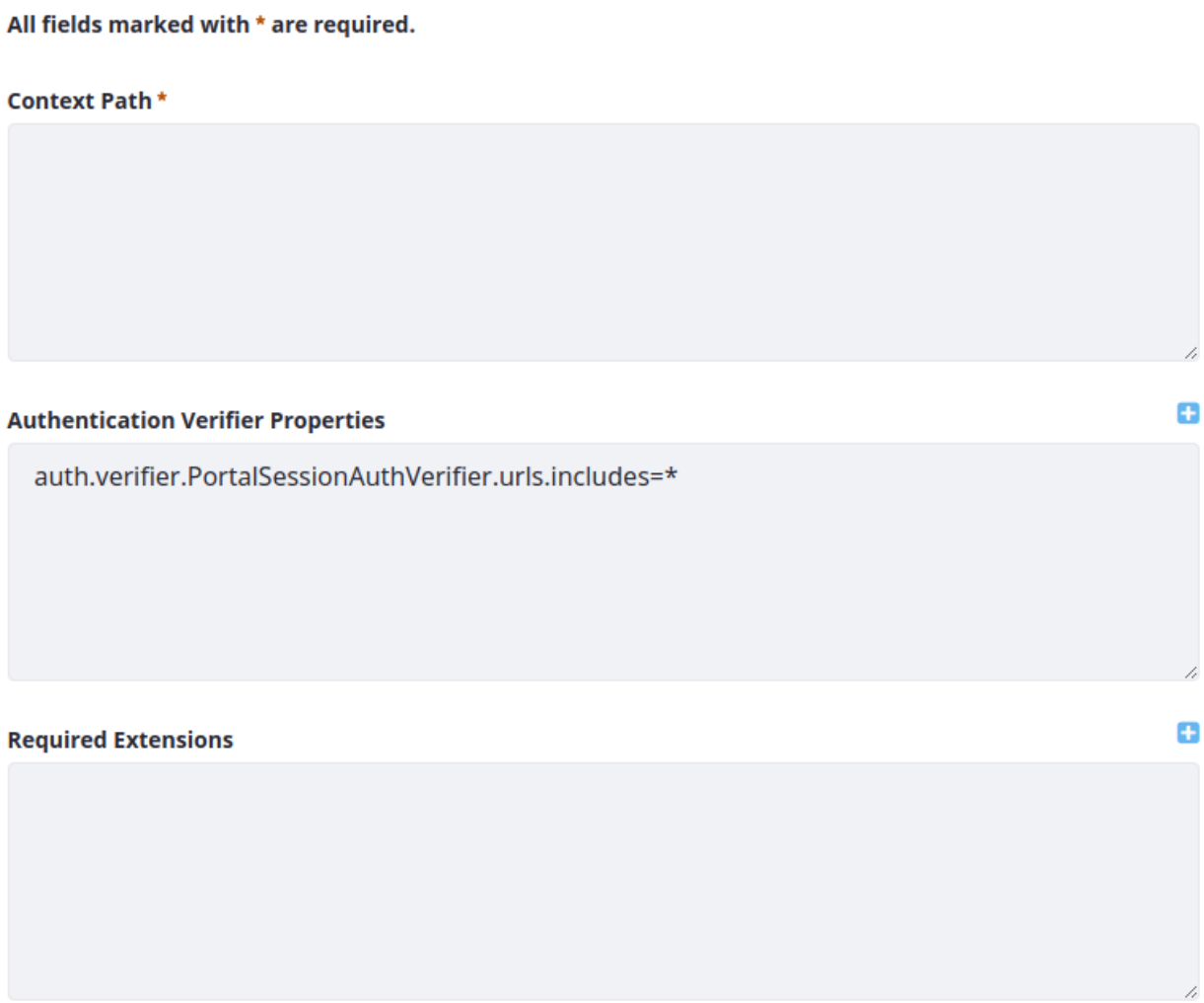

Figure 608.1: Fill out this form to create a CXF endpoint.

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.

608.2 Configuring Endpoints and Extenders Programmatically

To configure endpoints or extenders programmatically, you must use Liferay's configurator extender. The configurator extender provides a way for OSGi modules to deploy default configuration values. Modules that use the configurator extender must provide a `ConfigurationPath` header that points to the configuration files' location inside the module. For example, the following configuration sets the `ConfigurationPath` to `src/main/resources/configuration`:

```
Bundle-Name: Liferay Export Import Service JAX-WS
Bundle-SymbolicName: com.liferay.exportimport.service.jaxws
Bundle-Version: 1.0.0
Liferay-Configuration-Path: /configuration
Include-Resource: configuration=src/main/resources/configuration
Liferay-Releng-Module-Group-Description:
Liferay-Releng-Module-Group-Title: Data Management
```

Note 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
```

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.internal.configuration.SoapExtenderConfiguration-stagingjaxws.properties`:

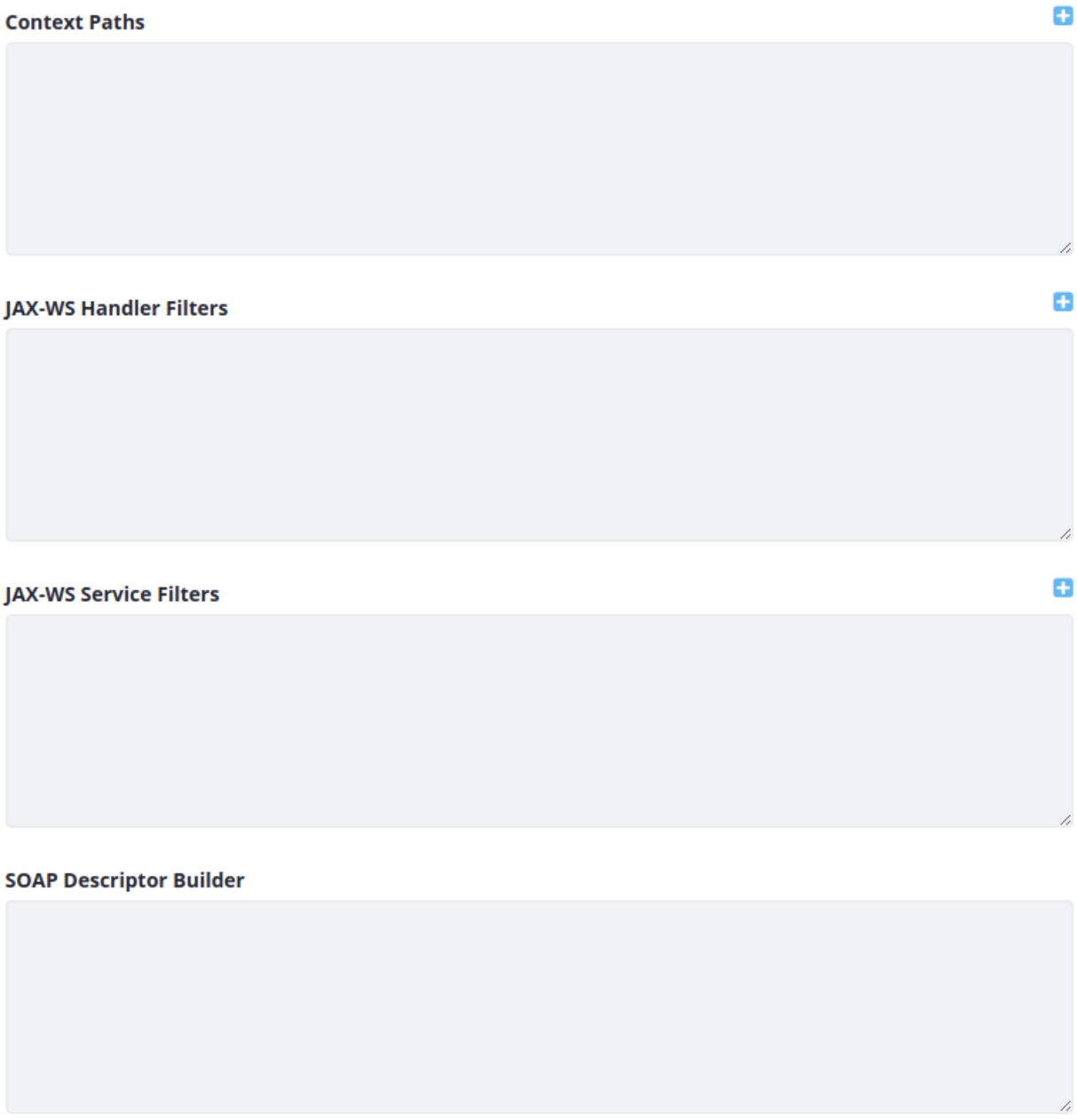

Figure 608.2: Fill out this form to create a SOAP extender.

```
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.internal.configuration.SoapExtenderConfiguration",
    localization = "content/Language", name = "soap.extender.internal.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.

608.3 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 609

GRAPHQL APIS

Liferay DXP exposes a full GraphQL API implementation, and lets your apps fetch several entities from the same request.

Here you'll learn how to navigate and consume Liferay DXP's GraphQL APIs. Since GraphQL APIs are discoverable, you'll start with that.

CHAPTER 610

GET STARTED: DISCOVER THE API

To begin consuming the GraphQL APIs, you must first know where they are, what operations you can invoke, and how to invoke them.

Because Liferay DXP's GraphQL APIs leverage the official specification, you don't need a service catalog. You only need to know the URL from which to discover the rest of the API.

Liferay DXP's GraphQL APIs are available here:

```
http://[host]:[port]/o/graphql
```

For example, if you're running Liferay DXP locally on port `8080`, the URL for discovering the GraphQL API is

```
http://localhost:8080/o/graphql
```

To inspect the GraphQL endpoint, use a GraphQL client, such as Altair (a Chrome extension) or GraphiQL.

You don't have to be authenticated to inspect the live documentation, but you must to be able to make requests. There are several ways of authenticating in GraphQL APIs (explained here) but the simplest way to test APIs locally is to use Basic Authentication, setting an `Authorization` header in Altair (first icon on the left). Remember that Basic Auth is a BASE64 transformation of `user:password`. This means it's insecure, and should never be used in production.

Most tools that introspect the GraphQL schema can autocomplete your query or fill all the fields in for you.

For a list of tools such as client generators, validators, and parsers supporting GraphQL, see Awesome GraphQL. Leveraging GraphQL provides standards support, extensive automatic documentation, and industry-wide conventions.

610.1 Unique endpoint and versioning

In contrast with the REST APIs, where endpoints are deployed by suite (headless-delivery, headless-admin-user...), GraphQL APIs are deployed under the same endpoint (/o/graphql). That way we can easily add relationships between entities to leverage GraphQL's powerful request characteristics.

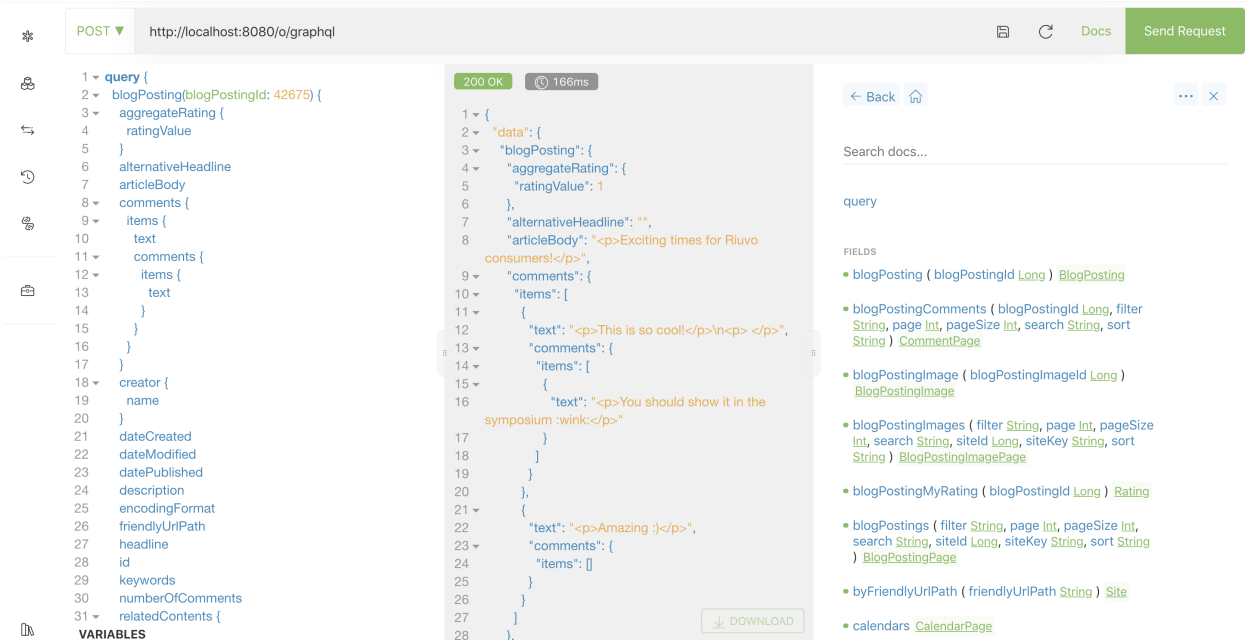

Figure 610.1: GraphQL APIs can be browsed in Altair.

Liferay DXP's GraphQL APIs also expose the latest published version of all entities available. If several versions of the same entity are deployed, only the latest one is exposed under the `/o/graphql` endpoint (REST APIs use different endpoints for different versions). This strategy follows GraphQL standards to avoid breaking versions by marking deprecated fields and always adding properties to an entity.

CHAPTER 611

GET STARTED: INVOKE A SERVICE

Once you know which API you want to call via the GraphQL-introspected documentation, you can send a request using a POST body. For example, suppose you want to retrieve all the blog entries from a Site. If you consult the GraphQL documentation you can find this endpoint:

Figure 611.1: GraphQL exposes a definition for BlogPostings.

If you add the full query with Altair/GraphiQL, you'll see a result like this:

```
query{
  blogPostings(filter: _____, page: _____, pageSize: _____, search: _____, siteId: _____, siteKey: _____, sort: _____){
    items
    page
    pageSize
    totalCount
  }
}
```

The only required parameter is `siteId` or `siteKey` (as of 7.2 FP4), the ID, or the internal name (like *guest*) of the blog posting's 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. For more information, see How to get SiteId.

A regular query would ignore optional parameters and return more elements of the list, identified by the property `items`:

```
query {
  blogPostings(siteKey: "guest") {
    items {
      alternativeHeadline
      articleBody
      creator {
```

```
        name
      }
      dateCreated
      dateModified
      datePublished
      description
      encodingFormat
      friendlyUrlPath
      headline
      id
      keywords
      numberOfComments
      relatedContents {
        title
      }
      siteId
      taxonomyCategoryIds
      viewableBy
    }
    page
    pageSize
    totalCount
  }
}
```

In GraphQL, you must list explicitly every field you want to return in the request. Complex objects, like `items`, `creator`, or `relatedContents` can not be returned fully: you must specify at least one field (or none).

To execute the query, make a `POST` request with an `Authentication` header and the query in a JSON object under the key `query`. Don't forget to escape strings!

The following request gets the Site's blog postings by providing the site key (`guest`):

```
curl -X "POST" "http://localhost:8080/o/graphql" \
     -H 'Content-Type: text/plain; charset=utf-8' \
     -u 'test@liferay.com:test' \
     -d $'{
  "query": "query { blogPostings(siteKey: \\"guest\\") { items { alternativeHeadline articleBody creator { id name } dateCreated dateModified datePul
  }'
```

If you send this request to a Site that contains some Blog entries, the response may look like this:

```
{
  "data": {
    "blogPostings": {
      "items": [
        {
          "alternativeHeadline": "",
          "articleBody": "<p>Content</p>",
          "creator": {
            "id": 20124,
            "name": "Test Test"
          },
          "dateCreated": "2019-10-29T17:48:03Z",
          "dateModified": "2019-10-29T17:48:03Z",
          "datePublished": "2019-10-29T17:47:00Z",
          "description": "",
          "encodingFormat": "text/html",
          "friendlyUrlPath": "title",
          "headline": "Title",
          "id": 37644,
          "keywords": [],
          "numberOfComments": 0,
          "relatedContents": [],
```

```
          "siteId": 20118,
          "taxonomyCategoryIds": null,
          "viewableBy": null
        }
      ],
      "page": 1,
      "pageSize": 20,
      "totalCount": 1
    }
  }
}
```

This response is a JSON object with information about the collection of blogs. The 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 (`37644`) and one for the blog post's creator (`20124`).

`page`: The current page's page number. 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 POST request with the `blogPostingId` to this query:

```
query {
  blogPosting(blogPostingId: 37644) {
    headline
    id
  }
}
```

611.1 GraphQL Clients

The examples above show the GraphQL requests as cURL operations but you can use any GraphQL clients available. We have made heavy use of Apollo, the official React client and the Vue integration without any issues.

CHAPTER 612

MAKING AUTHENTICATED REQUESTS

To make an authenticated request, you must authenticate as a specific User.

There are three 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), but should be used only for development purposes, as it's insecure.

OAuth 2.0: In 7.0, you can use OAuth 2.0 for authorization. See the OAuth 2.0 documentation for more information.

Cookie/Session authentication: From inside the portal you can do direct requests to the APIs by sending the session token.

First, you'll learn how to send requests with basic authentication.

612.1 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@liferay.com` with the password `Liferay`:

```
openssl base64 <<< test@liferay.com:Liferay
```

This returns the encoded value:

```
dGVzdEBsaWZlcmF5LmNvbTpMaWZlcmF5Cg==
```

If you don't have `openssl` installed, try the `base64` command:

```
base64 <<< test@liferay.com:Liferay
```

Warning: Encoding a string as shown here does not encrypt the resulting string. The 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 dGVzdEBsaWZlcmF5LmNvbTpMaWZlcmF5Cg==" http://localhost:8080/o/graphql ...
```

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.

612.2 OAuth 2.0 Authorization

7.0 supports authorization via OAuth 2.0, which is a token-based authorization 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.

612.3 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 Creating an Application. 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 Authorizing Account Access with OAuth 2.

612.4 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:

```
curl -H "Authorization: Bearer d5571ff781dc555415c478872f0755c773fa159" http://localhost:8080/o/graphql
```

The response contains the resources that the authenticated user has permission to access, just like the response from Basic authentication. The request could be prevented depending on the scopes defined. If POST a GraphQL query and there is scope disabling all request except `GET`, you see a 403.

612.5 Using Cookie Authentication or doing a request from the portal

You can call the GraphQL APIs using the existing session from outside the Liferay DXP by passing the session identifier (the cookie reference) and the Liferay Auth Token (a CSRF—Cross-Site Request Forgery—token).

To make an unauthenticated request from outside the Liferay DXP you must provide the `Cookie` identifier in the header:

```
curl -H 'Cookie: JSESSIONID=27D7C95648D7CDBE3347601FC4543F5D'
```

You must also provide the CSRF token by passing it as a query parameter called `p_auth` or by adding the URL to the whitelist of CSRF allowed URLs or disabling CSRF checks altogether with the `auth.verifier.auth.verifier.PortalSessionAuthVerifier.check.csrf.token` property (application level).

Here's a sample cURL request with the cookie and CSRF token:

```
curl -H 'Cookie: JSESSIONID=27D7C95648D7CDBE3347601FC4543F5D' http://localhost:8080/o/graphql?p_auth=O4dCU1Mj
```

To do an unauthenticated request from inside the Liferay DXP, from JavaScript code or a Java method, you don't need the session identifier. You must only provide the CRSF token or add the API to the whitelist of CSRF allowed URLs.

612.6 Making Unauthenticated Requests

Unauthenticated requests are disabled by default in Liferay DXP's GraphQL APIs. As all GraphQL APIs share the same endpoint, you cannot have the same level of granularity with Service Access Policies as in REST APIs. For that reason, we do not recommend disabling the security of the GraphQL APIs.

612.7 Related Topics

Get Started: Invoke a Service

Working with Collections of Data

CHAPTER 613

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 collection pagination.

613.1 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 GraphQL 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.

`items`: The collection elements present on 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.

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 blog postings that exist on the content set with the ID `42345`:

```
query {
  contentSetContentSetElements(contentSetId: 42345, page: 2) {
    items {
      id
      title
      content {
```

```
        ... on BlogPosting {
          headline
        }
      }
    }
    page
    pageSize
    totalCount
  }
}
```

CHAPTER 614

MUTATIONS

The GraphQL spec differentiates between retrieve operations (`query`) and create/update/delete operations (`mutations`).

Figure 614.1: The GraphQL Mutations list for Blog postings shows the possible operations.

To perform a mutation, do a `POST` request as you did with `query` operations, using cURL, a REST client, or a GraphQL Client like Apollo. The only difference is that create/update `mutations` require an Input type, a JSON object to create or update the content.

A create `mutation` to insert a new blog posting looks like this:

```
mutation {
```

```
  createSiteBlogPosting(
    blogPosting: {
      headline: "New GraphQL APIs!"
      articleBody: "WoW! This is cool!"
    }
    siteKey: "guest"
  ) {
    id
    headline
  }
}
```

Auto-complete also works as expected filling the `blogPosting` object. Here's a cURL request to create the same entry:

```
curl 'http://localhost:8080/o/graphql' -H 'Content-Type: application/json' -H 'Accept: application/json' -H 'Authorization: Basic dGVzdEBsaWZlcmF5LmN
data-binary '{"query":"mutation {createSiteBlogPosting(blogPosting: {headline: \"New GraphQL APIs!\" articleBody: \"WoW! This is cool!\"} siteKey: \'
```

CHAPTER 615

FRAGMENTS AND NODE PATTERNS

Liferay DXP's GraphQL APIs also supports GraphQL fragments, reusable sets of fields that are needed in different requests. A special type of fragments are inline fragments, which access the underlying concrete type when querying generic types or interfaces.

You'll use inline fragments to query objects that inherit from a common interface, like the kind of objects returned from a `ContentSet`. `ContentSets` allow defining lists of assets that comply with a set of rules, a segment, or are manually selected. `ContentSets` can return any type of asset. This makes them a perfect fit for inline fragments.

Here's an example of GraphQL querying `ContentSets`:

```
query {
  contentSetContentSetElements(contentSetId: 42345) {
    items {
      id
      title
      content {
        ... on BlogPosting {
          headline
        }
        ... on StructuredContent {
          relatedContents {
            id
            title
          }
        }
      }
    }
    page
    pageSize
    totalCount
  }
}
```

This query returns a set of objects, each of a different type.

615.1 Node pattern

`graphQLNode` is a special query that leverages the power of inline fragments. This query accepts a `dataType` and an ID and returns any kind of entity that has a query of the type `{dataType}` and

receives an `id` as a parameter. Inline fragments can specify the fields you want to return in this special case:

```
query{
  graphQLNode(dataType: _____, id: _____){
    id
    ... on BlogPosting {
      headline
    }
  }
}
```

You can also use `graphQLNode` as a field in entities that contain `contentType` and `id` properties. Those entities have a generated property called `GraphQLNode` that can return any type, queried by using inline fragments. A common use case is returning an asset linked as `relatedContents` (asset links).

CHAPTER 616

Language Negotiation

The same mechanism for requesting content in another language in the headless REST APIs is used in GraphQL.

APIs available in different languages return the options in a block called `availableLanguages`. For example, this block 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/graphql"  -H 'Accept-Language: es-ES'  -u 'test@liferay.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"
    }
```


616.1 Creating Content with Different Languages

By default, when sending a mutation request, the `Accept-Language` header is used as the content's language. There is one exception, however. Some entities require the first request to be in the Site's default language. In such cases, the first request for a different language results in an error.

After creating a new resource, a new request in a different language adds that translation.

CHAPTER 617

FILTER, SORT, AND SEARCH

You can use Liferay DXP's headless GraphQL APIs to search for the content you want. You can also sort and filter content.

617.1 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`. 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.

617.2 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 | `startswith(addressLocality, 'Lond')` |

617.3 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 requires a trailing space.

617.4 Grouping Operators

Operator | Description | Example | `( )` | Precedence grouping | `(price eq 5) or (addressLocality eq 'London')` |

617.5 String Functions

Function	Description	Example
`contains`	Contains	`contains(title,'edmon')`

617.6 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.

617.7 Escaping in Queries

You can escape a single quote in a value by escaping it with a backslash. For example, to filter for a blog posting whose headline is `New Headless APIs`, send this filter string to the filter parameter.

```
filter: \\"headline eq \'Title\'\\"
```

Here's an example of the full request:

```
curl -X "POST" "http://localhost:8080/o/graphql" \
     -H 'Content-Type: text/plain; charset=utf-8' \
     -H 'Cookie: COOKIE_SUPPORT=true; GUEST_LANGUAGE_ID=en_US; JSESSIONID=EFEEC1617529C7C85E8CCCE510B0F6CF' \
     -u 'test@liferay.com:test' \
     -d $'{
  "query": "query { blogPostings(siteKey: \\"guest\\", filter: \\"headline eq \'Title\'\\") { items {headline} page pageSize totalCount } }"
}'
```

And here's a possible response:

```
{
  "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
    }
  ],
  "page": 1,
  "pageSize": 20,
  "totalCount": 1
}
```

617.8 Filtering in Structured Content Fields (ContentField)

To filter for a `ContentField` value (dynamic values created by the end user), you must use the paths that are scoped to an individual `ContentStructure`.

Find the ID of the `ContentStructure` and use it in place of {`contentStructureId`} in this query:

```
"contentStructureStructuredContents"
```

617.9 Search

You can 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 Title:

```
curl -X "POST" "http://localhost:8080/o/graphql" \
     -H 'Content-Type: text/plain; charset=utf-8' \
     -u 'test@liferay.com:test' \
     -d $'{
  "query": "query { blogPostings(siteKey: \\"guest\\", search: \\"Title\\") { items {headline} page pageSize totalCount } }"
}'
```

```
{
  "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
    }
  ],
  "page": 1,
  "pageSize": 20,
  "totalCount": 1
}
```

617.10 Sorting

Collection results can be sorted. Note, however, that not all collections allow sorting. The ones that support it contain the optional parameter `{lb}?sort{rb}` in their GraphQL definition.

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:

```
sort:\"headline:desc,dateCreated:asc\"
```

617.11 Flatten

The `flatten` query parameter returns all resources and disregards folders or other hierarchical classifications. Collection GraphQL specifications define if `flatten` is available. Its 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 returns documents in any subfolders, regardless of how deeply those folders are nested. Setting `flatten` set to `true` and querying for documents in a Site's root folder returns all the documents in the Site.

CHAPTER 618

MULTIPART REQUESTS

Several mutations accept a binary file via a multipart request. For example, the definition for posting a file to a `DocumentFolder` specifies a multipart request, `Upload` type in GraphQL:

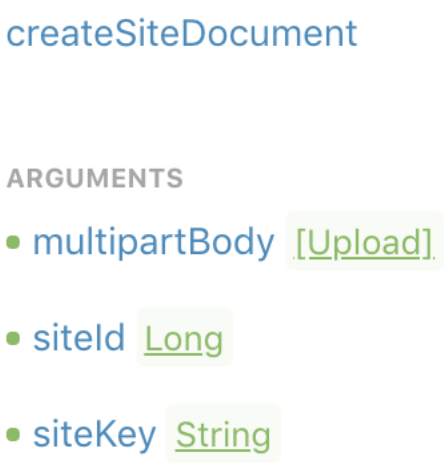

Figure 618.1: Create Document accepts a `multipartBody`.

The GraphQL specification doesn't support natively multipart uploads, but an extension contributed by the community covers that use case.

Liferay's implementation includes that extension and allows uploading files.

Multipart support in GraphQL is disabled by default. To enable it, add the configuration to upload multipart files in the Liferay application's `web.xml` file:

```
<servlet>
        <servlet-name>Module Framework Servlet</servlet-name>
        <servlet-class>
            com.liferay.portal.module.framework.ModuleFrameworkServletAdapter
        </servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
        <multipart-config>
```

```
            <location>/tmp</location>
            <max-file-size>20848820</max-file-size>
            <max-request-size>418018841</max-request-size>
            <file-size-threshold>1048576</file-size-threshold>
        </multipart-config>
</servlet>
```

To test, use the Altair configuration to upload files. Use the selector to upload one or multiple files and define a variable in the query.

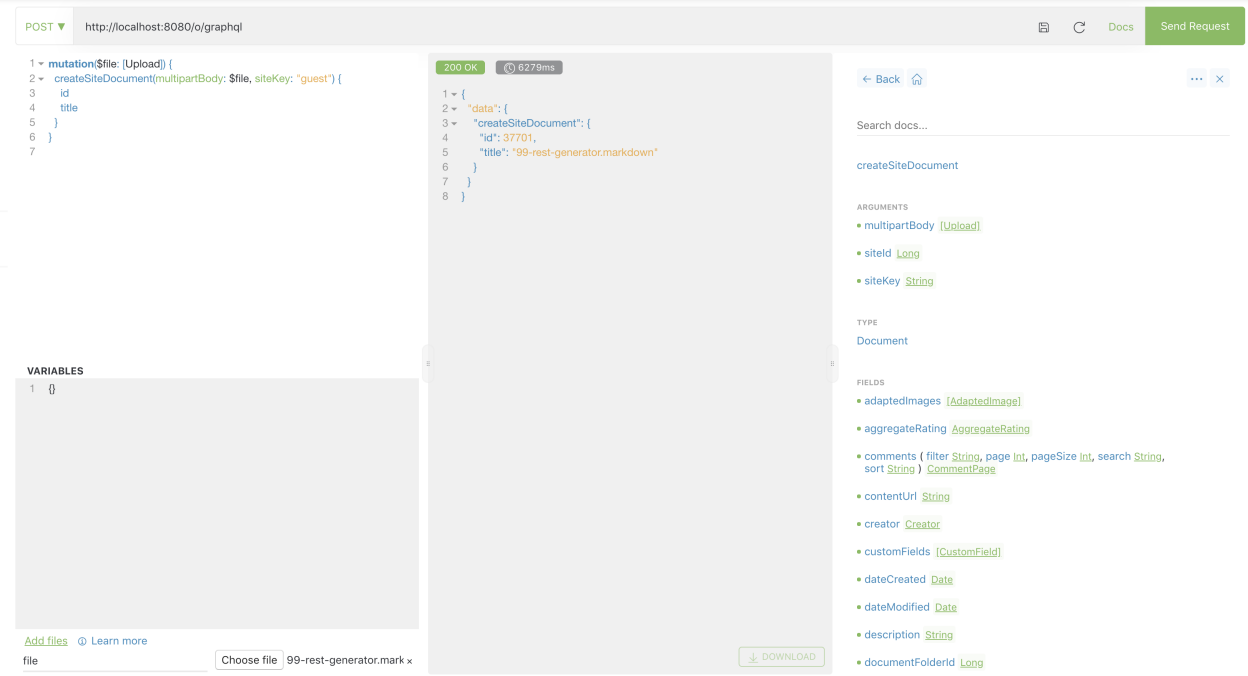

Figure 618.2: Creating a Document in Altair is easy with the selector.

```
mutation($file: [Upload]) {
  createSiteDocument(multipartBody: $file, siteKey: "guest") {
    id
    title
  }
}
```

The variable above is `file` because there's only one. If you wanted to upload several files, name the variable `$files` and each file should have a numeric sequence: `files.0`, `files.1`, `files.2`, etc.

All multipart APIs allow sending a JSON file containing the file's metadata (title, description, etc.). That parameter should be the second file uploaded (defined using the `file.0`, `file.1` syntax).

For example,

```
document={\"title\": \"Alternative name\"}"
```

And here's the response:

```
{
  "data": {
    "createSiteDocument": {
```

```
      "id": 37701,
      "title": "99-rest-generator.markdown"
    }
  }
}
```

The cURL request is slightly different (Altair fills out the variables):

```
curl 'http://localhost:8080/o/graphql' -H 'Authorization: Basic dGVzdEBsaWZlcmF5LmNvbTp0ZXN0' \
-F operations='{"query":"mutation($files: [Upload]) {createSiteDocument(multipartBody: $files, siteId: 20122) {id}}","variables": { "files": [null] }
-F map='{ "0": ["variables.files.0"]}' \
-F 0=@"99-rest-generator.markdown"
```

CHAPTER 619

USING GRAPHQL APIS

Liferay DXP's GraphQL APIs are independent of clients and can be used with any GraphQL client you want. The only usual requirements are setting up the `Authentication` header using OAuth, Cookie, Basic, etc.

For JavaScript applications, we recommend using Apollo Client or graphql-request, like this:

```
const { GraphQLClient } = require('graphql-request');

const graphQLClient = new GraphQLClient('http://localhost:8080/o/graphql', {
    headers: {
        authorization: `Basic ${AUTHORIZATION_TOKEN}`
    }
});

const getDestinationsQuery = ` {
  destinations: contentSetContentSetElements(contentSetId: ${DESTINATION_CONTENTSET_ID}) {
    items {
        id
        title
    }
    page
    pageSize
    totalCount
  }
}`;

...

const response = await graphQLClient.request(getDestinationsQuery);
```

Here are several examples of JavaScript applications using GraphQL APIs:

- Alexa skill using GraphQL APIs
- React application using Apollo Client for React
- Vue application using Apollo Client for Vue

CHAPTER 620

REST BUILDER

REST Builder is Liferay DXP's tool to build REST and GraphQL APIs. It's based on OpenAPI, following an API-first approach.

Using REST Builder takes only three steps:

1. Write the OpenAPI profile.
2. Use REST builder to generate the scaffolding.
3. Fill out the generated classes with your logic.

A good overview of the process is detailed here.

We'll see each step in detail but first, let's talk about why we want to use REST Builder.

620.1 Why we should use REST Builder

There are several reasons to prefer REST Builder over rolling our own JAX-RS services. Some of them are the following:

- Development speed: you avoid writing JAX-RS annotations, converters, adding support for multipart or layers to organize your code. Everything is generated.
- API scaffolding: pagination, filtering, searching, JSON writers, XML generation, even unit, and integration tests are generated.
- GraphQL support out of the box: write your REST API and get a GraphQL endpoint for free.
- Integration with Liferay's authentication pipelines: Basic, OAuth, Cookie, CORS handling. You don't have to search manually for the user or the company; everything is already there.
- JSON & XML support: APIs return whichever format the consumer prefers.
- Consistency: all APIs follow the same rules and conventions, enforced by REST builder.

CHAPTER 621

How to install REST Builder

Use the Gradle plugin to install REST builder by adding this gradle configuration to your project:

```
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"
```

To use it, run `gradlew buildREST`. Note that your Gradle wrapper may not be in your app's project directory, so you may need to use Blade to locate it:

```
blade gw buildREST
```

If you want to use a specific version of REST builder, you can specify it explicitly:

```
dependencies {
    restBuilder group: "com.liferay", name: "com.liferay.portal.tools.rest.builder", version: "1.0.30"
}
```

CHAPTER 622

REST BUILDER & OPENAPI

REST Builder is based on OpenAPI, and its philosophy is "OpenAPI first": first you write the profile and then you use it as the base of your implementation.

But first you must create a project with two empty bundles (a Blade template will follow soon). The bundles (-api and -impl) should have the files you are already used to: a `build.gradle` and a `bnd.bnd`. The novelty is two YAML files, a configuration file (`rest-config.yaml`) and the OpenAPI profile (`rest-openapi.yaml`). An example project is here.

Let's see the configuration file in detail. In the root of the -impl project we have to create a YAML file to specify paths and the basic configuration of our new API. A sample implementation would be:

```
apiDir: "../headless-test-api/src/main/java"
apiPackagePath: "com.liferay.headless.test"
application:
    baseURI: "/headless-test"
    className: "HeadlessTestApplication"
    name: "Liferay.Headless.Test"
author: "Javier Gamarra"
```

This file specifies the path of the -api bundle, the java package that we will use across all the bundles and the information of the JAX-RS application: the path of our application, the name of the class and the JAX-RS name of our API.

I've skipped two advanced features, generating a client and automated tests, will see them later.

Just one step left, writing our OpenAPI profile.

622.1 OpenAPI profile

The OpenAPI profile will be the source of all our APIs, in this file, we will add the paths and entities of our API. First, we'll create a YAML file called rest-openapi.yaml. Writing YAML files is tricky so we recommend using the swagger editor to do it, which validates the YAML file against YAML syntax and the OpenAPI specification.

A simple OpenAPI profile that retrieves a fictitious entity might look like this:

```
components:
  schemas:
```

```
    Entity:
      description: A very simple entity
      properties:
        name:
          description: The entity name.
          type: string
        id:
          description: The entity ID.
          type: integer
      type: object
info:
  description: ""
  title: "My API"
  version: v1.0
openapi: 3.0.1
paths:
  "/entities/{entityId}":
    get:
      parameters:
        - in: path
          name: entityId
          required: true
          schema:
            type: integer
      responses:
        200:
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Entity"
          description: ""
      tags: ["Entity"]
```

All OpenAPI profiles have three different sections: components, info, and paths. The easiest one is the information block. It contains the OpenAPI version, the title and the version of your API:

```
info:
  description: ""
  title: "My API"
  version: v1.0
openapi: 3.0.1
```

Indentations should be spaces. The swagger editor helps with formatting.

The components section specifies the schemas/entities to return or accept on your APIs. In this case, you define a schema called *Entity* that has two string fields: a name and an id.

```
components:
  schemas:
    Entity:
      description: A very simple entity
      properties:
        name:
          description: The entity name.
          type: string
        id:
          description: The entity ID.
          type: integer
      type: object
```

The OpenAPI specification defines many types and fields you can use in your schemas.

The other common type is `$ref`, a reference type that allows you to refer to an existing type like this:

```
$ref: '#/components/schemas/Entity'
```

The last block, called paths, defines the URLs that you'll expose in your APIs, with the type of HTTP verbs, list of parameters, status codes, etc.

```
paths:
  "/entities/{entityId}":
    get:
      parameters:
        - in: path
          name: entityId
          required: true
          schema:
            type: integer
      responses:
        200:
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Entity"
          description: ""
      tags: ["Entity"]
```

The pattern above, `"/entities/{entity}"`, follows a common pattern in REST APIs. This is the endpoint that retrieves one element, `"/entities"`. It returns a list of elements, and a POST request creates one.

For every path, it is mandatory to add a tag that points to an existing schema to indicate where to generate your code. REST Builder creates a method inside the class `[TAG]ResourceImpl.java`.

622.2 Generation

Once you've written your OpenAPI configuration and profile, it's time to generate your scaffolding for REST and GraphQL.

In the -impl or in the root module folder, execute this command:

```
gw buildREST
```

You can use `gw bR` if you want to save a few keystrokes.

If everything's indented properly and the OpenAPI profile validates, REST Builder generates your JAX-RS resources and the GraphQL endpoint. Next, you'll see what has been generated and how to implement our business logic.

622.3 Examples

Here's a complete example that defines all CRUD operations in OpenAPI.

622.4 GET Collection

```
paths:
    "/entities":
        get:
            responses:
                200:
                    content:
```

```
                    application/json:
                        schema:
                            items:
                                $ref: "#/components/schemas/Entity"
                            type: array
                description: ""
            tags: ["Entity"]
```

622.5 DELETE

```
paths:
    "/entities/{entityId}":
        delete:
            parameters:
                - in: path
                  name: entityId
                  required: true
                  schema:
                      type: integer
            responses:
                204:
                    content:
                        application/json: {}
                    description: ""
            tags: ["Entity"]
```

622.6 POST

```
paths:
    "/entities":
        post:
            requestBody:
                content:
                    application/json:
                        schema:
                            $ref: "#/components/schemas/Entity"
            responses:
                200:
                    content:
                        application/json:
                            schema:
                                $ref: "#/components/schemas/Entity"
                    description: ""
            tags: ["Entity"]
```

622.7 PUT

```
paths:
    "/entities/{entityId}":
        put:
            parameters:
                - in: path
                  name: entityId
                  required: true
                  schema:
                      type: integer
            requestBody:
                content:
                    application/json:
```

```
            schema:
                $ref: "#/components/schemas/Entity"
responses:
    200:
        content:
            application/json:
                schema:
                    $ref: "#/components/schemas/Entity"
        description: ""
tags: ["Entity"]
```

622.8 Summary

There are more examples showcasing all the supported OpenAPI syntax here and here. Your next step is to create your API.

CHAPTER 623

DEVELOPING AN API WITH REST BUILDER

After executing `gw buildREST`, you have two modules: *headless-test-api* and *headless-test-impl*.

- Headless Test API contains the interfaces for your resources and the POJOs of your schemas.
- Headless Test Impl contains your implementation and the JAX-RS application.

Your generated `EntityResource` looks like this:

```
public interface EntityResource {

    public Page<Entity> getEntitiesPage() throws Exception;

    public Entity postEntity(Entity entity) throws Exception;

    public void deleteEntity(Integer entityId) throws Exception;

    public Entity getEntity(Integer entityId) throws Exception;

    public Entity putEntity(Integer entityId, Entity entity) throws Exception;

    //Context methods

    public default void setContextAcceptLanguage(AcceptLanguage contextAcceptLanguage) {}

    public void setContextCompany(Company contextCompany);

    public default void setContextHttpServletRequest(HttpServletRequest contextHttpServletRequest) {}

    public default void setContextHttpServletResponse(HttpServletResponse contextHttpServletResponse) {}

    public default void setContextUriInfo(UriInfo contextUriInfo) {}

    public void setContextUser(User contextUser);
}
```

These are generated methods you defined in the OpenAPI profile (the full set as displayed in the examples).

REST builder also generates two implementation files, a base class, with all the JAX-RS, GraphQL and OpenAPI annotations and an empty implementation, `EntityResourceImpl`:

```
@Component(
    properties = "OSGI-INF/liferay/rest/v1_0/entity.properties",
```

```
    scope = ServiceScope.PROTOTYPE, service = EntityResource.class
)
public class EntityResourceImpl extends BaseEntityResourceImpl {
}
```

This is where you implement new methods, by overriding the base class implementation and returning your code. For example, here's a prototype implementation storing entities in a Map:

```
Map<Integer, Entity> entities = new HashMap<>();

@Override
public Entity getEntity(Integer entityId) throws Exception {
    return entities.get(entityId);
}

@Override
public Page<Entity> getEntitiesPage() throws Exception {
    return Page.of(entities.values());
}

@Override
public void deleteEntity(Integer entityId) throws Exception {
    entities.remove(entityId);
}

@Override
public Entity postEntity(Entity entity) throws Exception {
    entities.put(entity.getId(), entity);
    return entity;
}

@Override
public Entity putEntity(Integer entityId, Entity entity) throws Exception {
    entities.put(entity.getId(), entity);
    return entity;
}
```

For the collection, you return a `Page` object based on a list but there are also utility methods that return the pagination information:

```
Page.of(list, pagination, totalCount)
```

Don't touch the interfaces or the base classes (those are regenerated every time you run REST Builder). Like Service Builder, you only have to maintain the implementation classes, and if you change the API, run REST Builder again and the interfaces are updated. Your business logic could call other REST APIs, use Service Builder or another persistence mechanism.

623.1 Development Cycle

While implementing your API's business logic, you'll typically improve your API by adding parameters or other paths. For that, you'll modify the OpenAPI profile and regenerate the API again calling the `buildREST` command.

The cycle starts anew until you get to the final state and deploy your APIs. They become available at this URL pattern:

```
http://localhost:8080/o/[application class name]/[OpenAPI version]/
```

You can also execute `jaxrs:check` in the OSGi console to see all the JAX-RS endpoints.

GraphQL paths and entities are added automatically to the default GraphQL endpoint:

```
localhost:8080/o/graphql
```

You can disable GraphQL generation by adding `generateGraphQL: false` to your `rest-config.yaml` (`generateREST` controls the generation of the REST endpoints).

623.2 Wrapping Up

So... that's all!

When everything is ready, you might want to consider publishing your Headless API to Swaggerhub so others can consume it. You can use the following URL pattern for that:

```
http://localhost:8080/o/[application name]/[application version]/openapi.yaml
```

The URL for the example above, therefore, would be

```
http://localhost:8080/o/headless-test/v1.0/openapi.yaml
```

This URL has the content of `rest-openapi.yaml` plus the classes that REST Builder generated for you.

CHAPTER 624

MANAGING COLLECTIONS IN REST BUILDER

624.1 Pagination

To add pagination to your endpoints, add `page` and `pageSize` as query parameters to your OpenAPI profile, like this:

```
- in: query
  name: page
  schema:
      type: integer
- in: query
  name: pageSize
  schema:
      type: integer
```

Those two parameters add a `Pagination pagination` parameter in the method signature to restrict the number of entries to return in the `Page.of` constructor.

Pagination is highly recommended for entities that can have many elements, to avoid very large requests.

624.2 Filtering, sorting and searching

Adding support for filtering, sorting, and searching is trickier. The first step is to add the query parameters to the OpenAPI profile, like this:

```
- in: query
  name: filter
  schema:
      type: string
- in: query
  name: search
  schema:
      type: string
- in: query
  name: sort
  schema:
      type: string
```

The method signature then receives a Sort object, a Filter object, and string with the search request. Those objects use Liferay's indexing framework. This gives you many benefits, like support for permissions out-of-the-box and having to write very little code to achieve sort, filter, and search.

So first, you must make sure your entity is indexed and uses the indexing framework.

Once that's done you have three things to do:

1. Add an `EntityModel` to translate between our indexing framework and your code
2. Inject your `entityModel` into your resource implementation.
3. Call Search utilities to avoid boilerplate code.

624.3 Add an EntityModel

The `EntityModel` is a class that translates the name the property has in your API to the name used to index it.

```
public class EntityEntityModel implements EntityModel {
    public EntityEntityModel() {
        _entityFieldsMap = EntityModel.toEntityFieldsMap(
            new StringEntityField(
                "name", locale -> Field.getSortableFieldName(Field.NAME))
        );
    }

    @Override
    public Map<String, EntityField> getEntityFieldsMap() {
        return _entityFieldsMap;
    }

    private final Map<String, EntityField> _entityFieldsMap;
}
```

The `EntityModel` decouples the way you filter/sort from the way you index the information. You could use one field to sort, backed internally by several indexed fields or vice-versa.

624.4 Inject Your EntityModel

Injecting your `EntityModel` is really easy, our resource implementation just has to implement the `EntityModelResource` interface.

This entity model is simple and doesn't have any dynamic fields, so you can instantiate it directly and return it in the `getEntityModel` method, like this:

```
@Component(
    properties = "OSGI-INF/liferay/rest/v1_0/entity.properties",
    scope = ServiceScope.PROTOTYPE, service = EntityResource.class
)
public class EntityResourceImpl extends BaseEntityResourceImpl implements
    EntityModelResource {

    ...

    @Override
    public EntityModel getEntityModel(MultivaluedMap multivaluedMap) {
        return _entityEntityModel;
    }
```

```
    private EntityEntityModel _entityEntityModel = new EntityEntityModel();
}
```

624.5 Call search utilities

Finally, you must call `SearchUtil.search()` that links everything together. It requires these parameters:

`booleanQueryUnsafeConsumer`: a boolean query to restrict the information we want to retrieve.

`filter`: pass-through of the filter object.

`indexerClass`: the class of the entity that to filter/search.

`keywords`: pass-through of the search string.

`pagination`: pass-through of the pagination object (to read the row requested).

`queryConfigUnsafeConsumer`: the configuration for the fields that you want to return, typically the id to do a query later, in the `transformUnsafeFunction`.

`searchContextUnsafeConsumer`: global configuration of the query.

`transformUnsafeFunction`: the function that transforms from `Document` (of the indexing framework) to your entity, either searching in the database, your persistence, another API, etc.

`sorts`: pass-through of the sorts object.

The code would be similar to this:

```
@Override
public Page<Entity> getEntitiesPage(
    String search, Filter filter, Pagination pagination, Sort[] sorts)
    throws Exception {

    return SearchUtil.search(
        booleanQuery -> {},
        filter, Entity.class, search, pagination,
        queryConfig -> queryConfig.setSelectedFieldNames(
            Field.ENTRY_CLASS_PK),
        searchContext -> searchContext.setCompanyId(contextCompany.getCompanyId()),
        document -> new Entity(), //FILL with your implementation
        sorts);
}
```

624.6 Using Your filter, search, and sort

Lifeay uses OData to express our filter queries, following this syntax.

And that's it! Now you can can filter/search and sort by the fields you defined in your `EntityModel`.

CHAPTER 625

REST Builder Scaffolding

Apart from the JAX-RS annotations, basic structure and GraphQL support, there are some other things the REST Builder provides:

- Context Fields
- Automatic Transactions
- Test Generation
- Client Generation
- Common Utilities

These are useful and time saving additions to your development cycle.

625.1 Context fields

The `ResourceImpl` classes are JAX-RS-compliant resources. You can add `@Context` injections, mix JAX-RS endpoints, and use full power of the JAX-RS standard.

REST Builder injects several fields (as protected fields in the base class) to help you implement new APIs, like these:

- `contextAcceptLanguage`, containing the current `Locale`.
- `contextCompany`, containing the current `Company`.
- `contextHttpServletRequest`, containing the `HttpServletRequest`.
- `contextHttpServletResponse`, containing the `HttpServletResponse`.
- `contextUser`, containing the current logged-in `User`.
- `contextUriInfo`, containing the `UriInfo` information (paths, endpoints).

625.2 Automatic transactions

One little-known feature of REST Builder is that it wraps your API calls in transactions if the HTTP verb used is POST, PUT, PATCH or DELETE (doesn't apply for GET operations).

If you need to do several Service Builder calls in a sequence, you don't have to wrap them in a transaction; it happens automatically. The transaction commits if no exception is thrown and rolls back if an exception bubbles up to the resource implementation method.

625.3 Test generation

You can generate integration tests if you specify a `testDir` property:

```
testDir: "../headless-admin-taxonomy-test/src/testIntegration/java"
```

REST Builder generates integration tests under the -test module. Those tests check the API using a REST client. They check the response, doing an end-to-end test involving all steps from parsing the request to returning JSON.

The generated tests are ignored by default and, depending on the path, may force you to implement a creation method (to be able to add content to either update/delete or retrieve it).

The headless-delivery-test project contains many examples.

625.4 Client generation

You can generate a Java client if you specify a `clientDir` property:

```
clientDir: "../headless-admin-taxonomy-client/src/main/java"
```

This is a Java, typed, client that interacts with the APIs using static methods. The client project contains all the methods to call your paths and parses the requests and responses.

625.5 Common utilities

REST Builder provides several JAX-RS utilities, from exception mappers for the most common exceptions in the portal, XML and JSON Body Readers/Writers, Site validation (that works with `siteId` and `siteKey`), and support for Bean Validation.

There are also utility libraries that can be useful when developing new APIs:

- `ContentLanguageUtil`, to deal with the `ContentLanguage` header.
- `JaxRsLinkUtil`, to create links between APIs.
- `LocalDateTimeUtil`, to transform between date formats.
- `LocalizedMapUtil`, to create maps with locales as keys.
- `SearchUtil`, to use the search framework to return Pages of content.
- `TransformUtil`, to deal with lambdas and conversions.

CHAPTER 626

SUPPORT FOR ONEOF, ANYOF AND ALLOF

OpenAPI 3.0 added several ways of using inheritance and composition to create complex schemas. Specifically, it added support for *allOf, anyOf,* and *oneOf* , with these semantics:

- allOf – the value validates against all the subschemas
- anyOf – the value validates against any of the subschemas
- oneOf – the value validates against exactly one of the subschemas

Next, you'll learn each option's syntax and how its code is generated.

626.1 allOf

With `allOf` you can use the power of composition to create an entity that combines several others. It's the most potent way of reusing code without losing expressiveness and granularity: you can define small entities that can be reused by composing several to create a larger entity.

To use `allOf` you must follow this syntax:

```
EntityA:
    properties:
        nameA:
            type: string
EntityB:
    properties:
        nameB:
            type: string
EntityC:
    allOf:
        - $ref: '#/components/schemas/EntityA'
        - $ref: '#/components/schemas/EntityB'
```

This OpenAPI syntax generates the following Java code inside the `EntityC` class:

```
@Schema
@Valid
public EntityA getEntityA() {
    return entityA;
}
```

```
@Schema
@Valid
public EntityB getEntityB() {
    return entityB;
}
```

626.2 oneOf

`OneOf` is the simplest of the generics properties. It defines a property that can have different types. Since Java doesn't support Union Types, use an Object to model the property:

```
EntityA:
    properties:
        nameA:
            type: string
EntityB:
    properties:
        nameB:
            anyOf:
                - $ref: "#/components/schemas/EntityA"
                - type: object
                  properties:
                      name:
                          type: string
```

This syntax generates the following Java code:

```
@Schema
@Valid
public Object getNameB() {
    return nameB;
}
```

626.3 anyOf

The final generic keyword, `anyOf` leverages `JsonSubTypes` to extend entities with properties using inheritance. You can define parent relationships (in this example, `EntityC`) with two children containing the properties of the parent and their own properties. Here's how to define YAML to use inheritance:

```
EntityC:
    oneOf:
        - properties:
            nameA:
                type: string
        - properties:
            nameB:
                type: string
    properties:
        nameC:
            type: string
```

This generates a parent class and two children:

```
@JsonSubTypes(
    {
        @JsonSubTypes.Type(name = "nameA", value = NameA.class),
```

```
        @JsonSubTypes.Type(name = "nameB", value = NameB.class)
    }
)
@JsonTypeInfo(
    include = JsonTypeInfo.As.PROPERTY, property = "childType",
    use = JsonTypeInfo.Id.NAME
)
@Generated("")
@GraphQLName("EntityC")
@JsonFilter("Liferay.Vulcan")
@XmlRootElement(name = "EntityC")
public class EntityC {

    @Schema
    public String getNameC() {
        return nameC;
    }

    public void setNameC(String nameC) {
        this.nameC = nameC;
    }
```

And two children classes look like this:

```
@JsonTypeInfo(
    defaultImpl = NameA.class, include = JsonTypeInfo.As.PROPERTY,
    property = "childType", use = JsonTypeInfo.Id.NAME
)
@Generated("")
@GraphQLName("NameA")
@JsonFilter("Liferay.Vulcan")
@XmlRootElement(name = "NameA")
public class NameA extends EntityC {

    @Schema
    public String getNameA() {
        return nameA;
    }

    public void setNameA(String nameA) {
        this.nameA = nameA;
    }
```

CHAPTER 627

REST BUILDER LIFERAY CONVENTIONS

Liferay's headless APIs follow several patterns and conventions to provide consistency and uniformity in the APIs.

Below is a list the most important ones. It's a living list. Improvements are being made all the time, so check back to stay up to date on the changes.

627.1 YAML & OpenAPI restrictions

- *Tags are required*. We can't assign a class to a DELETE operation (doesn't return anything, doesn't receive an entity) so we need the tag to assign the method to a Java class.
- Responses must return a status code (default is not supported).
- Paths must be quoted.
- Paths only contain the method path (application and version are inherited from the JAX-RS application).

627.2 Conventions

- We use path parameters for information that is required (like the id in a DELETE operation) and query parameters for optional information (filtering, sorting...)
- We don't expose `className`. If you need to return information like the `className`, use `contentType` keyword.

CHAPTER 628

THE WORKFLOW FRAMEWORK

Blogs Entries, Journal Articles, and Forms Entries are just a few Assets supporting workflow. There's nothing stopping you from likewise enabling workflow for your custom Assets. Discover here how the workflow framework works, and find the steps and code samples for enabling your custom entities to use the workflow capabilities in subsequent articles.

A workflow process is a set of steps that an Asset must proceed through before it's marked with the workflow status *Approved*. The steps are defined in an XML file called a workflow definition. Each Asset is configured to run through a specific workflow definition via the Control Panel.

The workflow status is a database field that must be present for an entity to support workflow. If a database has the status field, but no workflow code has been written, it's auto-marked *Approved* by Liferay's Service Builder infrastructure, to assure that everything works smoothly by default.

628.1 Supporting Workflow in the Database

There are several database fields that must be present for an Asset to support workflow:

`int status` represents the workflow status of each Asset.

`long statusByUserId` is the ID of the user that set the status (for example, the initial User that hit the Submit for Publication button to add a new Asset.

`String statusByUserName` is the User Name of the User that set the status of the Asset.

`Date statusDate` is the date/time when the status was set.

For Service Builder applications, add these as entity columns in the `service.xml` file, run Service Builder, and you're good to go.

628.2 Setting the Status Fields

Once the database table has the proper status fields, set them in your Entity's `addEntity` service method. Initially, set the status as a DRAFT. It's what the workflow framework expects of an entity as it enters the workflow process. The status is an `int`, but you don't have to remember which number corresponds to the DRAFT status. Instead, use the `WorkflowConstants` in `portal-kernel`. For a draft, pass in

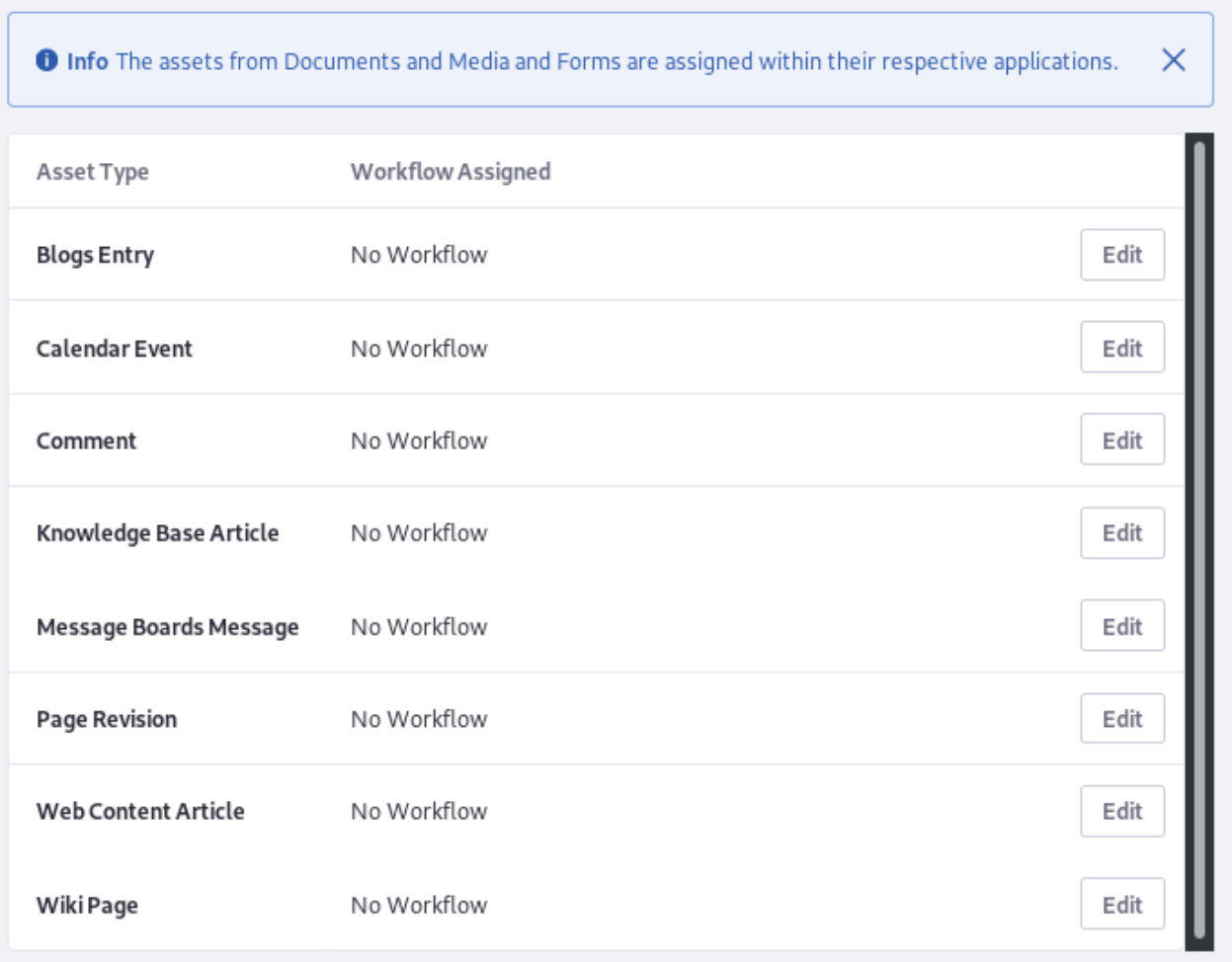

Figure 628.1: Enable workflow on your custom Asset, and it can be sent through a workflow process just like a native Asset.

`WorkflowConstants.STATUS_DRAFT`

If you're curious, the `int` represented by this constant is `2`. Another important status, APPROVED, is represented by the `int` value `0` and the constant

`WorkflowConstants.STATUS_APPROVED`

The User fields (`statusByUserID` and `statusByUserName`) are easy, since the `userId` of the User making the `addEntity` request is part of the request itself, and passed into the `addEntity` method for most assets. Use the ID directly as the `statusByUserId`, and get the full name associated with the User by using the ID to retrieve the `User` object.

```
entity.setStatusByUserId(userId);
entity.setStatusByUserName(user.getFullName());
```

The `statusDate` is usually best set as the date the entity was modified, and is part of the Service Context in the request:

```
entity.setStatusDate(serviceContext.getModifiedDate(null));
```

Once the status dates are set, the entity is ready to be sent into the workflow framework.

628.3 Sending the Entity to the Workflow Framework

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 call to `WorkflowHandlerRegistryUtil.startWorkflowInstance` in your persistence code.

628.4 Allowing the Workflow Framework to Handle the Entity

Once the entity is sent to the Workflow Framework, much of the process is automated, and you need not worry about the details. Write one class that gives the framework some information on how to process the entity. It's called a workflow handler (`WorkflowHandler<T>`), and you can create it by extending the handy abstract implementation, `BaseWorkflowHandler<T>`.

The workflow handler usually goes in the module containing service implementations. It's nice to keep your back-end code separate from your view layer and controller (ala the MVC pattern).

Make your workflow handler a Component class so it can be registered properly with OSGi runtime. It requires one Component property, `model.class.name`, which is the fully qualified class name for class you pass as the type parameter in the class declaration.

In addition to the property, declare the type of service you're providing in the Component: `WorkflowHandler.class`.

Workflow handlers extending the `BaseWorkflowHandler` must override three methods:

`getClassName` returns the model class's fully qualified class name (`com.my.app.package.model.FooEntity`, for example).

`getType` returns the model resource name (`model.resource.com.my.app.package.model.FooEntity`, for example).

`updateStatus` does most of the heavy lifting here. 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. 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.

628.5 Supporting Workflow in the Service Layer

The service layer must update the status of the entity when it returns from the Workflow Framework. Make an `updateStatus` method for this purpose, and make sure, at a minimum, to set the status fields again as the Asset comes out of the Workflow Framework, and call the persistence layer's `update` method.

After that, provide any additional logic you might want, like checking the status and updating the Asset's visibility (using the `assetEntryLocalService`) based on the condition (visible if *Approved*, not visible is any other status).

Return the entry once you're through here.

628.6 Database Cleanup: Delete the Workflow Instance Links

When you send an entity to the workflow framework via the `startWorkflowInstance` call, it creates an entry in the `workflowinstancelink` database table. In your service layer's deletion logic, you must delete the workflow instance links. This `delete` call ensures there are no orphaned entries in the `workflowinstancelinks` table.

To get the `WorkflowInstanceLocalService` injected into your `*LocalServiceBaseImpl` so you can call its methods in the `LocalServiceImpl`, add a `reference entity` to your entity declaration in `service.xml`, specifying `WorkflowInstancelink`.

628.7 Updating the User Interface

After you finish all the backend work, update your UI. Some common tasks here include:

- In any public facing portion of the application (accessible to guest Users), don't display the entity if the status is anything except *Approved*. This task requires the creation of an additional `finder` method that accounts for workflow status, and a corresponding `getter` to expose it in the service layer.
- In administrative portions of the application, display the entities, but also display their workflow status. There's a tag library for this.

See the next article for more concrete steps and code snippets.

CHAPTER 629

LIFERAY'S WORKFLOW FRAMEWORK

To workflow-enable your entities,

1. Create a Workflow Handler
2. Update the Service Layer
3. Update the User Interface

Time to get started.

629.1 Creating a Workflow Handler

If you're in a Service Builder application, the workflow handler goes in your `-service` module.

1. Create a Component class that extends `BaseWorkflowHandler<T>`.

```
@Component(immediate = true, service = WorkflowHandler.class)
public class FooEntityWorkflowHandler extends BaseWorkflowHandler<FooEntity>
```

2. Override three methods in the workflow handler.

```
@Override
public String getClassName() {
    return FooEntity.class.getName();
}

@Override
public String getType(Locale locale) {
    return ResourceActionsUtil.getModelResource(locale, getClassName());
}

@Override
public FooEntity updateStatus(int status, Map<String, Serializable> workflowContext) throws PortalException {
... }
```

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.

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. Here's an example `updateStatus` method:

```
@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(
        WorkflowConstants.CONTEXT_SERVICE_CONTEXT);

    return _fooEntityLocalService.updateStatus(
        userId, classPK, status, serviceContext);
}
```

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.

629.2 Updating the Service Layer

In most Liferay applications, Service Builder is used to create database fields. First, you must update the service layer:

1. Make sure your entity database table has `status`, `statusByUserId`, `statusByUserName`, and `statusDate` fields.

   ```
   <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`).

3. At the end of any method that adds a new entity to your database, call the workflow service to enable sending the entity into the workflow backend:

```
WorkflowHandlerRegistryUtil.startWorkflowInstance(
    fooEntity.getCompanyId(), fooEntity.getGroupId(), fooEntity.getUserId(),
    FooEntity.class.getName(), fooEntity.getPrimaryKey(), fooEntity,
    serviceContext);
```

4. Implement the `updateStatus` method that must be called in the workflow handler. In the end, persist 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);
```

5. Do anything else that might make sense here, like changing the visibility of the asset depending on its workflow status:

```
if (status == WorkflowConstants.STATUS_APPROVED) {
    assetEntryLocalService.updateVisible(
        FooEntity.class.getName(), fooEntityId, true);
}
else {
    assetEntryLocalService.updateVisible(
        FooEntity.class.getName(), fooEntityId, false);
}
```

   Here's what a full `updateStatus` method might look like:

```
@Indexable(type = IndexableType.REINDEX)
public FooEntity updateStatus(
    long userId, long fooEntityId, int status, ServiceContext serviceContext
) throws PortalException, SystemException {

        User user = userLocalService.getUser(userId);
        FooEntity fooEntity = getFooEntity(fooEntityId);

        fooEntity.setStatus(status);
        fooEntity.setStatusByUserId(userId);
        fooEntity.setStatusByUserName(user.getFullName());
        fooEntity.setStatusDate(new Date());

        fooEntityPersistence.update(fooEntity);

        if (status == WorkflowConstants.STATUS_APPROVED) {
            assetEntryLocalService.updateVisible(
          FooEntity.class.getName(), fooEntityId, true);
        }
    else {
            assetEntryLocalService.updateVisible(
          FooEntity.class.getName(), fooEntityId, false);
        }

        return fooEntity;
  }
```

6. Add a call to `deleteWorkflowInstanceLinks` in the `deleteEntity` method:

```
workflowInstanceLinkLocalService.deleteWorkflowInstanceLinks(
    fooEntity.getCompanyId(), fooEntity.getGroupId(),
    FooEntity.class.getName(), fooEntity.getFooEntityId());
```

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`.

Once you've accounted for workflow status in your database and service layer, there's only one thing left to do: update the user interface.

629.3 Workflow Status and the View Layer

This is dependent on the needs of your application, but often involves the following these steps:

Display only approved entities:

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()) %>"
/>
    ...
```

Display the workflow status:

When you want to display the workflow status, use the `<aui:worklfow-status>` tag.

```
<aui:workflow-status markupView="lexicon" showIcon="<%= false %>" showLabel="<%= false %>" status="<%= fooEntity.getStatus() %>" />
```

Once your user interface accounts for workflow, your Liferay application is fully workflow enabled.

Chapter 630

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 shows how to customize these WYSIWYG editors for your apps and sites.

CHAPTER 631

ADDING A WYSIWYG EDITOR TO A PORTLET

It's easy to include WYSIWYG editors in your portlet, thanks to the `<liferay-editor:editor />` tag. Follow these steps:

1. Add the liferay-editor taglib declaration to your portlet's JSP:

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

2. Add the editor to your JSP with the `<liferay-editor:editor />` tag. Configure it using the attributes shown in the table below:

```
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](https://docs.liferay.com/dxp/apps/frontend-editor/latest/taglibdocs/liferay-editor/editor.html)
for the complete list of supported attributes.

Below is an example configuration:

```html
<div class="alloy-editor-container">
    <liferay-editor:editor
            contents="Default Content"
            cssClass="my-alloy-editor"
            editorName="alloyeditor"
```
```

```
            name="myAlloyEditor"
            placeholder="description"
            showSource="true"
    />
</div>
```

3. Optionally 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>
```

As you can see, it's easy to include WYSIWYG editors in your portlets!

631.1 Related Topics

- Adding New Behavior to an Editor
- Modifying an Editor's Configuration
- Modifying the AlloyEditor

CHAPTER 632

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 article, you'll learn how to modify the default configuration for Liferay DXP's supported WYSIWYG editors to meet your requirements.

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 `4.13.1`. 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:

```java
@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/7-2/customization/-/knowledge_base/c/creating-a-custom-osgi-service):

```java
@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 article 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.

632.1 Related Topics

- Adding New Behavior to an Editor
- Modifying the AlloyEditor
- Adding a WYSIWYG Editor to a Portlet

CHAPTER 633

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.

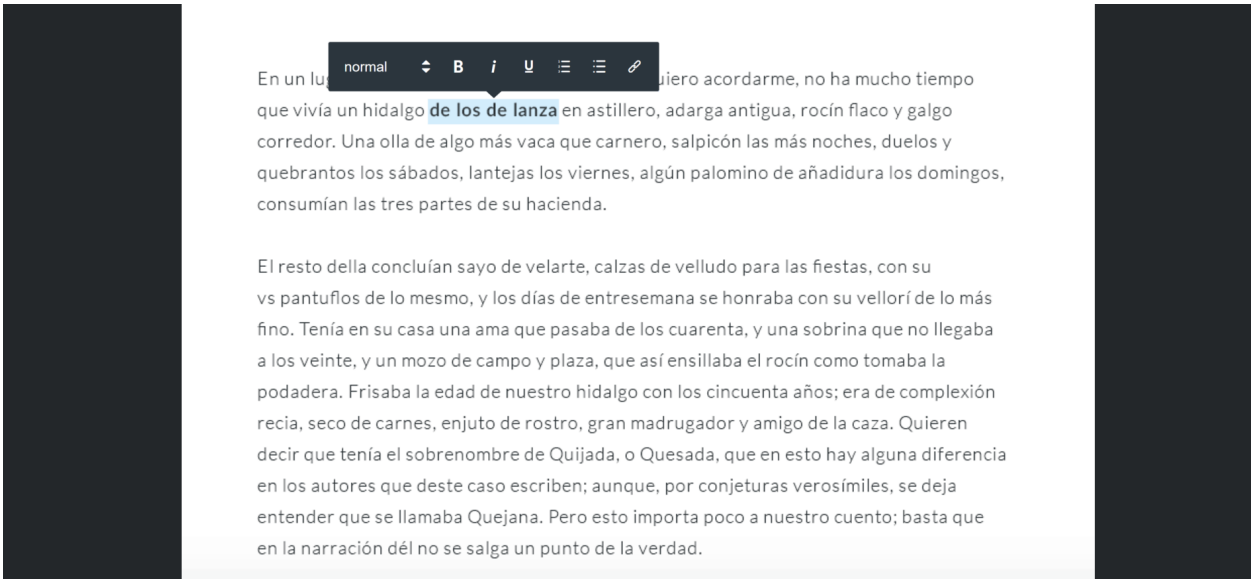

Figure 633.1: AlloyEditor is the default WYSIWYG editor built on top of CKEditor.

This section shows how to modify the default AlloyEditor configuration to meet your requirements.

CHAPTER 634

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 used as an example throughout this section. If you want to use it as a starting point for your own configuration or follow along with the articles, you can download the module's zip file from the Github repo.

CHAPTER 635

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 `4.13.1`. 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.

635.1 Related Topics

- Adding New Behavior to an Editor
- CKEditor Plugin Reference Guide

CHAPTER 636

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:

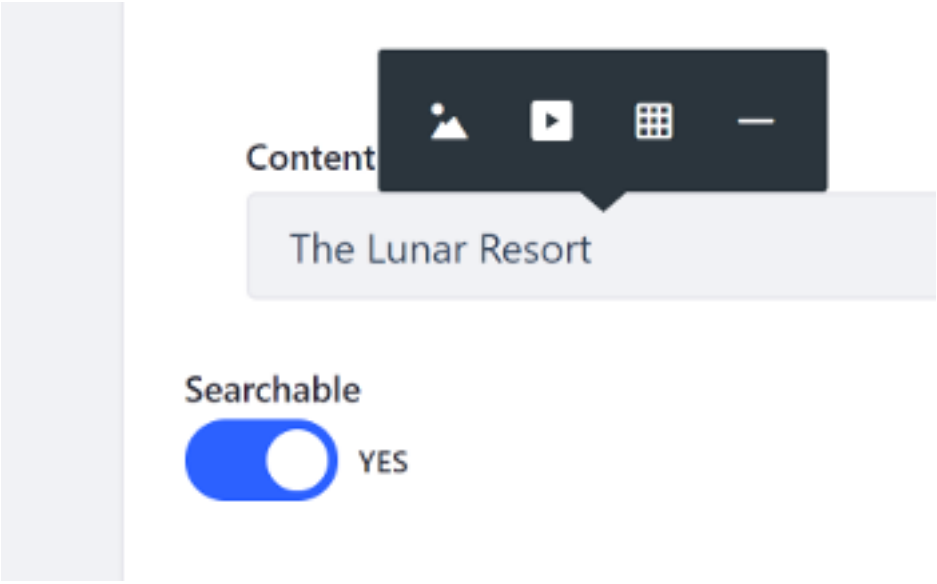

Figure 636.1: The Add toolbar lets you add content to the editor.

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:

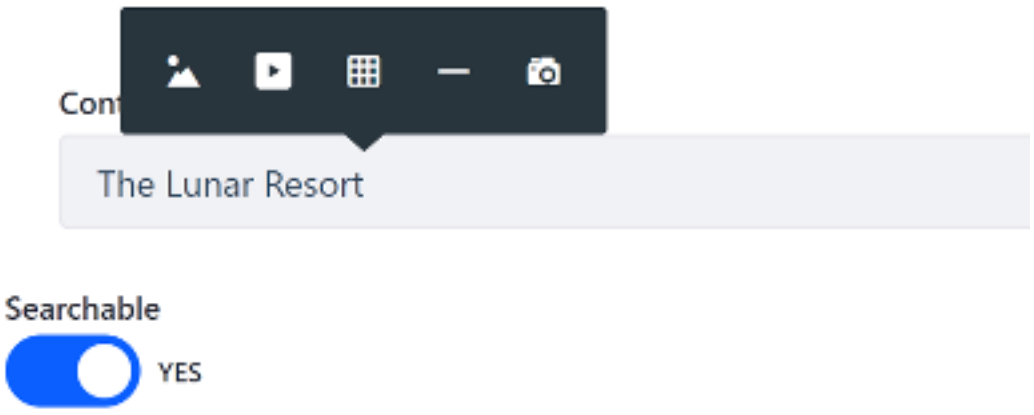

Figure 636.2: The Updated Add toolbar lets you add pictures from a camera directly to the editor.

636.1 Related Topics

- Adding New Behavior to an Editor
- Adding a Button to a Styles Toolbar

CHAPTER 637

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:

`embedurl`: Appears when embedded content is selected.

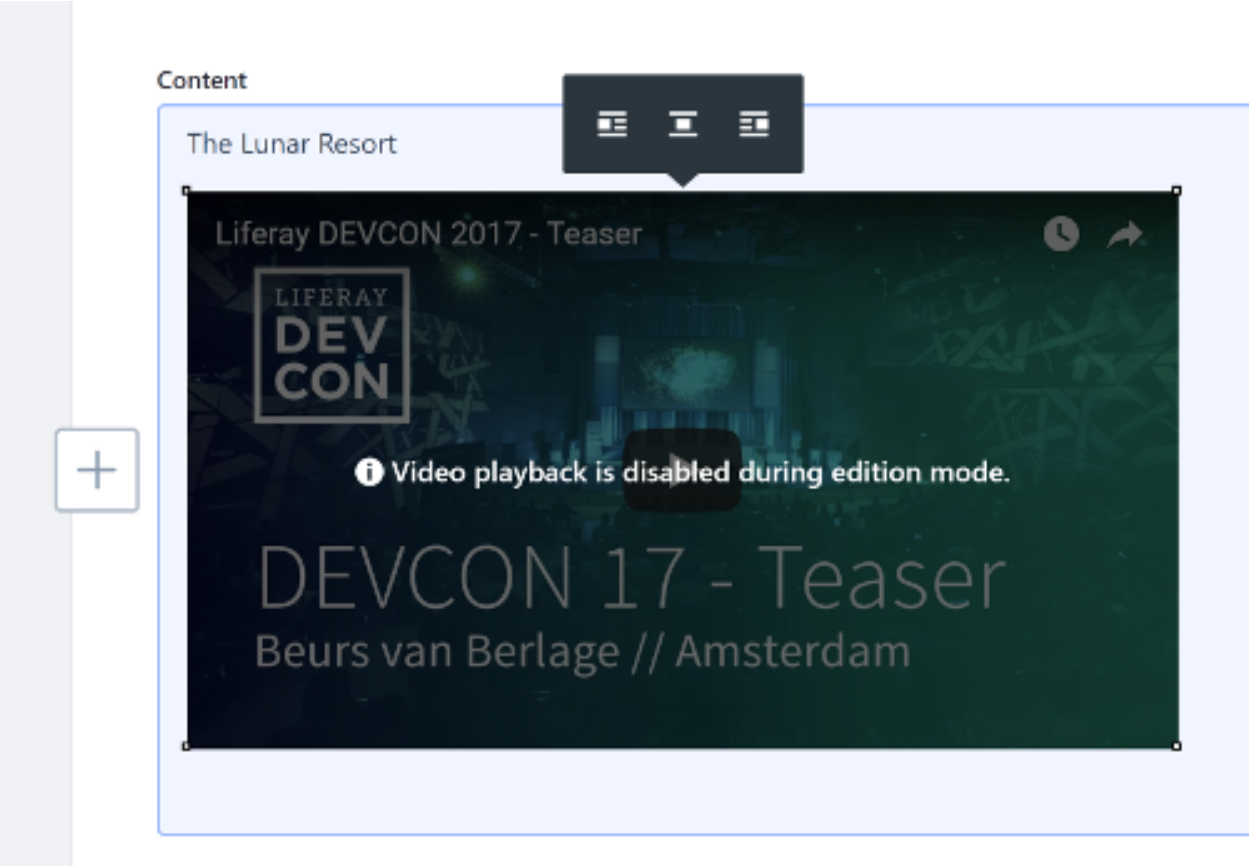

Figure 637.1: The embed URL Styles toolbar lets you format embedded content in the editor.

`image`: Appears when an image is selected.

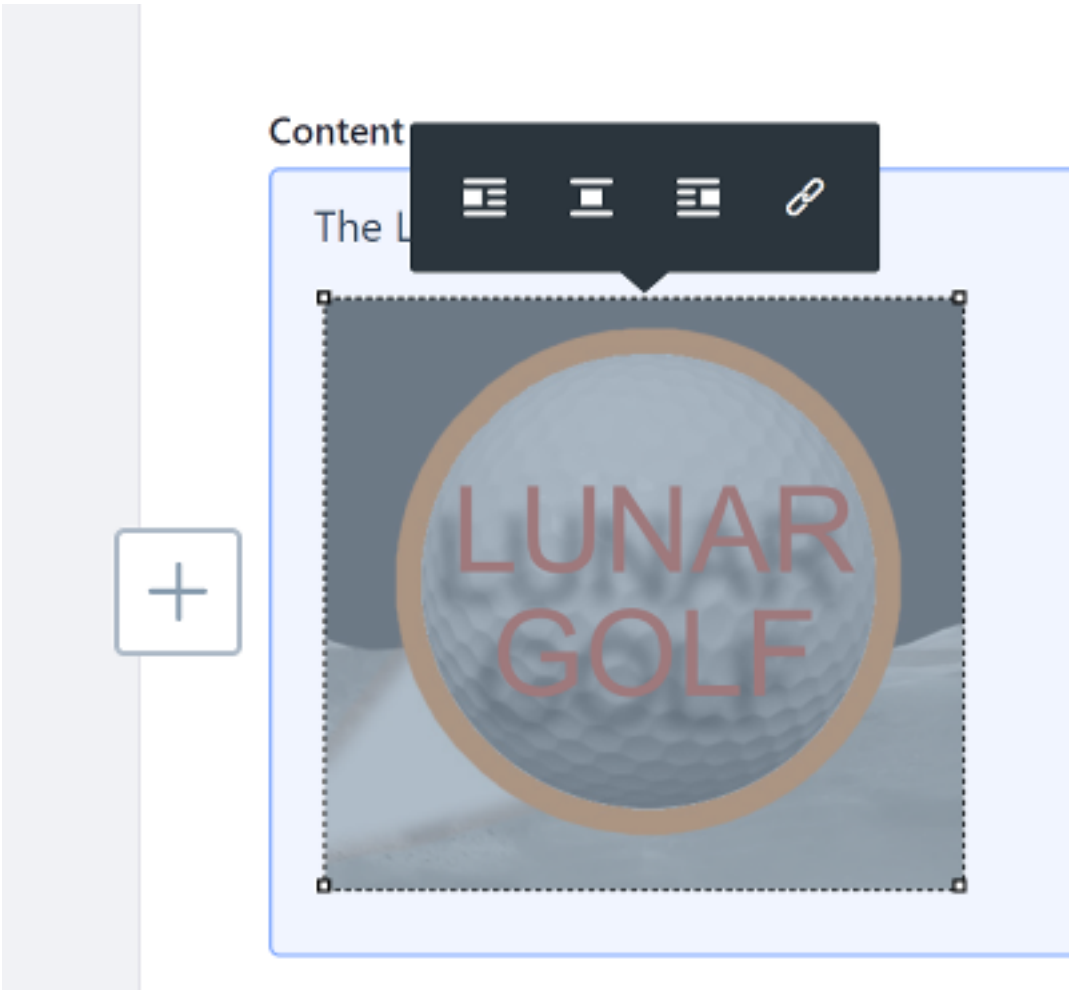

Figure 637.2: The image Styles toolbar lets you format images in the editor.

`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

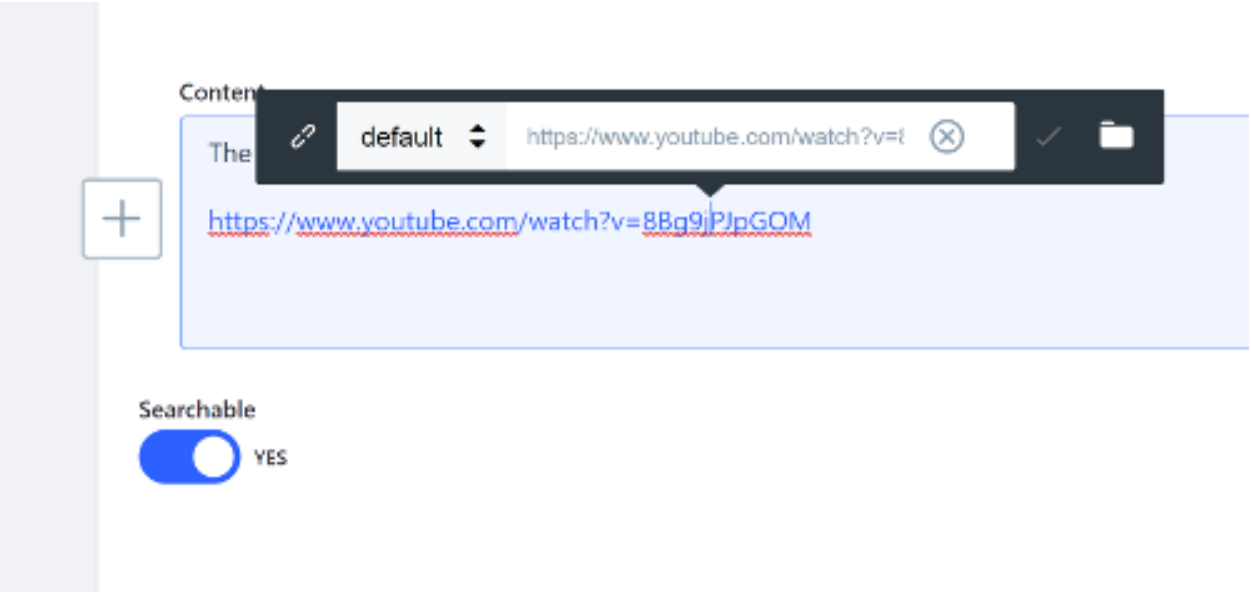

Figure 637.3: The link Styles toolbar lets you format hyperlinks in the editor.

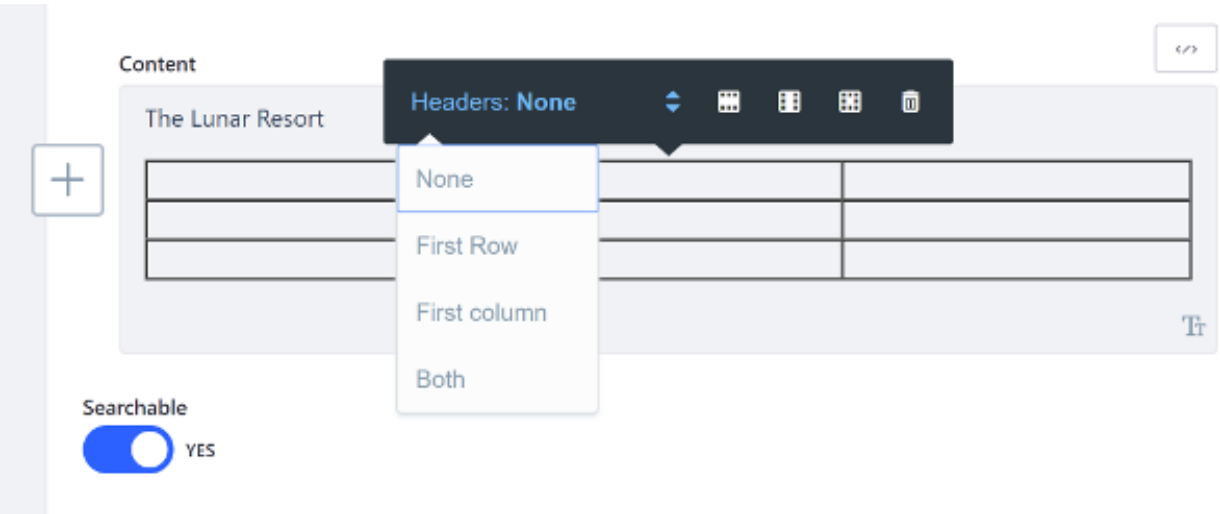

Figure 637.4: The table Styles toolbar lets you format tables in the editor.

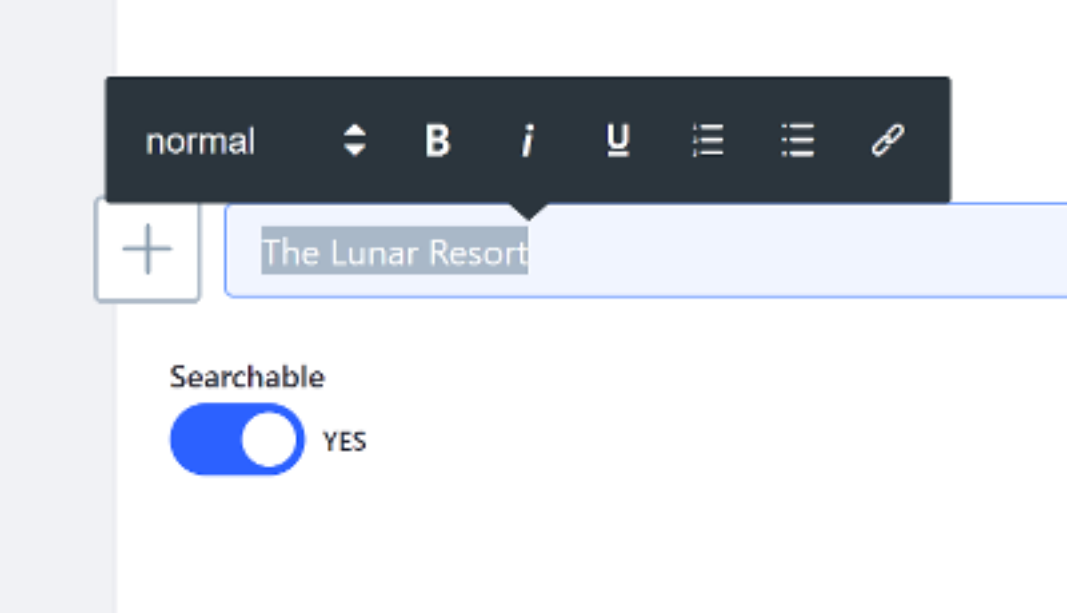

Figure 637.5: The text Styles toolbar lets you format highlighted text in the editor.

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:

```
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/camera.html) 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:

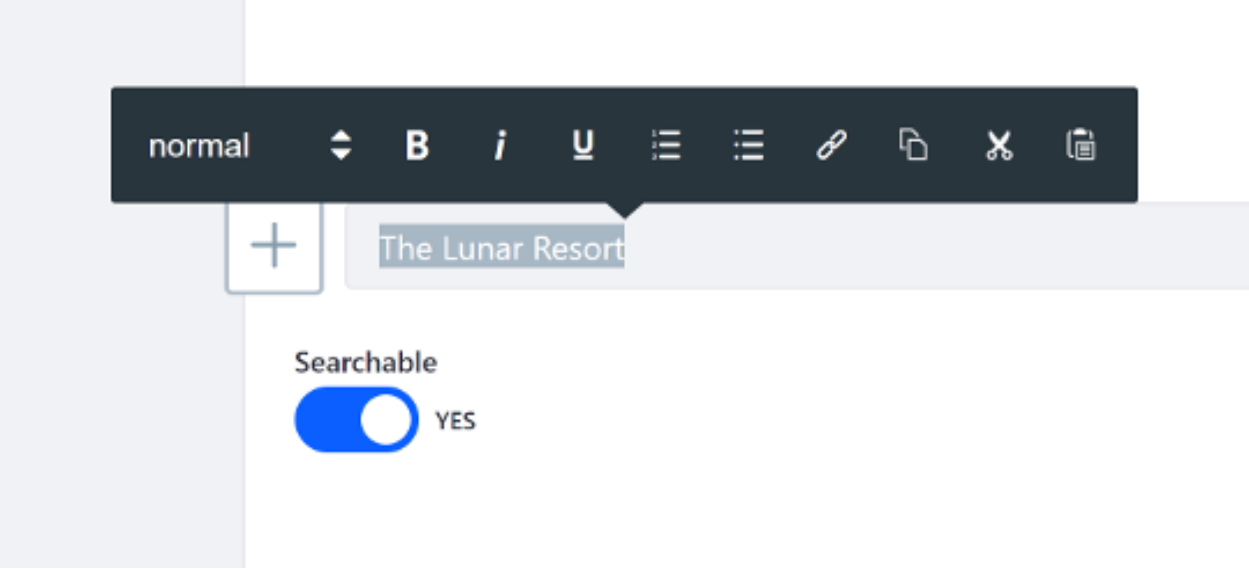

Figure 637.6: The Updated text styles toolbar lets you copy, cut, and paste text in the editor.

637.1 Related Topics

- Adding a Button to the Add Toolbar
- CKEditor Plugin Reference Guide

CHAPTER 638

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.

638.1 Related Topics

- Adding Buttons to AlloyEditor's Toolbars
- Adding New Behavior to an Editor

CHAPTER 639

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 code shown below to the editor:

```
CKEDITOR.dtd.$removeEmpty.i = 0
```

The `liferay-util:dynamic-include` extension point is in configurable editors' JSP files: it's the gateway for injecting JavaScript into your editor instance. In this article, you'll learn how to use this JavaScript extension point. Follow these steps to inject JavaScript into the WYSIWYG editor to modify its behavior:

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!

639.1 Related Topics

- Adding New Behavior to an Editor
- Embedding Portlets in Themes
- Portlets

Part V

Reference

CHAPTER 640

DEVELOPER REFERENCE

This developer reference contains lists of options for various APIs. The actual API reference is stored on docs.liferay.com. Here, you'll find higher level descriptions of those APIs, lists of tag libraries, descriptions of Gradle and Maven plugins, and much more.

One highlight here is a complete description of Liferay's development tooling. This includes not only our Blade CLI (a tool that bridges the gap between Gradle and Maven, bringing archetype-like functionality to Liferay Gradle projects), but also our plugins for IntelliJ and Eclipse, not to mention our Maven or Gradle-based Liferay Workspace. It also includes a complete description of our JS Generator, which helps front-end developers create pure JavaScript widgets.

CHAPTER 641

CLASSES MOVED FROM PORTAL-SERVICE.JAR

To leverage the benefits of modularization, many classes from 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 group, artifact ID, and version are listed, to facilitate configuring dependencies.

Classes Moved from portal-service.jar to Modules

This information was generated based on comparing classes in liferay-portal-src-6.2-ee-sp20 to classes in liferay-dxp-src-7.2.10-ga1.

Class

Package

Group ID, Artifact ID, and Version

ActionHandler

Old: com.liferay.portal.kernel.mobile.device.rulegroup.action New: com.liferay.mobile.device.rules.action

com.liferay com.liferay.mobile.device.rules.api 4.0.4

ActionHandlerManager

Old: com.liferay.portal.kernel.mobile.device.rulegroup New: com.liferay.mobile.device.rules.action

com.liferay com.liferay.mobile.device.rules.api 4.0.4

ActionHandlerManagerUtil

Old: com.liferay.portal.kernel.mobile.device.rulegroup New: com.liferay.mobile.device.rules.action

com.liferay com.liferay.mobile.device.rules.api 4.0.4

ActionTypeException

Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception

com.liferay com.liferay.mobile.device.rules.api 4.0.4

AlternateKeywordQueryHitsProcessor

Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.hits

com.liferay com.liferay.portal.search 6.0.14

ArticleContentException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception

com.liferay com.liferay.journal.api 4.2.1

ArticleContentSizeException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
ArticleCreateDateComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay com.liferay.journal.api 4.2.1
ArticleDisplayDateComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay com.liferay.journal.api 4.2.1
ArticleDisplayDateException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
ArticleExpirationDateException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
ArticleIDComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay com.liferay.journal.api 4.2.1
ArticleIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
ArticleModifiedDateComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay com.liferay.journal.api 4.2.1
ArticleResourcePKComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay com.liferay.journal.api 4.2.1
ArticleReviewDateComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay com.liferay.journal.api 4.2.1
ArticleReviewDateException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
ArticleSmallImageNameException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
ArticleSmallImageSizeException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
ArticleTitleComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay com.liferay.journal.api 4.2.1
ArticleTitleException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
ArticleVersionComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay com.liferay.journal.api 4.2.1
ArticleVersionException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
AuditMessageProcessor
Old: com.liferay.portal.kernel.audit New: com.liferay.portal.security.audit
com.liferay com.liferay.portal.security.audit.api 4.0.5
AutoDeleteFileInputStream
Old: com.liferay.portal.kernel.io New: com.liferay.petra.io
com.liferay com.liferay.petra.io 3.0.2
AverageStatistics
Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics
com.liferay com.liferay.portal.monitoring 7.0.7
BackgroundTaskLocalService
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskLocalServiceUtil
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskLocalServiceWrapper
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskModel
Old: com.liferay.portal.model New: com.liferay.portal.background.task.model
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskPersistence
Old: com.liferay.portal.service.persistence New: com.liferay.portal.background.task.service.persistence
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskService
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskServiceUtil
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskServiceWrapper
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskSoap
Old: com.liferay.portal.model New: com.liferay.portal.background.task.model
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskUtil
Old: com.liferay.portal.service.persistence New: com.liferay.portal.background.task.service.persistence
com.liferay com.liferay.portal.background.task.api 4.1.3
BackgroundTaskWrapper
Old: com.liferay.portal.model New: com.liferay.portal.background.task.model
com.liferay com.liferay.portal.background.task.api 4.1.3
BannedUserException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay com.liferay.message.boards.api 5.1.4
BaseCmisRepository
Old: com.liferay.portal.kernel.repository.cmis New: com.liferay.document.library.repository.cmis
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
BaseCmisSearchQueryBuilder
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
BaseDDLExporter
Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.internal.exporter
com.liferay com.liferay.dynamic.data.lists.service 3.0.12
BaseDDMDisplay
Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
BaseFieldRenderer
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
BaseScriptingExecutor
Old: com.liferay.portal.kernel.scripting New: com.liferay.portal.scripting
com.liferay com.liferay.portal.scripting.api 3.0.3
BaseStatistics
Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics
com.liferay com.liferay.portal.monitoring 7.0.7
BaseStorageAdapter
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
BlockingPortalCache
Old: com.liferay.portal.kernel.cache New: com.liferay.portal.cache
com.liferay com.liferay.portal.cache.api 2.0.1
BlogsEntry
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryFinder
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryLocalService
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryLocalServiceUtil
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5

BlogsEntryLocalServiceWrapper
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryModel
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryPersistence
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryService
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryServiceUtil
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryServiceWrapper
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntrySoap
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryUtil
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay com.liferay.blogs.api 5.0.5
BlogsEntryWrapper
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUser
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUserFinder
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUserLocalService
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUserLocalServiceUtil
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUserLocalServiceWrapper
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUserModel
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUserPersistence
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay com.liferay.blogs.api 5.0.5

BlogsStatsUserSoap
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUserUtil
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay com.liferay.blogs.api 5.0.5
BlogsStatsUserWrapper
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay com.liferay.blogs.api 5.0.5
BookmarksEntry
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryFinder
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryLocalService
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryLocalServiceUtil
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryLocalServiceWrapper
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryModel
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryPersistence
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryService
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryServiceUtil
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryServiceWrapper
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntrySoap
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksEntryUtil
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay com.liferay.bookmarks.api 4.0.5

BookmarksEntryWrapper
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolder
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderConstants
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderFinder
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderLocalService
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderLocalServiceUtil
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderLocalServiceWrapper
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderModel
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderPersistence
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderService
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderServiceUtil
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderServiceWrapper
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderSoap
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderUtil
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay com.liferay.bookmarks.api 4.0.5
BookmarksFolderWrapper
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay com.liferay.bookmarks.api 4.0.5

ByteArrayReportResultContainer
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay com.liferay.portal.reports.engine.api 5.0.1
CMISBetweenExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISConjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISContainsExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISContainsNotExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISContainsValueExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISCriterion
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISDisjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISFullTextConjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISInFolderExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISInTreeExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISJunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay com.liferay.document.library.repository.cmis.api 3.0.7
CMISNotExpression

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay com.liferay.document.library.repository.cmis.api 3.0.7

CMISParameterValueUtil

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay com.liferay.document.library.repository.cmis.api 3.0.7

CMISRepositoryHandler

Old: com.liferay.portal.kernel.repository.cmis New: com.liferay.document.library.repository.cmis

com.liferay com.liferay.document.library.repository.cmis.api 3.0.7

CMISRepositoryUtil

Old: com.liferay.portal.kernel.repository.cmis New: com.liferay.document.library.repository.cmis.internal

com.liferay com.liferay.document.library.repository.cmis.impl 4.0.8

CMISSearchQueryBuilder

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay com.liferay.document.library.repository.cmis.api 3.0.7

CMISSimpleExpression

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay com.liferay.document.library.repository.cmis.api 3.0.7

CMISSimpleExpressionOperator

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay com.liferay.document.library.repository.cmis.api 3.0.7

CharPool

Old: com.liferay.portal.kernel.util New: com.liferay.petra.string

com.liferay com.liferay.petra.string 3.0.1

CharsetDecoderUtil

Old: com.liferay.portal.kernel.nio.charset New: com.liferay.petra.nio

com.liferay com.liferay.petra.nio 3.0.1

CharsetEncoderUtil

Old: com.liferay.portal.kernel.nio.charset New: com.liferay.petra.nio

com.liferay com.liferay.petra.nio 3.0.1

ClassLoaderPool

Old: com.liferay.portal.kernel.util New: com.liferay.petra.lang

com.liferay com.liferay.petra.lang 3.0.1

ClassPathUtil

Old: com.liferay.portal.kernel.process New: com.liferay.petra.process

com.liferay com.liferay.petra.process 3.0.4

ClassResolverUtil

Old: com.liferay.portal.kernel.util New: com.liferay.petra.lang

com.liferay com.liferay.petra.lang 3.0.1

CollatedSpellCheckHitsProcessor

Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.hits

com.liferay com.liferay.portal.search 6.0.14

CompoundSessionIdServletRequest

Old: com.liferay.portal.kernel.servlet.filters.compoundsessionid New: com.liferay.portal.compound.session.id.internal

com.liferay com.liferay.portal.compound.session.id 4.0.5

Condition

Old: com.liferay.portlet.dynamicdatamapping.storage.query New: com.liferay.adaptive.media.image.media.query

com.liferay com.liferay.adaptive.media.image.api 3.0.3

ConsumerOutputProcessor

Old: com.liferay.portal.kernel.process New: com.liferay.petra.process

com.liferay com.liferay.petra.process 3.0.4

ContactConverterKeys

Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap

com.liferay com.liferay.portal.security.ldap.api 2.0.8

ContentException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

ContentNameException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

ContextClassloaderReportDesignRetriever

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

CountStatistics

Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics

com.liferay com.liferay.portal.monitoring 7.0.7

DDL

Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.util

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLExporter

Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.exporter

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLExporterFactory

Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.exporter

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecord

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordConstants

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordFinder

Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordLocalService
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordLocalServiceUtil
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordModel
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordPersistence
Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordService
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordServiceUtil
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordServiceWrapper
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordSet
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordSetConstants
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordSetFinder
Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordSetLocalService
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordSetLocalServiceUtil
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordSetLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordSetModel
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
DDLRecordSetPersistence

Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordSetService

Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordSetServiceUtil

Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordSetServiceWrapper

Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordSetSoap

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordSetUtil

Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordSetWrapper

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordSoap

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordUtil

Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordVersion

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordVersionModel

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordVersionPersistence

Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordVersionSoap

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordVersionUtil

Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordVersionVersionComparator

Old: com.liferay.portlet.dynamicdatalists.util.comparator New: com.liferay.dynamic.data.lists.util.comparator

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordVersionWrapper

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDLRecordWrapper

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

DDM

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContent

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContentLocalService

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContentLocalServiceUtil

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContentLocalServiceWrapper

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContentModel

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContentPersistence

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContentSoap

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContentUtil

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMContentWrapper

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMDisplay

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMDisplayRegistry

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMIndexer

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMIndexerUtil

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.asset.list.internal.dynamic.data.mapping.util

com.liferay com.liferay.asset.list.service 1.0.11

DDMStorageLink

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStorageLinkLocalService

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStorageLinkLocalServiceUtil

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStorageLinkLocalServiceWrapper

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStorageLinkModel

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStorageLinkPersistence

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStorageLinkSoap

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStorageLinkUtil

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStorageLinkWrapper

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureConstants

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureFinder

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLinkLocalService

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLinkLocalServiceUtil

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLinkLocalServiceWrapper

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLinkModel

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLinkPersistence

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLinkSoap

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLinkUtil

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLinkWrapper

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMStructureLocalService

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureLocalServiceUtil
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureModel
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructurePersistence
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureService
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureServiceUtil
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureServiceWrapper
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureSoap
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureUtil
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMStructureWrapper
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMTemplateConstants
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
DDMTemplateFinder
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateHelper

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateLocalService

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateLocalServiceUtil

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateLocalServiceWrapper

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateModel

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplatePersistence

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateService

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateServiceUtil

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateServiceWrapper

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateSoap

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateUtil

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMTemplateWrapper

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMUtil

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DDMXML

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

DLContent

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentDataBlobModel

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentLocalService

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.content.service

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentLocalServiceUtil

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.content.service

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentLocalServiceWrapper

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.content.service

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentModel

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentPersistence

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.content.service.persistence

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentSoap

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentUtil

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.content.service.persistence

com.liferay com.liferay.document.library.content.api 2.0.3

DLContentVersionComparator

Old: com.liferay.portlet.documentlibrary.util.comparator New: com.liferay.document.library.content.service.util.comparator

com.liferay com.liferay.document.library.content.service 2.0.3

DLContentWrapper

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay com.liferay.document.library.content.api 2.0.3

DLFileRank

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.file.rank.model

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankFinder

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.file.rank.service.persistence

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankLocalService

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.file.rank.service

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankLocalServiceUtil

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.file.rank.service

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankLocalServiceWrapper

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.file.rank.service

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankModel

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.file.rank.model

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankPersistence

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.file.rank.service.persistence

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankSoap

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.file.rank.model

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankUtil

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.file.rank.service.persistence

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLFileRankWrapper

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.file.rank.model

com.liferay com.liferay.document.library.file.rank.api 2.0.3

DLSyncConstants

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.constants

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEvent

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.model

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEventLocalService

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.sync.service

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEventLocalServiceUtil

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.sync.service

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEventLocalServiceWrapper

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.sync.service

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEventModel

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.model

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEventPersistence

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.sync.service.persistence

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEventSoap

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.model

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEventUtil

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.sync.service.persistence

com.liferay com.liferay.document.library.sync.api 2.0.3

DLSyncEventWrapper

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.model

com.liferay com.liferay.document.library.sync.api 2.0.3

Database

Old: com.liferay.portal.kernel.util New: com.liferay.portal.tools.db.upgrade.client

com.liferay com.liferay.portal.tools.db.upgrade.client 3.0.0

DefaultAttributesTransformer

Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.internal

com.liferay com.liferay.portal.security.ldap.impl 2.0.5

DefaultMessageBus

Old: com.liferay.portal.kernel.messaging New: com.liferay.portal.messaging.internal

com.liferay com.liferay.portal.messaging 6.0.5

DefaultSingleDestinationMessageSender

Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender

com.liferay com.liferay.portal.messaging 6.0.5

DefaultSingleDestinationSynchronousMessageSender

Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender

com.liferay com.liferay.portal.messaging 6.0.5

DefaultSynchronousMessageSender

Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender
com.liferay com.liferay.portal.messaging 6.0.5
DeleteFileFinalizeAction
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay com.liferay.petra.memory 3.0.1
DestinationStatisticsManager
Old: com.liferay.portal.kernel.messaging.jmx New: com.liferay.portal.messaging.internal.jmx
com.liferay com.liferay.portal.messaging 6.0.5
DestinationStatisticsManagerMBean
Old: com.liferay.portal.kernel.messaging.jmx New: com.liferay.portal.messaging.internal.jmx
com.liferay com.liferay.portal.messaging 6.0.5
DirectSynchronousMessageSender
Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender
com.liferay com.liferay.portal.messaging 6.0.5
Dummy
Old: com.liferay.portal.model New: com.liferay.exportimport.test.util.model
com.liferay com.liferay.exportimport.test.util 2.0.6
DummyContext
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.dummy
com.liferay com.liferay.portal.security.ldap.api 2.0.8
DummyDirContext
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.dummy
com.liferay com.liferay.portal.security.ldap.api 2.0.8
DummyFinalizeAction
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay com.liferay.petra.memory 3.0.1
DuplicateArticleIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
DuplicateFeedIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
DuplicateLDAPServerNameException
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap
com.liferay com.liferay.portal.security.ldap.api 2.0.8
DuplicateNodeNameException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
DuplicatePageException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
DuplicateRuleGroupInstanceException
Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception
com.liferay com.liferay.mobile.device.rules.api 4.0.4
DuplicateVoteException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception

com.liferay com.liferay.polls.api 6.0.3
EntryDisplayDateComparator
Old: com.liferay.portlet.blogs.util.comparator New: com.liferay.blogs.util.comparator
com.liferay com.liferay.blogs.api 5.0.5
EntryModifiedDateComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay com.liferay.bookmarks.api 4.0.5
EntryNameComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay com.liferay.bookmarks.api 4.0.5
EntryPriorityComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay com.liferay.bookmarks.api 4.0.5
EntrySmallImageNameException
Old: com.liferay.portlet.blogs New: com.liferay.blogs.exception
com.liferay com.liferay.blogs.api 5.0.5
EntryURLComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay com.liferay.bookmarks.api 4.0.5
EntryVisitsComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay com.liferay.bookmarks.api 4.0.5
EqualityWeakReference
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay com.liferay.petra.memory 3.0.1
Fact
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay com.liferay.portal.rules.engine.api 4.0.4
FeedContentFieldException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
FeedIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
FeedNameException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
FeedTargetLayoutFriendlyUrlException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
FeedTargetPortletIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1

FieldConstants

Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

FieldRenderer

Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

FieldRendererFactory

Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

Fields

Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

FileRankCreateDateComparator

Old: com.liferay.portlet.documentlibrary.util.comparator New: com.liferay.document.library.file.rank.util.comparator

com.liferay com.liferay.document.library.file.rank.service 2.0.6

FinalizeAction

Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory

com.liferay com.liferay.petra.memory 3.0.1

FinalizeManager

Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory

com.liferay com.liferay.petra.memory 3.0.1

FlagsEntryService

Old: com.liferay.portlet.flags.service New: com.liferay.flags.service

com.liferay com.liferay.flags.api 4.0.6

FlagsEntryServiceUtil

Old: com.liferay.portlet.flags.service New: com.liferay.flags.service

com.liferay com.liferay.flags.api 4.0.6

FlagsEntryServiceWrapper

Old: com.liferay.portlet.flags.service New: com.liferay.flags.service

com.liferay com.liferay.flags.api 4.0.6

FlagsRequest

Old: com.liferay.portlet.flags.messaging New: com.liferay.flags.internal.messaging

com.liferay com.liferay.flags.service 4.0.2

GroupConverterKeys

Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap

com.liferay com.liferay.portal.security.ldap.api 2.0.8

ImportFilesException

Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception

com.liferay com.liferay.wiki.api 4.0.7

JournalArticle

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model

com.liferay com.liferay.journal.api 4.2.1

JournalArticleConstants

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalArticleDisplay
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalArticleFinder
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalArticleLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalArticleLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalArticleLocalServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalArticleModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalArticlePersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalArticleResource
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalArticleResourceLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalArticleResourceLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalArticleResourceLocalServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalArticleResourceModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalArticleResourcePersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalArticleResourceSoap
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalArticleResourceUtil

Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence

com.liferay com.liferay.journal.api 4.2.1

JournalArticleResourceWrapper

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model

com.liferay com.liferay.journal.api 4.2.1

JournalArticleService

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service

com.liferay com.liferay.journal.api 4.2.1

JournalArticleServiceUtil

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service

com.liferay com.liferay.journal.api 4.2.1

JournalArticleServiceWrapper

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service

com.liferay com.liferay.journal.api 4.2.1

JournalArticleSoap

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model

com.liferay com.liferay.journal.api 4.2.1

JournalArticleUtil

Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence

com.liferay com.liferay.journal.api 4.2.1

JournalArticleWrapper

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model

com.liferay com.liferay.journal.api 4.2.1

JournalContent

Old: com.liferay.portlet.journalcontent.util New: com.liferay.journal.util

com.liferay com.liferay.journal.api 4.2.1

JournalContentSearch

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model

com.liferay com.liferay.journal.api 4.2.1

JournalContentSearchLocalService

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service

com.liferay com.liferay.journal.api 4.2.1

JournalContentSearchLocalServiceUtil

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service

com.liferay com.liferay.journal.api 4.2.1

JournalContentSearchLocalServiceWrapper

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service

com.liferay com.liferay.journal.api 4.2.1

JournalContentSearchModel

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model

com.liferay com.liferay.journal.api 4.2.1

JournalContentSearchPersistence

Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence

com.liferay com.liferay.journal.api 4.2.1

JournalContentSearchSoap

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalContentSearchUtil
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalContentSearchWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalConverter
Old: com.liferay.portlet.journal.util New: com.liferay.journal.util
com.liferay com.liferay.journal.api 4.2.1
JournalFeed
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalFeedConstants
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalFeedFinder
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalFeedLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFeedLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFeedLocalServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFeedModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalFeedPersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalFeedService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFeedServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFeedServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFeedSoap

Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalFeedUtil
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalFeedWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalFolder
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalFolderFinder
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalFolderLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFolderLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFolderLocalServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFolderModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalFolderPersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1
JournalFolderService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFolderServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFolderServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay com.liferay.journal.api 4.2.1
JournalFolderSoap
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalFolderUtil
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay com.liferay.journal.api 4.2.1

JournalFolderWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalSearchConstants
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
JournalStructureConstants
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay com.liferay.journal.api 4.2.1
LDAPFilterException
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.validator
com.liferay com.liferay.portal.security.ldap.api 2.0.8
LDAPGroup
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay com.liferay.portal.security.ldap.api 2.0.8
LDAPServerNameException
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap
com.liferay com.liferay.portal.security.ldap.api 2.0.8
LDAPToPortalConverter
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay com.liferay.portal.security.ldap.api 2.0.8
LDAPUser
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay com.liferay.portal.security.ldap.api 2.0.8
LDAPUtil
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.util
com.liferay com.liferay.portal.security.ldap.api 2.0.8
LockLocalService
Old: com.liferay.portal.service New: com.liferay.portal.lock.service
com.liferay com.liferay.portal.lock.api 4.1.1
LockLocalServiceUtil
Old: com.liferay.portal.service New: com.liferay.portal.lock.service
com.liferay com.liferay.portal.lock.api 4.1.1
LockLocalServiceWrapper
Old: com.liferay.portal.service New: com.liferay.portal.lock.service
com.liferay com.liferay.portal.lock.api 4.1.1
LockModel
Old: com.liferay.portal.model New: com.liferay.portal.lock.model
com.liferay com.liferay.portal.lock.api 4.1.1
LockPersistence
Old: com.liferay.portal.service.persistence New: com.liferay.portal.lock.service.persistence
com.liferay com.liferay.portal.lock.api 4.1.1
LockSoap
Old: com.liferay.portal.model New: com.liferay.portal.lock.model
com.liferay com.liferay.portal.lock.api 4.1.1
LockUtil
Old: com.liferay.portal.service.persistence New: com.liferay.portal.lock.service.persistence
com.liferay com.liferay.portal.lock.api 4.1.1

LockWrapper
Old: com.liferay.portal.model New: com.liferay.portal.lock.model
com.liferay com.liferay.portal.lock.api 4.1.1
LockedThreadException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay com.liferay.message.boards.api 5.1.4
LoggingOutputProcessor
Old: com.liferay.portal.kernel.process New: com.liferay.petra.process
com.liferay com.liferay.petra.process 3.0.4
MBBan
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBBanLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBBanLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBBanLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBBanModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBBanPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBBanService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBBanServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBBanServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBBanSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBBanUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBBanWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBCategory

Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryConstants
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.constants
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryDisplay
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.web.internal.display
com.liferay com.liferay.message.boards.web 3.0.17
MBCategoryFinder
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBCategorySoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBCategoryWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4

MBDiscussion
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBDiscussionLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBDiscussionLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBDiscussionLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBDiscussionModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBDiscussionPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBDiscussionSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBDiscussionUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBDiscussionWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMailingList
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMailingListLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBMailingListLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBMailingListLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBMailingListModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMailingListPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4

MBMailingListSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMailingListUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBMailingListWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMessage
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageConstants
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.constants
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageDisplay
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageFinder
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMessagePersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4

MBMessageSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBMessageWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUser
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUserLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUserLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUserLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUserModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUserPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUserSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUserUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBStatsUserWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBThread
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadConstants
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.constants
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFinder
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence

com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlag
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlagLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlagLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlagLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlagModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlagPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlagSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlagUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadFlagWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service

com.liferay com.liferay.message.boards.api 5.1.4
MBThreadServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay com.liferay.message.boards.api 5.1.4
MBThreadWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBTreeWalker
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay com.liferay.message.boards.api 5.1.4
MBeanRegistry
Old: com.liferay.portal.kernel.jmx New: com.liferay.portal.jmx
com.liferay com.liferay.portal.jmx.api 3.0.1
MDRAction
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionLocalService
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionLocalServiceUtil
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionLocalServiceWrapper
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionModel
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionPersistence
Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionService
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionServiceUtil
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionServiceWrapper
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionSoap
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionUtil
Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRActionWrapper
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRPermission
Old: com.liferay.portlet.mobiledevicerules.service.permission New: com.liferay.mobile.device.rules.web.internal.security.permission.resource
com.liferay com.liferay.mobile.device.rules.web 3.0.6
MDRRule
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRRuleGroup
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRRuleGroupFinder
Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRRuleGroupInstanceLocalService
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRRuleGroupInstanceLocalServiceUtil
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRRuleGroupInstanceLocalServiceWrapper
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay com.liferay.mobile.device.rules.api 4.0.4
MDRRuleGroupInstanceModel
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupInstancePermission

Old: com.liferay.portlet.mobiledevicerules.service.permission New: com.liferay.mobile.device.rules.web.internal.security.permission.resource

com.liferay com.liferay.mobile.device.rules.web 3.0.6

MDRRuleGroupInstancePersistence

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupInstanceService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupInstanceServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupInstanceServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupInstanceSoap

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupInstanceUtil

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupInstanceWrapper

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupLocalService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupLocalServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupLocalServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupModel

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupPermission

Old: com.liferay.portlet.mobiledevicerules.service.permission New: com.liferay.mobile.device.rules.web.internal.security.permission.resource

com.liferay com.liferay.mobile.device.rules.web 3.0.6

MDRRuleGroupPersistence

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupSoap

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupUtil

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleGroupWrapper

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleLocalService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleLocalServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleLocalServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleModel

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRulePersistence

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleSoap

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleUtil

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MDRRuleWrapper

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay com.liferay.mobile.device.rules.api 4.0.4

MailingListEmailAddressException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

MailingListInServerNameException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

MailingListInUserNameException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

MailingListOutEmailAddressException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

MailingListOutServerNameException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

MailingListOutUserNameException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

MemoryReportDesignRetriever

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

MessageBodyException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

MessageBusManager

Old: com.liferay.portal.kernel.messaging.jmx New: com.liferay.portal.messaging.internal.jmx

com.liferay com.liferay.portal.messaging 6.0.5

MessageBusManagerMBean
Old: com.liferay.portal.kernel.messaging.jmx New: com.liferay.portal.messaging.internal.jmx
com.liferay com.liferay.portal.messaging 6.0.5
MessageCreateDateComparator
Old: com.liferay.portlet.messageboards.util.comparator New: com.liferay.message.boards.util.comparator
com.liferay com.liferay.message.boards.api 5.1.4
MessageSubjectException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay com.liferay.message.boards.api 5.1.4
MessageThreadComparator
Old: com.liferay.portlet.messageboards.util.comparator New: com.liferay.message.boards.util.comparator
com.liferay com.liferay.message.boards.api 5.1.4
Modifications
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay com.liferay.portal.security.ldap.api 2.0.8
NoSuchArticleException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
NoSuchArticleImageException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
NoSuchArticleResourceException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
NoSuchBanException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay com.liferay.message.boards.api 5.1.4
NoSuchChoiceException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay com.liferay.polls.api 6.0.3
NoSuchContentException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
NoSuchContentSearchException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
NoSuchDiscussionException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay com.liferay.message.boards.api 5.1.4
NoSuchFeedException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay com.liferay.journal.api 4.2.1
NoSuchFileRankException
Old: com.liferay.portlet.documentlibrary New: com.liferay.document.library.file.rank.exception

com.liferay com.liferay.document.library.file.rank.api 2.0.3
NoSuchMailingListException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay com.liferay.message.boards.api 5.1.4
NoSuchNodeException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
NoSuchPageException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
NoSuchPageResourceException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
NoSuchQuestionException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay com.liferay.polls.api 6.0.3
NoSuchRecordException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
NoSuchRecordSetException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
NoSuchRecordVersionException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay com.liferay.dynamic.data.lists.api 4.0.5
NoSuchRuleException
Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception
com.liferay com.liferay.mobile.device.rules.api 4.0.4
NoSuchRuleGroupException
Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception
com.liferay com.liferay.mobile.device.rules.api 4.0.4
NoSuchRuleGroupInstanceException
Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception
com.liferay com.liferay.mobile.device.rules.api 4.0.4
NoSuchStatsUserException
Old: com.liferay.portlet.blogs New: com.liferay.blogs.exception
com.liferay com.liferay.blogs.api 5.0.5
NoSuchStorageLinkException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
NoSuchStructureLinkException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
NoSuchTemplateException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
NoSuchThreadException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay com.liferay.message.boards.api 5.1.4
NoSuchThreadFlagException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay com.liferay.message.boards.api 5.1.4
NoSuchVoteException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay com.liferay.polls.api 6.0.3
NodeNameException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
OutputProcessor
Old: com.liferay.portal.kernel.process New: com.liferay.petra.process
com.liferay com.liferay.petra.process 3.0.4
PageContentException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
PageCreateDateComparator
Old: com.liferay.portlet.wiki.util.comparator New: com.liferay.wiki.util.comparator
com.liferay com.liferay.wiki.api 4.0.7
PageTitleComparator
Old: com.liferay.portlet.wiki.util.comparator New: com.liferay.wiki.util.comparator
com.liferay com.liferay.wiki.api 4.0.7
PageTitleException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
PageVersionComparator
Old: com.liferay.portlet.wiki.util.comparator New: com.liferay.wiki.util.comparator
com.liferay com.liferay.wiki.api 4.0.7
PageVersionException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
PollsChoice
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceLocalService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceLocalServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceLocalServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceModel
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model

com.liferay com.liferay.polls.api 6.0.3
PollsChoicePersistence
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceSoap
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceUtil
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay com.liferay.polls.api 6.0.3
PollsChoiceWrapper
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsQuestion
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionLocalService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionLocalServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionLocalServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionModel
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionPersistence
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service

com.liferay com.liferay.polls.api 6.0.3
PollsQuestionSoap
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionUtil
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay com.liferay.polls.api 6.0.3
PollsQuestionWrapper
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsVote
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsVoteLocalService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsVoteLocalServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsVoteLocalServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsVoteModel
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsVotePersistence
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay com.liferay.polls.api 6.0.3
PollsVoteService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsVoteServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsVoteServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay com.liferay.polls.api 6.0.3
PollsVoteSoap
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PollsVoteUtil
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay com.liferay.polls.api 6.0.3
PollsVoteWrapper
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay com.liferay.polls.api 6.0.3
PoolAction
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory

com.liferay com.liferay.petra.memory 3.0.1

PortalCacheClusterChannel

Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal.cluster.link

com.liferay com.liferay.portal.cache.multiple 3.0.6

PortalCacheClusterChannelFactory

Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal.cluster.link

com.liferay com.liferay.portal.cache.multiple 3.0.6

PortalCacheClusterChannelSelector

Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal.cluster.link

com.liferay com.liferay.portal.cache.multiple 3.0.6

PortalCacheClusterEvent

Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal

com.liferay com.liferay.portal.cache.multiple 3.0.6

PortalCacheClusterEventCoalesceComparator

Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal

com.liferay com.liferay.portal.cache.multiple 3.0.6

PortalCacheClusterEventType

Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal

com.liferay com.liferay.portal.cache.multiple 3.0.6

PortalCacheClusterException

Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal

com.liferay com.liferay.portal.cache.multiple 3.0.6

PortalCacheClusterLink

Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal.cluster.link

com.liferay com.liferay.portal.cache.multiple 3.0.6

PortalExecutorFactory

Old: com.liferay.portal.kernel.executor New: com.liferay.portal.executor.internal

com.liferay com.liferay.portal.executor 4.0.2

PortalToLDAPConverter

Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport

com.liferay com.liferay.portal.security.ldap.api 2.0.8

PortletDisplayTemplate

Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.portlet.display.template

com.liferay com.liferay.portlet.display.template.api 2.0.2

PortletDisplayTemplateConstants

Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.portlet.display.template.constants

com.liferay com.liferay.portlet.display.template.api 2.0.2

PortletDisplayTemplateUtil

Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.roles.admin.web.internal.util

com.liferay com.liferay.roles.admin.web 3.0.6

PortletDisplayTemplateUtil

Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.roles.admin.web.internal.util

com.liferay com.liferay.roles.admin.web 3.0.6

PortletDisplayTemplateUtil

Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.roles.admin.web.internal.util

com.liferay com.liferay.roles.admin.web 3.0.6

ProcessUtil

Old: com.liferay.portal.kernel.process New: com.liferay.petra.process

com.liferay com.liferay.petra.process 3.0.4

QueryIndexingHitsProcessor

Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.hits

com.liferay com.liferay.portal.search 6.0.14

QuerySuggestionHitsProcessor

Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.hits

com.liferay com.liferay.portal.search 6.0.14

QueryType

Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine

com.liferay com.liferay.portal.rules.engine.api 4.0.4

QuestionChoiceException

Old: com.liferay.portlet.polls New: com.liferay.polls.exception

com.liferay com.liferay.polls.api 6.0.3

QuestionDescriptionException

Old: com.liferay.portlet.polls New: com.liferay.polls.exception

com.liferay com.liferay.polls.api 6.0.3

QuestionExpirationDateException

Old: com.liferay.portlet.polls New: com.liferay.polls.exception

com.liferay com.liferay.polls.api 6.0.3

QuestionExpiredException

Old: com.liferay.portlet.polls New: com.liferay.polls.exception

com.liferay com.liferay.polls.api 6.0.3

QuestionTitleException

Old: com.liferay.portlet.polls New: com.liferay.polls.exception

com.liferay com.liferay.polls.api 6.0.3

RecordSetDDMStructureIdException

Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

RecordSetDuplicateRecordSetKeyException

Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

RecordSetNameException

Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception

com.liferay com.liferay.dynamic.data.lists.api 4.0.5

RegistryAwareMBeanServer

Old: com.liferay.portal.kernel.jmx New: com.liferay.portal.jmx.internal

com.liferay com.liferay.portal.jmx 6.0.2

ReportCompilerRequestMessageListener

Old: com.liferay.portal.kernel.bi.reporting.messaging New: com.liferay.portal.reports.engine.messaging

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportDataSourceType

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportDesignRetriever

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportEngine

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportExportException

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportFormat

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportFormatExporter

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportFormatExporterRegistry

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportGenerationException

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportRequest

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportRequestContext

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportRequestMessageListener

Old: com.liferay.portal.kernel.bi.reporting.messaging New: com.liferay.portal.reports.engine.messaging

com.liferay com.liferay.portal.reports.engine.api 5.0.1

ReportResultContainer

Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay com.liferay.portal.reports.engine.api 5.0.1

RequestStatistics

Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics

com.liferay com.liferay.portal.monitoring 7.0.7

RequiredMessageException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

RequiredNodeException

Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
RequiredTemplateException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
RequiredTemplateException
Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
RuleGroupInstancePriorityComparator
Old: com.liferay.portlet.mobiledevicerules.util New: com.liferay.mobile.device.rules.util.comparator
com.liferay com.liferay.mobile.device.rules.api 4.0.4
RuleGroupProcessor
Old: com.liferay.portal.kernel.mobile.device.rulegroup New: com.liferay.mobile.device.rules.rule
com.liferay com.liferay.mobile.device.rules.api 4.0.4
RuleGroupProcessorUtil
Old: com.liferay.portal.kernel.mobile.device.rulegroup New: com.liferay.mobile.device.rules.rule
com.liferay com.liferay.mobile.device.rules.api 4.0.4
RuleHandler
Old: com.liferay.portal.kernel.mobile.device.rulegroup.rule New: com.liferay.mobile.device.rules.rule
com.liferay com.liferay.mobile.device.rules.api 4.0.4
RulesEngine
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay com.liferay.portal.rules.engine.api 4.0.4
RulesEngineException
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay com.liferay.portal.rules.engine.api 4.0.4
RulesEngineUtil
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay com.liferay.portal.rules.engine.api 4.0.4
RulesLanguage
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay com.liferay.portal.rules.engine.api 4.0.4
RulesResourceRetriever
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay com.liferay.portal.rules.engine.api 4.0.4
SearchUtil
Old: com.liferay.portal.kernel.search.util New: com.liferay.portal.vulcan.util
com.liferay com.liferay.portal.vulcan.api 3.2.2
SearchUtil
Old: com.liferay.portal.kernel.search.util New: com.liferay.portal.vulcan.util
com.liferay com.liferay.portal.vulcan.api 3.2.2
ServletContextReportDesignRetriever

Old: com.liferay.portal.kernel.bi.reporting.servlet New: com.liferay.portal.reports.engine.servlet

com.liferay com.liferay.portal.reports.engine.api 5.0.1

SoftReferencePool

Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory

com.liferay com.liferay.petra.memory 3.0.1

SortFactoryImpl

Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal

com.liferay com.liferay.portal.search 6.0.14

SplitThreadException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay com.liferay.message.boards.api 5.1.4

Statistics

Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics

com.liferay com.liferay.portal.monitoring 7.0.7

StatsUserLastPostDateComparator

Old: com.liferay.portlet.blogs.util.comparator New: com.liferay.blogs.util.comparator

com.liferay com.liferay.blogs.api 5.0.5

StorageAdapter

Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

StorageEngine

Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

StorageException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

StorageFieldNameException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

StringQueryImpl

Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.query

com.liferay com.liferay.portal.search 6.0.14

StructureDuplicateStructureKeyException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

StructureFieldException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

StructureIdComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

StructureModifiedDateComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

StructureStructureKeyComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

SummaryStatistics

Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics

com.liferay com.liferay.portal.monitoring 7.0.7

SynchronousMessageListener

Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender

com.liferay com.liferay.portal.messaging 6.0.5

TemplateDuplicateTemplateKeyException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

TemplateIdComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

TemplateModifiedDateComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

TemplateNameException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

TemplateNameException

Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

TemplateScriptException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

TemplateSmallImageNameException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1

TemplateSmallImageNameException

Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception

com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
TemplateSmallImageSizeException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
TemplateSmallImageSizeException
Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception
com.liferay com.liferay.dynamic.data.mapping.api 5.2.1
ThreadLastPostDateComparator
Old: com.liferay.portlet.messageboards.util.comparator New: com.liferay.message.boards.util.comparator
com.liferay com.liferay.message.boards.api 5.1.4
UniformPortalCacheClusterChannelSelector
Old: com.liferay.portal.kernel.cache.cluster New: com.liferay.portal.cache.multiple.internal.cluster.link
com.liferay com.liferay.portal.cache.multiple 3.0.6
UnknownRuleHandlerException
Old: com.liferay.portal.kernel.mobile.device.rulegroup.rule New: com.liferay.mobile.device.rules.rule
com.liferay com.liferay.mobile.device.rules.api 4.0.4
UserConverterKeys
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap
com.liferay com.liferay.portal.security.ldap.api 2.0.8
WikiFormatException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay com.liferay.wiki.api 4.0.7
WikiNode
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeLocalService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeLocalServiceUtil
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeLocalServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeModel
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiNodePersistence
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeServiceUtil

Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeSoap
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeUtil
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay com.liferay.wiki.api 4.0.7
WikiNodeWrapper
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPage
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPageConstants
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPageDisplay
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPageFinder
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay com.liferay.wiki.api 4.0.7
WikiPageLocalService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageLocalServiceUtil
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageLocalServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageModel
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPagePersistence
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResource
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResourceLocalService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResourceLocalServiceUtil

Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResourceLocalServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResourceModel
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResourcePersistence
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResourceSoap
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResourceUtil
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay com.liferay.wiki.api 4.0.7
WikiPageResourceWrapper
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPageService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageServiceUtil
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay com.liferay.wiki.api 4.0.7
WikiPageSoap
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7
WikiPageUtil
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay com.liferay.wiki.api 4.0.7
WikiPageWrapper
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay com.liferay.wiki.api 4.0.7

Chapter 642

Export/Import and Staging

Export/Import and Staging are frameworks that help you manage your content publication. This section provides reference documentation complementing the Content Publication Management section.

CHAPTER 643

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.

CHAPTER 644

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.

644.1 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'll dissect the anatomy structure next.

644.2 LAR Manifest

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 folders.

644.3 LAR Folders

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.

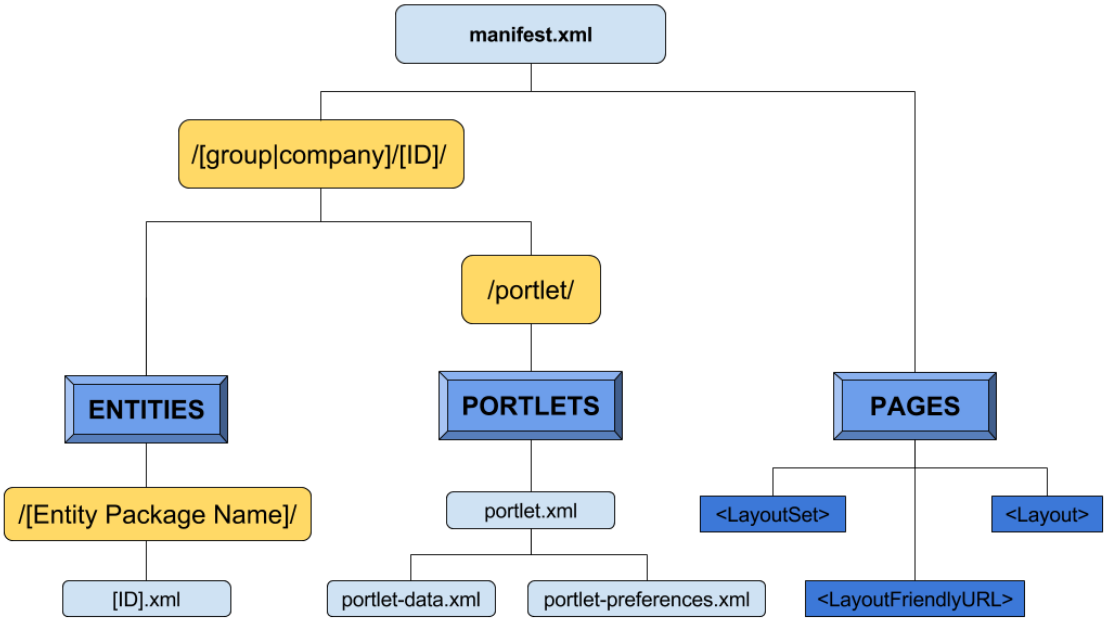

Figure 644.1: Entities, Portlets, and Pages are defined in a LAR in different places.

CHAPTER 645

FRONT-END REFERENCE

This section contains resources that you might find useful for Front-End development.

The topics below are covered in this section:

- Liferay DXP FreeMarker Macros
- FreeMarker Taglib Macros
- Front-end Taglibs
- Liferay npm Bundler
- Liferay JS APIs
- Setting up Your npm Environment
- Sitemap.json Page Configuration Options
- CKEditor Plugin Reference Guide
- Fully Qualified Portlet IDs List

CHAPTER 646

LIFERAY DXP FREEMARKER MACROS

Liferay DXP defines several macros in `FTL_Liferay.ftl` template that you can use in your theme templates to include theme resources, standard portlets, and more. Liferay DXP also exposes its taglibs as FreeMarker macros. See each taglib's documentation for more information on using the taglib in your FreeMarker templates. This reference guide lists the available FreeMarker macros that Liferay DXP offers.

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`
language	key	Prints the specified language key in the current locale	`<@liferay.language key="last-modified" />`

Macro	Parameters	Description	Example
language_format	argumentskey	Formats the given language key with the specified arguments. For example, passing `go-to-x` as the key and `Mars` as the arguments prints *Go to Mars*.	`<@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_preferencesinstance_id	Adds the Navigation Menu portlet with optional preferences and instance ID.	`<@liferay.navigation_menu />`
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 />`

A few reference examples are shown below.

646.1 Reference Examples

The example below includes a language key with the `language` macro directive along with its language `key` parameter:

```
<@liferay.language key="powered-by" />
```

This example includes the Search portlet with its Portlet Decorator portlet preference set 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.

CHAPTER 647

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:

- AUI: create common UI components such as forms, buttons, and more.
- Chart: visualize data. Create bar charts, line charts, scatter charts, spline charts, and much more.
- Clay: create Clay components, such as alerts, buttons, drop-down menus, form elements, and more for your apps.
- Frontend: create UI components commonly used throughout Portal's apps, such as add menus, cards, management bars, and more.
- Liferay UI: create common UI components such as icons, tabs, and more.
- Liferay Util: load additional resources, define parameters, buffer content, 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 articles 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, you'll learn how to use taglibs to build awesome user interfaces for your apps!

CHAPTER 648

LIFERAY THEME OBJECTS AVAILABLE IN JSPS

When you include the `<liferay-theme:defineObjects>` tag in your JSP, you gain access to several Liferay theme objects via variables. These objects are described in the table below:

Object	Description
`account`	The user's Account object. This object maps to the Account table in the Liferay database.
`colorScheme`	An object representing the current color scheme in the theme that is being rendered by the portal
`company`	The current Company object. This represents the portal instance on which the user is currently navigating.
`contact`	The user's Contact object. This object maps to the Contact table in the Liferay database.
`layout`	The page to which the user has currently navigated
`layoutTypePortlet`	This object can be used to programmatically add or remove portlets from a page.
`locale`	The current user's locale, as defined by Java
`permissionChecker`	An object that can determine—given a particular resource—whether the current user has a particular permission for that resource
`plid`	A portal layout ID. This is a unique identifier for any page that exists in the portal, across all portal instances.
`portletDisplay`	An object that gives the programmer access to many attributes of the current portlet, including the portlet name, the portlet mode, the ID of the column on the layout in which it resides, and more

Object	Description
realUser	When an administrator is impersonating a user, this variable tracks the administrator's User object.
scopeGroupId	By default, contains the groupId for the community or organization in which this portlet resides. If the scopeable attribute is set to true, this may contain a unique scope identifier for custom scopes, such as the page scope, if the portlet has been configured to use a custom scope.
theme	An object representing the current theme that is being rendered by the portal
themeDisplay	A runtime object that contains many useful items, such as the logged-in user, the layout, logo information, paths, and much more
timeZone	The current user's time zone, as defined by Java
user	The User object representing the current user

CHAPTER 649

LIFERAY PORTLET OBJECTS AVAILABLE IN JSPS

You may have noticed the `<liferay-portlet:defineObjects>` tag in your JSPs. Similar to the theme:defineObjects tag, when you include this tag in your JSP, you gain access to several variables that, in this case, return useful information about your portlet. Note that the JSR-286 specification defines four lifecycle methods for a portlet: processAction, processEvent, render, and serveResource. Some of the variables defined by the `<portlet:defineObjects/>` tag are only available to a JSP if the JSP was included during the appropriate phase of the portlet lifecycle. These objects are described in the table below:

Object	Description
`ActionRequest actionRequest`	Represents the request sent to the portlet to handle an action. `actionRequest` is only available to a JSP if the JSP was included during the action-processing phase.
`ActionResponse actionResponse`	Represents the portlet response to an action request. `actionResponse` is only available to a JSP if the JSP was included in the action-processing phase.
`EventRequest eventRequest`	Represents the request sent to the portlet to handle an event. `eventRequest` is only available to a JSP if the JSP was included during the event-processing phase.
`EventResponse eventResponse`	Represents the portlet response to an event request. `eventResponse` is only available to a JSP if the JSP was included in the event-processing phase.
`HeaderRequest headerRequest`	Represents the request sent to the portlet to handle its HTML header or HEAD section. `headerRequest` is only available to a JSP if the JSP was included during the header-processing phase.
`HeaderResponse headerResponse`	Represents the portlet response to a header request. `headerResponse` is only available to a JSP if the JSP was included in the header-processing phase.
`LiferayPortletRequest liferayPortletRequest`	Provides access to the `HttpServletRequest`, the `Portlet`, and the portlet name and lifecycle value. `liferayPortletRequest` is available in all portlet phases.

Object	Description
`LiferayPortletResponse` `liferayPortletResponse`	Includes the properties returned to the portal and provides a means to add or change properties. `liferayPortletResponse` is available in all portlet phases.
`RenderRequest renderRequest`	Represents the request sent to the portlet to render the portlet. `renderRequest` is only available to a JSP if the JSP was included during the render request phase.
`RenderResponse renderResponse`	Represents an object that assists the portlet in sending a response to the portal. `renderResponse` is only available to a JSP if the JSP was included during the render request phase.
`ResourceRequest resourceRequest`	Represents the request sent to the portlet for rendering resources. `resourceRequest` is only available to a JSP if the JSP was included during the resource-serving phase.
`ResourceResponse` `resourceResponse`	Represents an object that assists the portlet in rendering a resource. `resourceResponse` is only available to a JSP if the JSP was included in the resource-serving phase.
`PortletConfig portletConfig`	Represents the portlet's configuration including, the portlet's name, initialization parameters, resource bundle, and application context. `portletConfig` is always available to a portlet JSP, regardless of the request-processing phase in which it was included.
`PortletPreferences` `portletPreferences`	Provides access to a portlet's preferences. `portletPreferences` is always available to a portlet JSP, regardless of the request-processing phase in which it was included.
`Map<String, String[]>` `portletPreferencesValues`	Provides a Map equivalent to the `portletPreferences.getMap()` call or an empty Map if no portlet preferences exist.
`PortletSession portletSession`	Provides a way to identify a user across more than one request and to store transient information about a user. A `portletSession` is created for each user client. `portletSession` is always available to a portlet JSP, regardless of the request-processing phase in which it was included. `portletSession` is `null` if no session exists.
`Map<String, Object>` `portletSessionScope`	Provides a Map equivalent to the `PortletSession.getAtrributeMap()` call or an empty Map if no session attributes exist.

For more details, visit the Portlet 3.0 API Javadoc.

Chapter 650

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 covers how to create UI components with the Liferay UI taglibs. Each article contains code examples along with a screenshot of the resulting UI.

CHAPTER 651

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>
```

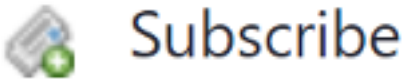

Figure 651.1: Use the image attribute to use a theme icon.

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:

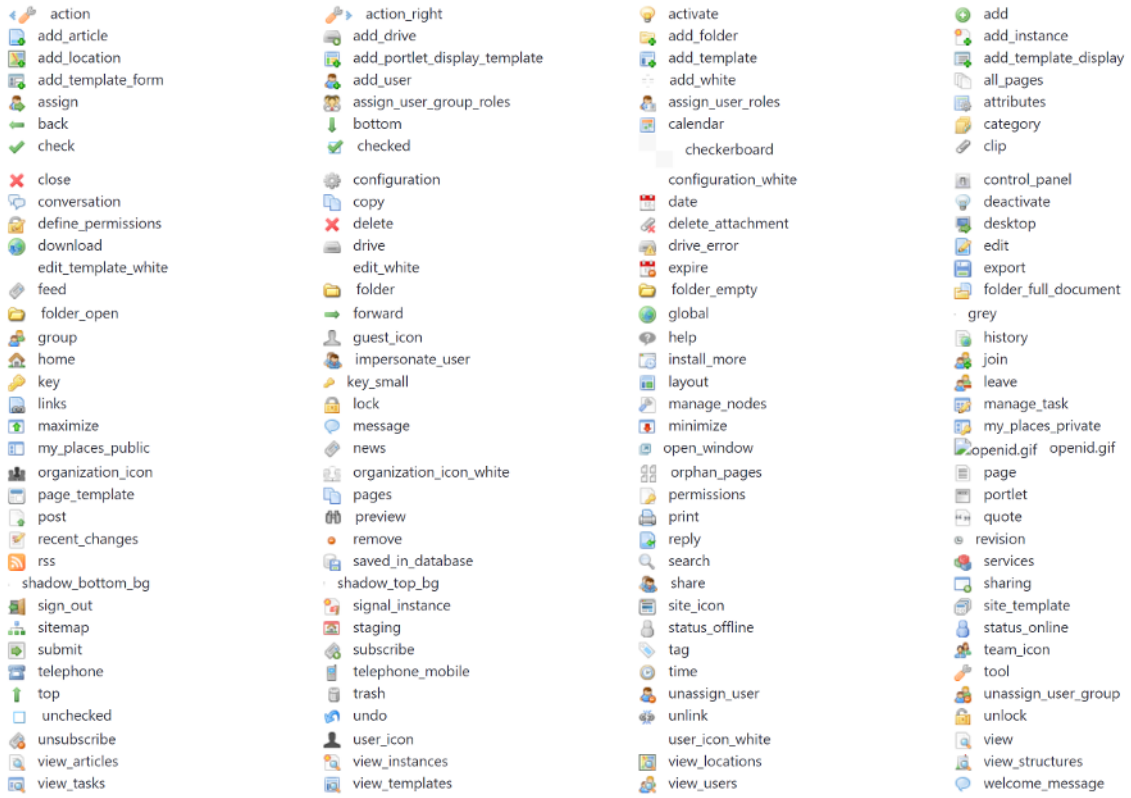

Figure 651.2: The Liferay UI taglib offers multiple icons for use in your app.

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.

Figure 651.3: Liferay UI icons can be configured based on language.

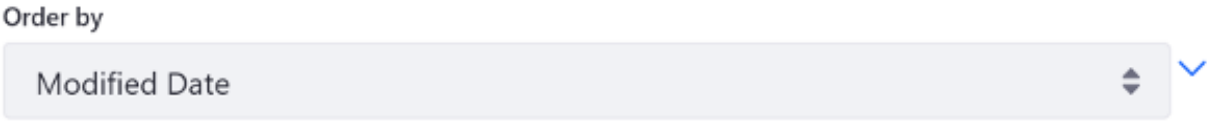

Figure 651.4: You can use the icon attribute to include Font Awesome icons in your app.

Figure 651.5: You can use Font Awesome icons in your app.

651.1 Related Topics

- Clay Icons
- Liferay UI Icon Lists
- Liferay UI Icon Menus

CHAPTER 652

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:

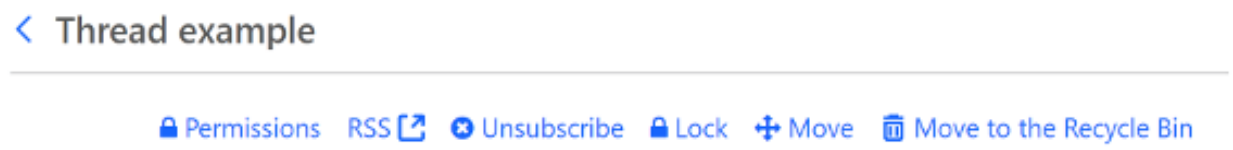

Figure 652.1: Icon lists display an app's actions at all times.

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.

652.1 Related Topics

- Clay Icons
- Liferay UI Icon Menus
- Liferay UI Icons

CHAPTER 653

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:

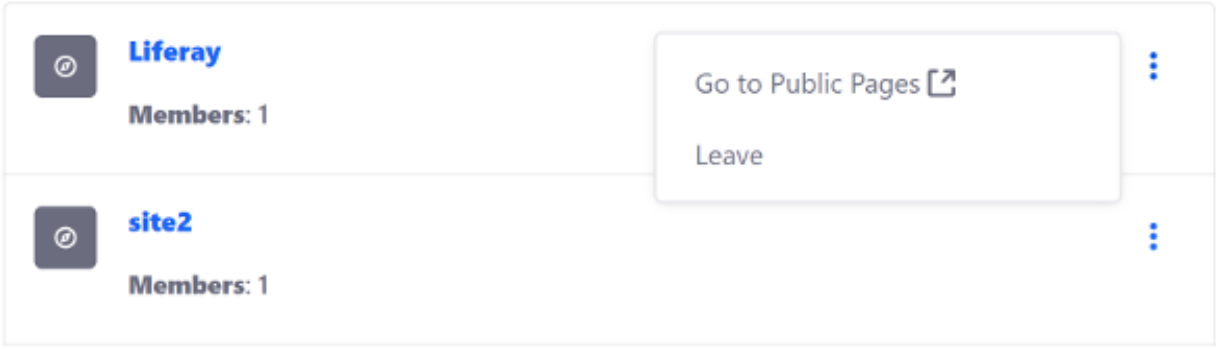

Figure 653.1: Setting up an icon menu is a piece of cake.

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.

653.1 Related Topics

- Clay Icons
- Liferay UI Icon Lists
- Liferay UI Icons

CHAPTER 654

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>
```

The example above uses some of the tab's available attributes. See the Tabs taglibdocs for the full list of attributes.

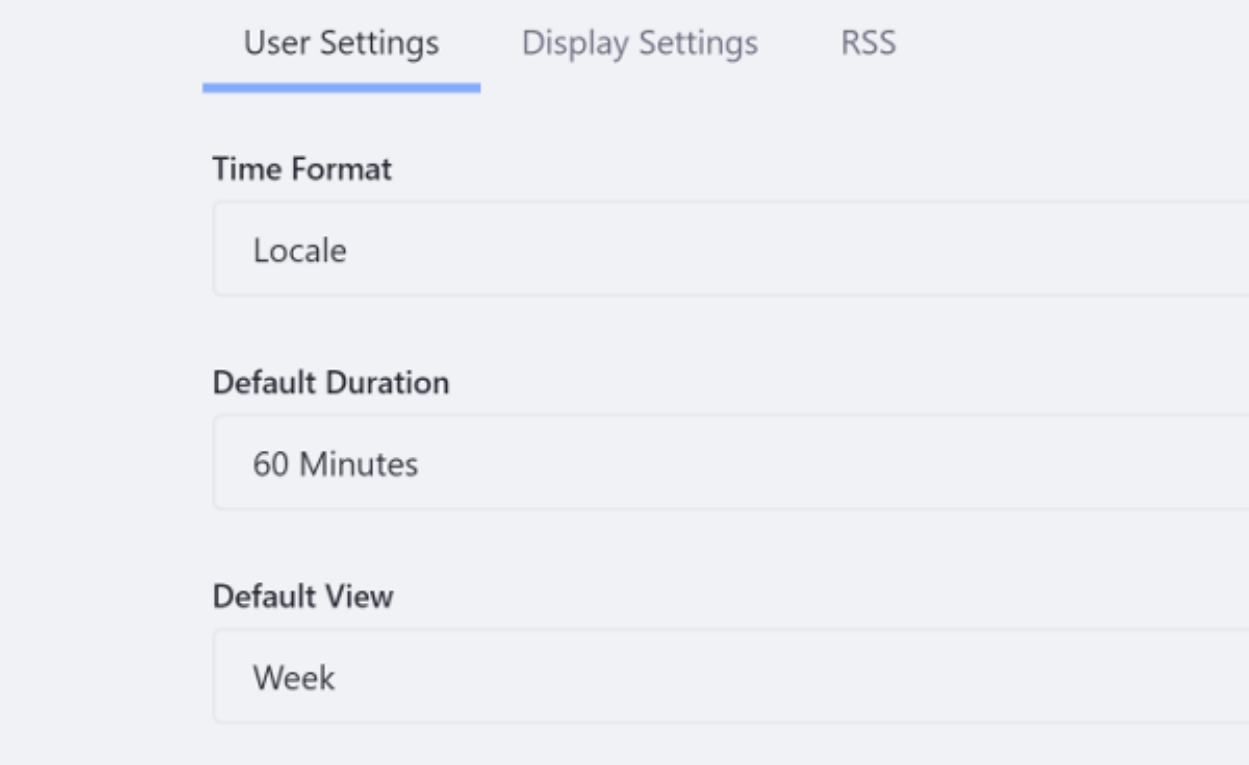

Figure 654.1: Tabs are a useful way to organize configuration options into individual sections within the same UI.

654.1 Related Topics

- Clay Navigation Bars
- Clay Dropdown Menus and Action Menus
- Liferay UI Icon Help

Chapter 655

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:

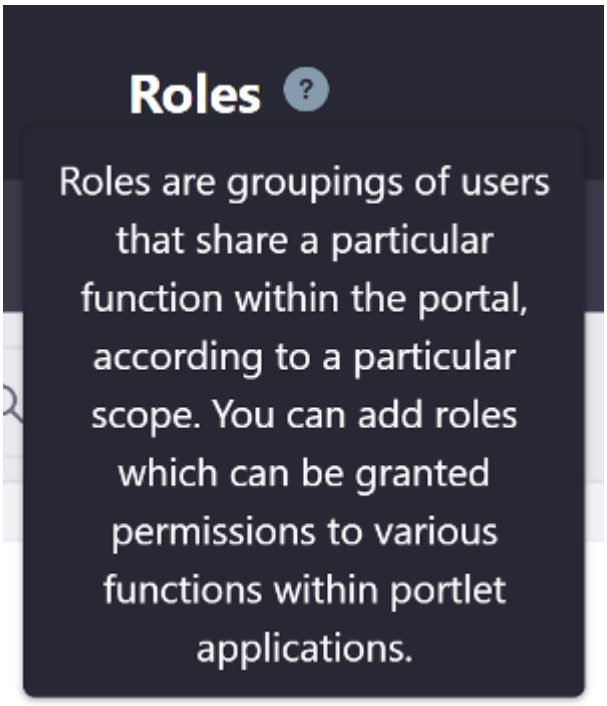

Figure 655.1: Here's an example of the icon help tag.

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 />
```

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.

655.1 Related Topics

- Clay Badges
- Clay Stickers
- Liferay UI Icon Menus

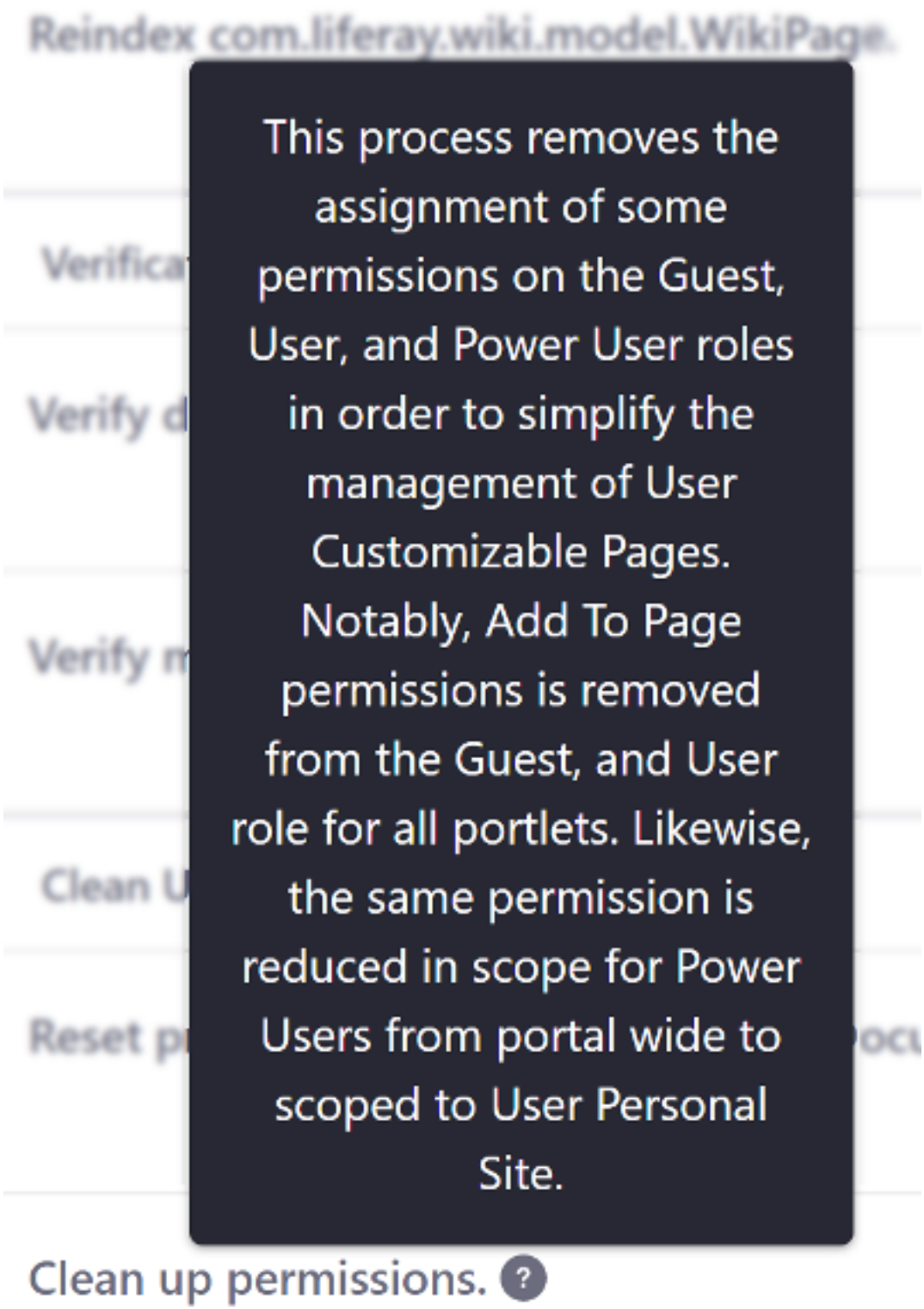

Figure 655.2: help icons are used throughout the Control Panel.

CHAPTER 656

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:

- Add Menu
- Cards
- Info Bar
- Management Bar

Each article contains a set of examples along with a screenshot of the resulting UI.

CHAPTER 657

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>
  <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>
</liferay-frontend:management-bar>
```

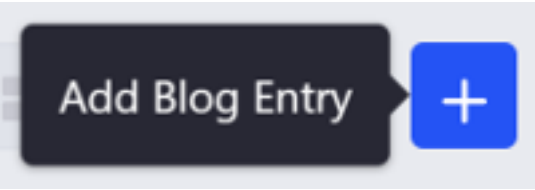

Figure 657.1: The add button pattern consists of an `add-menu` tag and at least one `add-menu-item` tag.

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>
```

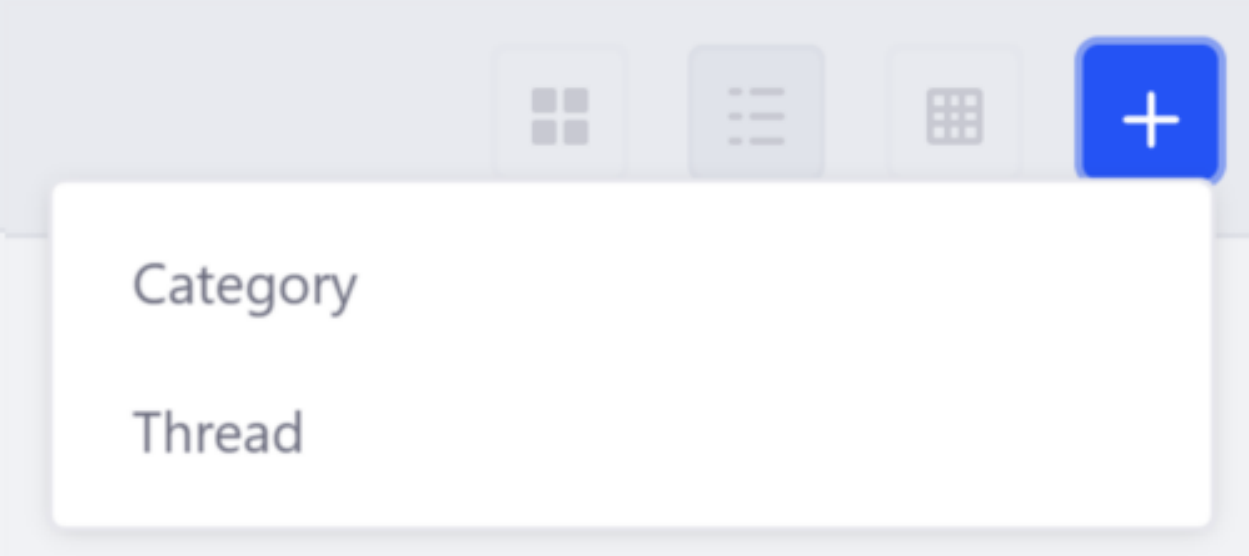

Figure 657.2: The add button pattern consists of an `add-menu` tag and at least one `add-menu-item` tag.

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.

657.1 Related Topics

- Liferay Frontend Cards
- Liferay Frontend Info Bar
- Liferay Frontend Management Bar

CHAPTER 658

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:

- Horizontal Cards
- Icon Cards
- Vertical Cards
- User Cards

Examples of each card are shown below.

658.1 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="https://portal.liferay.dev/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>
```

The `<liferay-frontend:horizontal-card-icon>` tag uses Clay Icons for its `icon` attribute.

658.2 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

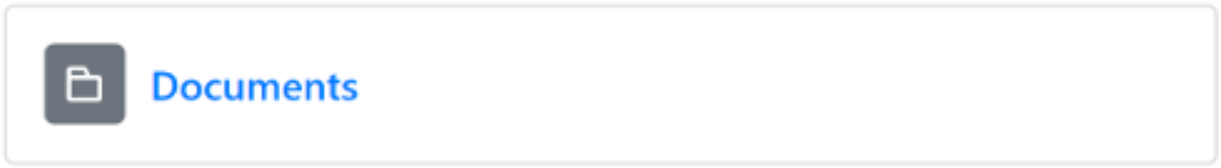

Figure 658.1: Horizontal cards are perfect to display files and documents.

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>
```

658.3 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
```

Figure 658.2: Vertical icon cards are perfect to display an entity selection, such as a web content article.

```
      markupView="lexicon"
      showIcon="<%= false %>"
      showLabel="<%= false %>"
      status="<%= article.getStatus() %>"
    />
  </liferay-frontend:vertical-card-footer>
</liferay-frontend:vertical-card>
```

658.4 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">
```

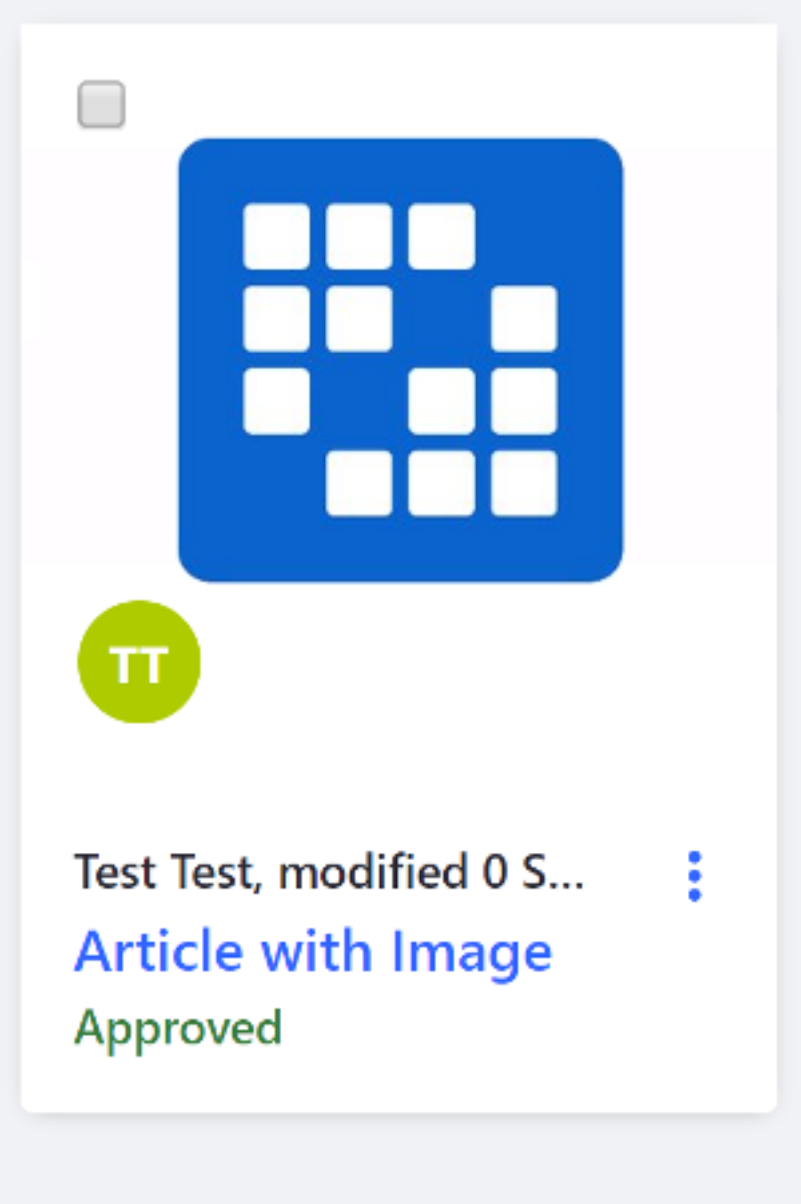

Figure 658.3: Vertical cards are perfect to display files and documents.

```
    <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 %>"
        title="My Video"
      >
      </liferay-frontend:html-vertical-card>
    </div>
  </div>
</div>
```

Figure 658.4: Html vertical cards let you display custom HTML in the card's header.

658.5 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>
```

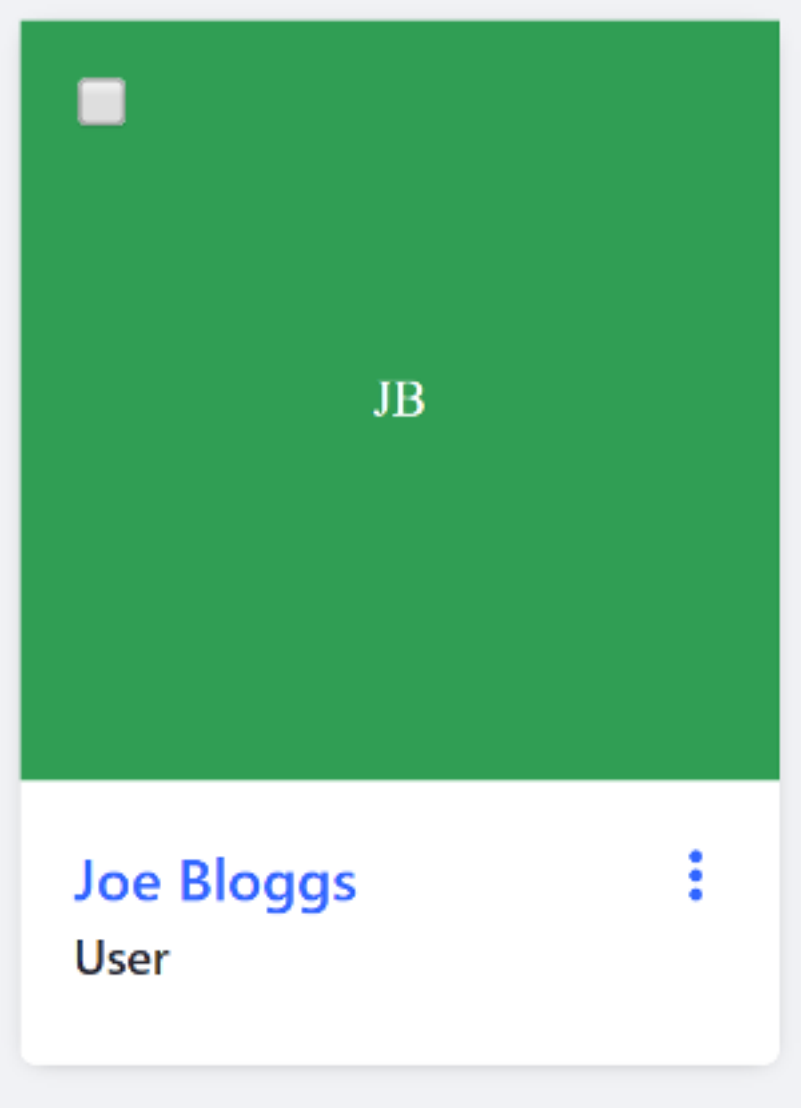

Figure 658.5: User vertical cards are perfect to display files and documents.

658.6 Related Topics

- Liferay Front-end Add Menu

- Liferay Front-end Info Bar
- Liferay Front-end Management Bar

CHAPTER 659

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:

```
<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.

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.

659.1 Related Topics

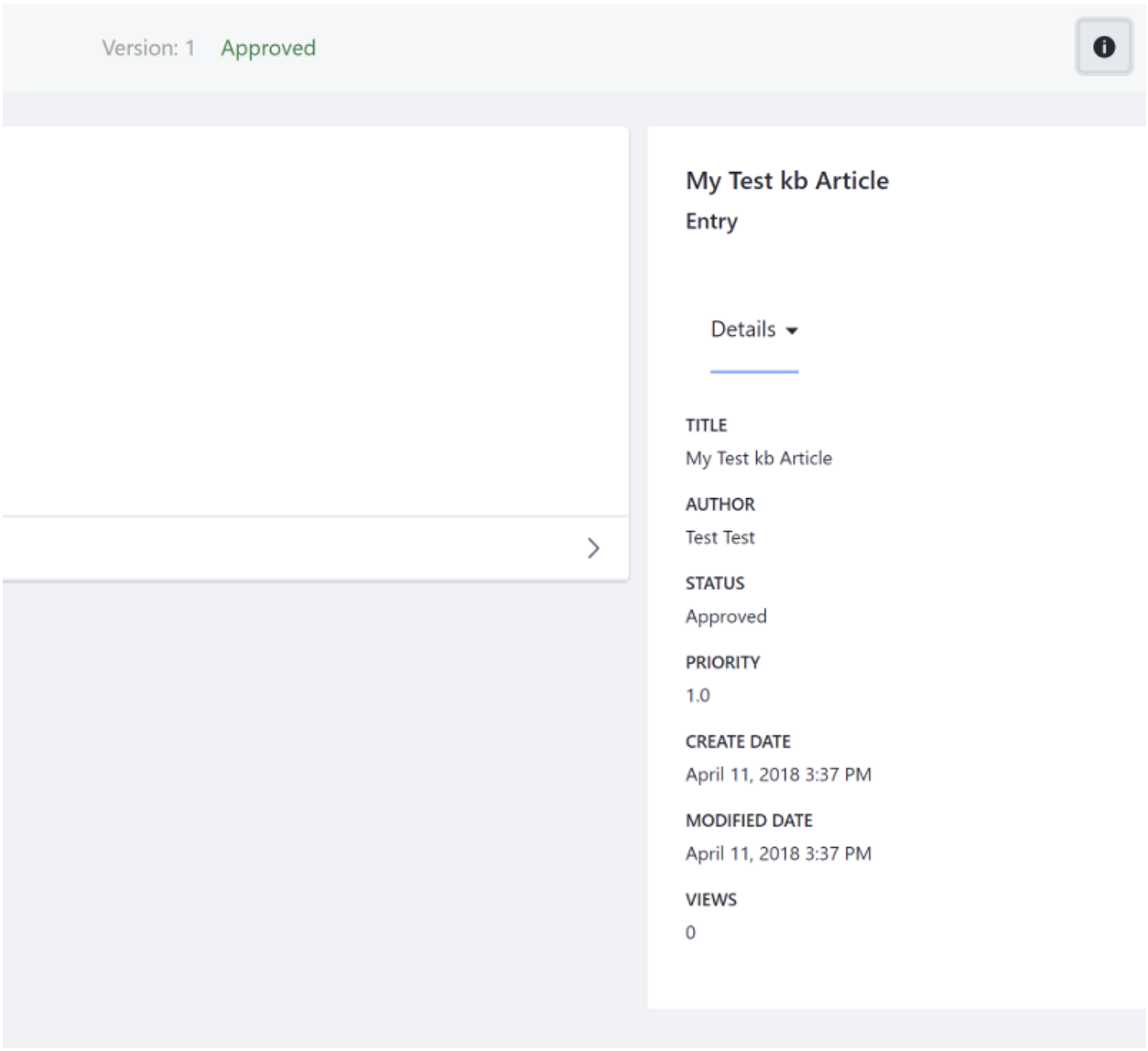

Figure 659.1: The info bar tags create a sidebar panel toggler that reveals additional info.

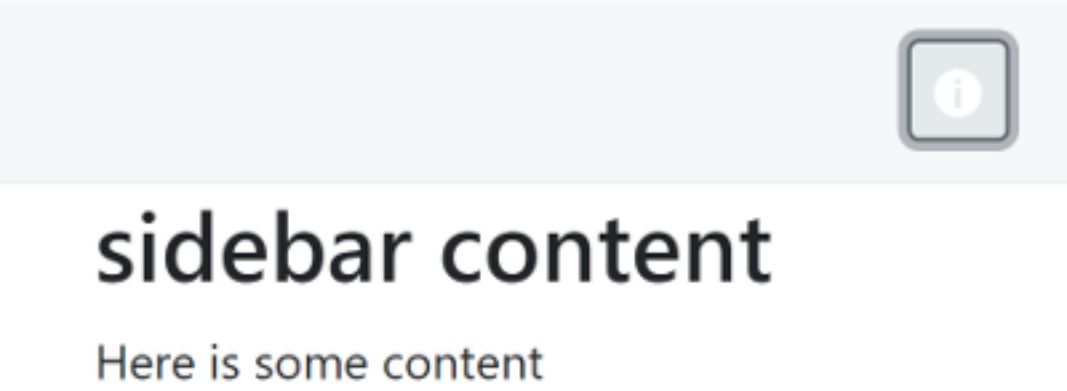

Figure 659.2: The info bar tags create a sidebar panel toggler that reveals additional info.

- Liferay Front-end Add Menu
- Liferay Front-end Cards
- Liferay Front-end Management Bar

CHAPTER 660

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.

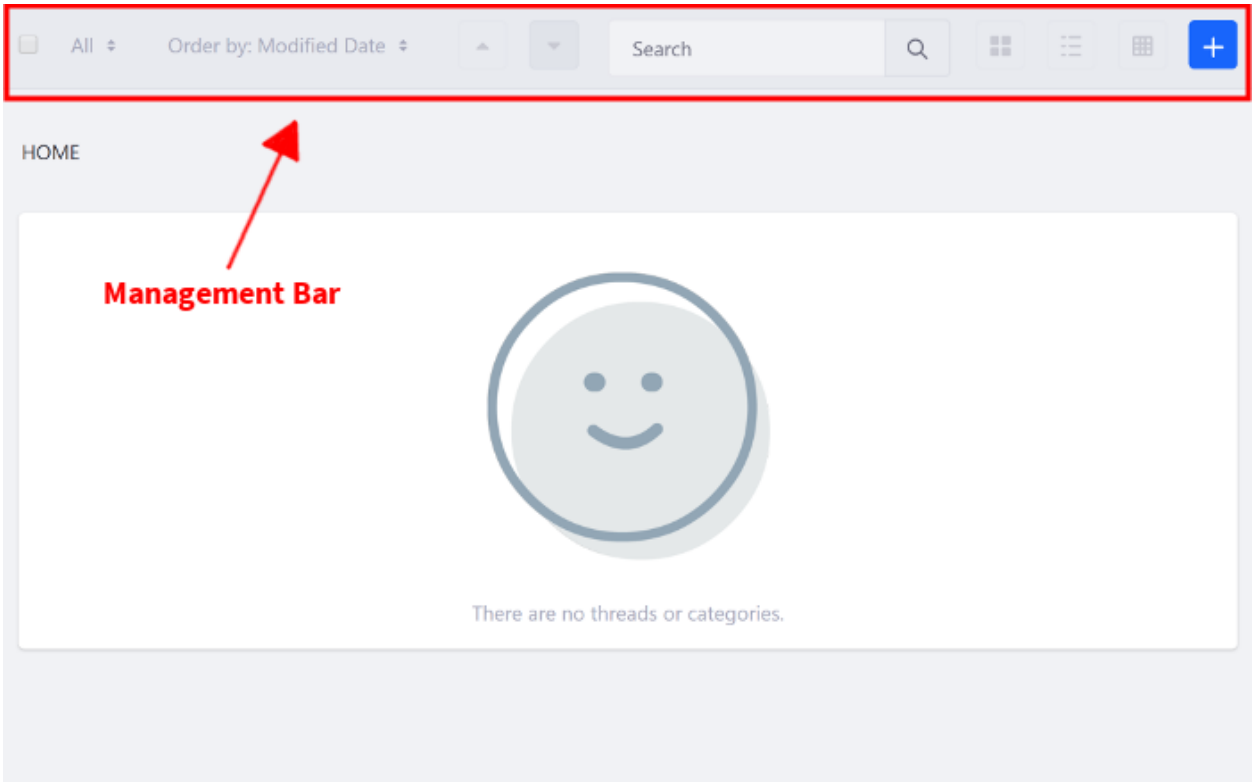

Figure 660.1: The Management Bar lets the user customize how the app displays content.

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:

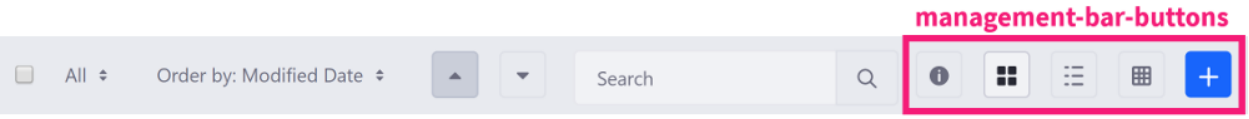

Figure 660.2: The `management-bar-buttons` tag contains the Management Bar's main buttons.

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.

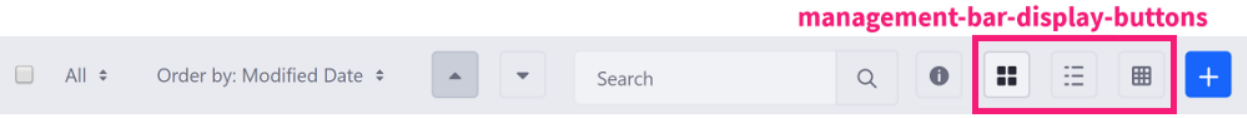

Figure 660.3: The `management-bar-display-buttons` tag contains the content's display options.

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.

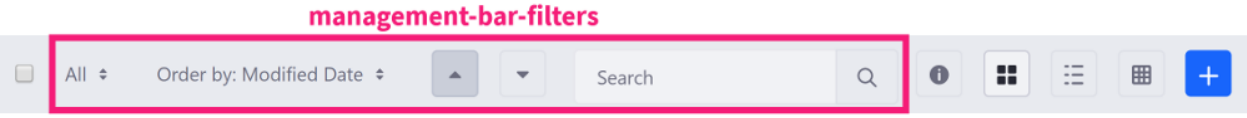

Figure 660.4: The `management-bar-filters` tag contains the content filtering options.

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.

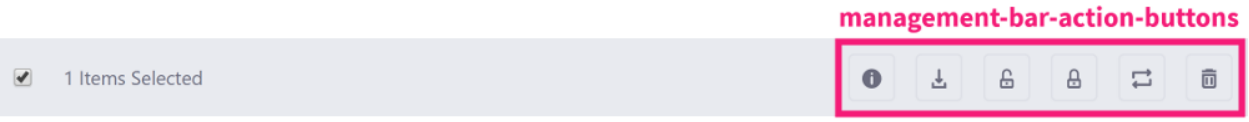

Figure 660.5: The management bar keeps track of the items selected and displays the actions to execute on them.

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>
```

CHAPTER 661

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.

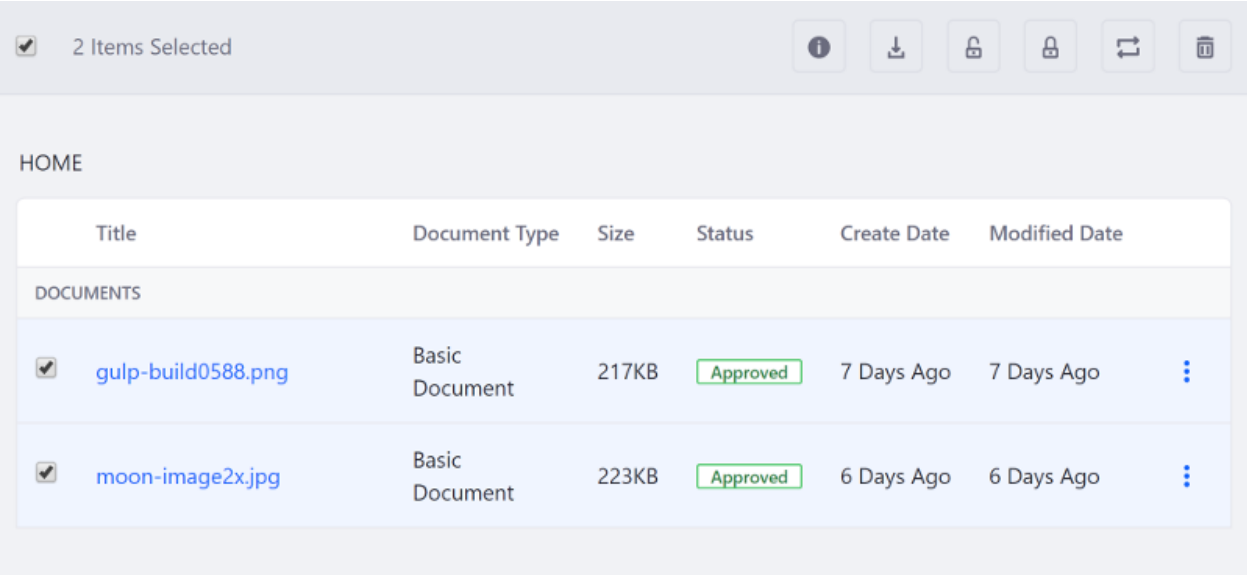

Figure 661.1: You can select individual results or all results at once.

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"
>
```

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>
```

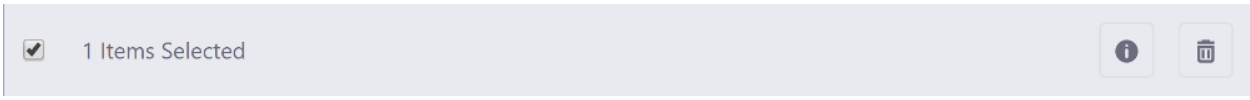

Figure 661.2: You can have as many actions as your app requires.

661.1 Related Topics

- Disabling All or Portions of the Management Bar
- Clay Management Toolbar

CHAPTER 662

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 %>"
/>
```

662.1 Related Topics

- Including Actions in the Management Bar
- Clay Management Toolbar

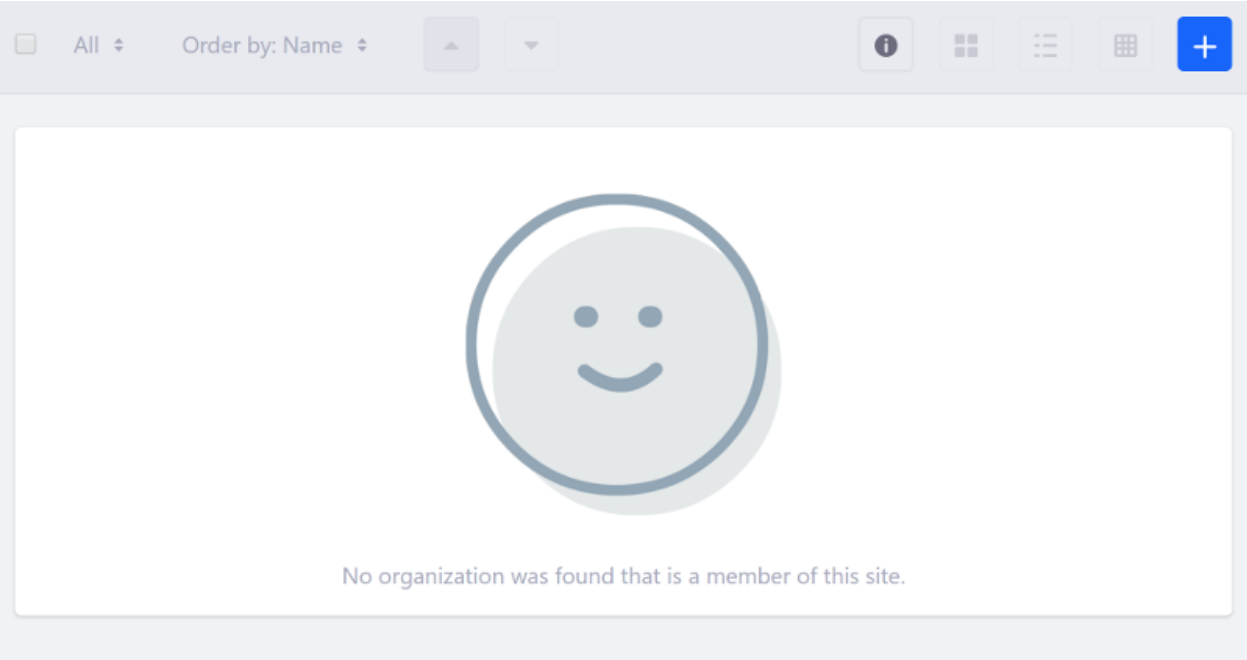

Figure 662.1: You can disable all or portions of the Management Bar.

CHAPTER 663

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 covers the available Liferay Util tags you can use in your app to inject content into portlets and themes.

CHAPTER 664

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.

664.1 Related Topics

- Using the Liferay Util HTML Body Top Tag
- Using the Liferay Util HTML Top Tag
- Using the Liferay UI Taglib

CHAPTER 665

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.

665.1 Related Topics

- Using the Liferay Util HTML Body Bottom Tag
- Using the Liferay Util HTML Bottom Tag
- Using the Clay Taglib

CHAPTER 666

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://www.liferay.com/"
            target="_blank">Liferay
        </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 666.1: You can use the Liferay Util Buffer tag to save pieces of markup to reuse in your JSP.

666.1 Related Topics

- JSP Overrides Using OSGi Fragments
- Using the Liferay Util Param Tag
- Using the Liferay Front-End Taglibs

CHAPTER 667

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 Dynamic Includes 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.

667.1 Related Topics

- Dynamic Includes
- Using the Liferay Util Body Top Tag
- Using the Chart Taglib

CHAPTER 668

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://www.liferay.com/" />
```

Here is an example that uses the var attribute:

```
<liferay-util:get-url url="https://www.liferay.com/" var="Liferay" />

<div>
        <h2>We borrowed <a href="https://www.liferay.com/">Liferay</a>. Here it is.</h2>

        <div class="Liferay">
                <%= Liferay %>
        </div>
</div>
```

Now you know how to use the `<liferay-util:get-url>` tag to scrape URLs.

668.1 Related Topics

- Using the Liferay Util Param Tag
- Using the Liferay Util Include Tag
- Using the AUI Taglib

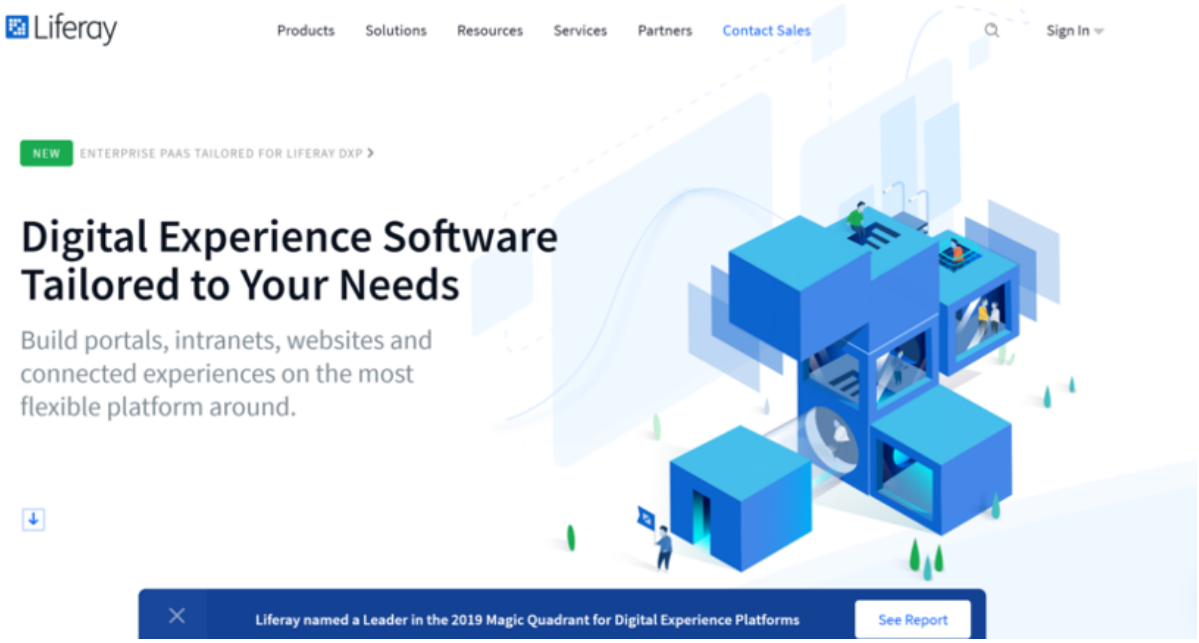

Figure 668.1: You can use the Liferay Util Get URL tag to scrape URLs.

CHAPTER 669

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.

669.1 Related Topics

- Using the Liferay Util HTML Body Bottom Tag
- Using the Liferay Util HTML Top Tag
- Using the Liferay UI Taglib

CHAPTER 670

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.

670.1 Related Topics

- Using the Liferay Util HTML Bottom Tag
- Using the Liferay Util Body Top Tag
- Using the Clay Taglib

CHAPTER 671

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.

671.1 Related Topics

- Using the Liferay Util Param Tag
- Using the Liferay Util Dynamic Include Tag
- Using the Liferay Front-End Taglibs

CHAPTER 672

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.

672.1 Related Topics

- Using the Liferay Util Include Tag
- Using the Liferay Util Body Top Tag
- Using the Chart Taglib

CHAPTER 673

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.

673.1 Related Topics

- Using the Liferay Util Param Tag
- Using the Liferay Util Buffer Tag
- Using the AUI Taglib

CHAPTER 674

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

This section covers how to create these components with the Clay taglibs. Each article contains a set of clay component examples along with a screenshot of the resulting UI.

CHAPTER 675

CLAY ALERTS

Clay alerts come in two types: embedded and stripe. Both types, along with several examples of each, are shown below.

675.1 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 675.1: 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"
/>
```

Info alert (embedded):

```
<clay:alert
    message="This is an info message."
    title="Info"
/>
```

Figure 675.2: The success alert notifies the user when an action is successful.

Figure 675.3: 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 675.4: The warning alert displays a warning message to the user.

675.2 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 675.5: 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"
/>
```

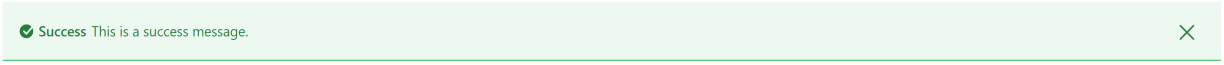

Figure 675.6: The success striped alert notifies the user that an action has completed successfully.

Info alert (stripe):

```
<clay:stripe
    message="This is an info message."
    title="Info"
/>
```

Figure 675.7: 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"
/>
```

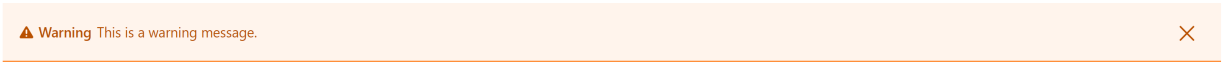

Figure 675.8: The warning striped alert warns the user about an action.

Now you know how to alert users!

675.3 Related Topics

- Clay Buttons
- Clay Form Elements
- Clay Labels and Links

CHAPTER 676

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 covers the different types of Clay badges you can add to your app.

676.1 Badge Types

The following badge styles are available:
Primary badge:

```
<div class="col-md-1">
    <clay:badge label="8" />

    <div>Primary</div>
</div>
```

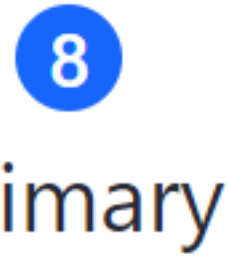

Figure 676.1: A primary badge is bright blue, commanding attention like the primary button of a form.

Secondary badge:

```
<div class="col-md-1">
    <clay:badge label="87" style="secondary" />

    <div>Secondary</div>
</div>
```

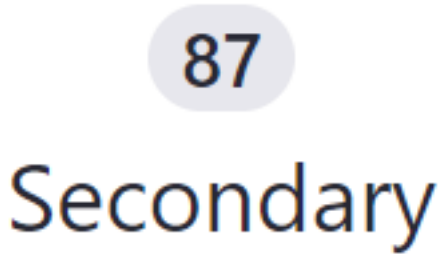

Figure 676.2: A secondary badge is light-grey and draws less focus than a primary button.

Info badge:

```
<div class="col-md-1">
    <clay:badge label="91" style="info" />

    <div>Info</div>
</div>
```

Figure 676.3: 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>
```

Success badge:

```
<div class="col-md-1">
    <clay:badge label="1111" style="success" />

    <div>Success</div>
</div>
```

Warning badge:

Figure 676.4: An error badge displays numbers related to an error.

Figure 676.5: A success badge displays numbers related to a successful action.

```
<div class="col-md-1">
    <clay:badge label="21" style="warning" />

    <div>Warning</div>
</div>
```

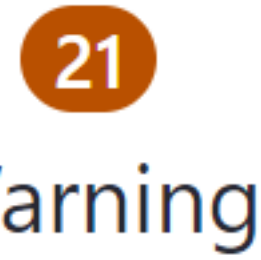

Figure 676.6: A warning badge displays numbers related to warnings that should be addressed.

Now you know how to use badges to keep track of values in your app.

676.2 Related Topics

- Clay Labels and Links
- Clay Cards
- Clay Stickers

CHAPTER 677

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.

677.1 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 677.1: 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>
```

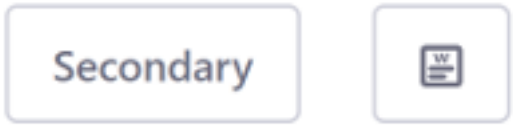

Figure 677.2: A secondary button draws less attention than a primary button and is meant for secondary actions.

```
<div class="col">
    <clay:button ariaLabel="Page Template" icon="page-template" style="borderless" />
</div>
```

Figure 677.3: 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 677.4: 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 677.5: 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 677.6: Buttons can be disabled if you don't want the user to interact with them.

677.2 Variations

Button with icon and text:

```
<clay:button icon="share" label="Share" />
```

Figure 677.7: Buttons can display both icons and text.

Button with monospaced text:

```
<clay:button icon="indent-less" monospaced="<%= true %>" style="secondary" />
```

Figure 677.8: Buttons can display monospaced text.

Block level button:

```
<clay:button block="<%= true %>" label="Button" />
```

Plus button:

```
<clay:button icon="plus" monospaced="<%= true %>" style="secondary" />
```

Action button:

```
<clay:button icon="ellipsis-v" monospaced="<%= true %>" style="borderless" />
```

Figure 677.9: Block level buttons span the entire width of the container.

Figure 677.10: : A plus button is used for add actions in an app.

Figure 677.11: : An action button is used to display actions menus.

677.3 Related Topics

- Clay Alerts
- Clay Buttons
- Clay Labels and Links

CHAPTER 678

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 below.

678.1 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 %>"
/>
```

Image Card empty:

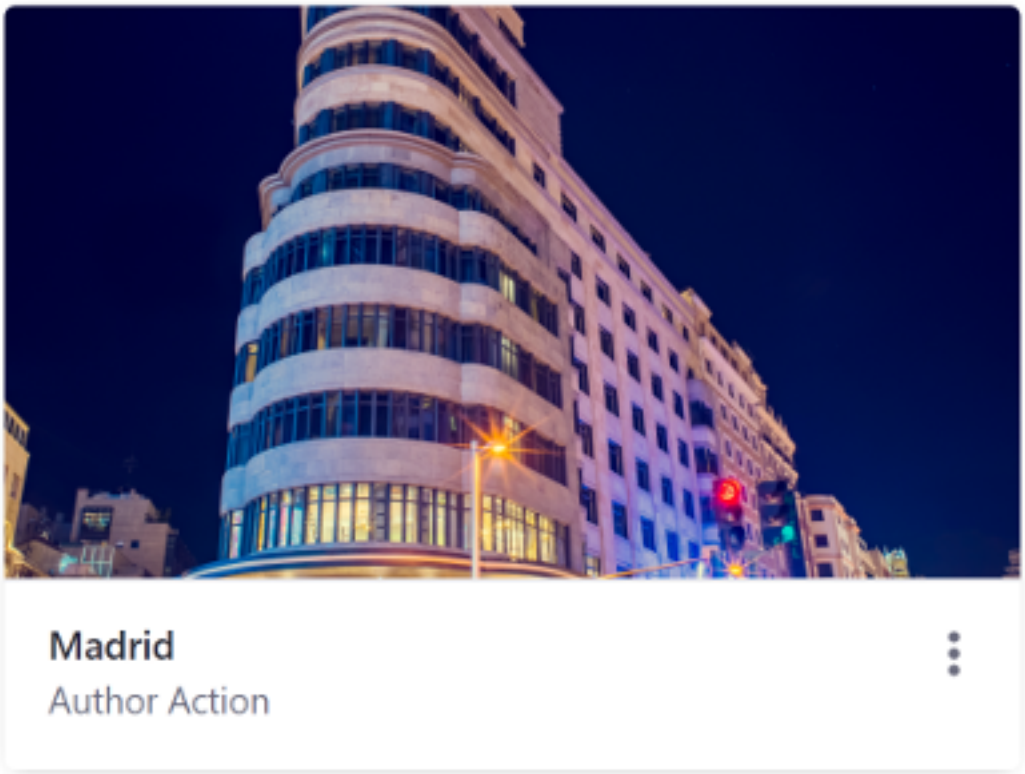

Figure 678.1: Image Cards display images and documents.

```
<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"
    labels="<%= cardsDisplayContext.getLabels() %>"
    subtitle="Author Action"
```

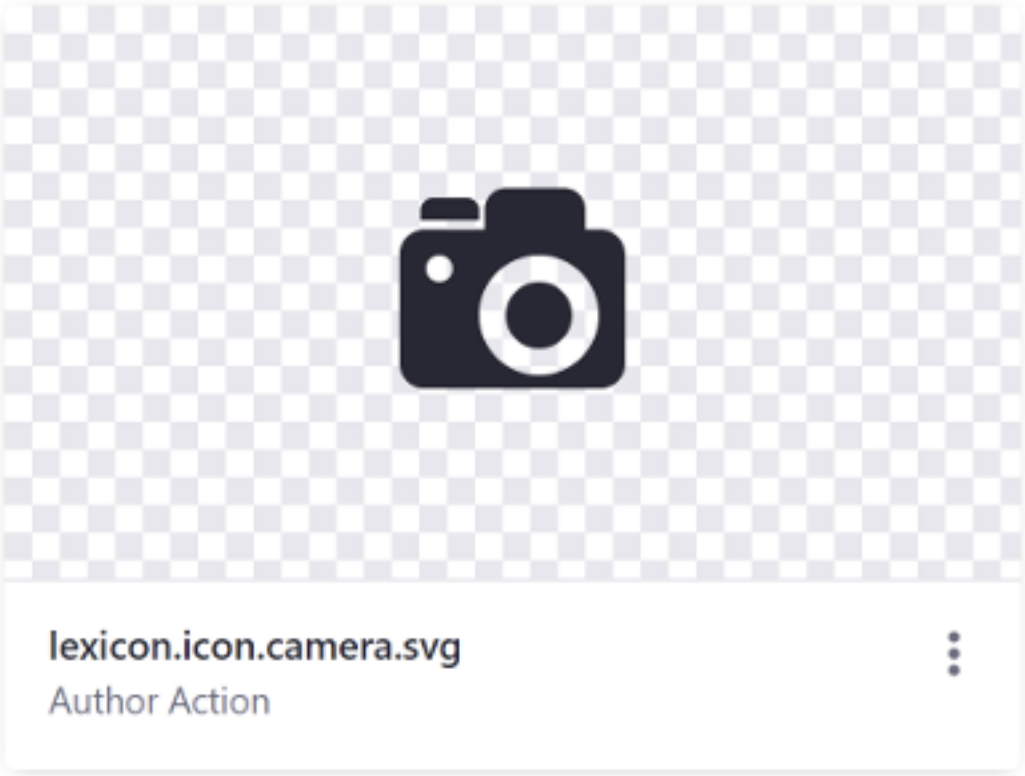

Figure 678.2: Image Cards can also display icons instead of images.

```
    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"
/>
```

678.2 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
```

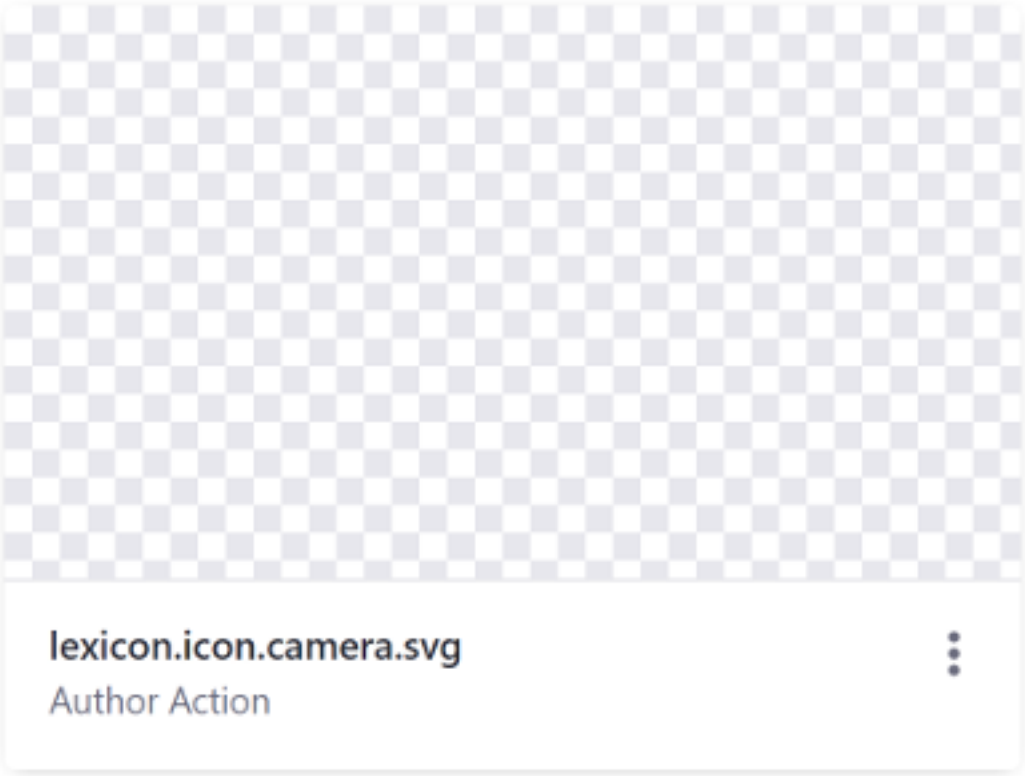

Figure 678.3: Cards can also display nothing.

```
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    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 678.4: Cards can also contain file types.

678.3 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"
/>
```

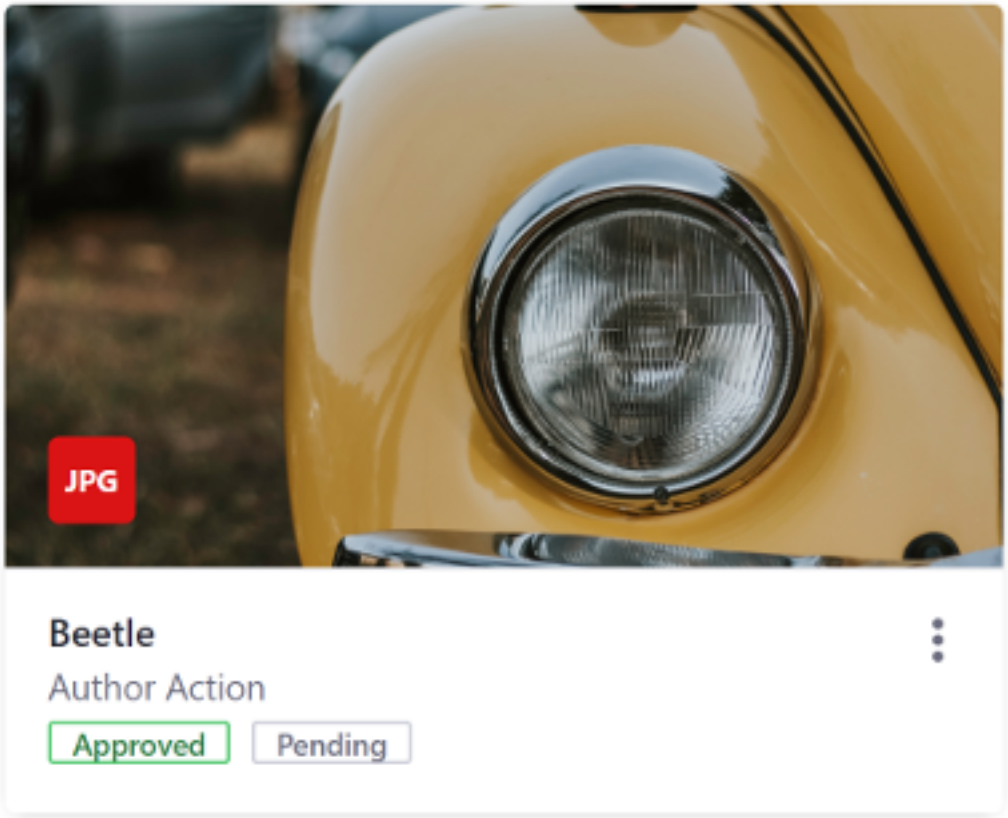

Figure 678.5: You can include labels in Cards.

678.4 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.

678.5 Related Topics

- Clay Badges
- Clay Labels and Links
- Clay Stickers

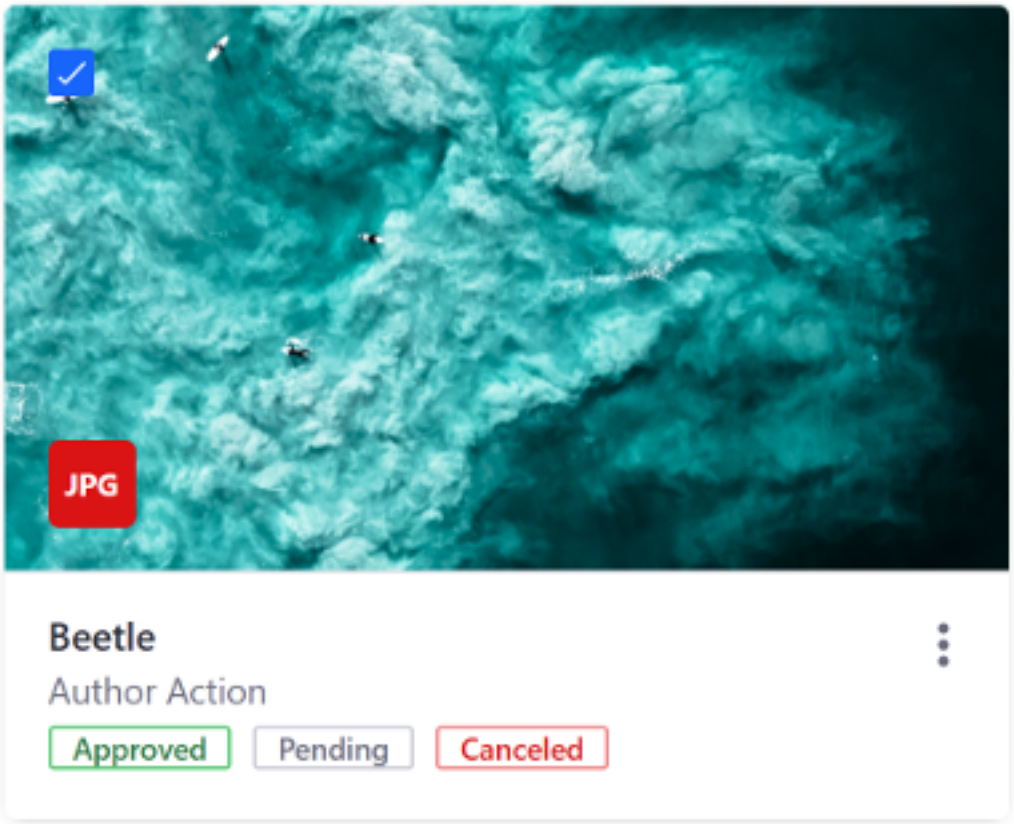

Figure 678.6: Cards can be selectable.

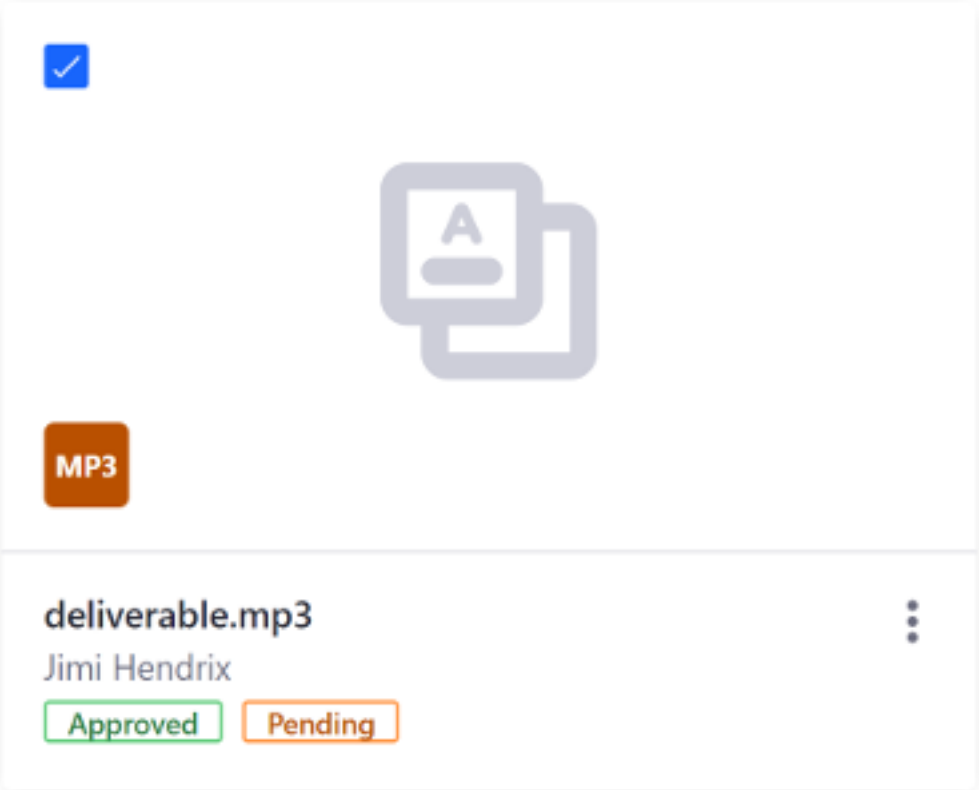

Figure 678.7: File Cards display file type icons.

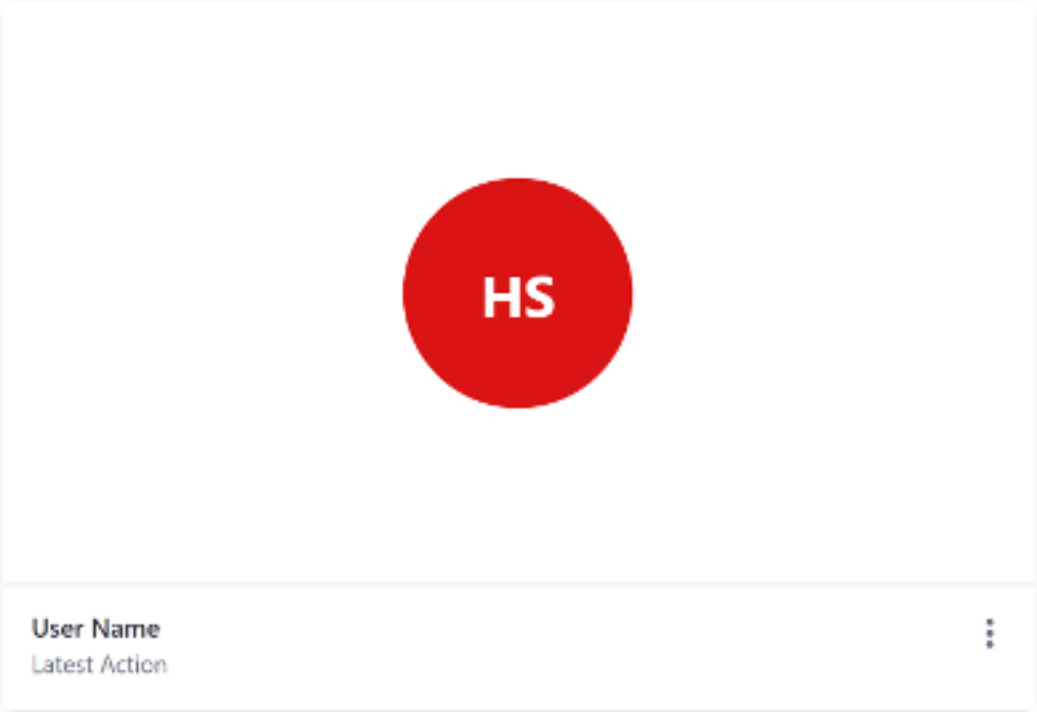

Figure 678.8: User Cards can display a user's initials.

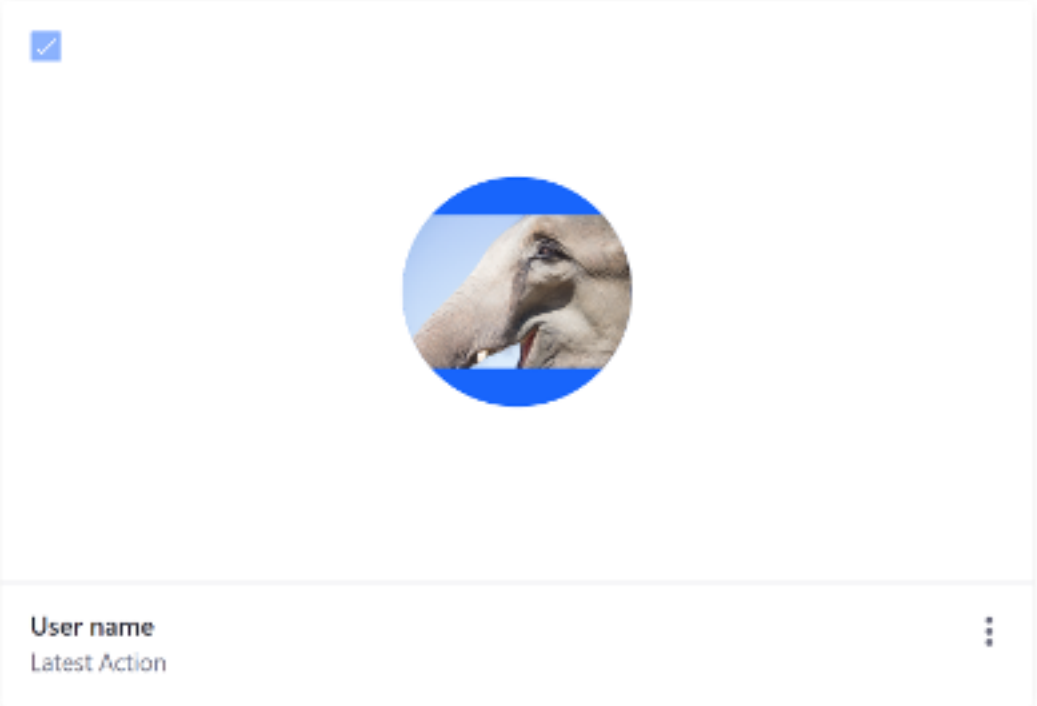

Figure 678.9: A User Card can also display a profile image.

Figure 678.10: : Horizontal Cards are good for displaying folders.

CHAPTER 679

CLAY DROPDOWN MENUS AND ACTION MENUS

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.

679.1 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;
}

_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);
```

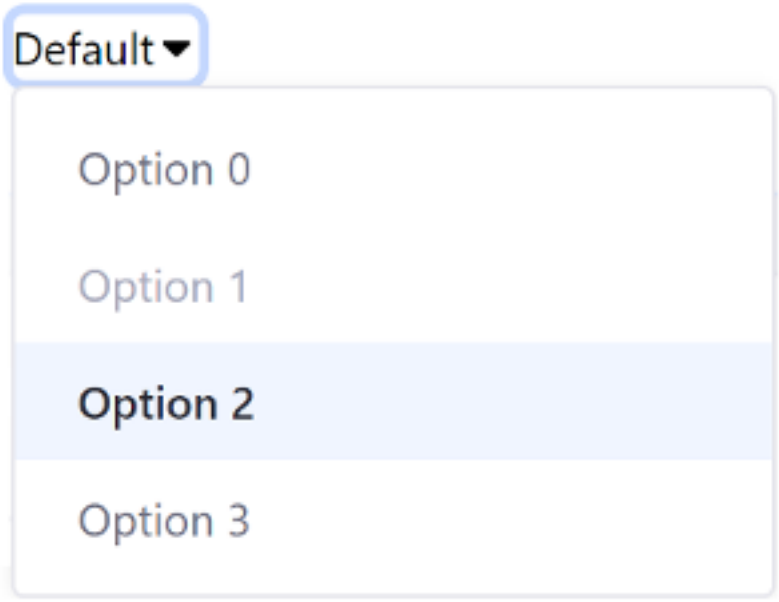

Figure 679.1: Clay taglibs provide everything you need to add dropdown menus to your app.

```
}

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:

```
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:

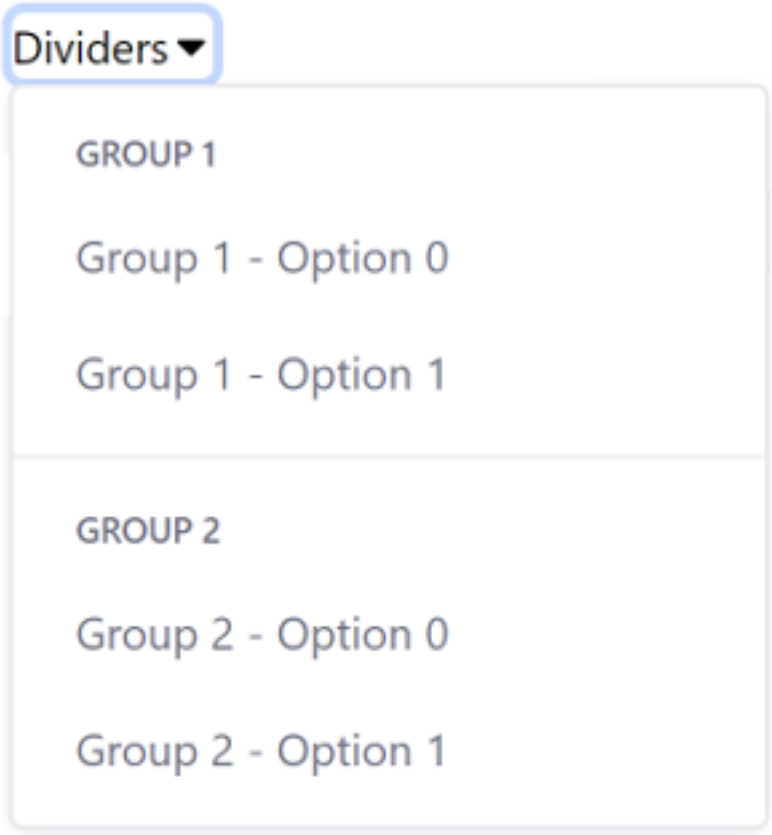

Figure 679.2: You can organize dropdown menu items into groups.

```
<clay:dropdown-menu
    buttonLabel="Done"
    items="<%= dropdownsDisplayContext.getInputDropdownItems() %>"
    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"
/>
```

679.2 Actions Menus

Basic actions menu:

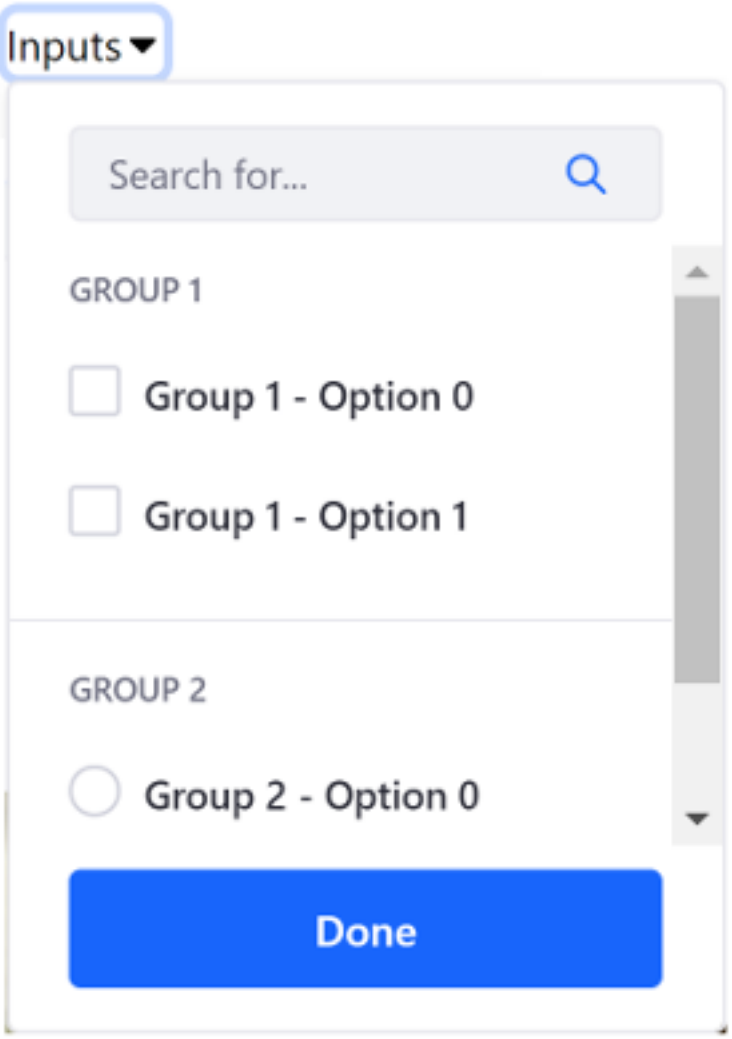

Figure 679.3: Inputs can be included in dropdown menus.

```
<clay:dropdown-actions
    items="<%= dropdownsDisplayContext.getDefaultDropdownItems() %>"
/>
```

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.

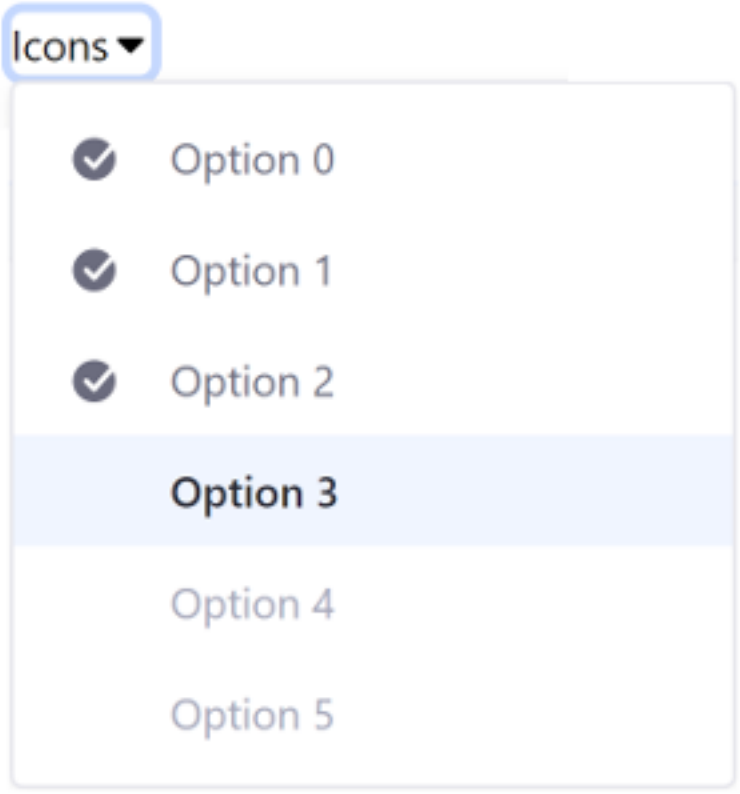

Figure 679.4: Icons can be included in dropdown menus.

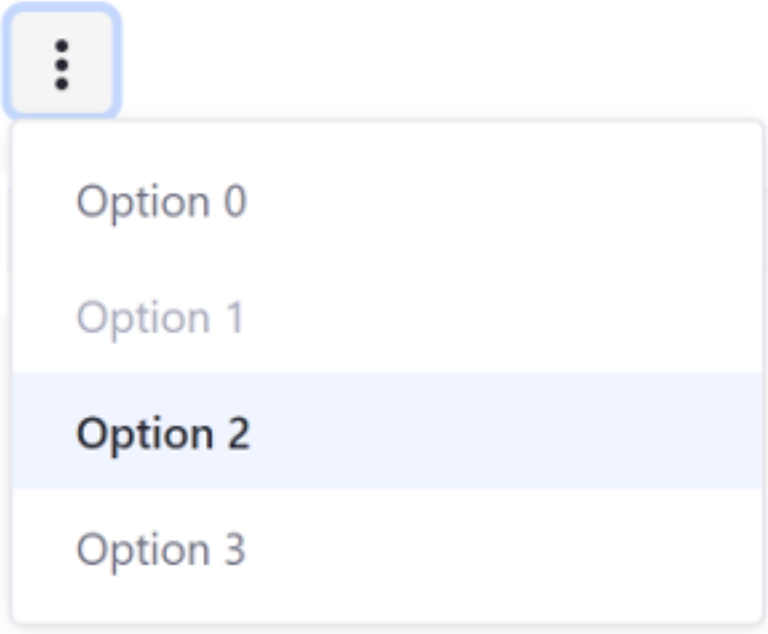

Figure 679.5: You can also create Actions menus with Clay taglibs.

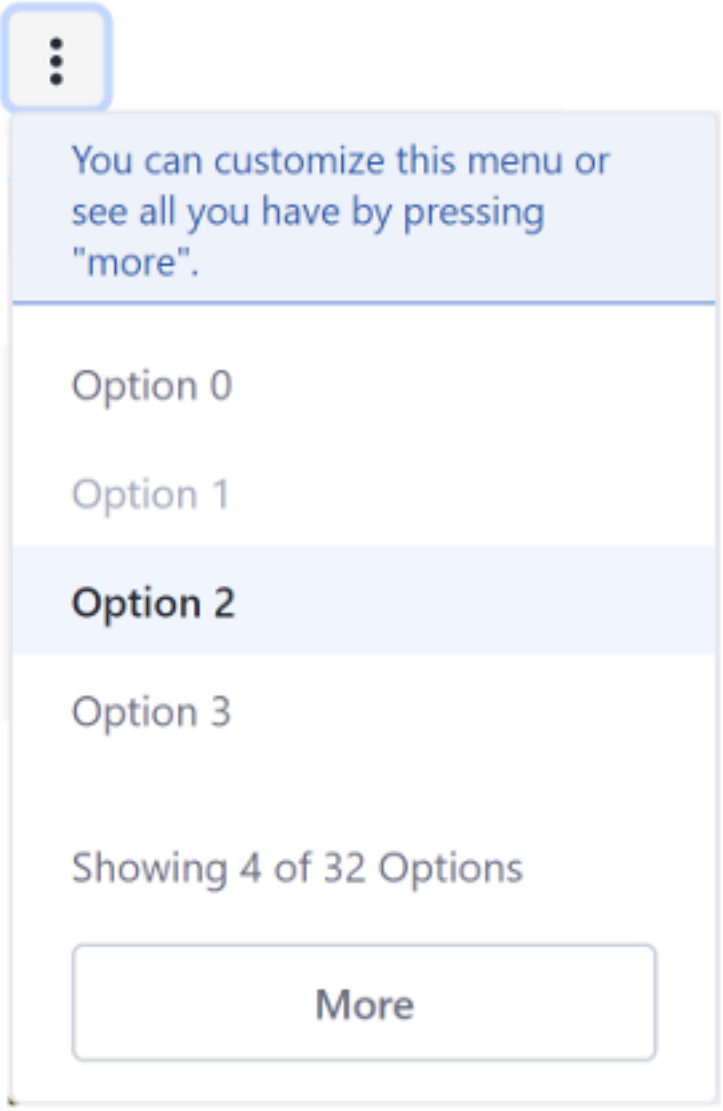

Figure 679.6: You can provide help text in Actions menus.

679.3 Related Topics

- Clay Form Elements
- Clay Navigation Bars
- Clay Progress Bars

CHAPTER 680

CLAY FORM ELEMENTS

The Liferay Clay tag library provides several tags for creating form elements. An example of each tag is shown below.

680.1 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 680.1: Clay taglibs provide checkboxes.

680.2 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 680.2: Clay taglibs provide radio buttons.

680.3 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!

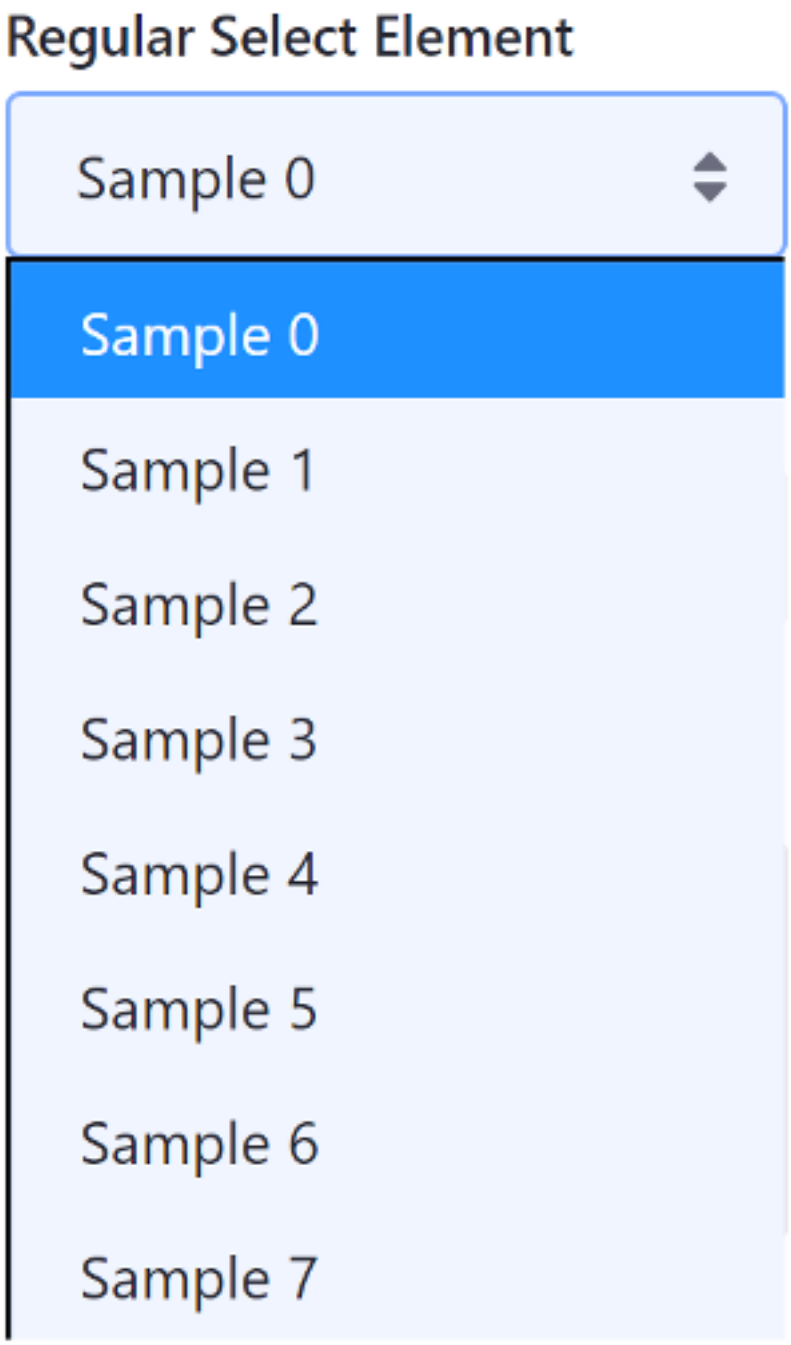

Figure 680.3: Clay taglibs provide select boxes.

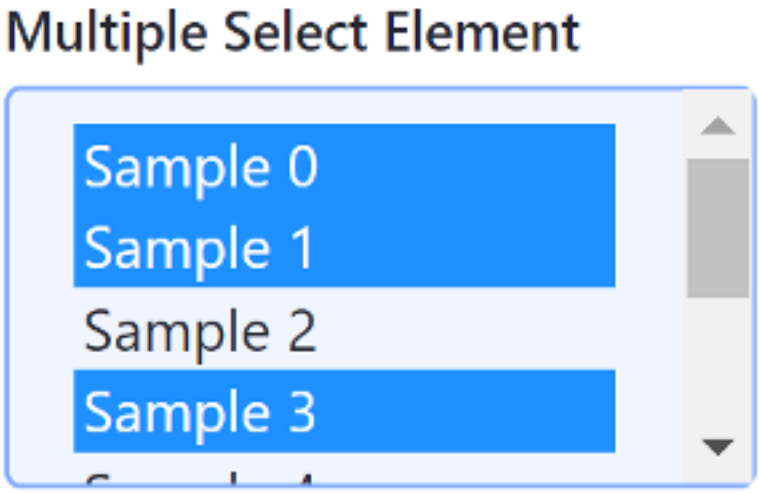

Figure 680.4: You can let users select multiple options from the select menu.

680.4 Related Topics

- Clay Buttons
- Clay Icons
- Clay Labels and Links

CHAPTER 681

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 681.1: 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:

681.1 Related Topics

- Clay Badges
- Clay Stickers
- Using Clay Icons in a Theme

Figure 681.2: The Clay taglib gives you access to several Liferay DXP icons.

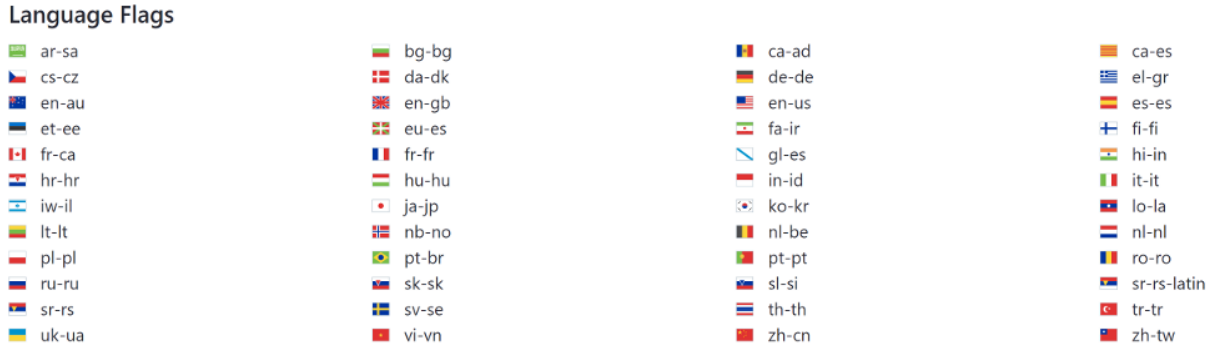

Figure 681.3: You can include language flags in your apps.

CHAPTER 682

CLAY LABELS AND LINKS

Liferay Clay taglibs provide tags for creating labels and links in your app. Both of these elements are covered below.

682.1 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.

682.2 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 682.1: 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 682.2: 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 682.3: 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" />
```

Figure 682.4: Danger labels convey a sense of urgency that must be addressed.

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" />
```

Figure 682.5: Success labels indicate a successful action.

Labels can also be bigger. Set the `size` attribute to `lg` to display large labels:

```
<clay:label label="Approved" size="lg" style="success" />
```

682.3 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 682.6: Labels can be removable.

682.4 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 682.7: Labels can also be links.

682.5 Links

You can add traditional hyperlinks to your app with the `<clay:link>` tag:

```
<clay:link href="#" label="link text" />
```

Figure 682.8: Clay taglibs also provide link elements.

Now you know how to add links and labels to your apps!

682.6 Related Topics

- Clay Badges
- Clay Cards
- Clay Form Elements

CHAPTER 683

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.

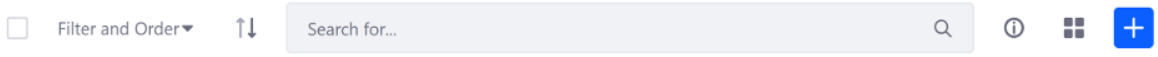

Figure 683.1: The Management ToolBar lets the user customize how the app displays content.

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.

683.1 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 Filtering and Sorting Items with the Management Toolbar. 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.

683.2 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 683.2: 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.

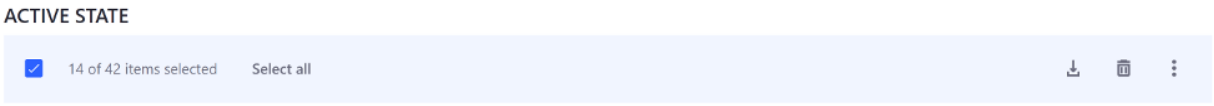

Figure 683.3: The Management Toolbar keeps track of the results selected and displays the actions to execute on them.

683.3 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.

`filterLabelItems`: Sets the search container's filter labels to display. This lets the user know which filters are currently applied.

`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");
            });
        }
    }
%>"


filterLabelItems="<%=
  new LabelItemList() {
    {
      add(
        labelItem -> {
          labelItem.setLabel("Filter 1");
        });

      add(
        labelItem -> {
          labelItem.setLabel("Filter 2");
        });
    }
  };
%>"
```

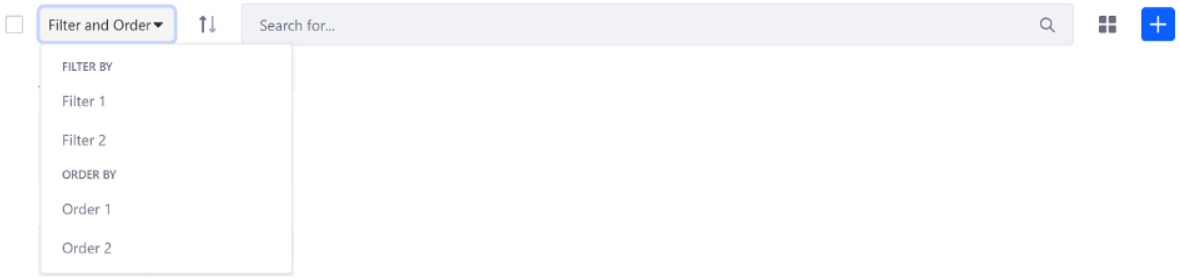

Figure 683.4: You can also sort and filter search container results.

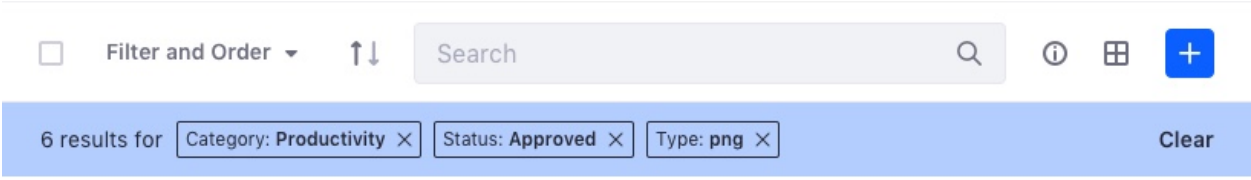

Figure 683.5: You can also sort and filter search container results.

683.4 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 value

`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 %>"
/>
```

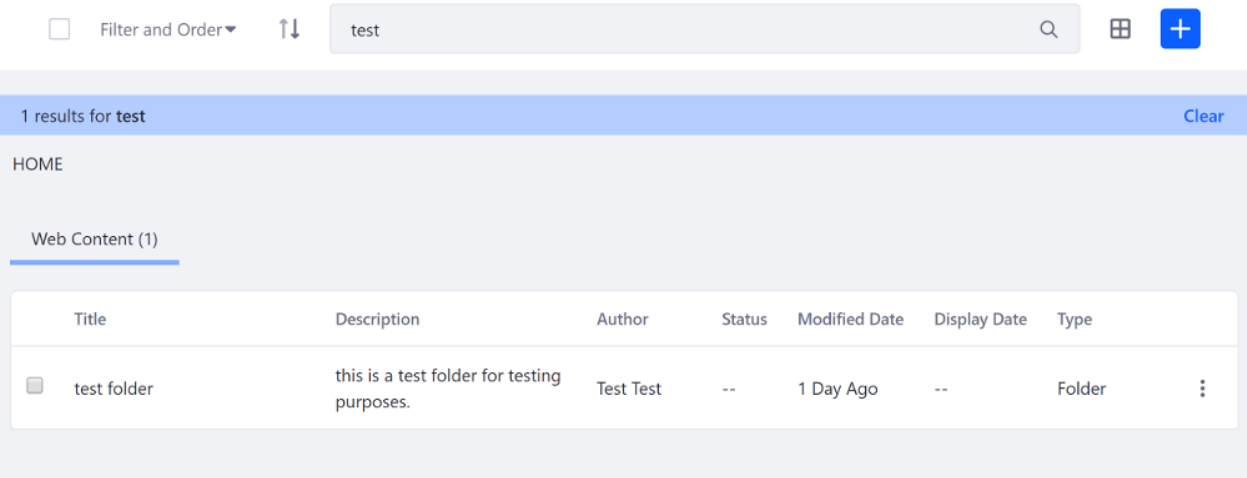

Figure 683.6: The search form comprises most of the Management Toolbar, letting users search through the search container results.

683.5 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:

```
<clay:management-toolbar
    displayContext="<%= journalDisplayContext %>"
    infoPanelId="infoPanelId"
    namespace="<%= renderResponse.getNamespace() %>"
    searchContainerId="≤% searchContainerId %>"
/>
```

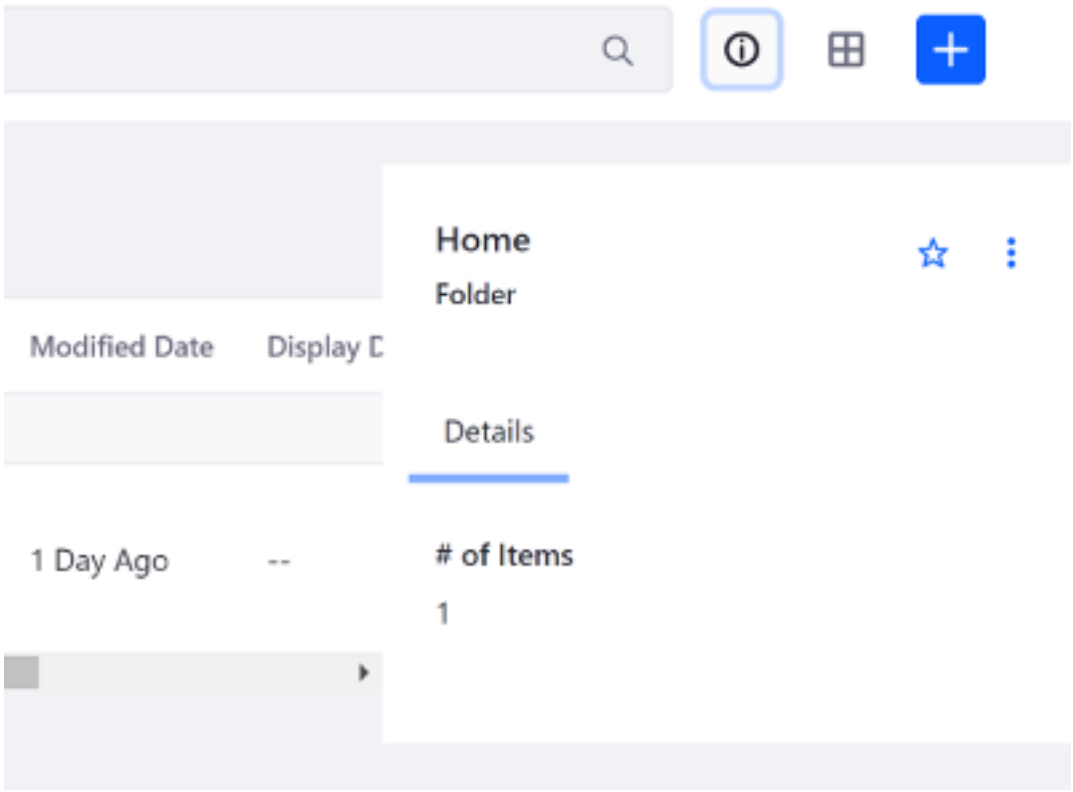

Figure 683.7: The info panel keeps your UI clutter-free.

683.6 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.

List: Displays a detailed description along with summarized details for the search result columns.

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) {
```

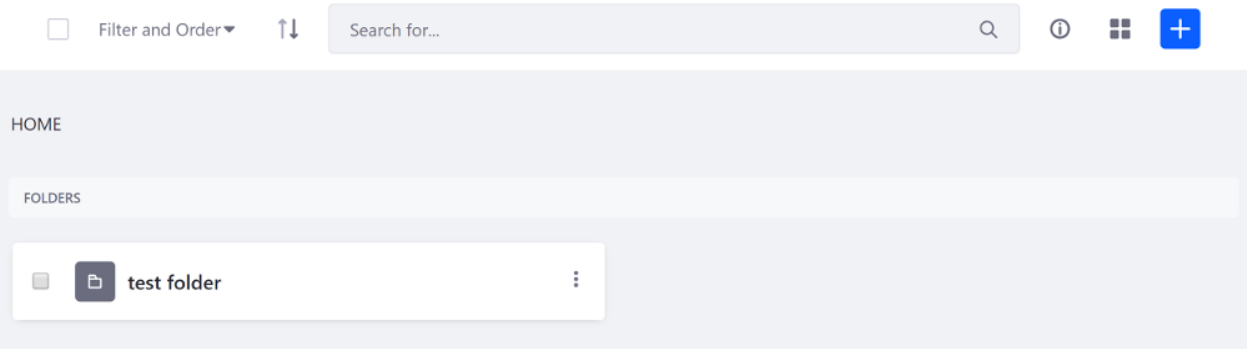

Figure 683.8: The Management Toolbar's icon display view gives a quick summary of the content's description and status.

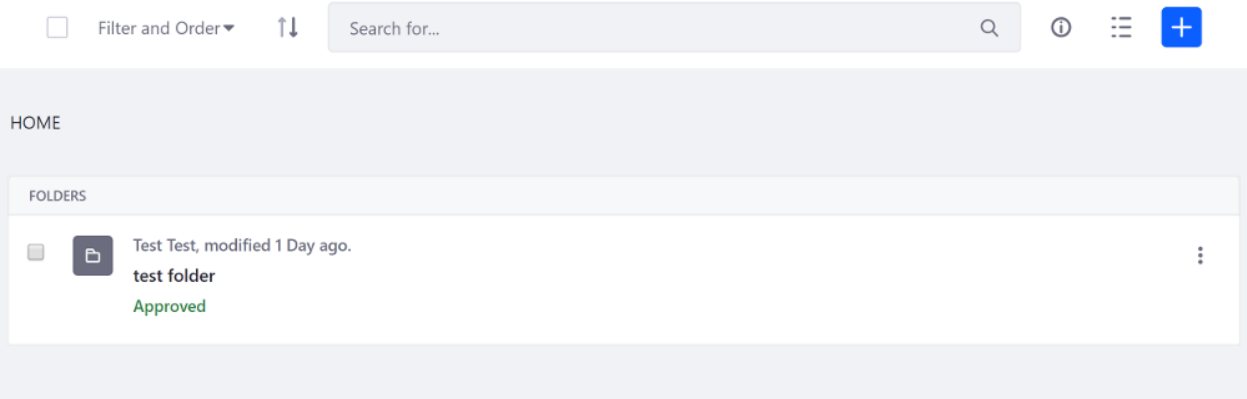

Figure 683.9: The Management Toolbar's List view type gives the content's full description.

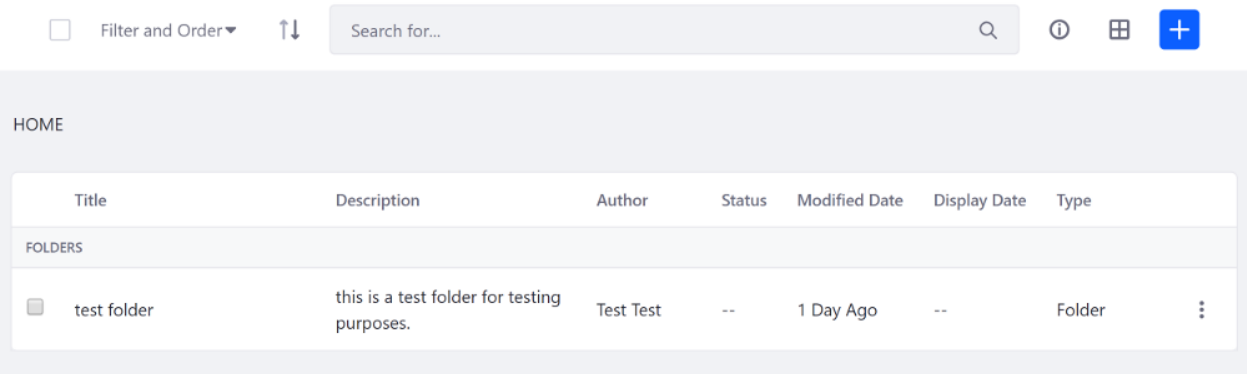

Figure 683.10: : The Management Toolbar's Table view type list the content's information in individual columns.

```
        {
            addCardViewTypeItem(
                viewTypeItem -> {
                    viewTypeItem.setActive(true);
                    viewTypeItem.setLabel("Card");
                });

            addListViewTypeItem(
                viewTypeItem -> {
                    viewTypeItem.setLabel("List");
                });

            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.

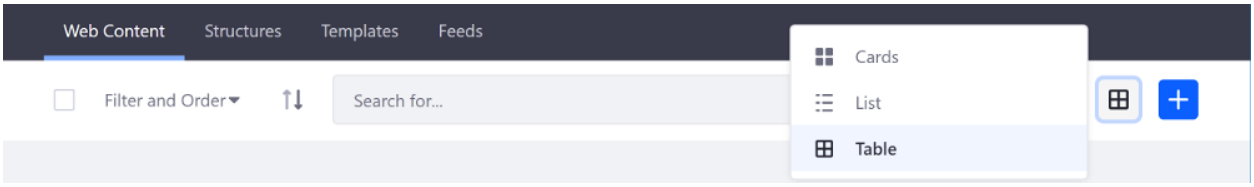

Figure 683.11: : The Management Toolbar offers three view type options.

683.7 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");
                });
        }
    };
%>"
```

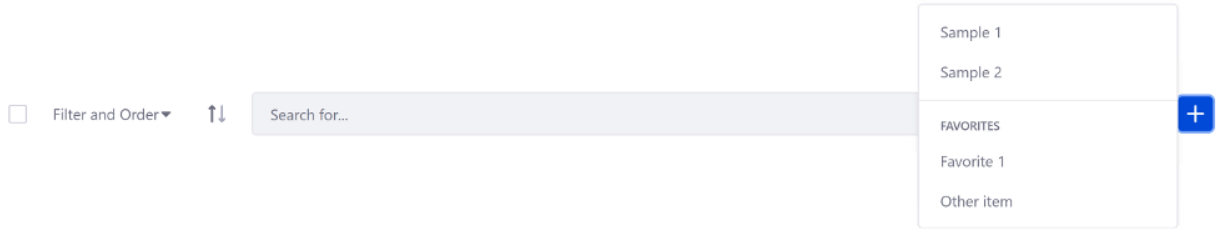

Figure 683.12: : The Management Toolbar lets you optionally add a Creation Menu for creating new entities.

683.8 Related Topics

- Clay Dropdown Menus and Action Menus
- Clay Icons
- Clay Navigation Bars

CHAPTER 684

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() %>"
/>
```

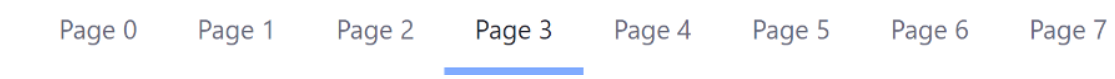

Figure 684.1: You can include navigation bars in your apps.

Inverted navigation bar (set `inverted` attribute to `true`):

```
<clay:navigation-bar
    inverted="<%= true %>"
    navigationItems="<%= navigationBarsDisplayContext.getNavigationItems() %>"
/>
```

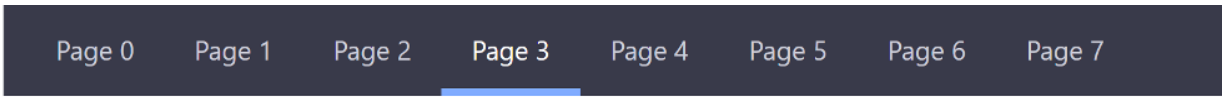

Figure 684.2: Navigation bars can be inverted if you prefer.

684.1 Related Topics

- Clay Dropdown Menus and Action Menus
- Clay Form Elements
- Clay Progress Bars

CHAPTER 685

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 685.1: You can include progress bars in your apps.

Warning progress bar:

```
<clay:progressbar
    maxValue="<%= 100 %>"
    minValue="<%= 0 %>"
    status="warning"
    value="<%= 70 %>"
/>
```


Figure 685.2: warning progress bars indicate that the progress has not completed due to an error.

Complete progress bar:

```
<clay:progressbar
    status="complete"
/>
```

Clay taglibs make it easy to track progress in your apps.

Figure 685.3: The complete progress bar indicates the progress is complete.

685.1 Related Topics

- Clay Dropdown Menus and Action Menus
- Clay Icons
- Clay Navigation Bars

CHAPTER 686

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 686.1: You can include stickers in your apps.

Square sticker with icon:

```
<clay:sticker icon="picture" />
```

Figure 686.2: Stickers can include icons.

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`:

```
<div class="aspect-ratio">

    <img class="aspect-ratio-item-fluid" src="](./images/thumbnail_hot_air_ballon.jpg" />

    <clay:sticker label="PDF" position="top-left" style="danger" />
</div>
```

Figure 686.3: You can also have circle stickers.

Figure 686.4: You can specify the position of the sticker within a container.

Now you know how to use Clay stickers in your app!

686.1 Related Topics

- Clay Badges
- Clay Cards
- Clay Icons

CHAPTER 687

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 covers the types of charts you can create with the Chart taglibs. Each article contains a set of chart examples along with sample Java data and a figure displaying the rendered results.

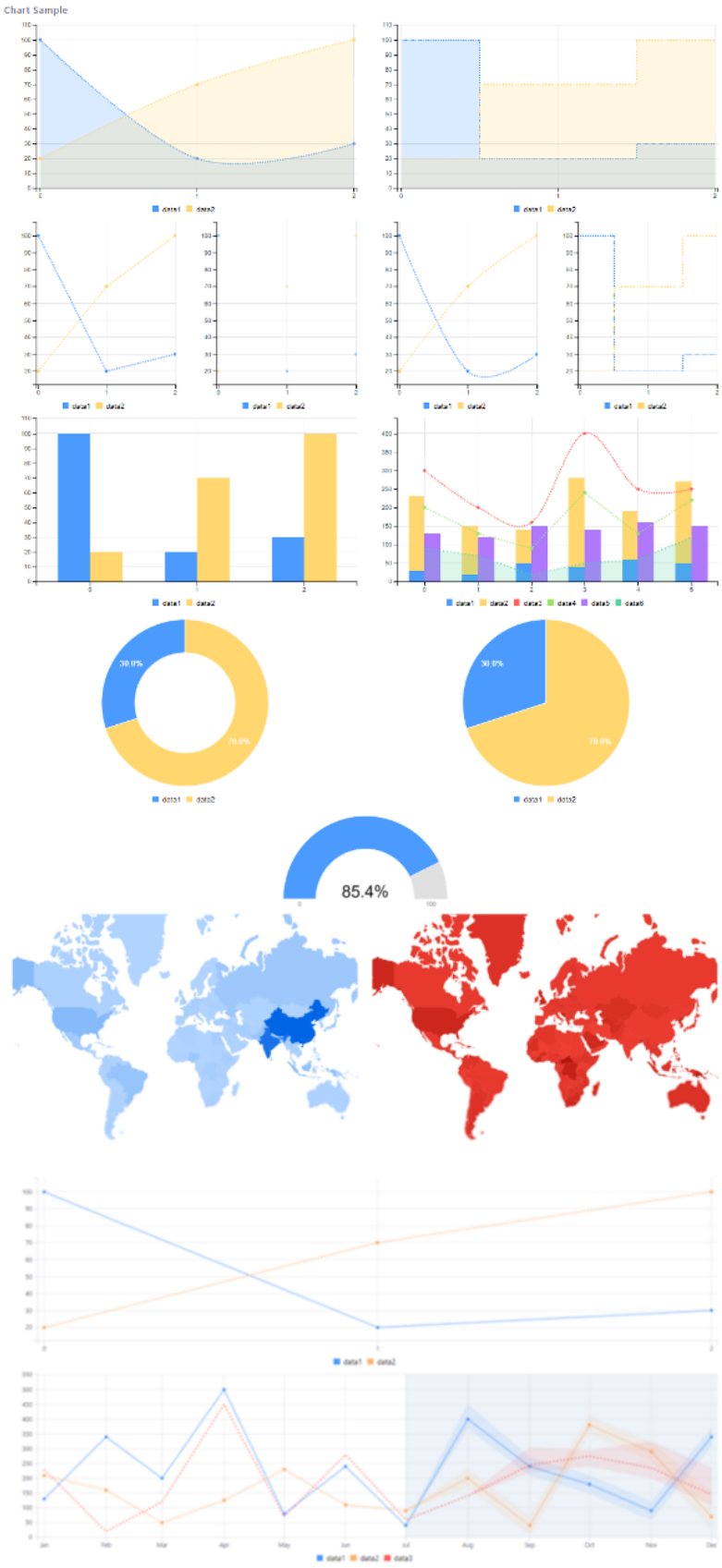

Figure 687.1: You can create many different types of charts with the chart taglibs.

CHAPTER 688

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. Follow these steps to configure your portlet to use bar charts.

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)
);
%>
```

3. Add the `<chart>` taglib to the `view.jsp`, passing the `_barChartConfig` as the `config` attribute's value:

```
<chart:bar
  config="<%= _barChartConfig %>"
/>
```

Awesome! Now you know how to create bar charts for your apps.

688.1 Related Topics

- Line Charts
- Donut Charts
- Combination Charts

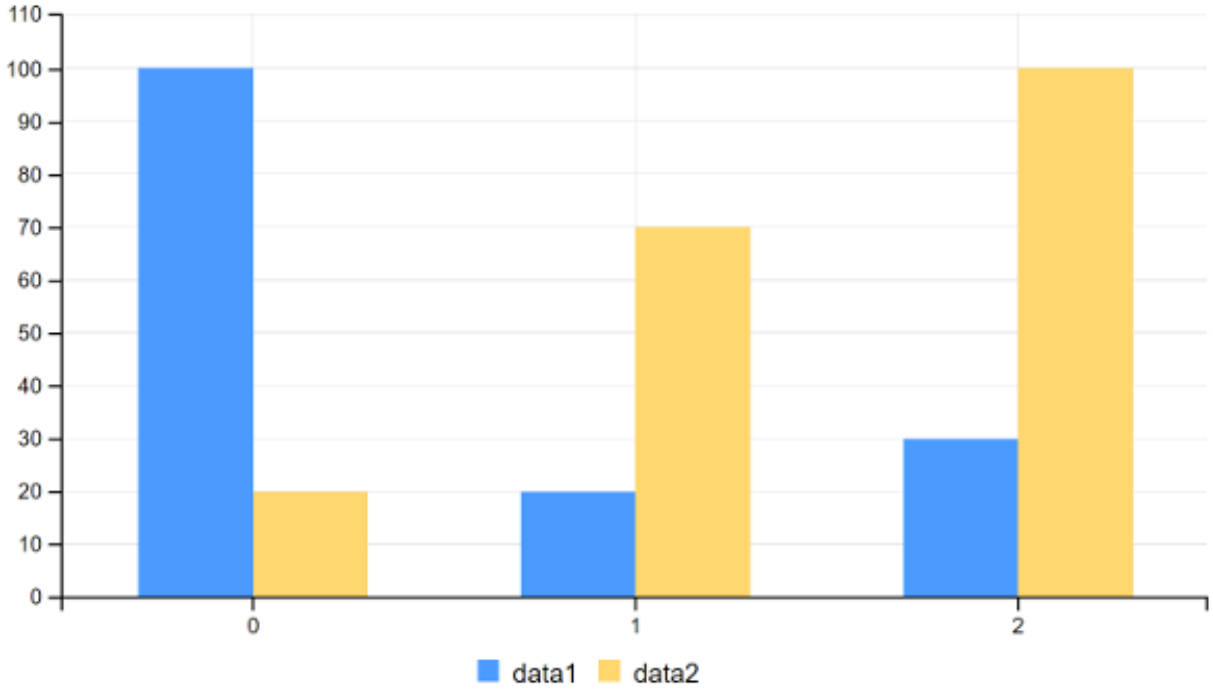

Figure 688.1: A bar chart models the data in bars.

CHAPTER 689

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. Follow these steps to configure your portlet to use line charts.

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 %>"
/>
```

Awesome! Now you know how to create line charts for your apps.

689.1 Related Topics

- Spline Charts
- Step Charts
- Predictive Charts

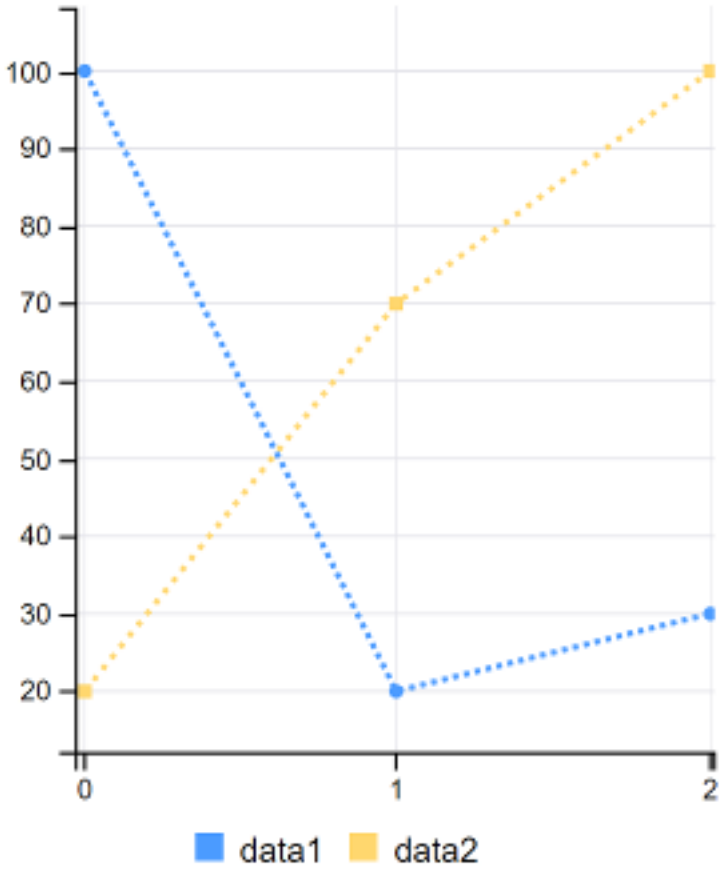

Figure 689.1: A Line chart displays the data linearly.

CHAPTER 690

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. Follow these steps to configure your portlet to use scatter charts.

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`:

```
<%
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 %>"
/>
```

Awesome! Now you know how to create scatter charts for your apps.

690.1 Related Topics

- Line Charts
- Step Charts
- Predictive Charts

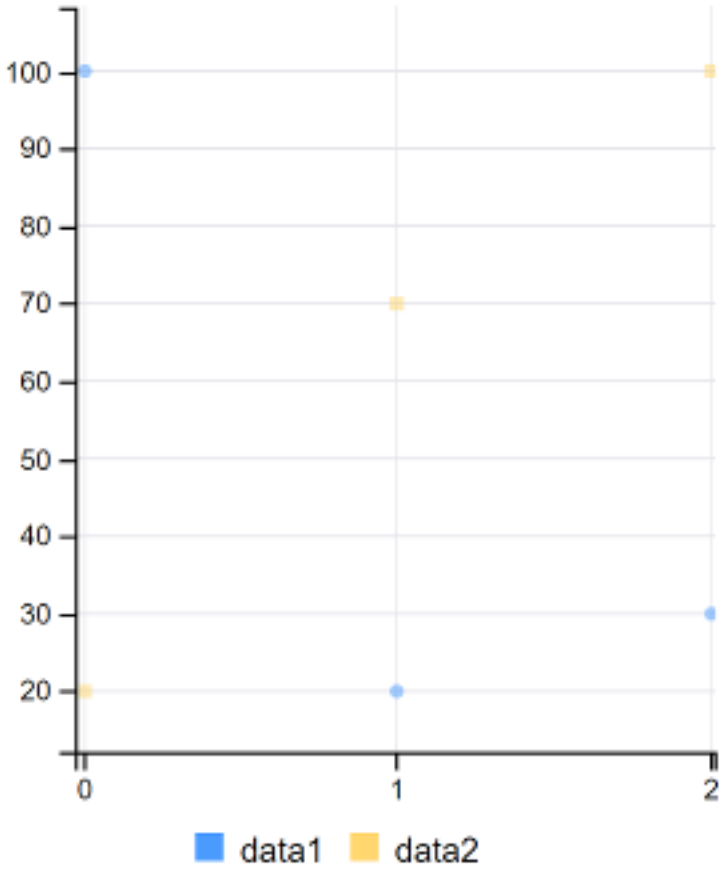

Figure 690.1: A scatter chart models the data as individual points.

CHAPTER 691

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. Follow these steps to configure your portlet to use spline charts.

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 %>"
/>
```

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 %>"
/>
```

Awesome! Now you know how to create spline charts for your apps.

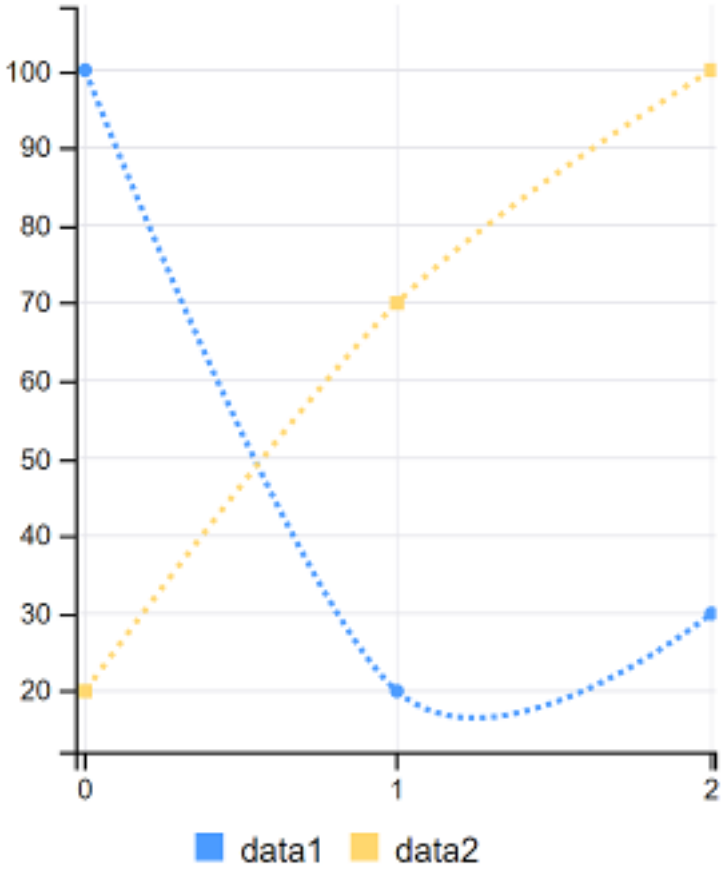

Figure 691.1: A spline chart connects points of data with a smooth curve.

691.1 Related Topics

- Line Charts
- Step Charts
- Scatter Charts

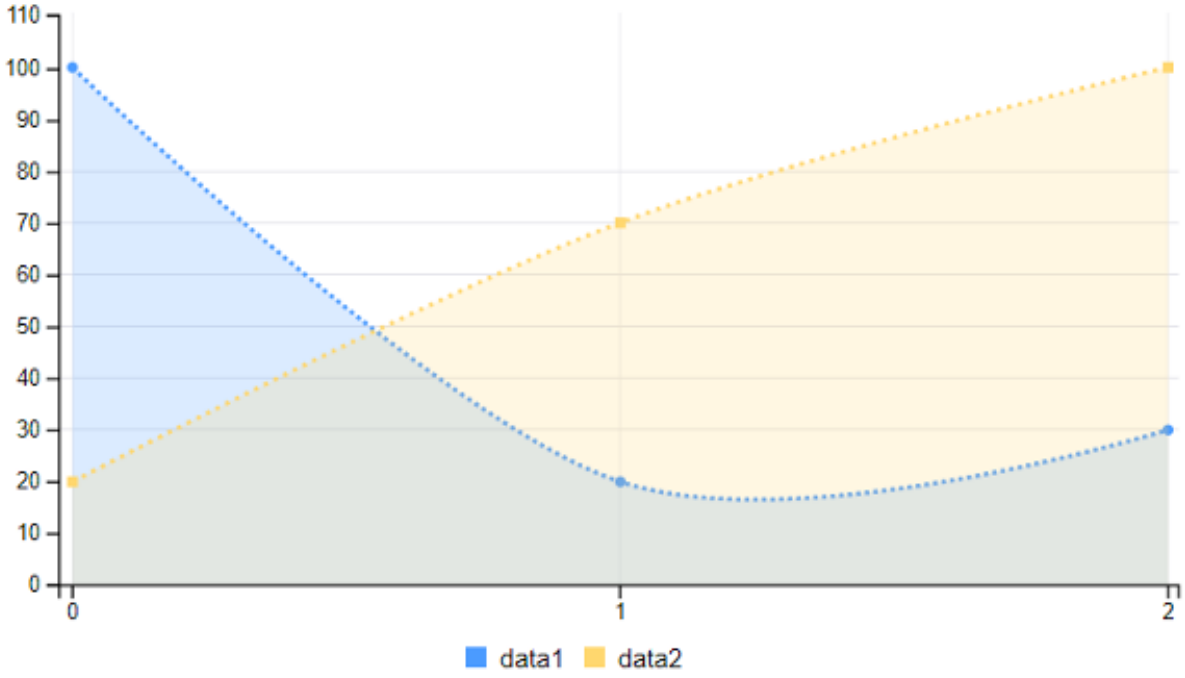

Figure 691.2: An area spline chart highlights the area under the spline curve.

CHAPTER 692

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. Follow these steps to configure your portlet to use step charts.

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 %>"
/>
```

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 %>"
/>
```

Awesome! Now you know how to create step charts for your apps.

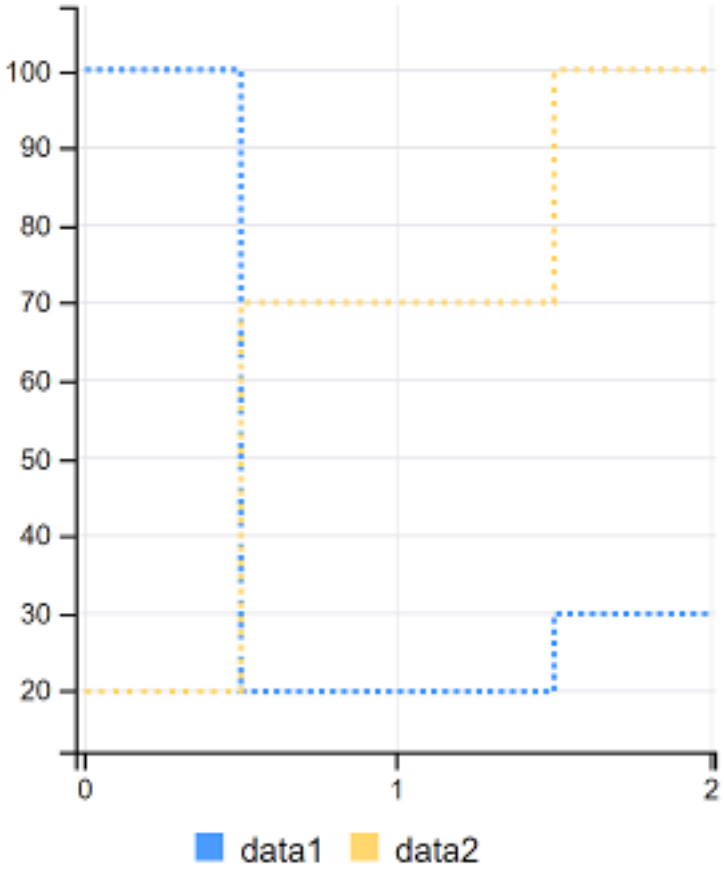

Figure 692.1: A step chart steps between the points of data, resembling steps.

692.1 Related Topics

- Line Charts
- Spline Charts
- Scatter Charts

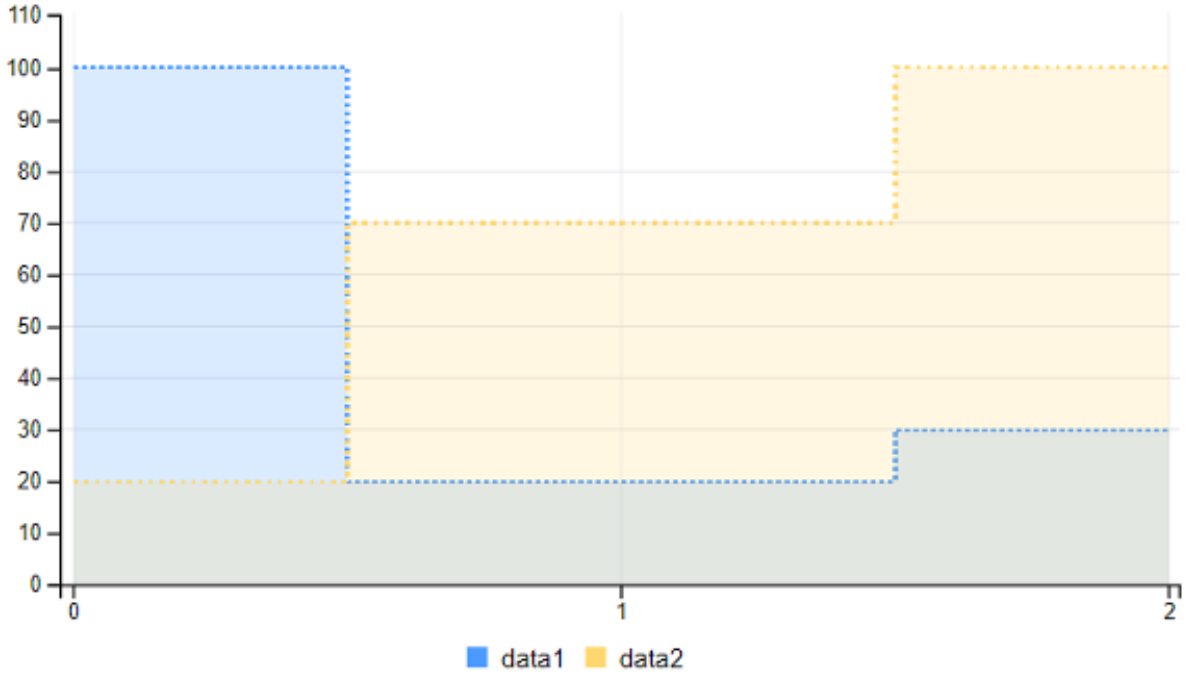

Figure 692.2: An area step chart highlights the area covered by a step graph.

CHAPTER 693

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. Follow these steps to configure your portlet to use combination charts.

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 %>"
/>
```

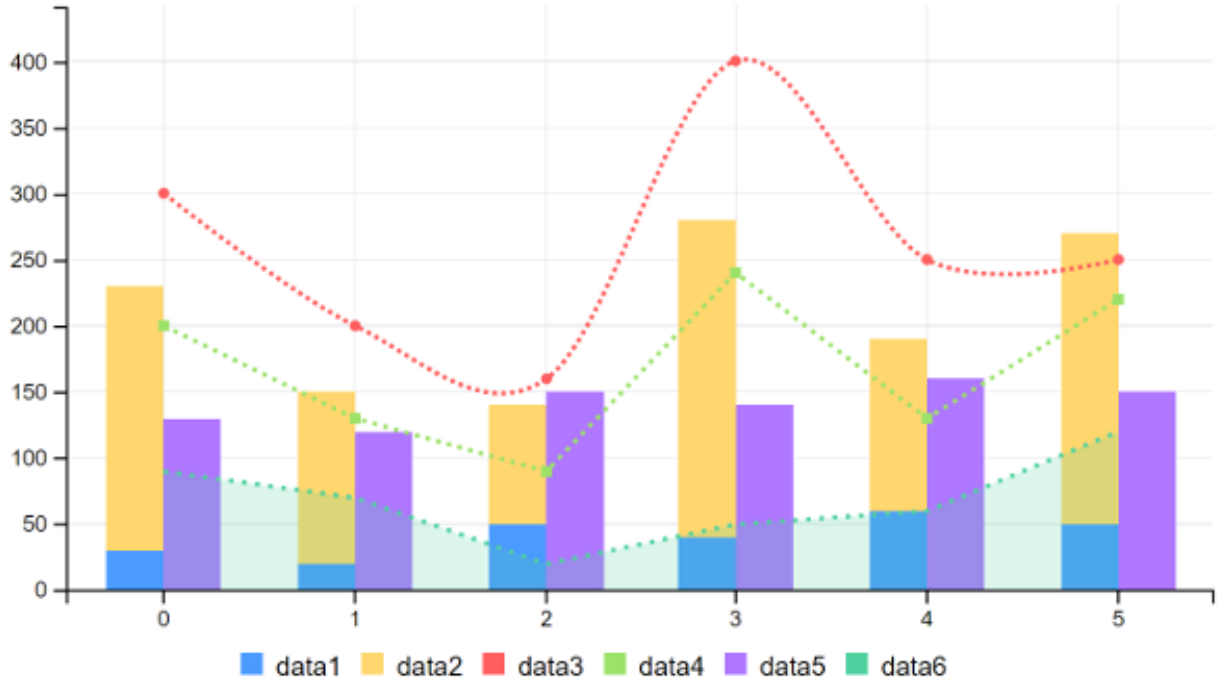

Figure 693.1: A combination chart displays a variety of data set types.

Awesome! Now you know how to create combination charts for your apps.

693.1 Related Topics

- Bar Charts
- Line Charts
- Geomap Charts

CHAPTER 694

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. Follow these steps to configure your portlet to use donut charts.

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 %>"
/>
```

Awesome! Now you know how to create donut charts for your apps.

694.1 Related Topics

- Pie Charts
- Gauge Charts
- Bar Charts

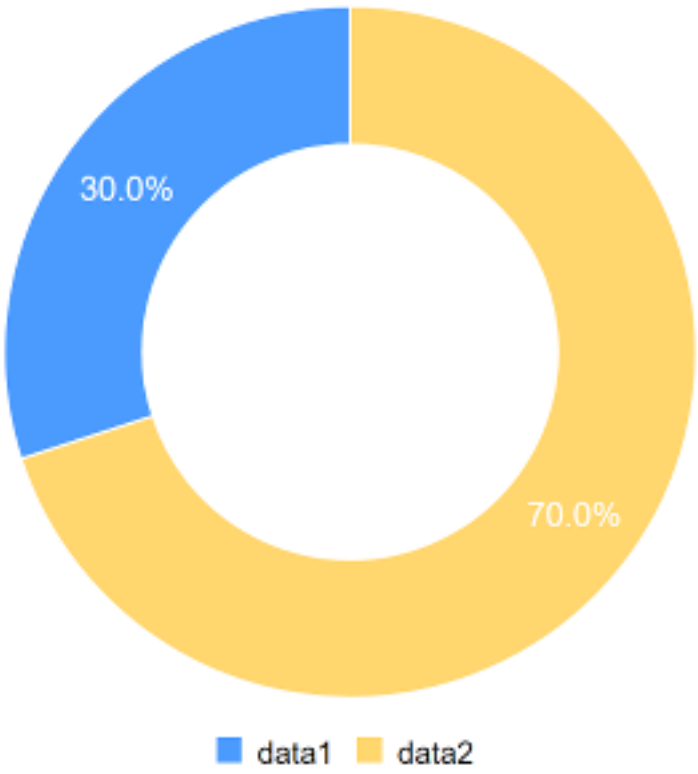

Figure 694.1: A donut chart is similar to a pie chart, but it has a hole in the center.

CHAPTER 695

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. Follow these steps to configure your portlet to use gauge charts.

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 %>"
/>
```

Awesome! Now you know how to create gauge charts for your apps.

695.1 Related Topics

- Pie Charts
- Donut Charts
- Bar Charts

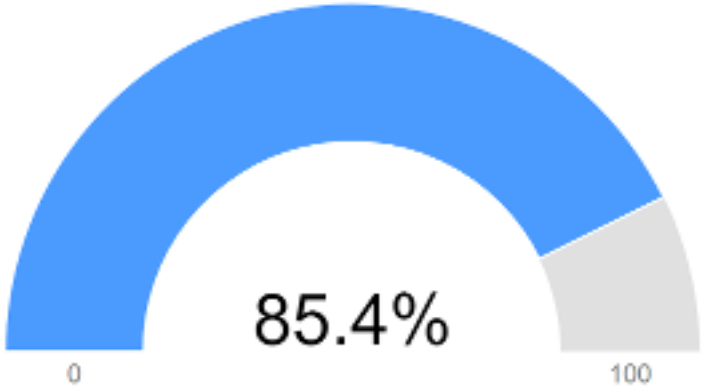

Figure 695.1: A gauge chart shows where percentage-based data falls over a given range.

CHAPTER 696

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. Follow these steps to configure your portlet to use pie charts.

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 %>"
/>
```

Awesome! Now you know how to create pie charts for your apps.

696.1 Related Topics

- Donut Charts
- Gauge Charts
- Spline Charts

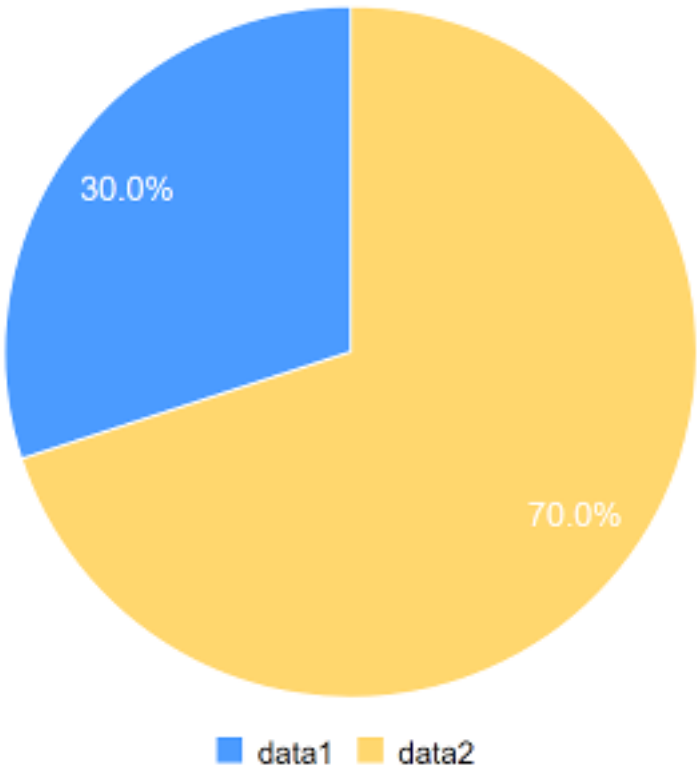

Figure 696.1: A pie chart models percentage-based data as individual slices of pie.

CHAPTER 697

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).

Follow these steps to configure your portlet to use geomap charts.

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");
```

Figure 697.1: A Geomap chart displays a heatmap representing the data.

```
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 697.2: Geomap charts can be customized to fit the look and feel you desire.

Awesome! Now you know how to create geomap charts for your apps.

697.1 Related Topics

- Bar Charts
- Pie Charts
- Combination Charts

CHAPTER 698

PREDICTIVE CHARTS

Predictive charts let you visualize current data along with predicted/forecasted data within a given value range.

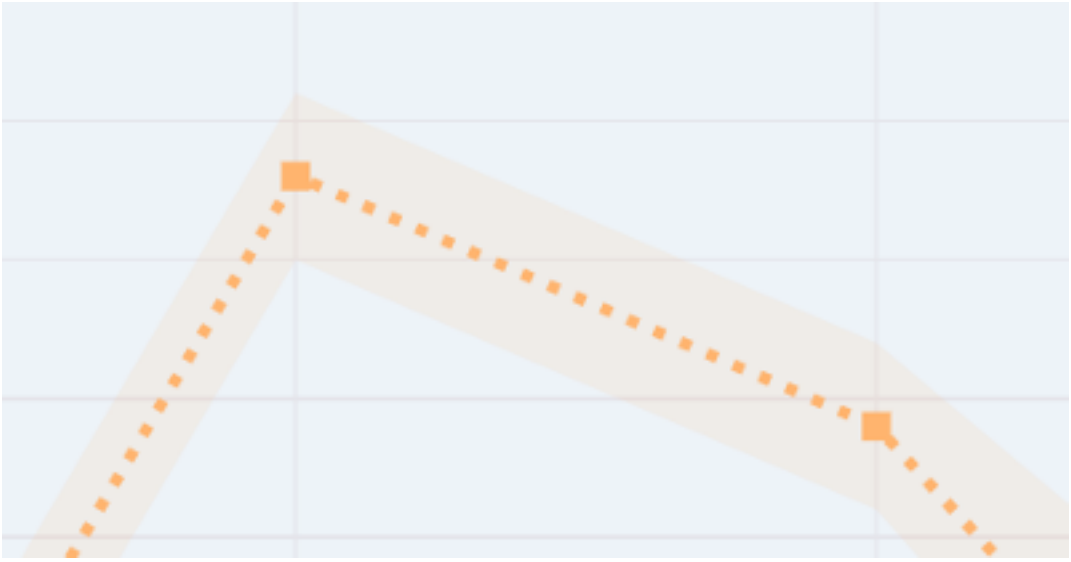

Figure 698.1: Predicted/forecasted data is surrounded by a highlighted area of possible values.

Follow these steps to use predictive charts.

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" %>
```

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 these 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:

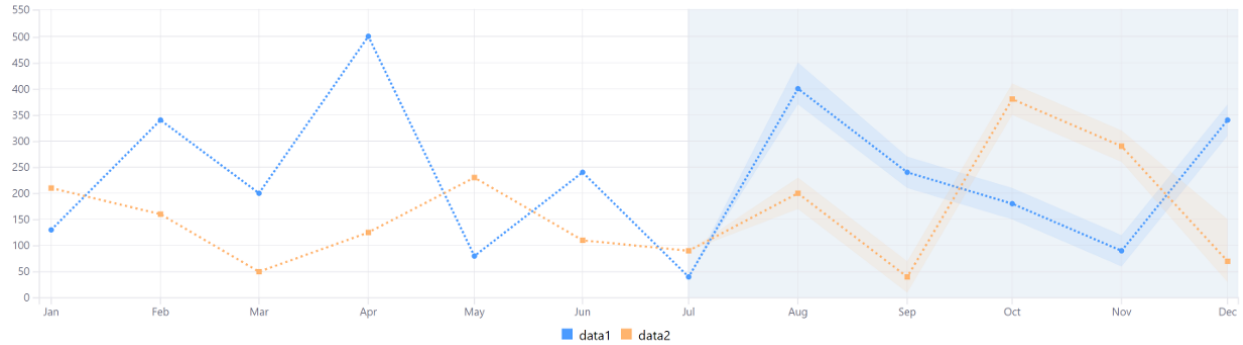

Figure 698.2: A predictive chart lets you visualize estimated future data alongside existing data.

Awesome! Now you know how to create predictive charts for your apps.

698.1 Related Topics

- Line Charts
- Combination Charts
- Geomap Charts

CHAPTER 699

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. Follow these steps to configure your chart to use real time data.

1. Add a new java scriptlet and create a new instance of the chart's object, and put the data into the `data` attribute. Finally, set the chart's polling interval with the `setPollingInterval()` method. An example `view.jsp` configuration is shown below:

```
```java
<%
LineChartConfig _pollingIntervalLineChartConfig = new LineChartConfig();

_pollingIntervalLineChartConfig.put("data", "/foo.json");

_pollingIntervalLineChartConfig.setPollingInterval(2000);
%>
```
```

2. Set the chart taglib's `config` attribute to the updated configuration object that you created in the last step, as shown in the example below:

```
```markup
<chart:line
    componentId="polling-interval-line-chart"
    config="<%= _pollingIntervalLineChartConfig %>"
/>
```
```

Now you know how to reflect real time data in your charts!

699.1 Related Topics

- Bar Charts
- Scatter Charts
- Donut Charts

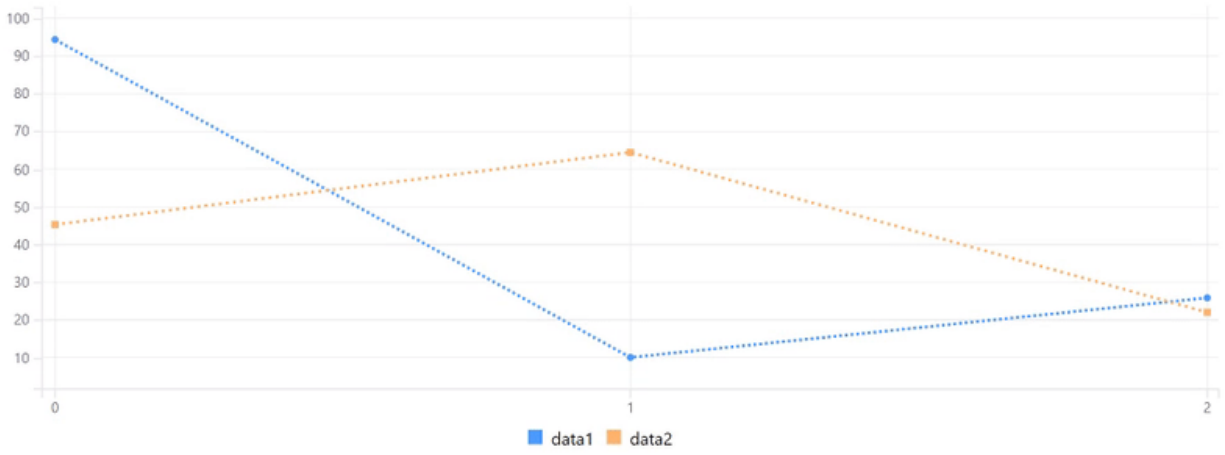

Figure 699.1: The polling interval property lets you refresh charts at a given interval to reflect real time data.

CHAPTER 700

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 covers how to create UI components with the AUI taglibs. Each article contains code examples along with a screenshot of the resulting UI.

CHAPTER 701

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.

Follow these steps to build a form using AUI tags:

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>
```

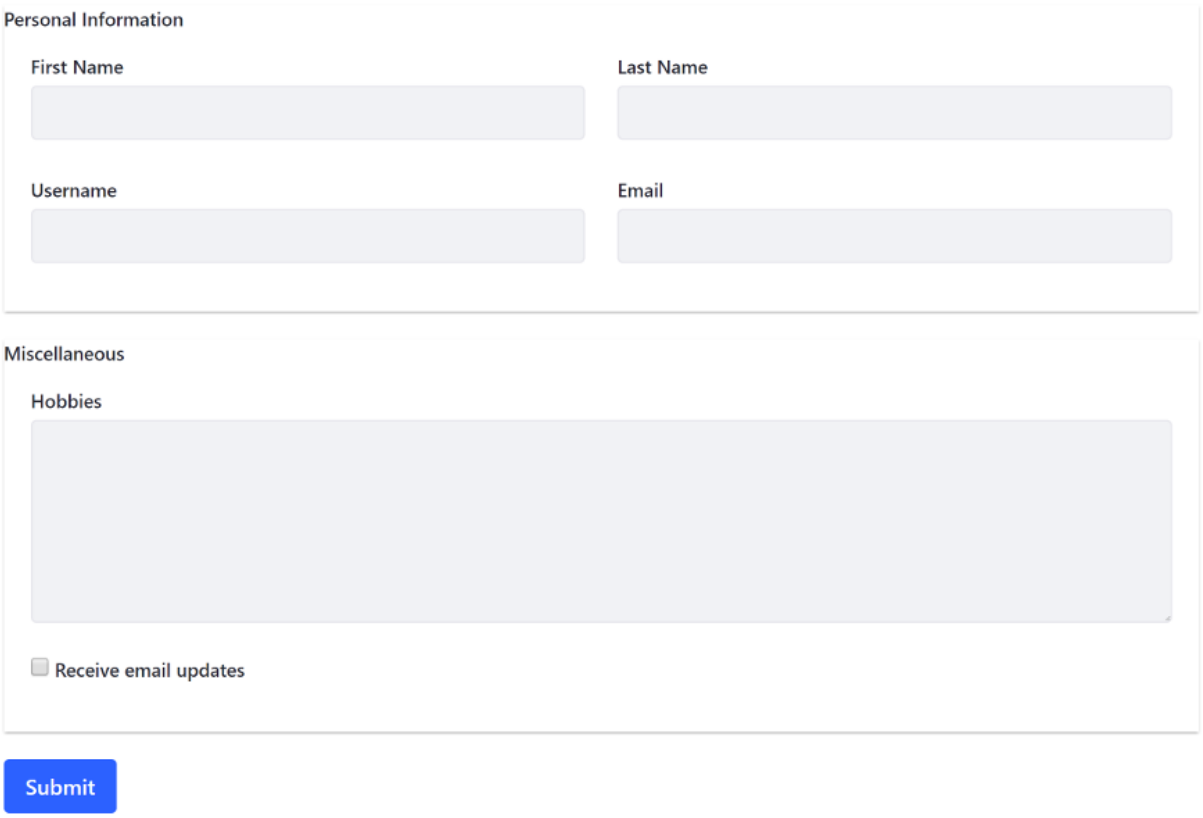

Figure 701.1: The AUI tags provide everything you need to build forms for your applications.

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">
        <aui:validator name="equalTo"
        errorMessage="The passwords much match. Please try again." >
        '#<portlet:namespace>password'
        </aui:validator>
    </aui:input>
</aui:form>
```

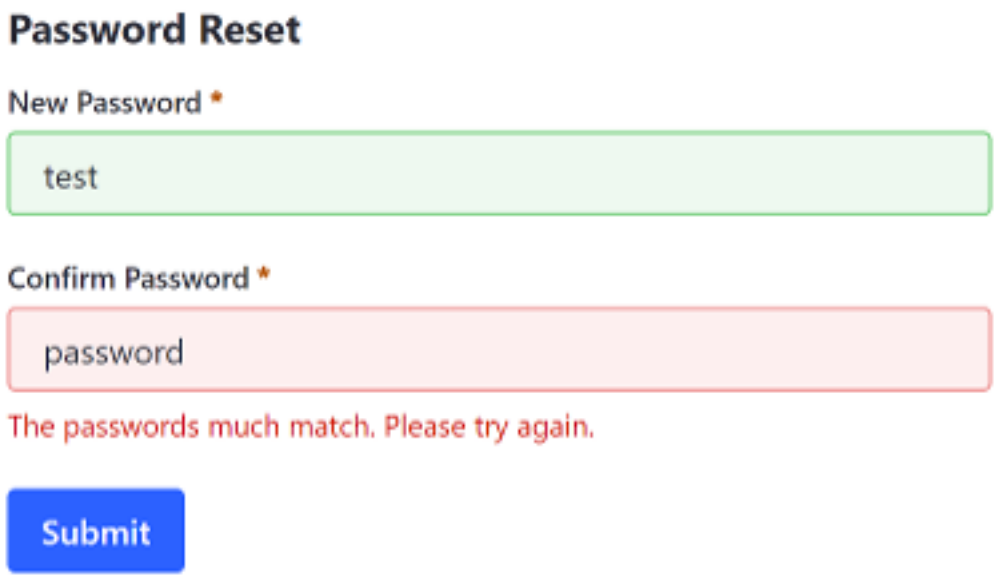

Figure 701.2: The AUI tags also provide validation for form fields.

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]
`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.

701.1 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 702

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 widgets (as opposed to typical web applications).

The workflow for running npm packages inside widgets 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 widgets share common versions of modules instead of each one loading its own copy. The liferay-npm-bundler handles this for you.

Note: You can also find information for the liferay-npm-bundler in the project's Wiki.

702.1 How the Liferay npm Bundler Works Internally

The liferay-npm-bundler takes a widget project and outputs its files (including npm packages) to a build folder, so the standard widget 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.

In this reference section, you'll learn more about the liferay-npm-bundler's configuration, default presets, format, and more.

CHAPTER 703

UNDERSTANDING THE .npmbundlerrc'S STRUCTURE

The liferay-npm-bundler is configured via a `.npmbundlerrc` file placed in the widget 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 Creating JavaScript Widgets with JavaScript Tooling to learn how to use the liferay-npm-bundler along with the Liferay JS Generator to create JavaScript widgets.

703.1 The 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",
"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.

703.2 Standard Configuration Options

Below are the standard configuration options for the `.npmbundlerrc` file:

`config`: Defines the global configuration that is made available to all liferay-npm-bundler and Babel plugins. Please refer to each plugin's documentation to find the available options for each specific plugin.

```
{
  "config": {
    "imports": {
      "vuejs-provider": {
        "vue": "^2.0.0"
      }
    }
  }
}
```

`dump-report`: Sets whether to generate a debugging report. If `true`, a `liferay-npm-bundler-report.html` file is generated in the project directory that describes all actions and decisions taken when processing project and npm modules. 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. All settings provided by the preset are inherited, but they can be overridden.

`verbose`: Sets whether to output detailed information about what the tool is doing to the console. The default value is `false`.

"/": 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).

`packages`: defines plugin configuration for npm packages, per package.

`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`.

`process-serially`: **Note**: removed since v 2.7.0. 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 that defines the loader to use, which can be specified using 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.

Here's an example configuration:

```
{
  "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`: rules apply to files in these project folders. 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"]
}
```

703.4 OSGi Bundle Creation Options

Since version 2.2.0, the liferay-npm-bundler can create widget OSGi bundles for you. See Creating and Bundling JavaScript Widgets with JavaScript Tooling 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, 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 (widget preferences, as defined in the Portlet spec) configuration to use. (see Configuring System Settings and Instance Settings for Your JavaScript Widgets 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": {
    "output-dir": "dist"
  }
}
```

- **create-jar.features.js-extender:** controls whether to process the OSGi bundle with the JS Portlet Extender. 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 use for the bundle (see Providing Localization in Your JavaScript Widgets for more information on using localization in your widget. 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:** **Note** that this option is deprecated. Use the create-jar.features.configuration option instead.

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!

CHAPTER 704

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.

CHAPTER 705

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.

705.1 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 widget, 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!

CHAPTER 706

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.

706.1 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$isobject@2.1.0/index.jsisobject@2.1.0/index.js)
- http://localhost/o/js/resolved-module/my-bundle-package$isarray@1.0.0/index.jsisarray@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.

706.2 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 widget projects below:

- `my-widget`
    - 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-widget
    - 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-widget` gets its `a-helper` at 1.0.0, while `another-widget` 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 widget 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.

706.3 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 widgets that compose the homepage of your site: `my-toolbar`, `my-menu`, and `my-content`. These widgets 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 widget'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 widgets. 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` widget:

```
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.

706.4 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.

CHAPTER 707

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!

CHAPTER 708

UNDERSTANDING HOW LIFERAY AMD LOADER CONFIGURATION IS EXPORTED

NOTE: This article is for users who know how Liferay AMD Loader works under the hood. See Liferay AMD Module Loader for more information.

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.

CHAPTER 709

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.

709.1 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).

709.2 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.

709.3 Improved Peer Dependency Support

In bundler 1.x, there was only one shared peer dependency package available between widgets. With isolated dependencies per widget, 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.

709.4 Manually De-duplicating Through Importing

Namespacing means that each widget 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 widget based architecture, sharing dependencies among different widgets 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 to migrate your 1.x projects to 2.x.

Chapter 710

Understanding liferay-npm-bundler's Loaders

liferay-npm-bundler's mechanism is inspired by webpack. Like webpack, the liferay-npm-bundler processes files using a set of rules that include 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 that 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 pattern to create custom loaders. A few example loader functions are shown below:

- Pass JavaScript 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 using 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 they are listed in the `use` property. The files are passed to the first loader, processed, sent to the next loader, and so on, until the files are processed by the rules. You can 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.

CHAPTER 711

DEFAULT LIFERAY-NPM-BUNDLER LOADERS

Several loaders are available for the liferay-npm-bundler by default:

`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, so you can 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 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.

CHAPTER 712

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.

CHAPTER 713

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 reference 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 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	

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.')
}
```

CHAPTER 714

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.

Note: Since Liferay DXP SP1 and Liferay Portal CE 7.2 GA2, the `Liferay.PortletURL` utilities are deprecated and have been replaced with `Liferay.Util.PortletURL` utilities. We recommend that you use the updated versions to ensure future compatibility. The examples below use the updated utilities.

This covers how to use the `Liferay` global JavaScript object to manipulate URLs. A list of the available methods and properties appears in the tables shown below. Example configurations are shown below the tables.

714.1 Portlet URL Methods

`Liferay.Util.PortletURL` Methods:

Method	Parameters	Returns
`createPortletURL`	`basePortletURL, parameters`	A portlet URL as a URL object
`createActionURL`	`basePortletURL, parameters`	A portlet URL as a URL object
`createRenderURL`	`basePortletURL, parameters`	A portlet URL as a URL object
`createResourceURL`	`basePortletURL, parameters`	A portlet URL as a URL object

714.2 Liferay Util PortletURL

`Liferay.Util.PortletURL` provides APIs for creating portlet URLs (`actionURL`, `renderURL`, and `resourceURL`) with JavaScript in your JSPs. Below is an example configuration for a JSP:

```
var basePortletURL = 'https://localhost:8080/group/control_panel/manage?p_p_id=com_liferay_roles_admin_web_portlet_RolesAdminPortlet';

var actionURL = Liferay.Util.PortletURL.createActionURL(
  basePortletURL,
  {
    'javax.portlet.action': 'addUser',
    foo: 'bar'
  }
);

console.log(actionURL.toString());
// https://localhost:8080/group/control_panel/manage?p_p_id=com_liferay_roles_admin_web_portlet_RolesAdminPortlet&javax.portlet.action=addUser&com_li
```

The same API is available as a module for use in your JavaScript files. The ES6 example below uses the `createActionURL` module:

```
import {createActionURL} from 'frontend-js-web';

var basePortletURL = 'https://localhost:8080/group/control_panel/manage?p_p_id=com_liferay_roles_admin_web_portlet_RolesAdminPortlet';

var actionURL = createActionURL(
  basePortletURL,
  {
    'p_p_id': Liferay.PortletKeys.DOCUMENT_LIBRARY,
    foo: 'bar'
  }
);
```

See the Portlet URL Methods section for more information about the method used in the example above.

714.3 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:

```
import {createActionURL} from 'frontend-js-web';

var basePortletURL = 'https://localhost:8080/group/control_panel/manage?p_p_id=com_liferay_roles_admin_web_portlet_RolesAdminPortlet';

var actionURL = createActionURL(
  basePortletURL,
  {
    'p_auth': Liferay.authToken
  }
);
```

714.4 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"
```

714.5 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"
```

Now you know how to manipulate URLs using methods within the `Liferay` global JavaScript object.

CHAPTER 715

LIFERAY DXP JAVASCRIPT UTILITIES

This reference explains some of the utility methods and objects inside the `Liferay` global JavaScript object.

715.1 Retrieve Browser Information

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

Method	Return Type	Description
isLinux	boolean	Returns whether the browser is being viewed on Linux
isMac	boolean	Returns whether the browser is being viewed on Mac
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

Below is an example configuration:

```
Liferay.Browser.isChrome(); //returns true in Chrome
```

715.2 Format XML

The `Liferay.Util.formatXML` utility takes XML content, as a String, and returns it formatted.

Parameters: - `content`: The XML string to format - `options`: An optional configuration object {} that contains additional parameters for formatting the XML

The default configuration contains these options:

```
const DEFAULT_OPTIONS = {
  newLine: NEW_LINE, //'\r\n'
  tagIndent: TAG_INDENT //'\t'
};
```

Below is an example configuration for a JSP that overwrites the default options:

```
var options = {newLine: '\n', tagIndent: ' '};

var input = `<?xml xlmns:a="http://www.w3.org/TR/html4/" version="1.0" encoding="UTF-8"?>
<!DOCTYPE note>
<a:note>                        <a:to>Foo</a:to>
<a:from>Bar</a:from><a:heading>FooBar</a:heading>
<a:body>FooBarBaz!</a:body>
                </a:note>
```

```
`;

var formattedXMLString = Liferay.Util.formatXML(input, options);

console.log(formattedXMLString);
/*results:
<?xml xlmns:a="http://www.w3.org/TR/html4/" version="1.0" encoding="UTF-8"?>\n'
<!DOCTYPE note>\n'
<a:note>\n'
<a:to>Foo</a:to>\n'
<a:from>Bar</a:from>\n'
<a:heading>FooBar</a:heading>\n'
<a:body>FooBarBaz!</a:body>\n'
</a:note>';
*/
```

715.3 Format Storage Size

The `Liferay.Util.formatStorage` utility takes a storage size number (in bytes) and returns it in the proper format (KB, MB, or GB) as a String.

Parameters: - `size`: The numerical value of the storage size in bytes - `options`: An optional configuration object {} that contains additional parameters for formatting the storage size

The default configuration contains these options:

```
const DEFAULT_OPTIONS = {
  addSpaceBeforeSuffix: false,
  decimalSeparator: '.',
  denominator: 1024.0,
  suffixGB: 'GB',
  suffixKB: 'KB',
  suffixMB: 'MB'
};
```

Below is an example configuration that overwrites some of the default options:

```
var formattedSize = Liferay.Util.formatStorage(1048576, {
  addSpaceBeforeSuffix: true,
  decimalSeparator: ',',
  suffixMB: 'megabytes'
});

console.log(formattedSize); //1,0 megabytes
```

715.4 Store and Retrieve Session Form data

`Liferay.Util.setSessionValue()`: Sets a key, value pair for the Store utility's fetch value.

Parameters: - `key`: The `formData` key (String) - `value`: The `formData` key's corresponding value (Object|String)

`Liferay.Util.getSessionValue()`: Retrieves the Store utility's fetch value for the given `key`.

Parameters: - `key`: The key (String) to fetch the value for.

Below is an example configuration for a JSP:

```
Liferay.Util.Session.set('state', 'open');

Liferay.Util.Session.get('state').then(function(value) {
  console.log(value); //open
});
```

Here is an example configuration that uses ES6:

```
import {getSessionValue, setSessionValue} from 'frontend-js-web';

setSessionValue('state', 'open');

getSessionValue('state').then(value =>{
  console.log(value); //open
});
```

CHAPTER 716

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).

This reference covers how to invoke these web services using JavaScript.

716.1 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.

716.2 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.

716.3 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.

716.4 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.

716.5 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);
	}
);
```

CHAPTER 717

HANDLING AJAX REQUESTS WITH Liferay.Util.fetch

When you make Ajax requests (referred to as Service Resource actions/requests in Liferay DXP), they must protect against CSRF and include the proper credentials. Since Liferay DXP 7.2 SP1 and Liferay CE Portal 7.2 GA2, Liferay DXP provides a `Liferay.Util.fetch` utility based on the standard `fetch` API that you can use to make AJAX requests. It includes these key features:

- A thin wrapper on ES6 `fetch` that shares the same API
- Sets `credentials:include` to each request
- Sets `x-csrf-token` header to each request
- Requires no dependencies

Below is an example configuration in ES6:

```
import {fetch} from 'frontend-js-web';

fetch(url, {
  body: new FormData(form),
  method: 'POST'
})
  .then(response => response.json())
  .then(response => processData(response))
  .then(response => failureCallback(error));
```

Example use case in JSPs:

```
Liferay.Util.fetch(url, {
  body: new FormData(form),
  method: 'POST'
}).then(function(response) {
  return response.json();
}).then(function(response) {
  message.innerHTML = response.message;
}).catch(function() {
  failureCallback();
});
```

NOTE: global access through `Liferay.Util` is only meant for use in JSP code. In ES6, you must use the `fetch` module, as shown in the JavaScript example above.

CHAPTER 718

WORKING WITH ADDRESSES

The `Liferay` global JavaScript Object exposes methods, objects, and properties that access the portal context. The `Liferay.Address` utility contains methods for retrieving information about the addresses country and region. The `Liferay` global object is automatically available at runtime, so no additional dependencies are required.

The available methods are listed below, along with an example configuration.

`Liferay.Address.getCountries(callback)`: returns an Array of the available countries.

Parameters: - `callback`: A callback function to post-process the Array of countries

The example below prints the list of available regions for the selected country (the United States in this case) in a table:

```
Liferay.Address.getCountries(function(e){console.table(e)}, 19);
```

`Liferay.Address.getRegions(callback, selectKey)`: returns an Array of the available regions, by country, for the specified region ID.

Parameters: - `callback`: A callback function to post-process the Array of regions - `selectKey`: The selected region ID

The example below prints the list of available countries in a table to the console:

```
Liferay.Address.getCountries(function(e){console.table(e)});
```

This example uses an AUI `DynamicSelect` module to create a pair of select fields in a JSP. The first field retrieves the countries with the `Liferay.Address.getCountries()` method, and the second select field is dynamically populated with the selected country's available regions with the `Liferay.Address.getRegions()` method:

```
<aui:script use="liferay-dynamic-select">
  new Liferay.DynamicSelect(
    [
      {
        select: '<portlet:namespace />countryId',
        selectData: Liferay.Address.getCountries,
        selectDesc: 'nameCurrentValue',
        selectId: 'countryId',
        selectSort: '<%= true %>',
        selectVal: '<%= countryId %>'
      },
      {
```

```
        select: '<portlet:namespace />regionId',
        selectData: Liferay.Address.getRegions,
        selectDesc: 'name',
        selectDisableOnEmpty: true,
        selectId: 'regionId',
        selectVal: '<%= regionId %>'
      }
    ]
  );
</aui:script>
```

CHAPTER 719

FREEMARKER TAGLIB MACROS

Liferay DXP's taglibs are mapped to FreeMarker macros, so you can use them in your FreeMarker templates. See the Taglib reference 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_sharing`
liferay-sharing
liferay-sharing.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

CHAPTER 720

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
   ```

CHAPTER 721

SITEMAP PAGE CONFIGURATION OPTIONS

If you're importing resources with your themes, you must define the pages for the site in the theme's `sitemap.json`. Below is the full list of available configuration options for pages in the theme's `sitemap.json`:

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`.

CHAPTER 722

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
```

CHAPTER 723

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

CHAPTER 724

AVAILABLE 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 available lifecycle events are listed in the table below:

Event	Description	Ex payload
`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.	`{ path: '/pages/page1.html', replaceHistory: false }`
`startNavigate`	Fires when navigation begins	‘{ form: ’

replaceHistory: false }‘ |

`endNavigate` | Fired after the content has been retrieved and inserted onto the page | `{ 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.

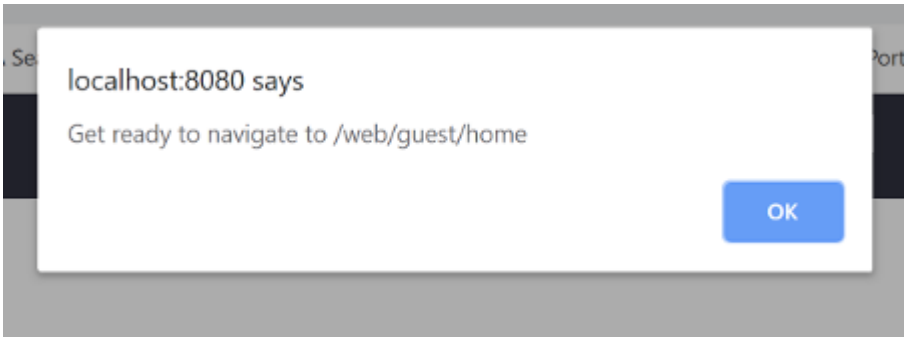

Figure 724.1: You can leverage SPA lifecycle events in your apps.

CHAPTER 725

THEME ANATOMY 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.

Themes built with the Liferay JS Theme Toolkit have the anatomy shown below:

- 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.

725.1 Theme Files

725.2 _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.

725.3 _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.

725.4 _custom.scss

Used for custom CSS styles. You should place all of your custom CSS modifications in this file.

725.5 _liferay_variables_custom.scss

Used for overwriting variables defined in _liferay_variables.scss without wiping out the whole file.

725.6 init_custom.ftl

Used for custom FreeMarker variables i.e. theme setting variables.

725.7 navigation.ftl

The theme template for the theme's navigation.

725.8 portal_normal.ftl

Similar to a static site's `index.html`, this file acts as a hub for all theme templates.

725.9 portal_pop_up.ftl

The theme template for pop up dialogs for the theme's portlets.

725.10 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.

725.11 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.

725.12 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.

725.13 main.js

Used for custom JavaScript.

725.14 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.

725.15 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 7.2 Properties documentation.

CHAPTER 726

FREEMARKER VARIABLE REFERENCE GUIDE

By default, FreeMarker templates have access to several variables defined in `init.ftl` that you can use in your themes to access several theme objects, settings, and resources. Several of these variables are listed below for reference:

Common Variables

Variable	Description
`theme_display`	Returns the `themeDisplay` Java Object and all its methods
`portlet_display`	Returns the `portletDisplay` Java Object and all its methods
`layoutSet`	Returns the page set
`theme_timestamp`	Prints the date in the current locale with the given format
`theme_settings`	Retrieves theme settings. See configurable theme settings for more information.
`root_css_class`	Returns the root CSS class which indicates the direction of the page (`ltr` (left-to-right) by default)
`css_class`	Returns a string of the current classes applied to the body of the page
`page_group`	Retrieves the page group
`css_folder`	Returns the path to the theme's `css` folder
`images_folder`	Returns the path to the theme's `images` folder
`javascript_folder`	Returns the path to the theme's `javascript` folder
`templates_folder`	Returns the path to the theme's `templates` folder
`full_css_path`	Returns the full path, which includes the servlet context, to the theme's `css`
`full_templates_path`	returns the full path, which includes the servlet context, to the theme's `templates`
`css_main_file`	Returns the path to `main.css`

Variable	Description
`js_main_file`	Returns the path to `main.js`
`company_id`	Returns the company ID
`company_name`	Returns the company name
`company_logo`	Returns the company logo's URL
`company_logo_height`	Returns the company logo's height
`company_logo_width`	Returns the company logo's width
`company_url`	Returns the URL of the home page for the company
`time_zone`	Returns the time zone for the current user
`is_login_redirect_required`	Returns whether a login redirect is required for the user
`is_signed_in`	Returns whether the user is signed in
`group_id`	Returns the group ID for the current user
`time_zone`	Returns the time zone for the current user
`is_default_user`	Returns if the user has a default role
`is_female`	Returns if the current user is Female
`is_male`	Returns if the current user is Male
`is_setup_complete`	Returns whether the user has configured their profile
`language`	Returns the native language for the current user
`language_id`	Returns the ID of the current locale
`user_birthday`	Returns the current user's birthday
`user_comments`	Returns comments from the user's profile
`user_email_address`	Returns the user's email address
`user_first_name`	Returns the user's first name
`user_greeting`	Returns the user's greeting
`user_id`	Returns the ID of the current user
`user_last_login_ip`	Returns the IP address that the user last logged in from
`user_last_name`	Returns the last name of the current user
`user_login_ip`	Returns the current user's current IP address
`user_middle_name`	Returns the user's middle name
`user_name`	Returns the current user's username
`w3c_language_id`	Returns the W3C language code of the current language

URLs

Variable	Description
`show_control_panel`	Returns whether the current user has permission to view the Control Panel

Variable	Description
control_panel_text	Returns the “control-panel” language key in the current user’s locale, if they have permission to view the Control Panel
control_panel_url	Returns the URL to the Control Panel, if the current user has permission to view the Control Panel
show_home	Returns whether the current user is on a page
home_text	Returns the “home” language key in the current user’s locale
home_url	Returns the URL to the home page
show_my_account	Returns whether the current user’s account icon is visible
my_account_text	Returns the “my-account” language key in the current user’s locale, if the user’s account icon is visible
my_account_url	Returns the URL to the user’s Account Settings page if the user’s account icon is visible
show_sign_in	Returns whether the sign in link is visible
sign_in_text	Returns the “sign-in” language key in the current user’s locale, if they are signed out
sign_in_url	Returns the sign in URL, if the current user is signed out
show_sign_out	Returns whether the sign out link is visible
sign_out_text	Returns the “sign-out” language key in the current user’s locale, if they are signed in
sign_out_url	Returns the sign out URL, if the current user is signed in

Page

Variable	Description
the_title	Returns the current page’s title
selectable	Returns whether the current page is selectable
is_maximized	Returns whether the page is maximized
page	Returns the current page (layout)
is_first_child	Returns whether the current page is the first child page in the navigation
is_first_parent	Returns whether the current page is the first parent page in the navigation
is_portlet_page	Returns whether the current page is a widget page (portlet)
site_name	Returns the site’s name

Variable	Description
`is_guest_group`	Returns whether the current page group is for guests
`site_type`	Returns the type of the current site: site, company site, organization site, or user site
`site_default_url`	Returns the default URL for the site
`layout_friendly_url`	Returns the friendly URL of the current page
`portlet_id`	Returns the portlet ID for the specified portlet

Logo

Variable	Description
`logo_css_class`	Returns a string of the current classes applied to the logo.
`use_company_logo`	Returns whether the logo is displayed
`site_logo_height`	Returns the logo's height
`site_logo_width`	Returns the logo's width
`show_site_name_supported`	Returns whether the logo is configured to show the site name. The value is `true` if `show_site_name_default` is true.
`show_site_name_default`	Returns whether the Show Site Name Default theme setting is enabled
`show_site_name`	Returns whether the `showSiteName` property for the current pageset is enabled
`logo_description`	Returns the Site's name or nothing if `show_site_name` is enabled. It is used for alternate text for the logo by default.

Navigation

Variable	Description
`has_navigation`	Returns whether navigation exists (i.e. at least one page exists)
`nav_items`	Returns the current pages as list
`nav_css_class`	Returns a string of the current classes applied to the page's navigation

My Sites

Variable	Description
show_my_sites	Returns whether the current user has a My Sites page
show_my_places	Returns whether the current user has a My Sites page
my_sites_text	Returns the “my-sites” language key in the current user’s locale
my_places_text	Returns whether the current user has a My Sites page

Includes

Variable	Description
dir_include	Returns “/html”
body_bottom_include	Returns “$dir_include}/common/themes/body_bottom.jsp"|`body_top_include}/common/themes/body_top.jsp”
bottom_include	Returns “$dir_include}/common/themes/bottom.jsp"|`top_head_include}/common/themes/top_head.jsp”
top_messages_include	Returns “${dir_include}/common/themes/top_messages.jsp”

Date

Variable	Description
date	Gives access to the dateUtil Java Object and all its methods
current_time	Returns the current time
the_year	Returns the current year

CHAPTER 727

GRADLE PLUGINS

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.

CHAPTER 728

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.

728.1 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.

728.2 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.

728.3 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`

CHAPTER 729

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.

729.1 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()
}
```

729.2 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.

729.3 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`).

729.4 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.

729.5 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.

729.6 Additional Configuration

There are additional configurations that can help you baseline your OSGi bundle.

729.7 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.

729.8 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
```

CHAPTER 730

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.

730.1 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"
```

730.2 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]`

730.3 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.

CHAPTER 731

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.

731.1 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"
    }
}
```

731.2 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

731.3 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.

731.4 Additional Configuration

There are additional configurations that can help you use the CSS Builder.

731.5 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"
}
```

731.6 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"
}
```

CHAPTER 732

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.

732.1 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"
    }
}
```

732.2 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"`

732.3 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.

732.4 Additional Configuration

There are additional configurations that can help you use the Deployment Helper.

732.5 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"
}
```

732.6 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"
}
```

CHAPTER 733

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.

733.1 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"
```

733.2 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

733.3 Additional Configuration

There are additional configurations that can help you use the Deployment Helper.

733.4 Project Properties

It is possible to set the default values of the `ignoreFailures` property via the project property `dependencyCheckerIgnoreFailures`:

```
-PdependencyCheckerIgnoreFailures=false
```

CHAPTER 734

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.

734.1 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"
    }
}
```

734.2 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.

734.3 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`.

734.4 Additional Configuration

There are additional configurations that can help you use the Deployment Helper.

734.5 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"
}
```

CHAPTER 735

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.

735.1 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"
```

735.2 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.

735.3 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.

735.4 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.

735.5 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.

735.6 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}"]`

CHAPTER 736

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.

736.1 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.

736.2 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.

736.3 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.

CHAPTER 737

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.

737.1 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"
    }
}
```

737.2 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`

737.3 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`.

737.4 Additional Configuration

There are additional configurations that can help you use Jasper JSPC.

737.5 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.

737.6 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"
}
```

CHAPTER 738

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.

738.1 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"
    }
}
```

738.2 Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `formatJavadoc` | - | `FormatJavadocTask` | Runs the Liferay Javadoc Formatter to format files.

738.3 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.

738.4 Additional Configuration

There are additional configurations that can help you use the Javadoc Formatter.

738.5 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.

738.6 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`

CHAPTER 739

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.

739.1 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.

739.2 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.

739.3 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.

739.4 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.

CHAPTER 740

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.

740.1 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"
  ```

740.2 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.

740.3 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.

740.4 Tasks

740.5 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.

CHAPTER 741

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.

741.1 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.

741.2 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`).

741.3 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.

741.4 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.

741.5 Tasks

741.6 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`.

CHAPTER 742

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.

742.1 Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.lang.builder", version: "3.0.12"
    }

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

apply plugin: "com.liferay.lang.builder"
```

Since the plugin automatically resolves the Liferay Lang Builder library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

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

See this page on the *Liferay Developer Network* for more information about usage of the Lang Builder Gradle plugin.

742.2 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`

742.3 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.

742.4 Additional Configuration

There are additional configurations that can help you use the Lang Builder.

742.5 Liferay Lang Builder Dependency

By default, the plugin creates a configuration called `langBuilder` and adds a dependency to the latest released version of the Liferay Lang Builder. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `langBuilder` configuration:

```
dependencies {
    langBuilder group: "com.liferay", name: "com.liferay.lang.builder", version: "1.0.31"
}
```

CHAPTER 743

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.

743.1 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"
```

743.2 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.

743.3 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.

743.4 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.

743.5 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.

743.6 Additional Configuration

There are additional configurations that can help you use the Maven Plugin Builder.

743.7 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"
}
```

743.8 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
```

CHAPTER 744

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.

744.1 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"
```

744.2 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
    }
}
```

744.3 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.

744.4 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.

744.5 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.

744.6 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.

744.7 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.

744.8 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.

744.9 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`.

744.10 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.

744.11 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.

744.12 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.

CHAPTER 745

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.

745.1 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"
    }
}
```

745.2 Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `buildREST` | - | `BuildRESTTask` | Runs the Liferay REST Builder.

745.3 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.

745.4 Additional Configuration

There are additional configurations added to use REST Builder.

745.5 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"
}
```

CHAPTER 746

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.

746.1 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"
    }
}
```

746.2 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.

746.3 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.

746.4 Additional Configuration

There are additional configurations that can help you use Service Builder.

746.5 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.

CHAPTER 747

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.

747.1 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"
    }
}
```

747.2 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.

747.3 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.

747.4 Additional Configuration

There are additional configurations that can help you use the Source Formatter.

747.5 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"
}
```

747.6 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
```

CHAPTER 748

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.

748.1 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"
    }
}
```

748.2 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: `[]`

748.3 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"
}
```

748.4 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: []

748.5 Tasks

748.6 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.

748.7 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.

748.8 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.

CHAPTER 749

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.

749.1 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: "2.0.0"
    }

    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. If you have a multi-module Gradle project, you only need to apply these plugins to the root 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"
    }
}
```

749.2 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.

749.3 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).

749.4 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.

749.5 Tasks

749.6 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.

749.7 Additional Configuration

There are additional configurations that you can use to configure the target platform.

749.8 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.portal", name: "release.portal.bom", version: "7.2.0"
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom.compile.only", version: "7.2.0"
}
```

749.9 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"
}
```

749.10 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.portal", name: "release.portal.distro", version: "7.2.0"
}
```

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")
}
```

749.11 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"
}
```

CHAPTER 750

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.

750.1 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"
    }
}
```

750.2 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.

750.3 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.

750.4 Additional Configuration

There are additional configurations that can help you use the CSS Builder.

750.5 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"
}
```

750.6 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.

CHAPTER 751

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.

751.1 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"
    }
}
```

751.2 Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `formatTLD` | - | `FormatTLDTask` | Runs the Liferay TLD Formatter to format files.

751.3 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.

751.4 Additional Configuration

There are additional configurations that can help you use the TLD Formatter.

751.5 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"
}
```

CHAPTER 752

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.

752.1 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"
    }
}
```

752.2 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>
```

752.3 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

752.4 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.

752.5 Tasks

752.6 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.

752.7 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.

752.8 Additional Configuration

There are additional configurations that can help you use the TLDDoc Builder.

752.9 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"
}
```

CHAPTER 753

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.

753.1 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.

753.2 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.

753.3 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.

753.4 Additional Configuration

There are additional configurations that can help you use Whip.

753.5 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.

CHAPTER 754

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.

754.1 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"
    }
}
```

754.2 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
```

754.3 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.

754.4 Additional Configuration

There are additional configurations that can help you use the WSDD Builder.

754.5 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"
}
```

CHAPTER 755

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.

755.1 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"
    }
}
```

755.2 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.

755.3 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.

755.4 Additional Configuration

There are additional configurations that can help you use WSDL Builder.

755.5 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.

CHAPTER 756

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.

756.1 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"
    }
}
```

756.2 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`).

756.3 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.

756.4 Additional Configuration

There are additional configurations that can help you use the XML Formatter.

756.5 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"
}
```

CHAPTER 757

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.

757.1 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"
    }
}
```

757.2 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.

757.3 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.

757.4 Additional Configuration

There are additional configurations that can help you use the XSD Builder.

757.5 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 758

LIFERAY FACES

Liferay Faces is an umbrella project that provides support for the JavaServer™ Faces (JSF) standard within Liferay DXP. 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.

CHAPTER 759

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.

759.1 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.

759.2 Liferay Faces Alloy

Provides a suite of JSF components that utilize AlloyUI.

Branch|Example Artifact|AlloyUI|JSF API|Additional Info| master (4.x)|com.liferay.faces.alloy-4.1.0.jar|3.1.x|2.2+|*AlloyUI 3.1.x is the version that comes bundled with Liferay Portal 7.3.*| 3.x|com.liferay.faces.alloy-3.1.0.jar|3.0.x|2.2+|*AlloyUI 3.0.x is the version that comes bundled with Liferay Portal 7.0/7.1/7.2.*| 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.*|

759.3 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|*Under "Final Review" by the JCP and scheduled for release in 2020.*| 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.*|

759.4 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|

759.5 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 | Portlet API| JSF API| 6.x|com.liferay.faces.portal-6.0.0.jar|7.2+|3.0|2.3| 5.x|com.liferay.faces.portal-5.0.0.jar|7.2+|3.0|2.2| 4.x|com.liferay.faces.portal-4.0.0.jar|7.2/7.3|2.0|2.2| 3.x|com.liferay.faces.portal-3.0.1.jar|7.0/7.1/7.2|2.0|2.2| 2.x|com.liferay.faces.portal-2.0.1.jar|6.2|2.0|2.1/2.2| 1.x|com.liferay.faces.portal-1.0.1.jar|6.2|2.0|1.2|

759.6 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.

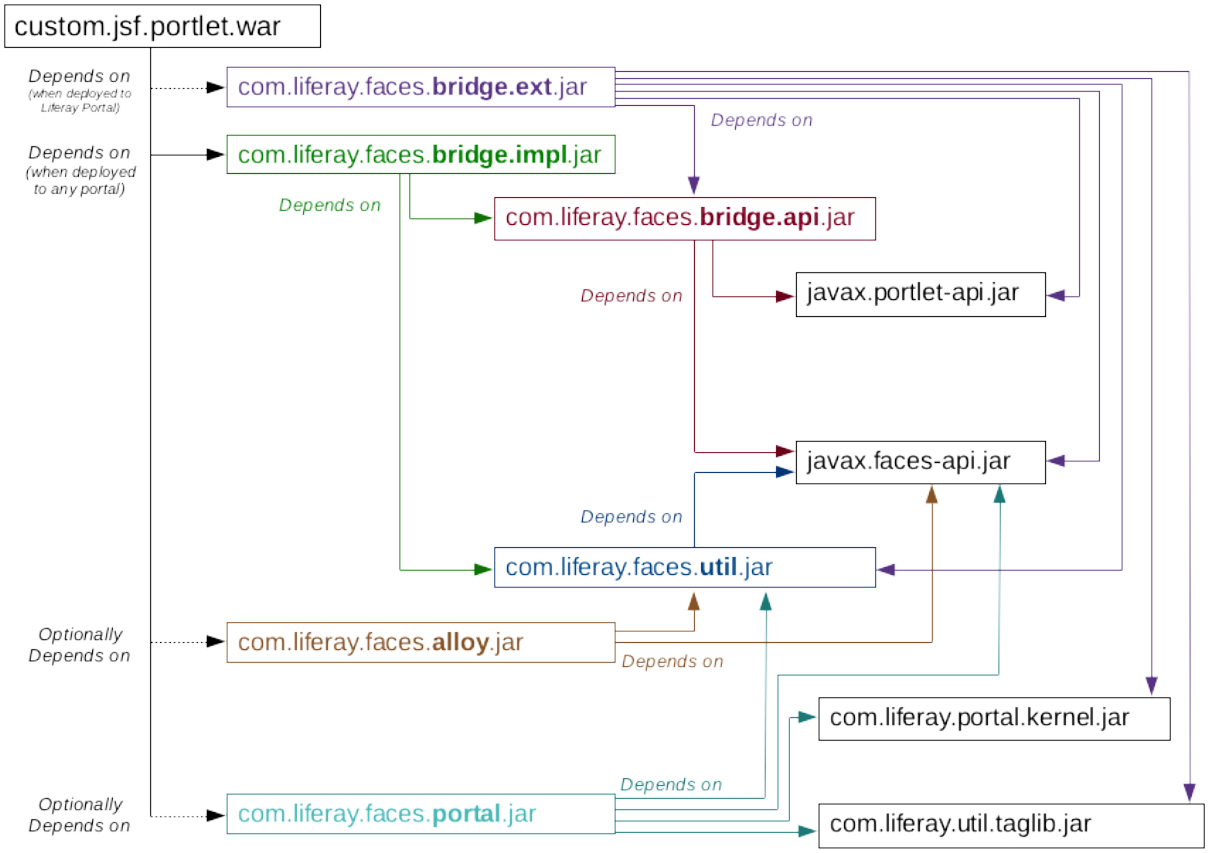

Figure 759.1: The Liferay Faces dependency diagram helps visualize how components interact and depend on each other.

Next, you can view some example configurations to see the new versioning scheme in action.

CHAPTER 760

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 performed 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.

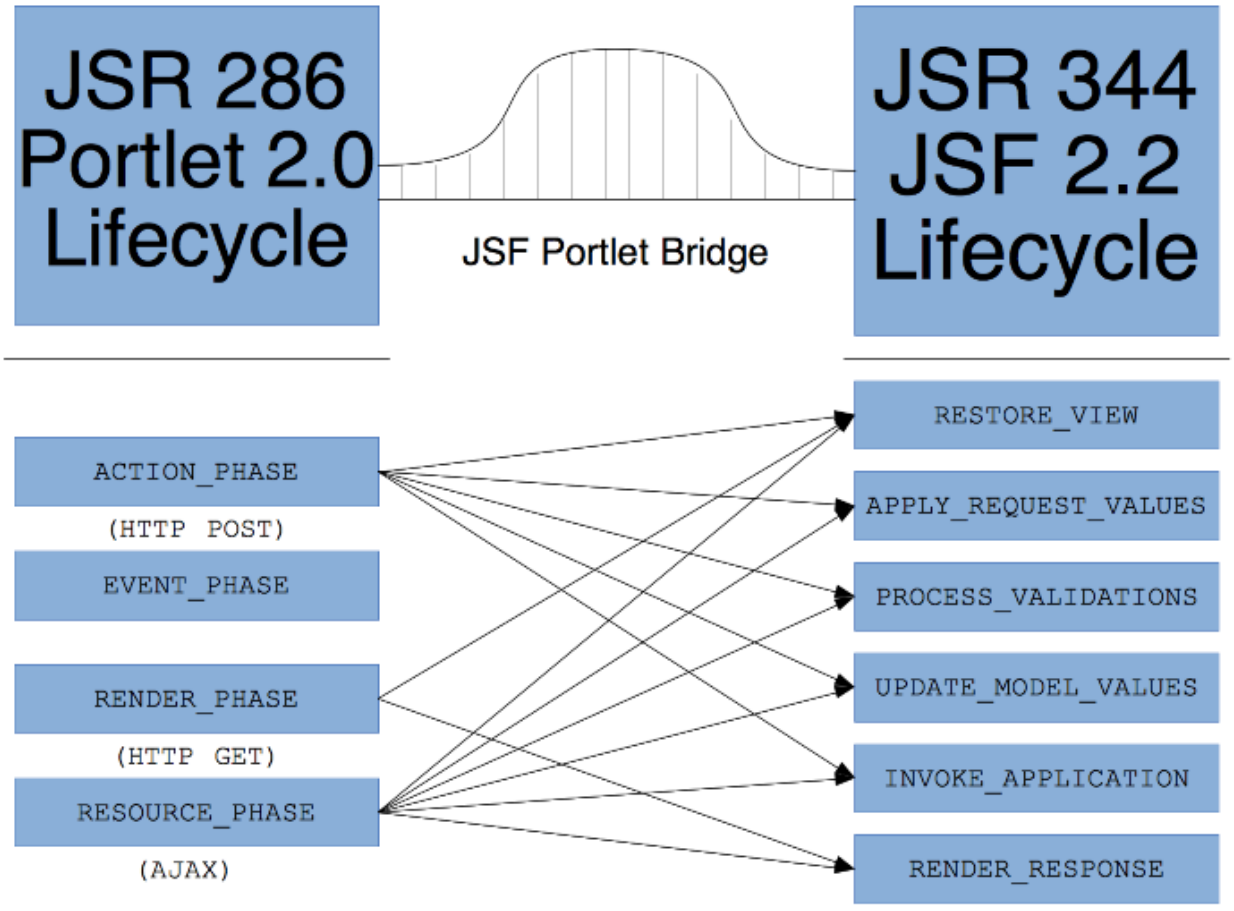

Figure 760.1: The different phases of the JSF Lifecycle are executed depending on which phase of the Portlet lifecycle is being executed.

Besides ensuring that the two lifecycles connect correctly, the JSF portlet bridge also acts as a mediator between the portal URL generator and JSF navigation rules. JSF portlet bridges ensure that URLs created by the portal comply with JSF navigation rules, so that a JSF portlet is able to switch to different views.

The JSR 329/378 standards defines several configuration options prefixed with the `javax.portlet.faces` namespace. Liferay Faces Bridge defines additional implementation-specific options prefixed with the `com.liferay.faces.bridge` namespace.

Liferay Faces Bridge is an essential part of the JSF development process for Liferay DXP. Visit the JSF Portlets with Liferay Faces section of tutorials for more information on JSF development for Liferay DXP.

760.1 Related Topics

Understanding Liferay Faces Alloy
    Understanding Liferay Faces Portal
    Service Builder

CHAPTER 761

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, to use AlloyUI in a way that is consistent with JSF development.

Note: AlloyUI is deprecated in Liferay DXP 7.2.

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.

761.1 Related Topics

Developing a JSF Portlet Application
    Understanding Liferay Faces Bridge
    Understanding Liferay Faces Portal

CHAPTER 762

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.

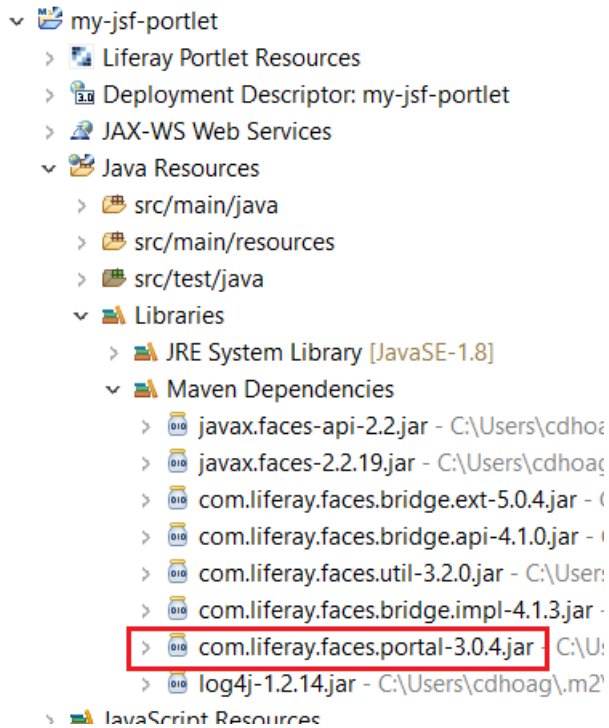

Figure 762.1: The required `.jar` files are downloaded for your JSF portlet based on the JSF UI Component Suite you configured.

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.

762.1 Related Topics

Developing a JSF Portlet Application
    Understanding Liferay Faces Bridge
    Understanding Liferay Faces Alloy

CHAPTER 763

MAVEN PLUGINS

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 articles.

CHAPTER 764

BUNDLE SUPPORT PLUGIN

The Bundle Support plugin lets you use Liferay Workspace as a Maven project.

764.1 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>
```

764.2 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.

764.3 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.

764.4 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.

764.5 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.

764.6 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.

764.7 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.

CHAPTER 765

CSS BUILDER PLUGIN

The CSS Builder plugin lets you compile Sass files in your project.

765.1 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.

765.2 Goals

The plugin adds one Maven goal to your project:
Name | Description `css-builder:build` | Compiles the Sass files in the project.

765.3 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.

CHAPTER 766

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.

766.1 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.

766.2 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.

766.3 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.

CHAPTER 767

DEPLOYMENT HELPER PLUGIN

The Deployment Helper plugin lets you create a cluster deployable WAR from your OSGi artifacts.

767.1 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.

767.2 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.

767.3 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.

CHAPTER 768

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.

768.1 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.

768.2 Goals

The plugin adds one Maven goal to your project:

Name | Description `javadoc-formatter:format` | Runs the Liferay Javadoc Formatter to format files.

768.3 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.

CHAPTER 769

LANG BUILDER PLUGIN

The Lang Builder plugin lets you sort and translate the language keys in your project.

769.1 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.

769.2 Goals

The plugin adds one Maven goal to your project:

Name | Description `lang-builder:build` | Runs Liferay Lang Builder to translate language property files.

769.3 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.

CHAPTER 770

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.

770.1 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.

770.2 Goals

The plugin adds one Maven goal to your project:
Name | Description `rest-builder:build` | Runs the Liferay REST Builder.

770.3 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.

CHAPTER 771

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.

771.1 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.

771.2 Goals

The plugin adds one Maven goal to your project:
Name | Description `service-builder:build` | Runs the Liferay Service Builder.

771.3 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.

CHAPTER 772

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.

772.1 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.

772.2 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.

772.3 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.

CHAPTER 773

THEME BUILDER PLUGIN

The Theme Builder plugin lets you build Liferay theme files in your project. Visit the Building a Theme with Maven tutorial to learn more about applying Theme Builder to your Maven project.

773.1 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.

773.2 Goals

The plugin adds one Maven goal to your project:
Name | Description `theme-builder:build` | Builds the theme files.

773.3 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.

CHAPTER 774

TLD FORMATTER PLUGIN

The TLD Formatter plugin lets you format a project's TLD files.

774.1 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.

774.2 Goals

The plugin adds one Maven goal to your project:
Name | Description `tld-formatter:format` | Runs the Liferay TLD Formatter to format files.

774.3 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.

CHAPTER 775

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.

775.1 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.

775.2 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.

775.3 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.

CHAPTER 776

XML FORMATTER PLUGIN

The XML Formatter plugin lets you format a project's XML files.

776.1 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.

776.2 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.

776.3 Available Parameters

You can set the following parameters in the `<configuration>` section of the POM:

Parameter Name | Type | Default Value | Description `fileName` | `String` | `null` | The XML file to format. This plugin only lets you format one XML file at a time. `stripComments` | `boolean` | `false` | Whether to remove all the comments from the XML file.

CHAPTER 777

PORTLETMVC4SPRING

PortletMVC4Spring integrates Spring, the Spring Web Framework, and the MVC design pattern with portlet development. As such, it uses configuration files from each of these areas and provides new annotations that facilitate portlet application development. Here are the PortletMVC4Spring reference topics:

- PortletMVC4Spring Annotations
- PortletMVC4Spring Configuration Files

CHAPTER 778

PORTLETMVC4SPRING PROJECT ANATOMY

PortletMVC4Spring portlets are packaged in WARs. Liferay provide Maven archetypes for creating projects configured to use JSP/JSPX and Thymeleaf templates. Their commands are listed below. The PortletMVC4Spring project structure follows the commands.

778.1 Maven Commands for Generating PortletMVC4Spring Projects

Here are Maven commands for generating PortletMVC4Spring portlet projects that use JSPX and Thymeleaf View templates:

778.2 SP/JSPX Form Portlet

```
mvn archetype:generate \
-DarchetypeGroupId=com.liferay.portletmvc4spring.archetype \
-DarchetypeArtifactId=com.liferay.portletmvc4spring.archetype.form.jsp.portlet \
-DarchetypeVersion=5.1.0 \
-DgroupId=com.mycompany \
-DartifactId=com.mycompany.my.form.jsp.portlet
```

778.3 Thymeleaf Form Portlet

```
mvn archetype:generate \
-DarchetypeGroupId=com.liferay.portletmvc4spring.archetype \
-DarchetypeArtifactId=com.liferay.portletmvc4spring.archetype.form.thymeleaf.portlet \
-DarchetypeVersion=5.1.0 \
-DgroupId=com.mycompany \
-DartifactId=com.mycompany.my.form.thymeleaf.portlet
```

778.4 Project Structure

The Maven commands generate a project that includes Model and Controller classes, View templates, a resource bundle, a stylesheet, and more. The Spring contexts and configuration files set PortletMVC4Spring development essentials. Here's the resulting project structure:

- `[com.mycompany.my.form.jsp.portlet]/` → Arbitrary project name
  - `src/`
    - `main/`
      - `java/[my-package-path]/`
      - `controller/` → Sub-package for Controller classes (optional)
      - `dto/` → Sub-package for Model (data transfer object) classes (optional)
      - `resources/` → Resources to include in the class path - `content/` → Resource bundles - `log4j.properties` → Log4J logging configuration
      - `webapp/`
      - `resources/`
      - `css/` → Style sheets
      - `images/` → Images
      - `WEB-INF/`
      - `spring-context/` → Contexts
      - `portlet/` → Portlet contexts
      - `portlet1-context.xml` → Portlet context
      - `portlet-application-context.xml` → Application context
      - `views/` → View templates
      - `liferay-display.xml` → Portlet display configuration
      - `liferay-plugin-package.properties` → Packaging descriptor
      - `liferay-portlet.xml` → Liferay-specific portlet configuration
      - `portlet.xml` → Portlet configuration
      - `web.xml` → Web application configuration
    - `test/java/` → Test source files
  - `build.gradle` → Gradle build file
  - `pom.xml` → Maven POM

CHAPTER 779

PORTLETMVC4SPRING ANNOTATIONS

PortletMVC4Spring provides several annotations for mapping requests to controller classes and controller methods.

779.1 `@RenderMapping` Annotation Examples

The following table describes some `@RenderMapping` annotation examples.

Example	Description
`@RenderMapping`	Handle primary render requests if no other handler methods match the render request.
`@RenderMapping(params = "javax.portlet.action=success")`	Handle the render request if has parameter a parameter setting `javax.portlet.action=success`.
`@RenderMapping(param = "foo")`	Handle the request if it has a parameter named `foo`, regardless of its value.
`@RenderMapping(param = "!bar")`	Handle the request as long as it has not parameter named `bar`.
`@RenderMapping(windowState = "MAXIMIZED")`	Handle the request if the window state is `MAXIMIZED`. Note, supported portlet window states are `NORMAL`, `MAXIMIZED`, and `MINIMIZED`.

779.2 `@ActionMapping` Annotation Examples

The table below describes some `@ActionMapping` annotation examples.

Example	Description
`@ActionMapping`	Handle primary action requests if no other handler methods match the action request.
`@ActionMapping(params = some.param=yourValue")`	Handle the action request if has parameter a parameter setting `javax.portlet.action=success`.
`@ActionMapping(param = "foo")`	Handle the request if it has a parameter named `foo`, regardless of its value.
`@ActionMapping(param = "!bar")`	Handle the request as long as it has not parameter named `bar`.

CHAPTER 780

PORTLETMVC4SPRING CONFIGURATION FILES

A PortletMVC4Spring application has these descriptors, Spring contexts, and properties files in its `WEB-INF` folder:

- `web.xml` → Web application descriptor
- `portlet.xml` → Portlet application descriptor
- `liferay-portlet.xml` → Liferay-specific portlet descriptor
- `liferay-display.xml` → Liferay-specific display descriptor
- `spring-context/portlet-application-context.xml` → Portlet application context
- `spring-context/portlet/[portlet]-context.xml` → Portlet context
- `liferay-plugin-package.properties` → Packaging descriptor

Examples of each file are provided and portlet-specific content is highlighted.

780.1 web.xml

The servlet container processes the `web.xml`. This file specifies the servlet that render's the portlet and the portlet application's context, servlet, filters, listeners, and more. Here's an example `web.xml`:

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

<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xm
app_3_1.xsd">
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-context/portlet-application-context.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>ViewRendererServlet</servlet-name>
        <servlet-class>com.liferay.portletmvc4spring.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>
    <filter>
        <filter-name>delegatingFilterProxy</filter-name>
```

```
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>delegatingFilterProxy</filter-name>
        <url-pattern>/WEB-INF/servlet/view</url-pattern>
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>INCLUDE</dispatcher>
    </filter-mapping>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
</web-app>
```

The `<context-param/>` element gives the path to the portlet application context (discussed later):

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring-context/portlet-application-context.xml</param-value>
</context-param>
```

The `<servlet/>` and `<servlet-mapping/>` elements set the servlet and the internal location for its views.

```
<servlet>
    <servlet-name>ViewRendererServlet</servlet-name>
    <servlet-class>com.liferay.portletmvc4spring.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>
```

The `ViewRendererServlet`. converts portlet requests into servlet requests and enables the view to be rendered using the Spring Web MVC infrastructure and the infrastructure's renderers for JSP, Thymeleaf, Velocity, and more.

The filter and filter mappings are set to forward and include servlet views as necessary.

```
<filter>
    <filter-name>delegatingFilterProxy</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
    <filter-name>delegatingFilterProxy</filter-name>
    <url-pattern>/WEB-INF/servlet/view</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

A listener is configured for processing the application's contexts.

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Liferay's project archetypes generate all this boilerplate code.

780.2 portlet.xml

The `portlet.xml` file describes the portlet application to the portlet container. Here's an example:

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

<portlet-app xmlns="http://xmlns.jcp.org/xml/ns/portlet" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.o
app_3_0.xsd" version="3.0">
    <portlet>
        <portlet-name>portlet1</portlet-name>
        <display-name>com.mycompany.my.form.jsp.portlet</display-name>
        <portlet-class>com.liferay.portletmvc4spring.DispatcherPortlet</portlet-class>
        <init-param>
            <name>contextConfigLocation</name>
            <value>/WEB-INF/spring-context/portlet/portlet1-context.xml</value>
        </init-param>
        <expiration-cache>0</expiration-cache>
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
        </supports>
        <resource-bundle>content.portlet1</resource-bundle>
        <portlet-info>
            <title>com.mycompany.my.form.jsp.portlet</title>
            <short-title>com.mycompany.my.form.jsp.portlet</short-title>
            <keywords>com.mycompany.my.form.jsp.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>
    <filter>
        <filter-name>SpringSecurityPortletFilter</filter-name>
        <filter-class>com.liferay.portletmvc4spring.security.SpringSecurityPortletFilter</filter-class>
        <lifecycle>ACTION_PHASE</lifecycle>
        <lifecycle>RENDER_PHASE</lifecycle>
        <lifecycle>RESOURCE_PHASE</lifecycle>
    </filter>
    <filter-mapping>
        <filter-name>SpringSecurityPortletFilter</filter-name>
        <portlet-name>portlet1</portlet-name>
    </filter-mapping>
</portlet-app>
```

This application has one portlet named `portlet1`.

```
<portlet-name>portlet1</portlet-name>
<display-name>com.mycompany.my.form.jsp.portlet</display-name>
<portlet-class>com.liferay.portletmvc4spring.DispatcherPortlet</portlet-class>
```

The `<portlet-name/>` is internal and the `<display-name/>` is shown to users. `<portlet-class/>` specifies the portlet's Java class.

Important: All PortletMVC4Spring portlets must specify `<portlet-class>com.liferay.portletmvc4spring.Dispatc` `class>`.

The <supports/> element must declare the mime type that the portlet templates use.

The <resource-bundle/> sets the path to the portlet's localized Java message properties. For example, the element refers to properties at content/portlet1.properties:

```
<resource-bundle>content.portlet1</resource-bundle>
```

The <portlet-info/> element lists the portlet's titles and reserved keyword.

The <security-role-ref/> elements declare default user roles the portlet accounts for.

Lastly, the <filter/> named SpringSecurityPortletFilter prevents Cross-Site Request Forgery (CSRF).

```
<filter>
    <filter-name>SpringSecurityPortletFilter</filter-name>
    <filter-class>com.liferay.portletmvc4spring.security.SpringSecurityPortletFilter</filter-class>
    <lifecycle>ACTION_PHASE</lifecycle>
    <lifecycle>RENDER_PHASE</lifecycle>
    <lifecycle>RESOURCE_PHASE</lifecycle>
</filter>
<filter-mapping>
    <filter-name>SpringSecurityPortletFilter</filter-name>
    <portlet-name>portlet1</portlet-name>
</filter-mapping>
```

The portlet XSD defines the portlet.xml. The Liferay-specific portlet descriptor is next.

780.3 liferay-portlet.xml

The liferay-portlet.xml file applies Liferay-specific settings that provide more developer features. Here's an example:

```
<?xml version="1.0"?>
<!DOCTYPE liferay-portlet-app PUBLIC "-//Liferay//DTD Portlet Application 7.1.0//EN" "http://www.liferay.com/dtd/liferay-
portlet-app_7_1_0.dtd">

<liferay-portlet-app>
    <portlet>
        <portlet-name>portlet1</portlet-name>
        <icon>/resources](./images/icon.png</icon>
        <requires-namespaced-parameters>false</requires-namespaced-parameters>
    </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>
```

This <portlet/> element associates an icon with the portlet and indicates that name-spaced parameters aren't required.

The <role-mapper/> elements associate the portlet with default Liferay DXP user roles.

The liferay-portlet-app DTD defines the `liferay-portlet.xml` file.

780.4 liferay-display.xml

The `liferay-display.xml` applies display characteristics to the portlet. For example, this descriptor associates the portlet with a Widget category in Liferay DXP's Add Widget menu.

```
<?xml version="1.0"?>
<!DOCTYPE display PUBLIC "-//Liferay//DTD Display 7.2.0//EN" "http://www.liferay.com/dtd/liferay-display_7_2_0.dtd">

<display>
<category name="category.sample">
    <portlet id="portlet1" />
</category>
</display>
```

See the liferay-display DTD for details.

It's time to look at the application contexts.

780.5 Portlet Application Context

This context applies to all of the application's portlets. This is where you specify view resolvers, resource bundles, security beans, proxies, and more. Here's an example:

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

<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    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
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
    <context:annotation-config />
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" id="viewResolver">
        <property name="contentType" value="text/html;charset=UTF-8" />
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jspx" />
        <property name="viewClass" value="com.liferay.portletmvc4spring.PortletJstlView" />
    </bean>
    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>content.portlet1</value>
            </list>
        </property>
        <property name="defaultEncoding" value="UTF-8" />
    </bean>
    <bean id="springSecurityPortletConfigurer" class="com.liferay.portletmvc4spring.security.SpringSecurityPortletConfigurer" />
    <bean id="delegatingFilterProxy" class="org.springframework.web.filter.DelegatingFilterProxy">
        <property name="targetBeanName" value="springSecurityFilterChain" />
    </bean>
</beans>
```

The view resolver bean above handles JSPX view templates. To resolve Thymeleaf view templates, for example, you could specify these beans:

```
<bean class="org.thymeleaf.templateresolver.ServletContextTemplateResolver" id="templateResolver">
    <property name="prefix" value="/WEB-INF/views/"/>
    <property name="suffix" value=".html"/>
    <property name="templateMode" value="HTML"/>
</bean>
<bean class="org.thymeleaf.spring5.SpringTemplateEngine" id="templateEngine">
    <property name="templateResolver" ref="templateResolver"></property>
    <property name="enableSpringELCompiler" value="true" />
</bean>
<bean class="org.thymeleaf.spring5.view.ThymeleafViewResolver" id="viewResolver">
    <property name="templateEngine" ref="templateEngine"/>
    <property name="order" value="1"/>
</bean>
```

The context's `springSecurityPortletConfigurer` bean facilitates using Spring Security:

```
<bean id="springSecurityPortletConfigurer"
    class="com.liferay.portletmvc4spring.security.SpringSecurityPortletConfigurer" />
```

You can also designate contexts for each portlet in the application.

780.6 Portlet Contexts

Beans specific to a portlet, go in the portlet's context. Since annotations are the easiest way to develop PortletMVC4Spring portlets, you should specify MVC annotation scanning in the portlet context:

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

<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <context:component-scan base-package="portlet1**"/>
    <mvc:annotation-driven/>
</beans>
```

The portlet context naming convention is `[portlet-name]-context.xml`. To associate your portlet with its own context, edit your application's `portlet.xml` file and add an `<init-param/>` element that maps the `<portlet/>` element to the portlet's context:

```
<init-param>
    <name>contextConfigLocation</name>
    <value>/WEB-INF/spring-context/portlet/portlet1-context.xml</value>
</init-param>
```

What's left is to describe your application package.

780.7 liferay-plugin-package.properties

This file specifies the application's name, version, Java package imports/exports, and OSGi metadata. Here's an example package properties file:

```
author=N/A
change-log=
licenses=N/A
liferay-versions=7.1.0+
long-description=
module-group-id=com.mycompany
module-incremental-version=1
name=com.mycompany.my.form.jsp.portlet
page-url=
short-description=my portlet short description
tags=myTag
Bundle-Version: 1.0.0
Import-Package: com.liferay.portal.webserver,com.liferay.portal.kernel.servlet.filters.invoker
```

It uses this OSGi metadata header to import required Java packages:

```
Import-Package: com.liferay.portal.webserver,\
com.liferay.portal.kernel.servlet.filters.invoker
```

On deploying the portlet application WAR file, the WAB Generator adds the specified OSGi metadata to the resulting web application bundle (WAB) that's deployed to Liferay's runtime framework.

The liferay-plugin-package reference document describes the `liferay-plugin-package.properties` file.

Congratulations! You've successfully toured the PortletMVC4Spring configuration files.

780.8 Related Topics

PortletMVC4Spring Annotations

Migrating to PortletMVC4Spring

CHAPTER 781

PROJECT TEMPLATES

Liferay provides project templates that you can use to generate starter projects formatted in an opinionated way. These templates can be used by most build tools (e.g., Gradle, Maven, Dev Studio) to generate your desired project structure.

If you're using Blade CLI, execute the following command to display a full list of project templates:

```
blade create -l
```

If you're using Maven, you can view and use the project templates as Maven archetypes. Execute the following command to list them:

```
mvn archetype:generate -Dfilter=liferay
```

Archetypes with the `com.liferay.project.templates` prefix are the latest templates offered by Liferay.

If you're using Dev Studio, navigate to *File → New → Liferay Module Project* and view the project templates from the *Project Template Name* drop-down menu.

In this section of reference articles, each project template is outlined with the appropriate generation command and folder structure. Visit the project template article you're most interested in to start building your own project!

CHAPTER 782

ACTIVATOR TEMPLATE

In this article, you'll learn how to create a Liferay activator as a Liferay module. To create a Liferay activator via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t activator [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.activator \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `activator`. Suppose you want to create an activator project called `my-activator-project` with a package name of `com.liferay.docs.activator` and a class name of `Activator`. You could run the following command to accomplish this:

```
blade create -t activator -p com.liferay.docs.activator -c Activator my-activator-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.activator \
    -DgroupId=com.liferay \
    -DartifactId=my-activator-project \
    -Dpackage=com.liferay.docs.activator \
    -Dversion=1.0 \
    -DclassName=Activator \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

Note that in your class, you're implementing the `org.osgi.framework.BundleActivator` interface.

After running the Blade command above, your project's directory structure looks like this:

- my-activator-project
    - gradle
        * wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        * main
            - java
            - com/liferay/docs/activator
            - Activator.java

            - resources
    - 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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

CHAPTER 783

API TEMPLATE

In this tutorial, you'll learn how to create a Liferay API as a Liferay module. To create a Liferay API via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t api [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.api \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `api`. The `api` template creates a simple `api` module with an empty public interface. For example, suppose you want to create an API project called `my-api-project` with a package name of `com.liferay.docs.api` and a class name of `MyApi`. You could run the following command to accomplish this:

```
blade create -t api -p com.liferay.docs -c MyApi my-api-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.api \
    -DgroupId=com.liferay \
    -DartifactId=my-api-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=MyApi \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

- my-api-project
    - gradle
        * wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        * main
            - java
            - com/liferay/docs/api
            - MyApi.java

            - resources
            - com/liferay/docs/api
            - packageinfo
    - bnd.bnd
    - build.gradle
    - gradlew

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 784

CONTROL MENU ENTRY TEMPLATE

In this article, you'll learn how to create a Liferay Control Menu entry as a Liferay module. To create a Liferay Control Menu entry via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t control-menu-entry [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.control.menu.entry \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `control-menu-entry`. Suppose you want to create a control menu entry project called `my-control-menu-entry-project` with a package name of `com.liferay.docs.entry.control.menu` and a class name of `SampleProductNavigationControlMenuEntry`. You could run the following command to accomplish this:

```
blade create -t control-menu-entry -p com.liferay.docs.entry -c Sample my-control-menu-entry-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.control.menu.entry \
    -DgroupId=com.liferay \
    -DartifactId=my-control-menu-entry-project \
    -Dpackage=com.liferay.docs.entry \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure would look like this:

```
- my-control-menu-entry-project
    - gradle
        * wrapper
            · gradle-wrapper.jar
            · gradle-wrapper.properties
    - src
        * main
            · java
            · com/liferay/docs/entry/control/menu
            · SampleProductNavigationControlMenuEntry.java

            · resources
            · content
            · Language.properties
    - bnd.bnd
    - build.gradle
    - gradlew
```

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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the control-menu-entry sample project for a more expanded sample of a Control Menu entry.

CHAPTER 785

FORM FIELD TEMPLATE

In this article, you'll learn how to create a Liferay form field as a Liferay module. To create a Liferay form field via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t form-field [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.form.field \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `form-field`. Suppose you want to create a form field project called `my-form-field-project` with a package name of `com.liferay.docs.form.field` and a class name prefix of `MyFormField`. You could run one of the following commands to accomplish this:

```
blade create -t form-field -p com.liferay.docs -c MyFormField my-form-field-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.form.field \
    -DgroupId=com.liferay \
    -DartifactId=my-form-field-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=MyFormField \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

- `my-form-field-project`

```
- gradle

    * wrapper

        · gradle-wrapper.jar
        · gradle-wrapper.properties

- src

    * main

        · java
        · com/liferay/docs/form/field
        · MyFormFieldDDMFormFieldRenderer.java
        · MyFormFieldDDMFormFieldType.java

        · resources
        · content
        · Language.properties

        · META-INF
        · resources
        · config.js
        · my-form-field-project.soy
        · my-form-field-project_field.js
        · my-form-field-project_field.es.js

- 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 form field and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

CHAPTER 786

FRAGMENT TEMPLATE

In this article, you'll learn how to create a Liferay fragment as a Liferay module. You can learn more about fragment modules in the Declaring a Fragment Host article and in section 3.14 of the OSGi Alliance's core specification document.

To create a Liferay fragment via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t fragment [-h hostBundleName] [-H hostBundleVersion] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.fragment \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `fragment`. Suppose you want to create a fragment project called `my-fragment-project` with a host bundle symbolic name of `com.liferay.login.web` and host bundle version of `1.0.0`. You could run the following command to accomplish this:

```
blade create -t fragment -h com.liferay.login.web -H 1.0.0 my-fragment-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.fragment \
    -DgroupId=com.liferay \
    -DartifactId=my-fragment-project \
    -Dversion=1.0 \
    -Dpackage= \
    -DhostBundleSymbolicName=com.liferay.login.web \
    -DhostBundleVersion=1.0.0 \
    -DliferayVersion=7.2
```

The folder structure is created, but there are no files. The only files created are the `bnd.bnd` and `build.gradle` files, which specify your host bundle and its information, and your build tool's files. After running the Blade command above, your project's directory structure looks like this:

```
• my-fragment-project

    – gradle

        * wrapper

            · gradle-wrapper.jar
            · gradle-wrapper.properties

    – src

        * main

            · java
            · my/fragment/project-resources

    – 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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

CHAPTER 787

FREEMARKER PORTLET TEMPLATE

In this article, you'll learn how to create a Liferay FreeMarker portlet application as a Liferay module. To create a Liferay FreeMarker portlet application via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t freemarker-portlet [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.freemarker.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `freemarker-portlet`. Suppose you want to create a FreeMarker portlet project called `my-freemarker-portlet-project` with a package name of `com.liferay.docs.freemarkerportlet` and a class name of `MyFreemarkerPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.util.bridges.freemarker.FreeMarkerPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t freemarker-portlet -p com.liferay.docs.freemarkerportlet -c MyFreemarkerPortlet my-freemarker-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.freemarker.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-freemarker-portlet-project \
    -Dpackage=com.liferay.docs.freemarkerportlet \
    -Dversion=1.0 \
    -DclassName=MyFreemarkerPortlet \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

```
• my-freemarker-portlet-project

    – gradle

        * wrapper

            · gradle-wrapper.jar
            · gradle-wrapper.properties

    – src

        * main

            · java
            · com/liferay/docs/freemarkerportlet
            · constants
            · MyFreemarkerPortletKeys.java

            · portlet
            · MyFreemarkerPortlet.java

            · resources
            · content
            · Language.properties

            · META-INF
            · resources
            · css
            · main.scss

            · templates
            · init.ftl
            · view.ftl

    – bnd.bnd
    – build.gradle
    – gradlew
```

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 788

LAYOUT TEMPLATE

In this article, you'll learn how to create a Liferay layout template as a WAR project. To create a Liferay layout template via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t layout-template projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.layout.template \
    -DartifactId=[projectName] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `layout-template`. Suppose you want to create a layout template project called `my-layout-template-project`. You could run one of the following commands to accomplish this:

```
blade create -t layout-template my-layout-template-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.layout.template \
    -DgroupId=com.liferay \
    -DartifactId=my-layout-template-project \
    -Dversion=1.0 \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

- `my-layout-template-project`
    - `gradle`

```
        * wrapper

              · gradle-wrapper.jar
              · gradle-wrapper.properties

    - src

        * main

              · webapp
              · WEB-INF
              · liferay-layout-templates.xml
              · liferay-plugin-package.properties

              · my-layout-template-project.ftl
              · my-layout-template-project.png

    - 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 WAR is a working layout template and is deployable to a Liferay DXP instance. To build upon the generated layout template, modify the project by adding logic and additional files to the folders outlined above.

CHAPTER 789

MODULES EXT TEMPLATE

In this article, you'll learn how to create an Ext module. To create an Ext module via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t modules-ext [-p packageName] [-m originalModuleName] [-M originalModuleVersion] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.modules.ext \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DoriginalModuleName=[originalModuleName] \
    -DoriginalModuleVersion=[originalModuleVersion] \
    -DliferayVersion=7.2
```

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 `modules-ext`. Suppose you want to create an Ext module called `my-ext-module-project` that overrides the `com.liferay.test.web` module (BSN) with version `1.0.0`. If you have Target Platform enabled, you're not required to specify the intended module version to override. Also, the override module has a package path of `com.liferay.docs.test`. You must use the exact path of the original module when creating an Ext module. You could run the following command to accomplish this:

```
blade create -t modules-ext -p com.liferay.docs.test -m com.liferay.test.web -M 1.0.0 my-ext-module-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.modules.ext \
    -DartifactId=my-ext-module-project \
    -Dpackage=com.liferay.docs.test \
    -DoriginalModuleName=com.liferay.test.web \
    -DoriginalModuleVersion=1.0.0 \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

```
• my-ext-module-project

    – gradle

        * wrapper

            · gradle-wrapper.jar
            · gradle-wrapper.properties

    – src

        * main

            · java
            · com/liferay/docs/test

            · resources

    – 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.

To build upon the generated project, modify the project by adding logic and additional files to the folders outlined above.

CHAPTER 790

MVC PORTLET TEMPLATE

In this article, you'll learn how to create a Liferay MVC portlet application as a Liferay module. To create a Liferay MVC portlet application via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t mvc-portlet [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.mvc.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `mvc-portlet`. Suppose you want to create an MVC portlet project called `my-mvc-portlet-project` with a package name of `com.liferay.docs.mvcportlet` and a class name of `MyMvcPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t mvc-portlet -p com.liferay.docs.mvcportlet -c MyMvcPortlet my-mvc-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.mvc.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-mvc-portlet-project \
    -Dpackage=com.liferay.docs.mvcportlet \
    -Dversion=1.0 \
    -DclassName=MyMvcPortlet \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

```
• my-mvc-portlet-project

    – gradle

        * wrapper

            · gradle-wrapper.jar
            · gradle-wrapper.properties

    – src

        * main

            · java
            · com/liferay/docs/mvcportlet
            · constants
            · MyMvcPortletKeys.java

            · portlet
            · MyMvcPortlet.java

            · resources
            · content
            · Language.properties

            · META-INF
            · resources
            · init.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.

CHAPTER 791

PANEL APP TEMPLATE

In this article, you'll learn how to create a Liferay panel app and category as a Liferay module. To create a Liferay panel app and category via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t panel-app [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.panel.app \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `panel-app`. Suppose you want to create a panel app project called `my-panel-app-project` with a package name prefix of `com.liferay.docs` and a class name prefix of `Sample`. You could run the following command to accomplish this:

```
blade create -t panel-app -p com.liferay.docs -c Sample my-panel-app-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.panel.app \
    -DgroupId=com.liferay \
    -DartifactId=my-panel-app-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure would look like this

- `my-panel-app-project`

```
- gradle

    * wrapper

        · gradle-wrapper.jar
        · gradle-wrapper.properties

- src

    * main

        · java
        · com/liferay/docs/
        · application/list
        · SamplePanelApp.java
        · SamplePanelCategory.java

        · constants
        · SamplePanelCategoryKeys.java
        · SamplePortletKeys.java

        · portlet
        · SamplePortlet.java

        · resources
        · content
        · Language.properties

        · META-INF
        · resources
        · init.jsp
        · view.jsp

- bnd.bnd
- build.gradle
- gradlew
```

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 functional and is deployable to a Liferay DXP instance. The generated module, by default, creates a panel category with a panel app in Liferay DXP's Product Menu. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

CHAPTER 792

PORTLET CONFIGURATION ICON

In this article, you'll learn how to create a Liferay portlet configuration icon as a Liferay module. To create a portlet configuration icon via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t portlet-configuration-icon [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.configuration.icon \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `portlet-configuration-icon`. Suppose you want to create a portlet configuration icon project called `my-portlet-config-icon` with a package name of `com.liferay.docs.portlet.configuration.icon` and a class name of `SamplePortletConfigurationIcon`. You could run the following command to accomplish this:

```
blade create -t portlet-configuration-icon -p com.liferay.docs -c Sample my-portlet-config-icon
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.configuration.icon \
    -DgroupId=com.liferay \
    -DartifactId=my-portlet-config-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure would look like this

- my-portlet-config-icon
    - gradle
        * wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        * main
            - java
            - com/liferay/docs/portlet/configuration/icon
            - SamplePortletConfigurationIcon.java

            - resources
            - content
            - Language.properties
    - bnd.bnd
    - build.gradle
    - gradlew

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 functional and is deployable to a Liferay DXP instance. The generated module, by default, creates a sample link in the Hello World portlet's Options menu. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the portlet-configuration-icon sample project for a more expanded sample of a portlet configuration icon.

CHAPTER 793

PORTLET PROVIDER TEMPLATE

In this article, you'll learn how to create a Liferay portlet provider as a Liferay module. To create a Liferay portlet provider via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t portlet-provider [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.provider \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `portlet-provider`. Suppose you want to create a portlet provider project called `my-portlet-provider-project` with a package name of `com.liferay.docs.portlet` and a class name prefix of `Sample`. You could run the following command to accomplish this:

```
blade create -t portlet-provider -p com.liferay.docs -c Sample my-portlet-provider-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.provider \
    -DgroupId=com.liferay \
    -DartifactId=my-portlet-provider-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure would look like this

- my-portlet-provider-project
    - gradle
        * wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        * main
            - java
            - com/liferay/docs
            - constants
            - SamplePortletKeys.java
            - SampleWebKeys.java
            - portlet
            - SampleAddPortletProvider.java
            - SamplePortlet.java
            - resources
            - META-INF
            - resources
            - init.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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the Providing Portlets to Manage Requests tutorial for instructions on customizing a portlet provider project.

CHAPTER 794

PORTLET TOOLBAR CONTRIBUTOR TEMPLATE

In this article, you'll learn how to create a Liferay portlet toolbar contributor as a Liferay module. To create a portlet toolbar contributor entry via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t portlet-toolbar-contributor [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.toolbar.contributor \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `portlet-toolbar-contributor`. Suppose you want to create a portlet toolbar contributor project called `my-portlet-toolbar-contributor` with a package name of `com.liferay.docs.portlet.toolbar.contributor` and a class name of `SamplePortletToolbarContributor`. You could run the following command to accomplish this:

```
blade create -t portlet-toolbar-contributor -p com.liferay.docs -c Sample my-portlet-toolbar-contributor
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.toolbar.contributor \
    -DgroupId=com.liferay \
    -DartifactId=my-portlet-toolbar-contributor \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure would look like this

- my-portlet-toolbar-contributor
    - gradle
        * wrapper
            · gradle-wrapper.jar
            · gradle-wrapper.properties
    - src
        * main
            · java
            · com/liferay/docs/portlet/toolbar/contributor
            · SamplePortletToolbarContributor.java

            · resources
            · content
            · Language.properties
    - bnd.bnd
    - build.gradle
    - gradlew

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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. This generated project, by default, creates a new button on the Hello World portlet's toolbar. You can visit the portlet-toolbar-contributor sample project for a more expanded sample of a portlet toolbar contributor.

CHAPTER 795

REST TEMPLATE

In this article, you'll learn how to create a Liferay RESTful web service packaged in a Liferay module. To create a Liferay RESTful web service via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t rest [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.rest \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `rest`. Suppose you want to create a RESTful web service project called `my-rest-project` with a package name of `com.liferay.docs.application` and a class name prefix of `Rest`. You could run one of the following commands to accomplish this:

```
blade create -t rest -p com.liferay.docs -c Rest my-rest-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.rest \
    -DgroupId=com.liferay \
    -DartifactId=my-rest-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Rest \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

- `my-rest-project`

- gradle
    - wrapper
        - gradle-wrapper.jar
        - gradle-wrapper.properties
- src
    - main
        - java
        - com/liferay/docs/application
        - RestApplication.java
        - resources
        - configuration
        - com.liferay.portal.remote.cxf.common.configuration.CXFEndpointPublisherConfiguration-cxf.properties
        - com.liferay.portal.remote.rest.extender.configuration.RestExtenderConfiguration-rest.properties
- 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 RESTful web service and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

CHAPTER 796

SERVICE BUILDER TEMPLATE

In this article, you'll learn how to create a Liferay portlet application that uses Service Builder as Liferay modules. To create a Liferay Service Builder project via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t service-builder [-p packageName] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service.builder \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DapiPath=[apiPath] \
    -DdependencyInjector=[dependencyInjector] \
    -DliferayVersion=7.2
```

By default, the Service Builder project uses OSGi Declarative Services (`ds`) for its dependency injector. If you prefer using Spring, you can set the parameter `--dependency-injector spring` with Blade CLI or `-DdependencyInjector=spring` with Maven. See the Dependency Injection section for more information on these options.

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `service-builder`. Suppose you want to create a Service Builder project called `tasks` with a package name of `com.liferay.docs.tasks` using OSGi Declarative Services. You could run the following command to accomplish this:

```
blade create -t service-builder -p com.liferay.docs.tasks tasks
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service.builder \
    -DgroupId=com.liferay \
    -DartifactId=tasks \
    -Dpackage=com.liferay.docs.tasks \
    -Dversion=1.0 \
    -DapiPath=com.liferay.api.path \
    -DdependencyInjector=ds \
    -DliferayVersion=7.2
```

This task creates the `tasks-api` and `tasks-service` folders. In many cases, a Service Builder project also requires a `-web` folder to hold, for example, portlet classes. This should be created manually. After running the Blade command above, your project's directory structure looks like this:

- tasks
    - gradle
        - wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - tasks-api
        - bnd.bnd
        - build.gradle
    - tasks-service
        - bnd.bnd
        - build.gradle
        - service.xml
    - build.gradle
    - gradlew
    - settings.gradle

The Maven-generated project includes a `pom.xml` file and does not include the Gradle-specific files, but otherwise, appears exactly the same.

To generate your service and API classes for the `*-api` and `*-service` modules, replace the `service.xml` file in the `*-service` module. Depending on your build tool, you can build your services by executing

```
blade gw buildService
```

or

```
mvn service-builder:build
```

from the `tasks` root directory. Note that `blade gw` only works if the Gradle wrapper can be detected. To ensure the availability of the Gradle wrapper, be sure to work in a Liferay Workspace.

The `mvn service-builder:build` command only works if you're using the `com.liferay.portal.tools.service.build` plugin version 1.0.145+. Maven projects using an earlier version of the Service Builder plugin should update their POM accordingly.

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

For more information on Service Builder, see the Service Builder section.

CHAPTER 797

SERVICE TEMPLATE

In this article, you'll learn how to create a Liferay service as a Liferay module. To create a Liferay service via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t service [-p packageName] [-c className] [-s serviceName] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className]
    -DserviceName=[serviceName] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `service`. Suppose you want to create a service project called `my-service-project` with a package name of `com.liferay.docs.service` and a class name of `Service`. Also, you'd like to create a service of type `com.liferay.portal.kernel.events.LifecycleAction` that also implements that same service. You could run the following command to accomplish this:

```
blade create -t service -p com.liferay.docs.service -c Service -s com.liferay.portal.kernel.events.LifecycleAction my-service-
project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service \
    -DgroupId=com.liferay \
    -DartifactId=my-service-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Service \
    -DclassName=com.liferay.portal.kernel.events.LifecycleAction \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure would look like this

```
• my-service-project

    – gradle

        * wrapper

            · gradle-wrapper.jar
            · gradle-wrapper.properties

    – src

        * main

            · java
            · com/liferay/docs/service
            · Service.java

            · resources

    – 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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

CHAPTER 798

SERVICE WRAPPER TEMPLATE

In this article, you'll learn how to create a Liferay service wrapper as a Liferay module. To create a Liferay service wrapper via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t service-wrapper [-p packageName] [-c className] [-s serviceWrapperClass] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service.wrapper \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DserviceWrapperClass=[serviceWrapperClass] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `service-wrapper`. Suppose you want to create a service wrapper project called `service-override` with a package name of `com.liferay.docs.serviceoverride` and a class name of `UserLocalServiceOverride`. Also, you'd like to create a service of type `com.liferay.portal.kernel.service.ServiceWrapper` that extends the `com.liferay.portal.service.UserLocalServiceWr` class. You could run the following command to accomplish this:

```
blade create -t service-wrapper -p com.liferay.docs.serviceoverride -c UserLocalServiceOverride -s com.liferay.portal.kernel.service.UserLocalServic
override
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service.wrapper \
    -DgroupId=com.liferay \
    -DartifactId=service-override \
    -Dpackage=com.liferay.docs.serviceoverride \
    -Dversion=1.0 \
    -DclassName=UserLocalServiceOverride \
    -DserviceWrapperClass=com.liferay.portal.kernel.service.UserLocalServiceWrapper \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*.

After running the Blade command above, your project's directory structure looks like this:

```
• service-override

    – gradle

        * wrapper

            · gradle-wrapper.jar
            · gradle-wrapper.properties

    – src

        * main

            · java
            · com/liferay/docs/serviceoverride
            · UserLocalServiceOverride.java

            · resources

    – 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 799

SIMULATION PANEL ENTRY TEMPLATE

In this article, you'll learn how to create a Liferay simulation panel entry as a Liferay module. To create a simulation panel entry via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t simulation-panel-entry [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.simulation.panel.entry \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `simulation-panel-entry`. Suppose you want to create a simulation panel entry project called `my-simulation-panel-entry` with a package name of `com.liferay.docs.application.list` and a class name of `SampleSimulationPanelApp`. You could run the following command to accomplish this:

```
blade create -t simulation-panel-entry -p com.liferay.docs -c Sample my-simulation-panel-entry
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.simulation.panel.entry \
    -DgroupId=com.liferay \
    -DartifactId=my-simulation-panel-entry \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure would look like this

- my-simulation-panel-entry
    - gradle
        - wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        - main
            - java
            - com/liferay/docs/application/list
            - SampleSimulationPanelApp.java
            - resources
            - content
            - Language.properties
            - META-INF
            - resources
            - simulation_panel.jsp
    - bnd.bnd
    - build.gradle
    - gradlew

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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the simulation-panel-app sample project for a more expanded sample of a control menu entry.

CHAPTER 800

SOCIAL BOOKMARK TEMPLATE

In this article, you'll learn how to create a Liferay social bookmark as a Liferay module. To create a social bookmark as a module via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t social-bookmark [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.social.bookmark \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

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 `social-bookmark`. Suppose you want to create a social bookmark project called `my-social-bookmark-project` with a package name of `com.liferay.docs.socialbookmark` and a class name of `TestSocialBookmark`. You could run the following command to accomplish this:

```
blade create -t social-bookmark -p com.liferay.docs.socialbookmark -c Test my-social-bookmark-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.social.bookmark \
    -DgroupId=com.liferay \
    -DartifactId=my-social-bookmark-project \
    -Dpackage=com.liferay.docs.socialbookmark \
    -Dversion=1.0 \
    -DclassName=Test \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

```
- my-social-bookmark-project
    - gradle
        * wrapper
            · gradle-wrapper.jar
            · gradle-wrapper.properties
    - src
        * main
            · java
            · com/liferay/docs/socialbookmark
            · social/bookmark
            · TestSocialBookmark

            · resources
            · content
            · Language.properties

            · META-INF
            · resources
            · icons.svg
            · init.jsp
            · page.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. An unmodified module generated as described above creates a social bookmark named *Test* that searches the current URL using Google Search.

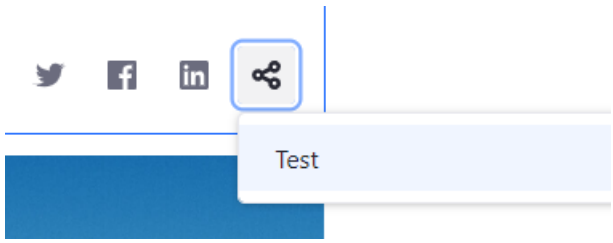

Figure 800.1: Click the magnifying glass icon to search the current URL using Google Search.

To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. For more information on developing social bookmarks, see the Social API section of tutorials. For information on configuring social bookmarks for the Blogs widget, see the Displaying Blogs article.

CHAPTER 801

SPRING MVC PORTLET TEMPLATE

In this article, you'll learn how to create a Liferay Spring MVC portlet application as a WAR. To create a Liferay Spring MVC portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t spring-mvc-portlet [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.spring.mvc.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `spring-mvc-portlet`. Suppose you want to create a Spring MVC portlet project called `my-spring-mvc-portlet-project` with a package name of `com.liferay.docs.springmvcportlet` and a class name of `MySpringMvcPortlet`. Also, you'd like to create a Spring-annotated portlet class named `MySpringMvcPortletViewController`.

```
blade create -t spring-mvc-portlet -p com.liferay.docs.springmvcportlet -c MySpringMvcPortlet my-spring-mvc-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.spring.mvc.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-spring-mvc-portlet-project \
    -Dpackage=com.liferay.docs.springmvcportlet \
    -Dversion=1.0 \
    -DclassName=MySpringMvcPortlet \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure looks like this:

- my-spring-mvc-portlet-project
    - gradle
        * wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        * main
            - java
            - com/liferay/docs/springmvcportlet/portlet
            - MySpringMvcPortletViewController

            - resources
            - content
            - Language.properties

            - webapp
            - css
            - main.scss

            - WEB-INF
            - jsp
            - init.jsp
            - view.jsp

            - spring-context
            - portlet
            - my-spring-mvc-portlet-project.xml

            - portlet-application-context.xml

            - tld
            - liferay-portlet.tld
            - liferay-portlet-ext.tld
            - liferay-security.tld
            - liferay-theme.tld
            - liferay-ui.tld
            - liferay-util.tld

            - liferay-display.xml
            - liferay-plugin-package.properties
            - liferay-portlet.xml
            - portlet.xml
            - web.xml

```
            · icon.png

    – build.gradle
    – gradlew
```

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 WAR is a working application and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the springmvc-portlet sample project for a more expanded sample of a Spring MVC portlet.

CHAPTER 802

TEMPLATE CONTEXT CONTRIBUTOR TEMPLATE

In this article, you'll learn how to create a Liferay template context contributor as a Liferay module. To create a template context contributor via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t template-context-contributor [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.template.context.contributor \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `template-context-contributor`. Suppose you want to create a template context contributor project called `my-template-context-contributor` with a package name of `com.liferay.docs.theme.contributor` and a class name of `SampleTemplateContextContributor`. You could run the following command to accomplish this:

```
blade create -t template-context-contributor -p com.liferay.docs -c Sample my-template-context-contributor
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.template.context.contributor \
    -DgroupId=com.liferay \
    -DartifactId=my-template-context-contributor \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's directory structure would look like this

- my-template-context-contributor
    - gradle
        * wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        * main
            - java
            - com/liferay/docs/context/contributor
            - SampleTemplateContextContributor.java

            - resources
    - 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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the template-context-contributor sample project for a more expanded sample of a template context contributor. Likewise, see the Context Contributors tutorial for instructions on customizing a template context contributor project.

CHAPTER 803

THEME CONTRIBUTOR TEMPLATE

In this article, you'll learn how to create a Liferay theme contributor as a Liferay module. To create a theme contributor via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t theme-contributor [--contributorType contributorType] [-p packageName] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.theme.contributor \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DcontributorType=[contributorType] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `theme-contributor`. Suppose you want to create a theme contributor project called `my-theme-contributor` with a package name of `com.liferay.docs.theme.contributor` and a contributor type of `my-contributor`. You could run the following command to accomplish this:

```
blade create -t theme-contributor --contributor-type my-contributor -p com.liferay.docs.theme.contributor my-theme-contributor
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.theme.contributor \
    -DgroupId=com.liferay \
    -DartifactId=my-theme-contributor \
    -Dpackage=com.liferay.docs.theme.contributor \
    -Dversion=1.0 \
    -DcontributorType=my-contributor \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's folder structure would look like this:

```
• my-theme-contributor

    – gradle

        * wrapper

            · gradle-wrapper.jar
            · gradle-wrapper.properties

    – src

        * main

            · java
            · com/liferay/docs/theme/contributor

            · resources
            · META-INF
            · resources
            · css
            · my-contributor
            · _body.scss
            · _control_menu.scss
            · _product_menu.scss
            · _simulation_panel.scss

            · my-contributor.scss

            · js
            · my-contributor.js

    – bnd.bnd
    – build.gradle
    – 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 functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the Blade Theme Contributor sample project for a more expanded sample of a theme contributor. Likewise, see the Theme Contributors tutorial for instructions on customizing a theme contributor project.

CHAPTER 804

THEME TEMPLATE

In this article, you'll learn how to create a Liferay theme as a WAR project. To create a Liferay theme via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t theme projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.theme \
    -DartifactId=[projectName] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `theme`. Suppose you want to create a theme project called `my-theme-project` as a WAR file. You could run the following command to accomplish this:

```
blade create -t theme my-theme-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.theme \
    -DgroupId=com.liferay \
    -DartifactId=my-theme-project \
    -Dversion=1.0 \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's folder structure looks like this:

- my-theme-project
    - gradle
        * wrapper

```
            · gradle-wrapper.jar
            · gradle-wrapper.properties

- src

    * main

            · resources
            · resources-importer
            · sitemap.json

            · webapp
            · css
            · _custom.scss

            · WEB-INF
            · liferay-plugin-package.properties
            · web.xml

- build.gradle
- gradlew
```

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 theme is functional and is deployable to a Liferay DXP instance. To build upon the generated project, modify the project by adding logic and additional files to the folders outlined above. You can visit the simple-theme project for a more expanded sample of a theme. Likewise, see the Themes section for more information on creating themes.

CHAPTER 805

WAR Core Ext

In this article, you'll learn how to create a Liferay WAR core Ext project. To create a WAR core Ext project via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t war-core-ext projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.core.ext \
    -DartifactId=[projectName] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `war-core-ext`. Suppose you want to create a WAR core Ext project called `my-war-core-ext-project`. You could run the following command to accomplish this:

```
blade create -t war-core-ext my-war-core-ext-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.core-ext \
    -DgroupId=com.liferay \
    -DartifactId=my-war-core-ext-project \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's folder structure looks like this:

- my-war-core-ext-project
    - gradle
        * wrapper

        - gradle-wrapper.jar
        - gradle-wrapper.properties

- src

    * extImpl

        - java
        - resources
        - META-INF
        - ext-model-hints.xml
        - ext-spring.xml
        - portal-log4j-ext.xml

    * extKernel

        - java
        - resources
        - META-INF

    * extUtilBridges

        - java
        - resources
        - META-INF

    * extUtilJava

        - java
        - resources
        - META-INF

    * extUtilTaglib

        - java
        - resources
        - META-INF

    * main

        - webapp
        - WEB-INF
        - ext-web
        - docroot
        - WEB-INF
        - liferay-portlet-ext.xml
        - portlet-ext.xml
        - struts-config-ext.xml
        - tiles-defs-ext.xml
        - web.xml

        - liferay-plugin-package.properties

- 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.

Note: If you generate a WAR Ext project using Gradle outside of Liferay Workspace, you must set the `app.server.parent.dir` property in the project's `gradle.properties`. The app server location is required for this project to compile.

The generated WAR Ext project is functional and is deployable to a Liferay DXP instance. To build upon the generated project, modify the project by adding logic and additional files to the folders outlined above. Deploying WAR Ext projects is only supported for limited use cases; it is recommended to leverage provided extension points offered in Liferay DXP. You can visit the Customization with Ext section for info on how to do this.

CHAPTER 806

WAR HOOK TEMPLATE

In this article, you'll learn how to create a Liferay WAR hook project. To create a Liferay WAR hook via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t war-hook [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.hook \
    -DartifactId=[projectName]
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `war-hook`. Suppose you want to create a WAR hook project called `my-war-hook-project` with a package name of `com.liferay.docs` and a class name of `MyWarHook`. You could run the following command to accomplish this:

```
blade create -t war-hook -p com.liferay.docs -c MyWarHook my-war-hook-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.hook \
    -DgroupId=com.liferay \
    -DartifactId=my-war-hook-project \
    -Dpackage=com.liferay.docs \
    -DclassName=MyWarHook \
    -Dversion=1.0 \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's folder structure looks like this:

- `my-war-hook-project`

```
- gradle

    * wrapper

        · gradle-wrapper.jar
        · gradle-wrapper.properties

- src

    * main

        · java
        · com/liferay/docs
        · MyWarHookLoginPostAction
        · MyWarHookStartupAction

        · resources
        · portal.properties

        · webapp
        · WEB-INF
        · liferay-hook.xml
        · liferay-plugin-package.properties
        · web.xml

- build.gradle
- 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 WAR hook is functional and is deployable to a Liferay DXP instance. To build upon the generated project, modify the project by adding logic and additional files to the folders outlined above. Deploying WAR hooks is supported for 7.0, however, it is recommended to optimize your WAR hooks to fragments or other applicable module projects. You can visit the Liferay Customization section for info on how to do this for many project types. See the Customizing Liferay Portal section for more information on WAR hooks.

CHAPTER 807

WAR MVC PORTLET TEMPLATE

In this article, you'll learn how to create a Liferay MVC portlet project as a WAR file. To create a Liferay MVC portlet project as a WAR via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t war-mvc-portlet [-p packageName] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.mvc.portlet \
    -DartifactId=[projectName]
    -Dpackage=[packageName] \
    -DliferayVersion=7.2
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `war-mvc-portlet`. Suppose you want to create a WAR MVC portlet project called `my-war-mvc-portlet-project` with a package name of `com.liferay.docs.war.mvc` and a class name of `MyWarMvcPortlet`. You could run the following command to accomplish this:

```
blade create -t war-mvc-portlet -p com.liferay.docs.war.mvc my-war-mvc-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.mvc.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-war-mvc-portlet-project \
    -Dpackage=com.liferay.docs.war.mvc \
    -Dversion=1.0 \
    -DliferayVersion=7.2
```

After running the Blade command above, your project's folder structure looks like this:

- `my-war-mvc-portlet-project`
    - `gradle`

```
        * wrapper

              · gradle-wrapper.jar
              · gradle-wrapper.properties

    – src

        * main

              · java
              · com/liferay/docs/war/mvc

              · resources
              · content
              · Language.properties

              · webapp
              · css
              · main.scss

              · WEB-INF
              · tld
              · liferay-portlet.tld
              · liferay-portlet-ext.tld
              · liferay-security.tld
              · liferay-theme.tld
              · liferay-ui.tld
              · liferay-util.tld

              · liferay-display.xml
              · liferay-plugin-package.properties
              · liferay-portlet.xml
              · portlet.xml
              · web.xml

              · init.jsp
              · view.jsp

    – build.gradle
    – 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 WAR MVC portlet is functional and is deployable to a Liferay DXP instance. To build upon the generated project, modify the project by adding logic and additional files to the folders outlined above. Deploying WAR MVC portlets is supported for 7.0, however, it is recommended to optimize your WAR portlet to a module project, if possible. You can visit the From Liferay Portal 6 to 7 section for info on how to do this.

CHAPTER 808

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. You can also generated them using Blade CLI.

The sample projects are grouped into three different parent folders based on the build tools used to generate them:

- `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.

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.).

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 article 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.

CHAPTER 809

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 810

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!

CHAPTER 811

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.

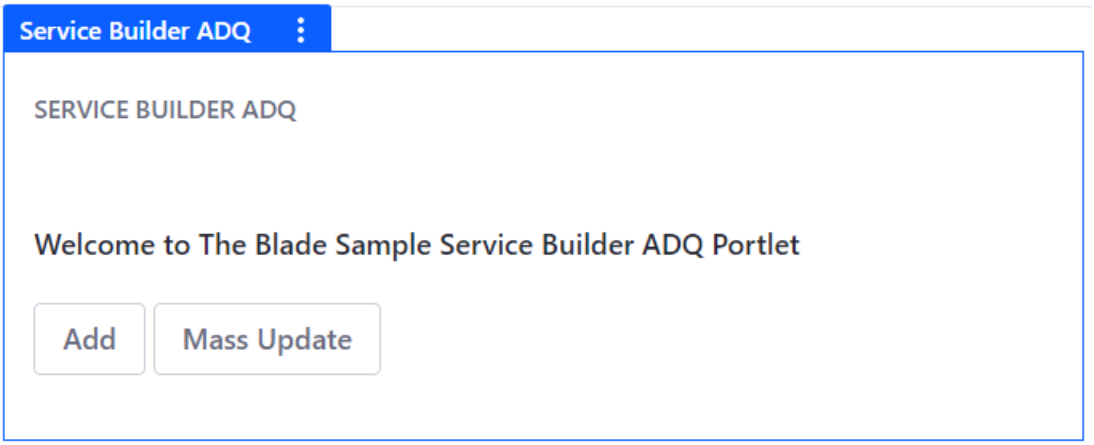

Figure 811.1: This sample provides options to add entities and perform a mass update.

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!

811.1 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.

811.2 How does this sample leverage the API(s) and/or code component?

An action request is sent to the `JSPPortlet` with a `cmd` request parameter. When the `JSPPortlet`'s `processAction` method processes the request, the value of the `cmd` parameter is parsed and then the portlet's `massUpdate` method is invoked. The `massUpdate` method, in turn, invokes the `massUpdate` method defined in the `adq-service` module's `BarLocalServiceImpl`. This is where the sample leverages the actionable dynamic query API:

```
public void massUpdate() {
    ActionableDynamicQuery adq = getActionableDynamicQuery();

    adq.setAddCriteriaMethod(
        new ActionableDynamicQuery.AddCriteriaMethod() {

            @Override
            public void addCriteria(DynamicQuery dynamicQuery) {
                dynamicQuery.add(RestrictionsFactoryUtil.lt("field3", 100));
            }

        });

    adq.setPerformActionMethod(
        new ActionableDynamicQuery.PerformActionMethod<Bar>() {

            @Override
            public void performAction(Bar bar) {
                int field3 = bar.getField3();

                field3++;
                bar.setField3(field3);

                updateBar(bar);
            }

        });

    try {
        adq.performActions();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

For more information on the actionable dynamic query API, visit its dedicated tutorial.

CHAPTER 812

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.

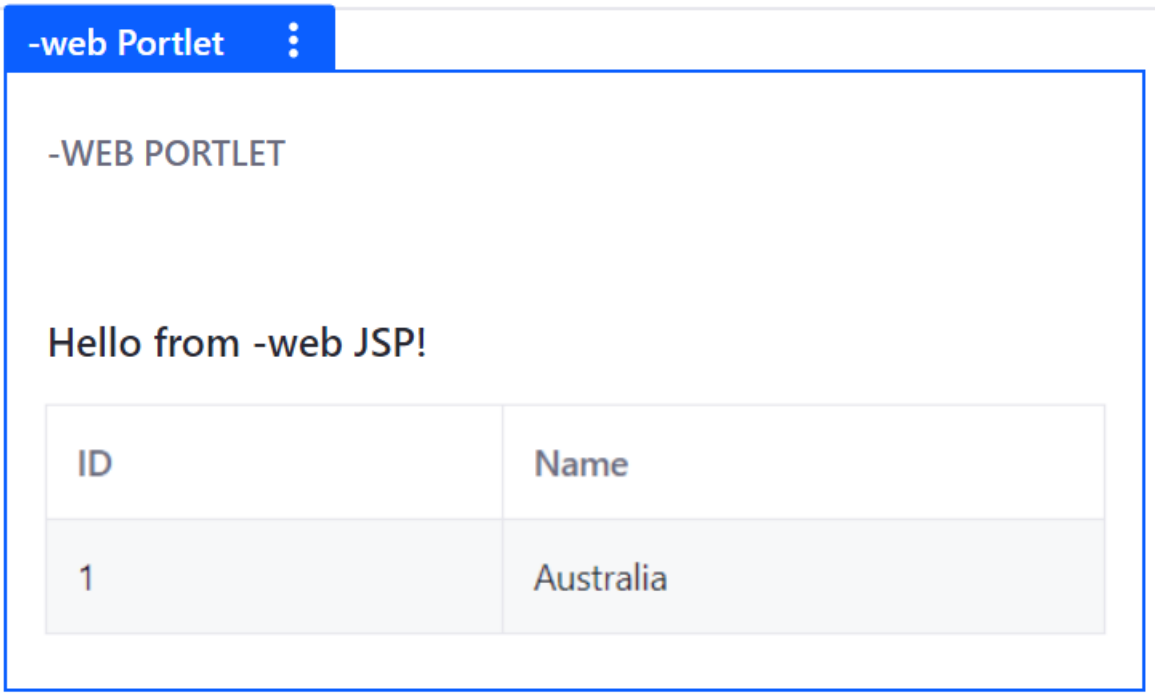

Figure 812.1: This sample prints out the values previously inputted into the database.

A sample table is printed in the portlet's view, representing the info inputted into the database.

812.1 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.

812.2 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.

812.3 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.

812.4 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.

CHAPTER 813

SERVICE BUILDER APPLICATION USING EXTERNAL DATABASE VIA JNDI

The apps/service-builder/jndi sample demonstrates how to connect a Liferay Service Builder application to an external database via a JNDI connection configured on the application server. 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 your application server to use it.

Important: Connecting to an external data source using JNDI is broken in Portal CE 7.2 GA1 and GA2, and in DXP 7.2 releases prior to FP5/SP2. See LPS-107733 for details.

Follow the steps below to make the required preparations before deploying the application.

1. Create an external database based on sample application's service.xml.

   service.xml:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.2.0//EN" "http://www.liferay.com/dtd/liferay-service-builder_7_2_0.dtd">

<service-builder package-path="com.liferay.blade.samples.jndiservicebuilder">
    <namespace>REGION</namespace>
    <!--<entity data-source="sampleDataSource" local-service="true" name="Foo" remote-service="false" session-factory="sampleSessionFactory" table="bar" tx-manager="sampleTransactionManager uuid="true"">-->
    <entity
        data-source="extDataSource"
        local-service="true"
        name="Region"
        remote-service="false"
        table="region"
        uuid="false"
    >
        <column db-name="id" name="regionId" primary="true" type="long" />
        <column db-name="name" name="regionName" type="String" />
    </entity>
</service-builder>
```

The entity's data source name `extDataSource` is arbitrary but must be specified in the data source configuration in the application server (next step).

Here are MariaDB commands to create the database:

```
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');
```

The database name is arbitrary; the data source configuration in your application server (next step), however, must specify this same database. The database table called `region` represents the service entity. The table has a `BIGINT` column called `Id` and a `VARCHAR(255)` column called `Name`.

Add at least one record to this table. Running `select * from region;` should return the record you added.

2. In your application server configuration, define a JNDI connection to your database and map it to the `data-source` name (i.e., `extDataSource`) that the sample `service.xml` entities specify.

   For example, if Tomcat is your application server, open your `[LIFERAY_HOME]/tomcat-version/conf/server.xml` file and add a `Resource` element like this one 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?useUnicode=true&amp;characterEncoding=UTF-8&amp;useFastDateParsing=false"
       username="[place user name here]"
       password="[place password here]"
       maxActive="20"
       maxIdle="5"
       maxWait="10000"
   />
   ```

   Replace the user name and password values and see the Database Templates for the URL parameters to use for your database.

3. If you are using Tomcat, open your `[LIFERAY_HOME]/tomcat-version/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.

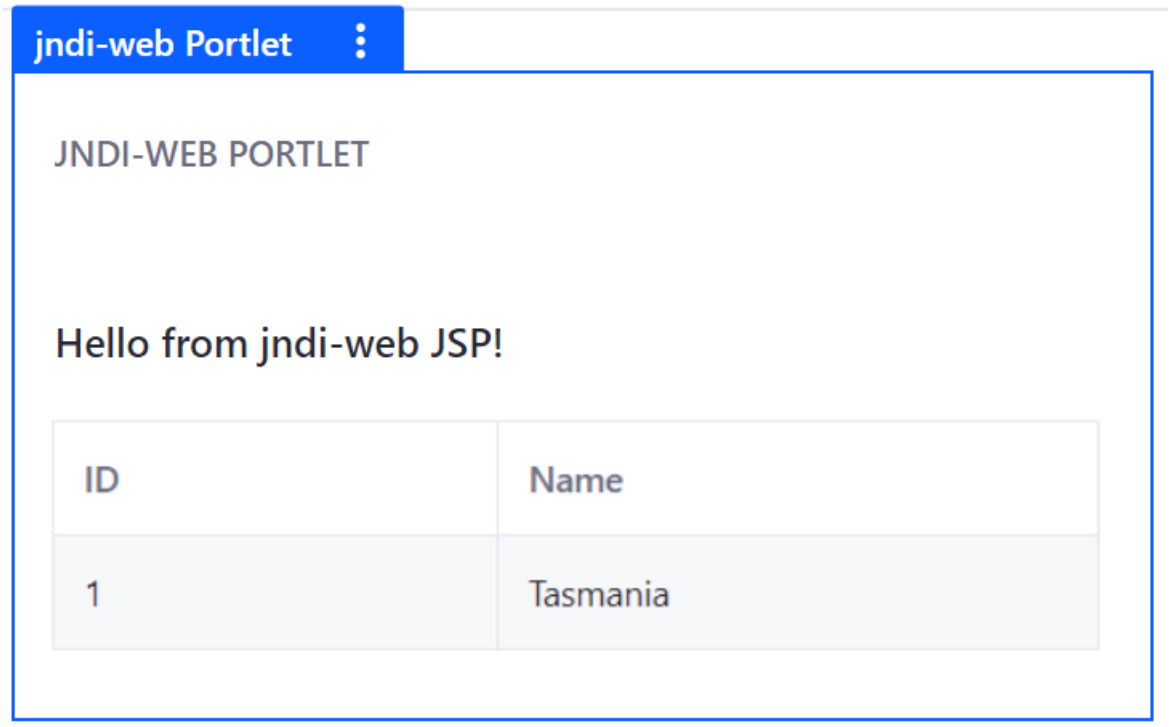

Figure 813.1: This sample prints out the values previously inputted into the database.

A sample table is printed in the portlet's view, representing the info inputted into the database.

813.1 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.

813.2 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.

813.3 Additional Information

- Connecting to an External Data Source

CHAPTER 814

WORKFLOW SAMPLES

This section focuses on Liferay's Workflow Framework sample projects built with various build tools. You can view these samples by visiting the `apps/workflow` folder corresponding to your preferred build tool:

- Gradle Workflow sample apps
- Liferay Workspace Workflow sample apps
- Maven Workflow sample apps

The following Workflow samples are documented:

- Workflow application
- Workflow application with Asset Integration

Visit a particular sample page to learn more!

CHAPTER 815

WORKFLOW ASSET APPLICATION

This sample demonstrates workflow enabling a model entity that is an asset.

To see the Workflow sample in action, complete the following steps:

1. Add the sample widget to a page by navigating to *Add* (+) → *Widgets* → *Sample* → *Workflow Asset* and dragging it to the page.
2. Go to *Control Panel* → *Workflow* → *Process Builder* → *Configuration* and assign a workflow to the Qux entity.
3. Select the app's *Add* button and add an entity. Do this several times to create multiple entities.
4. Go to *User* → *My Workflow Tasks* → *Assigned to My Roles* and assigned the task to me and Approve the Task.

Now you've taken the entity and successfully run it through a workflow.

815.1 What API(s) and/or code components does this sample highlight?

This sample demonstrates Liferay DXP's Workflow Handler API. Specifically, it demonstrates how to create a `WorkflowHandler` for your custom entity that is asset enabled.

815.2 How does this sample leverage the API(s) and/or code component?

The basic implementation of `WorkflowHandler` is done via extension of the `BaseWorkflowHandler` class. This is where the sample leverages the basic methods required for the entity's `WorkflowHandler`.

```
@Override
public String getClassName() {
  return Qux.class.getName();
}

@Override
public String getTitle(long classPK, Locale locale) {
  return String.valueOf(classPK);
```

```
}

@Override
public String getType(Locale locale) {
  return ResourceActionsUtil.getModelResource(locale, getClassName());
}

@Override
public Qux 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));

  return _quxLocalService.updateStatus(userId, classPK, status);
}
```

For more information on the workflow framework, visit its dedicated documentation.

CHAPTER 816

WORKFLOW APPLICATION

The basic sample demonstrates workflow enabling an entity that is not an asset.

To see the Workflow sample in action, complete the following steps:

1. Add the sample widget to a page by navigating to *Add* (+) → *Widgets* → *Sample* → *Workflow Basic* and dragging it to the page.
2. Go to *Control Panel* → *Workflow* → *Process Builder* → *Configuration* and assign a workflow to the Baz entity.
3. Select the app's *Add* button and add an entity. Do this several times to create multiple entities.
4. Go to *User* → *My Workflow Tasks* → *Assigned to My Roles* and assigned the task to me and Approve the Task.

Now you've taken the entity and successfully run it through a workflow.

816.1 What API(s) and/or code components does this sample highlight?

This sample demonstrates Liferay DXP's Workflow Handler API. Specifically, it demonstrates how to create a `WorkflowHandler` for your custom entity.

816.2 How does this sample leverage the API(s) and/or code component?

The basic implementation of `WorkflowHandler` is done via extension of the `BaseWorkflowHandler` class. This is where the sample leverages the basic methods required for the entity's `WorkflowHandler`.

```
@Override
public String getClassName() {
  return Baz.class.getName();
}

@Override
public String getTitle(long classPK, Locale locale) {
  return String.valueOf(classPK);
```

```
}

@Override
public String getType(Locale locale) {
  return ResourceActionsUtil.getModelResource(locale, getClassName());
}

@Override
public Baz 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));

  return _bazLocalService.updateStatus(userId, classPK, status);
}
```

For more information on the workflow framework, visit its dedicated documentation.

CHAPTER 817

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.

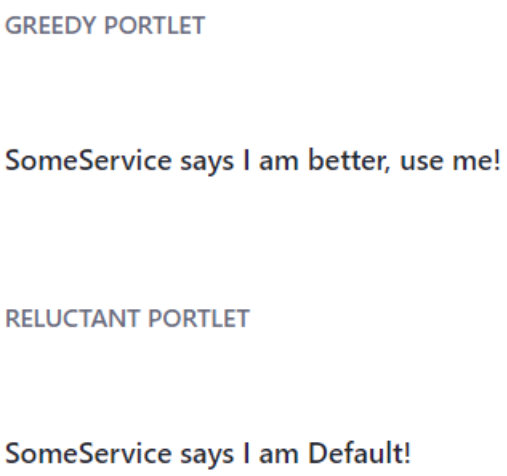

Figure 817.1: The Greedy Policy Option app provides two portlets that only print text. You'll dive deeper later to discover their interesting capabilities involving services.

These two portlets do not provide anything useful out-of-the-box. They are, however, very effective at demonstrating the ability to reference services using greedy and reluctant policy options. You'll learn how to do this later.

817.1 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`

817.2 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.

817.3 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 → Widgets → 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.

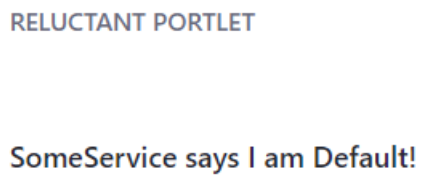

Figure 817.2: *Reluctant Portlet* displays the message "SomeService says I am Default!"

4. Add the *Greedy Portlet* from the *Add → Widgets → Sample* category to a site page.

   The portlet displays the message "SomeService says I am better, use me!". Both portlets are referencing a DefaultService instance.

Since DefaultService is the only active SomeService instance in the system, the portlets refer to it for their SomeService fields.

The DefaultService and portlets *Reluctant Portlet* and *Greedy Portlet* are active. Let's activate a higher ranked SomeService instance and see how the portlets react to it.

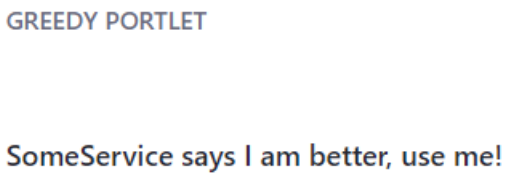

Figure 817.3: *Greedy Portlet* displays the message "SomeService says I am better, use me!"

817.4 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.

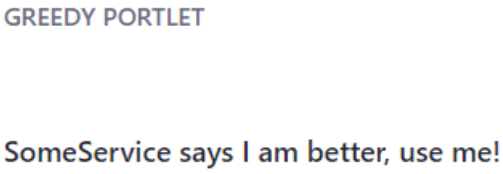

Figure 817.4: The *Greedy Portlet* is using a `HigherRankedService` instance

Next, learn how to bind the *Reluctant Portlet* to a `HigherRankedService` instance.

817.5 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.

Reluctant Portlet displays a new message "SomeService says I am better, use me!".

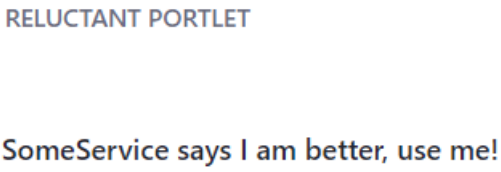

Figure 817.5: *Reluctant Portlet* is using the `HigherRankedService` instance instead of a `DefaultService` instance.

Reluctant Portlet is using `HigherRankedService` instance instead of a `DefaultService` instance. You've configured *Reluctant Portlet* to use a `HigherRankedService` instance!

817.6 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 818

KOTLIN PORTLET

The Kotlin Portlet sample provides an input form that accepts a name. Once submitting a name, the portlet renders a greeting message.

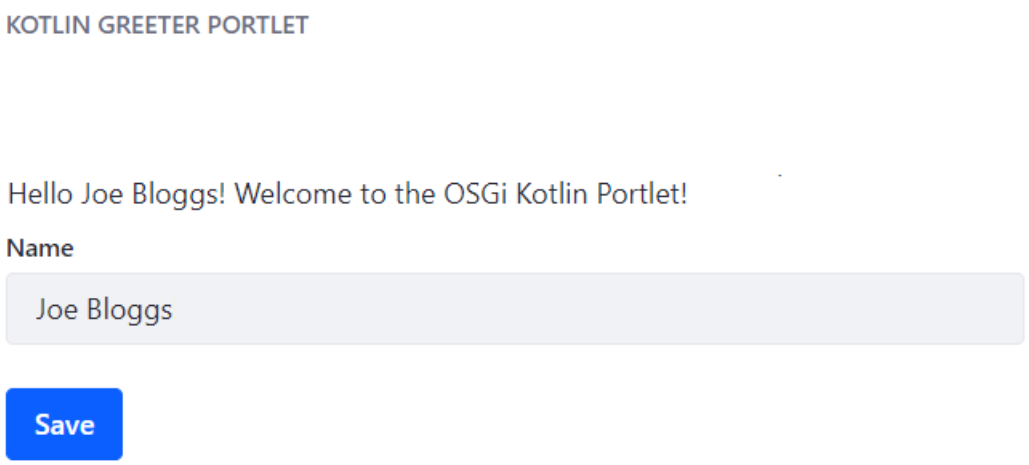

Figure 818.1: After saving the inputted name, it's displayed as a greeting on the portlet page.

818.1 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.

818.2 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`.

818.3 Where Is This Sample?

This sample is built with the following build tools:

- Gradle
- Liferay Workspace

CHAPTER 819

SHARED LANGUAGE KEYS

The Shared Language Keys sample provides a JSP portlet that displays language keys.

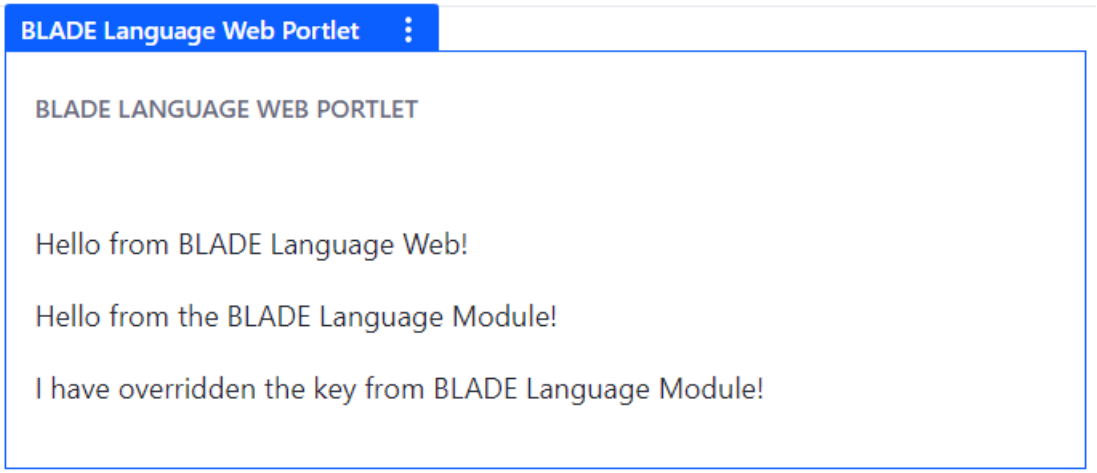

Figure 819.1: The sample JSP portlet displays three language keys.

The language keys displayed in the portlet come from two different modules.

819.1 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.

819.2 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:

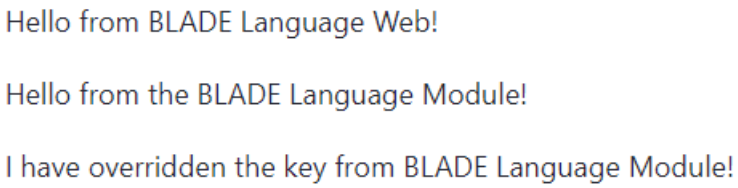

Figure 819.2: The Language Web portlet displays three phrases, two of which are shared from a different module.

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.

819.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 820

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.

820.1 What API(s) and/or code components does this sample highlight?

This sample leverages the PanelApp API.

820.2 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.

820.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 821

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!

CHAPTER 822

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.

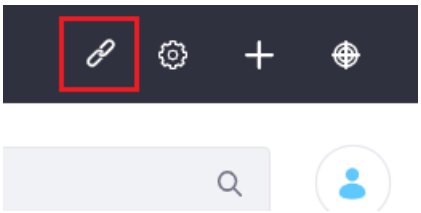

Figure 822.1: The User area of the Control Menu is provided an additional link button when the Control Menu Entry sample is deployed to Liferay DXP.

The button navigates the user to Liferay's website: https://www.liferay.com.

822.1 What API(s) and/or code components does this sample highlight?

This sample leverages the ProductNavigationControlMenuEntry API.

822.2 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.

822.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 823

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:

823.1 What API(s) and/or code components does this sample highlight?

This sample leverages the PortletConfigurationIcon API.

823.2 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.

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

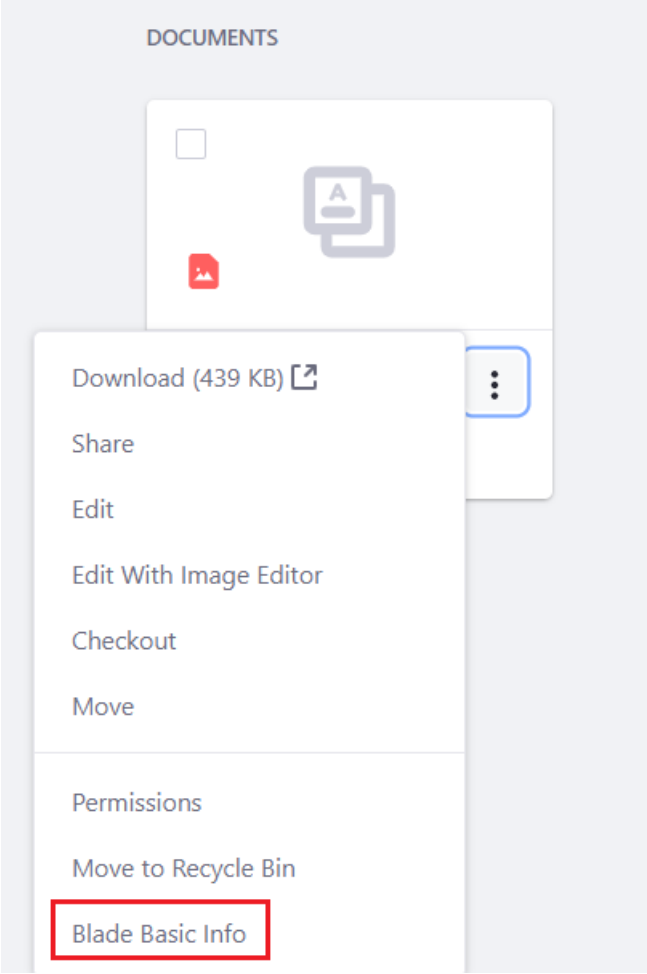

Figure 823.1: The new *Blade Basic Info* option is available from the entry's Options menu.

(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.

823.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 824

GOGO SHELL COMMAND

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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.
4. Execute `usercount` to execute the new custom command. The number of users on your running Liferay Portal installation is printed.

824.1 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.

824.2 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:

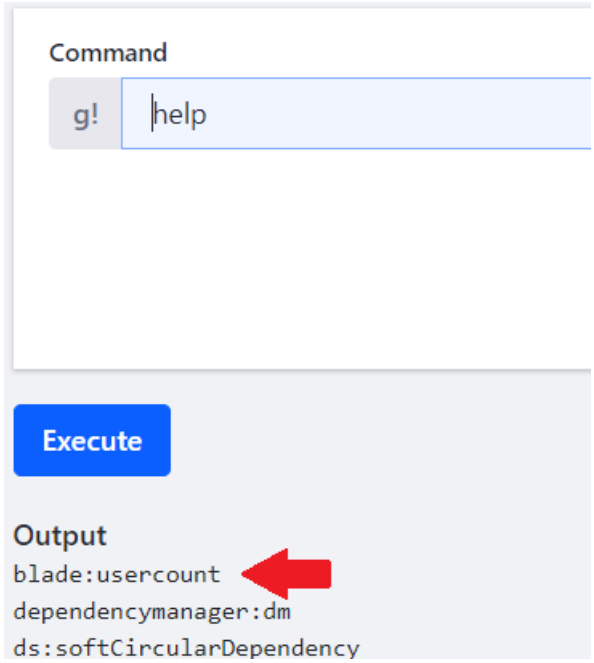

Figure 824.1: The sample Gogo shell command is listed with all the available commands.

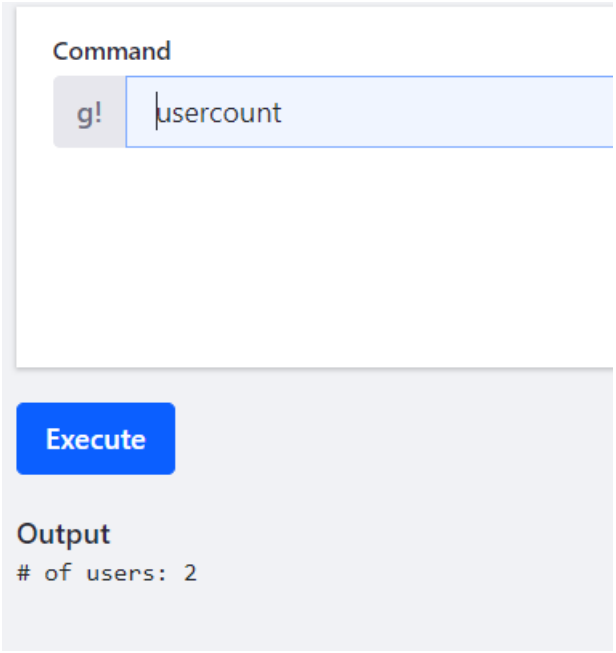

Figure 824.2: The outcome of executing the `usercount` command.

```
@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.

824.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 825

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.

825.1 What API(s) and/or code components does this sample highlight?

This sample leverages the IndexSettingsContributor API.

Figure 825.1: This sample added four new index properties.

825.2 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
)

> If using Elasticsearch 7, the value of the `service` property is instead `com.liferay.portal.search.elasticsearch7.settings.IndexSettingsContributo
```

This sample demonstrates the essentials needed to contribute your own index settings.

825.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 826

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
```

826.1 What API(s) and/or code components does this sample highlight?

This sample leverages the IndexerPostProcessor API.

826.2 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 article.

826.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 827

MODEL LISTENER

The Model Listener sample demonstrates adding a custom model listener to a Liferay Portal out-of-the-box entity. When deploying this sample with no customizations, a custom model listener is added to the portal's layouts, listening for `onBeforeCreate` events. This means that any page creation will trigger this listener, which will execute before the new page is created.

For example, if a new page is added with the name *My Test Page*, the following message is printed to the console:

```
==> ./catalina.out <==
About to create layout: My Test Page
```

Figure 827.1: The sample model listener's message in the console.

You can also verify that the model listener sample was executed by navigating to the new page's *Options → Configure Page → SEO* option. The HTML Title field looks like this:

827.1 What API(s) and/or code components does this sample highlight?

This sample leverages the ModelListener API.

827.2 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

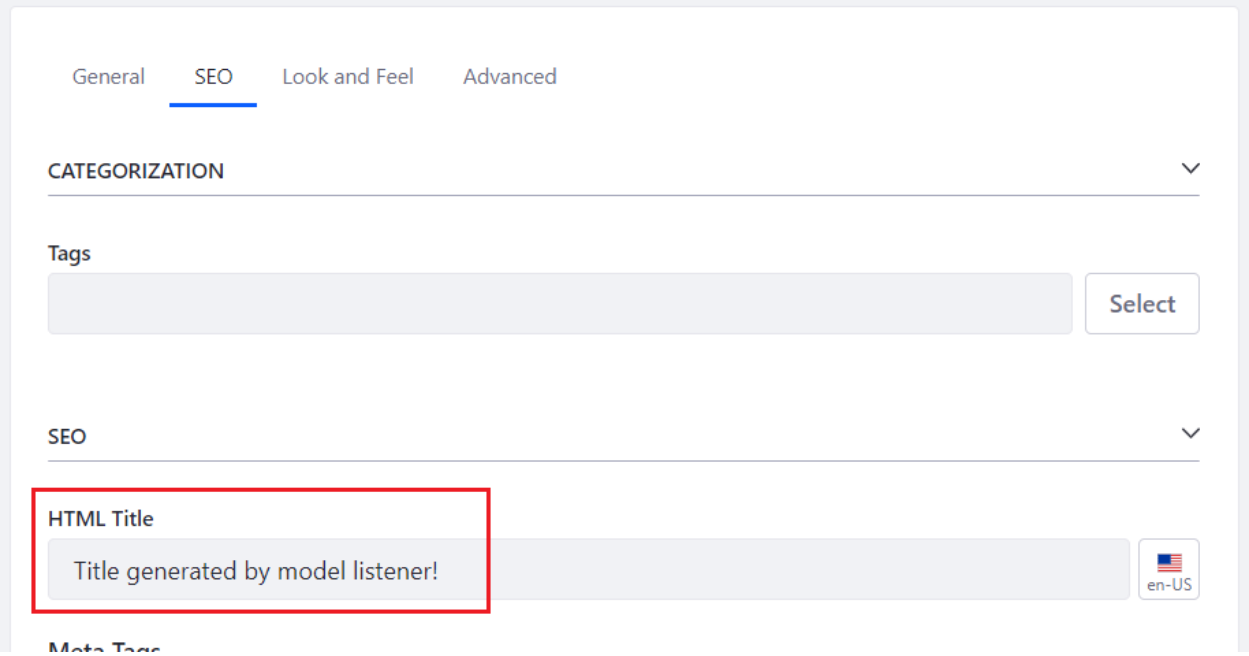

Figure 827.2: The page's HTML title updated by the model listener sample.

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.

827.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 828

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*.

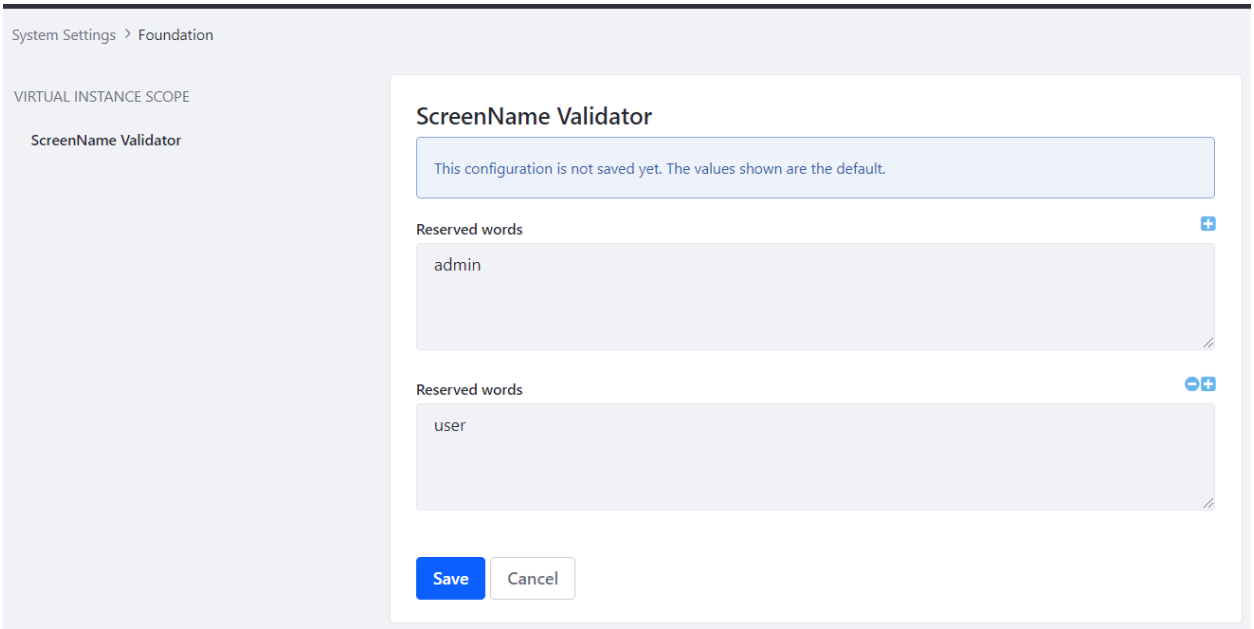

Figure 828.1: Enter reserved words for the screen name validator.

You can test this sample by following the following steps:

1. Deploy the Screen Name Validator to your portal installation.
2. Navigate to the *Control Panel → Users → Users and Organizations* menu.
3. Create a new user by selecting the *Add User* ( ) button.
4. Adding a screen name that contains the word *admin* or *user*.

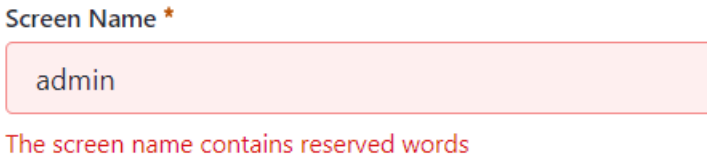

Figure 828.2: The error message displays when inputting a reserved word for the screen name.

828.1 What API(s) and/or code components does this sample highlight?

This sample leverages the ScreenNameValidator API.

828.2 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.

828.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 829

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.

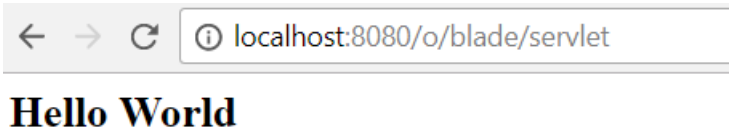

Figure 829.1: The servlet displays *Hello World* from the configured servlet page URL.

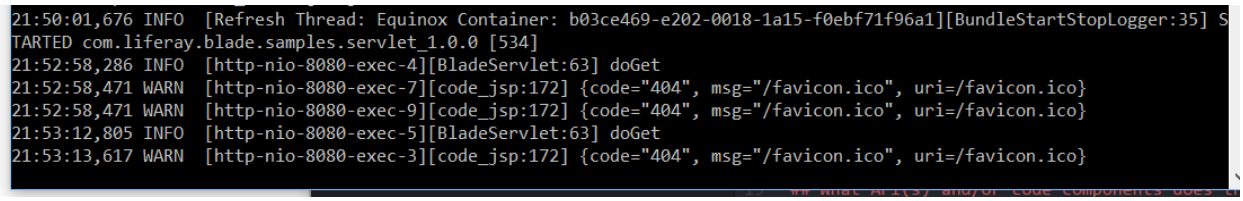

Figure 829.2: The servlet also logs info in the console.

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.

829.1 What API(s) and/or code components does this sample highlight?

This sample leverages the HttpServlet API.

829.2 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.

829.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 830

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!

CHAPTER 831

MODULE JSP OVERRIDE

The Module JSP Override sample conveys how to override an application's JSP by leveraging OSGi fragment modules. This is not the recommended practice for overriding JSPs in 7.0. See the Customizing JSPs article for better options.

This example overrides the default `login.jsp` file in the `com.liferay.login.web` bundle by adding the red text *changed* to the Sign In form.

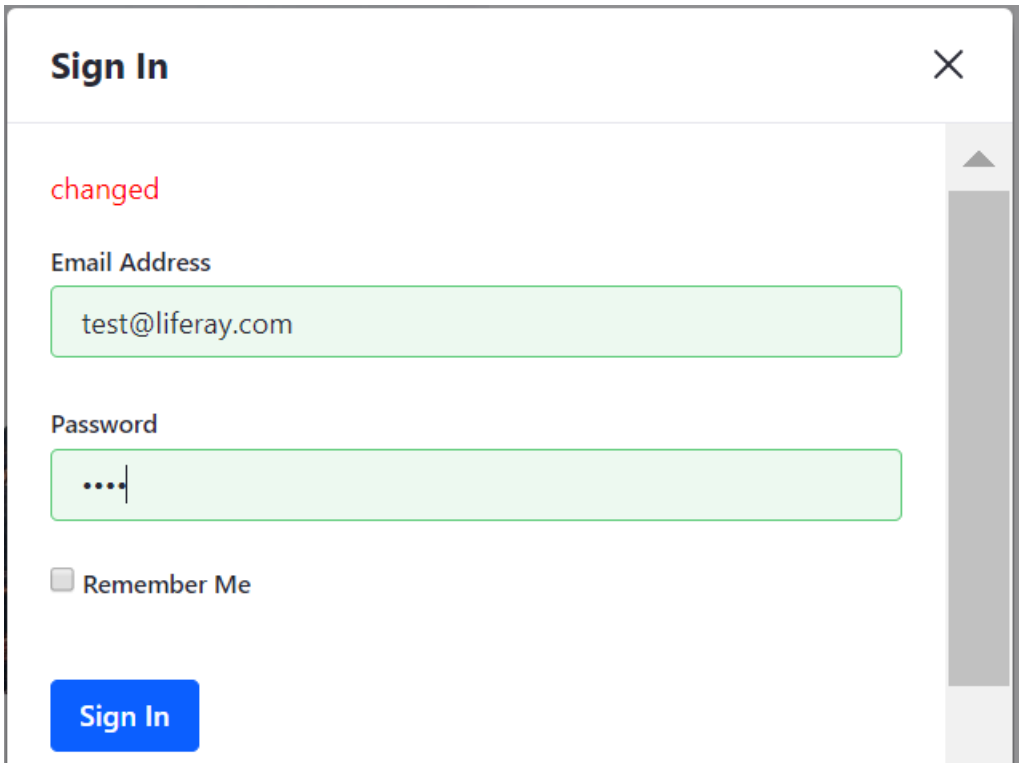

Figure 831.1: The customized Sign In form with the new *changed* text.

831.1 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.

831.2 How does this sample leverage the API(s) and/or code component?

You can create your own JSP override by

- Declaring the fragment host.
- Providing the JSP that will override the original one.

To properly declare the fragment host in the `bnd.bnd` file, you must specify the host module's (where the original JSP is located) Bundle Symbolic Name and the host module's exact version to which the fragment belongs. In this example, this is configured like this:

```
Fragment-Host: com.liferay.login.web;bundle-version="1.0.0"
```

Then you must provide the new JSP intended to override the original one. Be sure to mimic the host module's folder structure when overriding its JAR. For this example, since the original JSP is in the folder `/META-INF/resources/login.jsp`, the new JSP file resides in the folder `src/main/resources/META-INF/resources/login.jsp`.

If needed, you can also target the original JSP following one of the two possible naming conventions: `original` or `portal`. This pattern looks like

```
<liferay-util:include
    page="/login.original.jsp"
    servletContext="<%= application %>"
/>
```

or

```
<liferay-util:include
    page="/login.portal.jsp"
    servletContext="<%= application %>"
/>
```

This approach can be used to override any application JSP (i.e., JSPs residing in a module). You can also add new JSPs to an existing module with this technique. For more information on other ways to customize JSPs, see the Customizing JSPs articles.

831.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 832

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*.

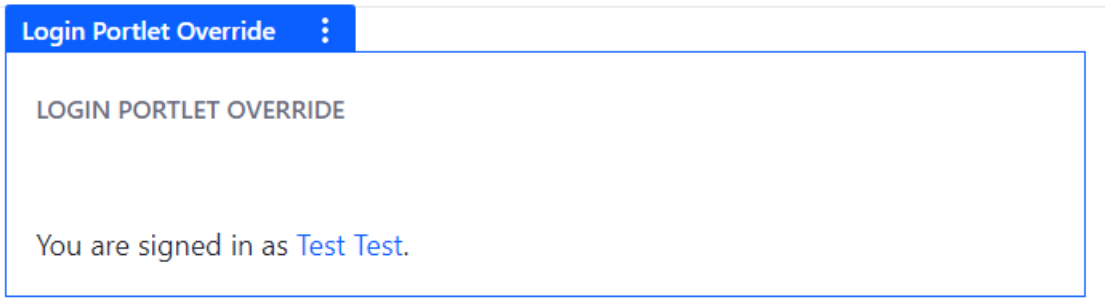

Figure 832.1: The customized Login portlet displays the new language key.

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.

832.1 What API(s) and/or code components does this sample highlight?

This sample leverages the `Provide-Capability` OSGi manifest header.

832.2 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 article.

For more information on using a resource bundle to override a module's language keys, see the Overriding a Module's Language Keys tutorial.

832.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 833

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!

CHAPTER 834

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.

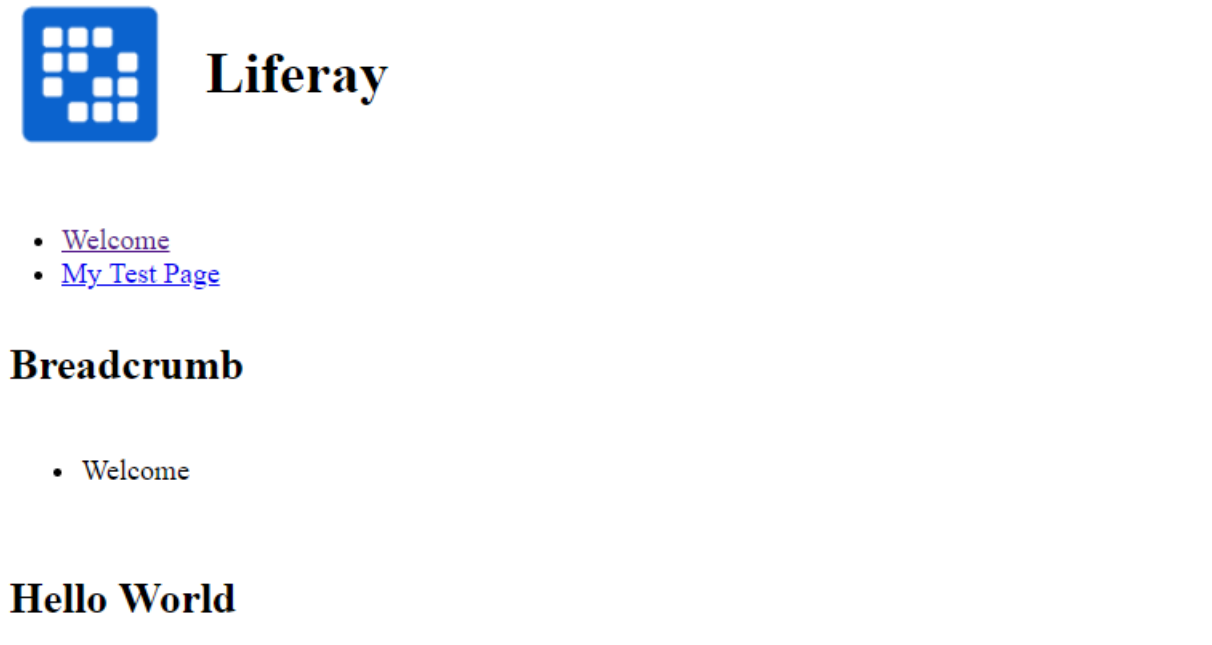

Figure 834.1: A theme based off of the Styled base theme is created when the Theme Blade sample is deployed to Liferay Portal.

For more information on themes, visit the Introduction to Themes tutorial.

834.1 What API(s) and/or code components does this sample highlight?

This sample demonstrates a way to create a simple theme in Liferay DXP.

834.2 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.

834.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 835

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.

835.1 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.

835.2 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.

835.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 836

THEME CONTRIBUTOR

The Theme Contributor sample contributes updates to the UI of the theme body, Control Menu, Product Menu, and Simulation Panel. When deploying this sample with no customizations, the colors of the theme and aforementioned menus are updated.

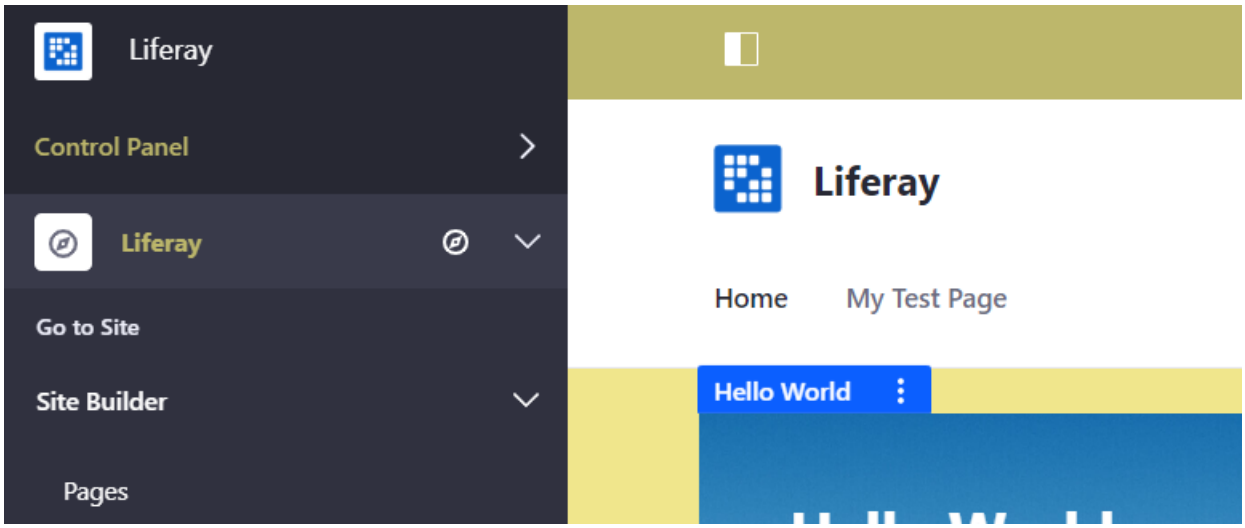

Figure 836.1: Your Liferay DXP pages and menu fonts now have a yellow tint.

Also, there's a simple JavaScript update that is provided, which logs a message to the browser's console window that states *Hello Blade Theme Contributor!*.

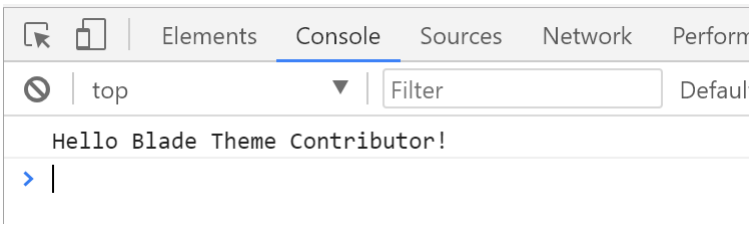

Figure 836.2: The message is printed to your browser's console window using JavaScript.

836.1 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.

836.2 How does this sample leverage the API(s) and/or code component?

To modify this sample, replace the corresponding JS or SCSS file with the JavaScript or styles that you want, or add your own JS or SCSS files. For example, this sample provides an update to the Control Menu's `background-color` in its `src/main/resources/META-INF/resources/css/blade.theme.contributor/_control_menu.scss` file:

```
body {
        .control-menu {
                background-color: darkkhaki;
        }
}
```

All of the SCSS files used in this sample are imported into the main `blade.theme.contributor.scss` file:

```
@import "bourbon";
@import "mixins";

@import "blade.theme.contributor/body";
@import "blade.theme.contributor/control_menu";
@import "blade.theme.contributor/product_menu";
@import "blade.theme.contributor/simulation_panel";
```

If you add your own SCSS files, you must add them to the list of imports in the `blade.theme.contributor.scss` file.

Likewise, the sample `blade.theme.contributor.js` logs a message to your browser's console window using the following JS logic:

```
console.log('Hello Blade Theme Contributor!');
```

For more information on Theme Contributors, visit the Theme Contributors tutorial.

836.3 Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 837

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!

CHAPTER 838

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.

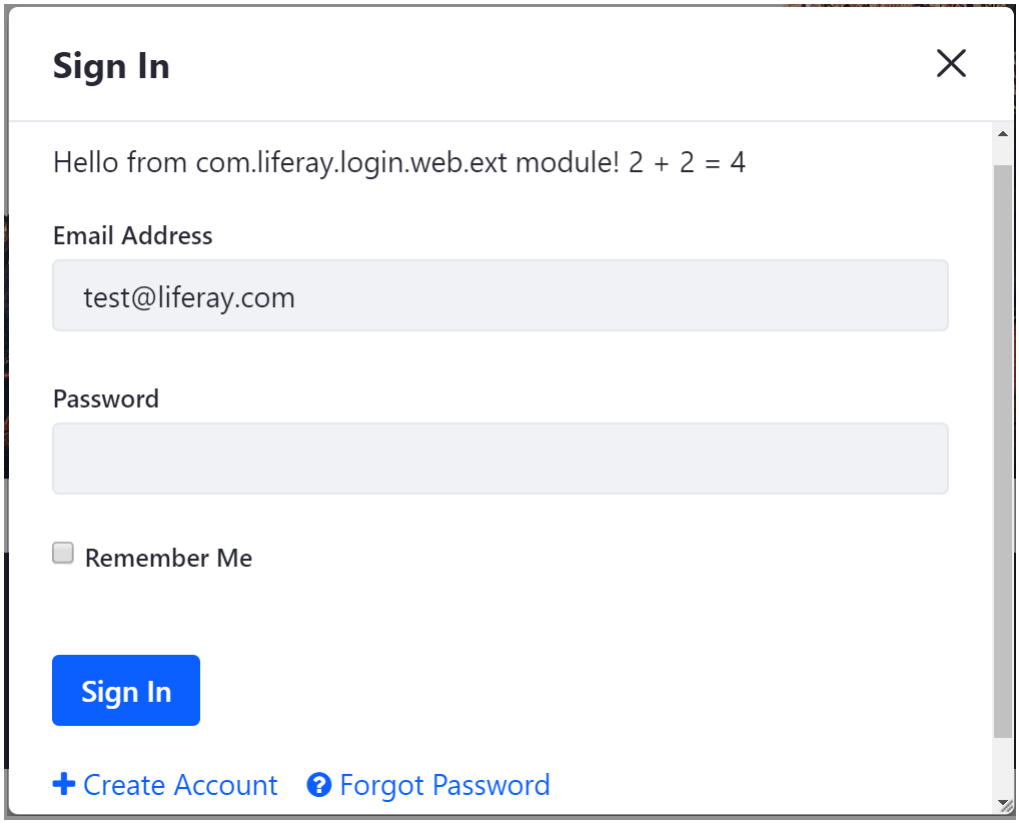

Figure 838.1: The Login Ext module customizes the original Login module.

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:

   ```
   1580|Active   |   10|com.liferay.login.web (4.0.5)
   ```

   Make note of the ID (e.g., `1580`).
3. Stop the bundle:

   ```
   stop 1580
   ```

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 deployment feature.

838.1 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.

838.2 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: "4.0.5"
```

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.

838.3 Where Is This Sample?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

CHAPTER 839

SEGMENTATION AND PERSONALIZATION REFERENCE

Browse this section's reference articles for additional information on the Segmentation and Personalization framework.

CHAPTER 840

DEFINING SEGMENTATION CRITERIA

There are three categories for defining Segment criteria:

- User Properties
- Organization Properties
- Session Properties

There are several types of information that can be collected by the User Segment interface. Some data is entered in text boxes, while others use selectors to select specific criteria or tools like a date picker. In addition, some fields use an operator, which, depending on the specific context lets you select the relationship between the user or agent data and the criteria:

- *equals*
- *not equals*
- *greater than*
- *greater than or equals*
- *less than*
- *less than or equals*
- *contains*
- *does not contain*

Depending on the nature of the criteria, the operator selection may contain different combinations. For example, the *Date* selection described below contains options for all the above option except *contains* and *does not contain,* whereas the *Email Address* selection has *equals, not equals, contains* and *does not contain*.

In between each criteria and each category, you can define an "and" or "or" conjunction. For "and" all criteria must be true in order for the criteria to be satisfied. With "or" it will be true if any of the defined criteria are true. You can also mix operators to create complex cases.

840.1 User Properties

The following are the criteria available for defining user properties:

Date Modified: Provides a date picker and an relationship selector to select the date that user information was last changed

Email Address: Provides a text box to enter the email provided in the user's profile.

First Name: Enter the first name provided in the user's profile.

Group: Select a site that the user is a member of.

Job Title: Enter the job title provided in the user's profile.

Last Name: Enter the last name provided in the user's profile.

Role: Select a role that the user is a member of.

Screen Name: Enter the users' screen name.

Team: Select a team that the user is a member of.

User Group: Select a user group that the user is a member of.

User: Select a specific user from a list.

Name: The full name of the user.

840.2 Organization Properties

Date Modified: Enter the date that the organization information was last modified.

Name: Enter the name of the organization.

Hierarchy Path: Enter the name of an ancestor organization.

Organization: Select a specific organization.

Parent Organization: Select a specific parent organization.

Type: Select the type of organization, if organization types have been defined.

840.3 Session Properties

Browser: Enter a property from the browser.

Cookies: Enter the name of a browser cookie.

Device Brand: Enter the brand name of the device being used.

Device Model: Enter the model name of the device being used.

Device Screen Resolution Height: Enter the screen resolution height value.

Device Screen Resolution Width: Enter the screen resolution width value.

Language: Select the current Language.

Last Sign In Date: Select the date of the user's last sign in.

Local Date: Select the current date where the user is located.

Referrer URL: Enter the URL that the user last visited.

Signed In: Select whether the user is signed in.

URL: Enter the current URL.

User Agent: Enter a User Agent property.

CHAPTER 841

TOOLING

You can write code for Liferay DXP using any standard toolset. Liferay is tool-agnostic, which 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, Maven, and NodeJS). They include

- Blade CLI: a command line interface used to build and manage Liferay Workspaces and Liferay DXP projects. This CLI is intended for Gradle or Maven development.
- Liferay Workspace: a generated Gradle/Maven environment built to hold and manage Liferay DXP projects.
- Liferay Dev Studio: an Eclipse-based IDE supporting development for Liferay DXP.
- Liferay IntelliJ Plugin: a plugin providing support for Liferay DXP development with IntelliJ IDEA.
- Liferay Theme Generator: a generator that creates themes, layouts templates, and themelets for Liferay DXP development.
- Liferay JS Generator: a generator that creates JavaScript portlets with JavaScript tooling.

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 answers your tooling questions.

CHAPTER 842

CREATING A PROJECT

Liferay provides many project templates you can use to generate starter projects formatted in an opinionated way. Visit the Project Templates reference section for more information on the available project templates. Each project template has different configurable options, so be sure to research a project template before generating it.

You can use your desired tool to generate a project. The following tools are preconfigured for Liferay project generation:

- Blade CLI
- Liferay Dev Studio
- Liferay IntelliJ Plugin
- Maven

It's recommended to create Liferay projects within a Liferay Workspace. Most tools, however, support creating projects in a standalone environment (except for IntelliJ). Visit the appropriate section to learn how to create a project with the highlighted tool.

842.1 Blade CLI

1. Print the available project templates by executing this:

   ```
   blade create -l
   ```

   Note the project template you want to generate; you'll use it in the next step.

2. Run the following command to create a Gradle project with Blade CLI:

   ```
   blade create -t [projectTemplate] [option1] [option2] ... [optionN] [projectName]
   ```

Note: 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.

The available configuration options are documented for each project template. Run `blade create --help` for the entire list of available options.

842.2 Liferay Dev Studio

1. Navigate to *File → New → Liferay Module Project*.
2. Specify the project name, location, build type, Liferay DXP version, and template type.

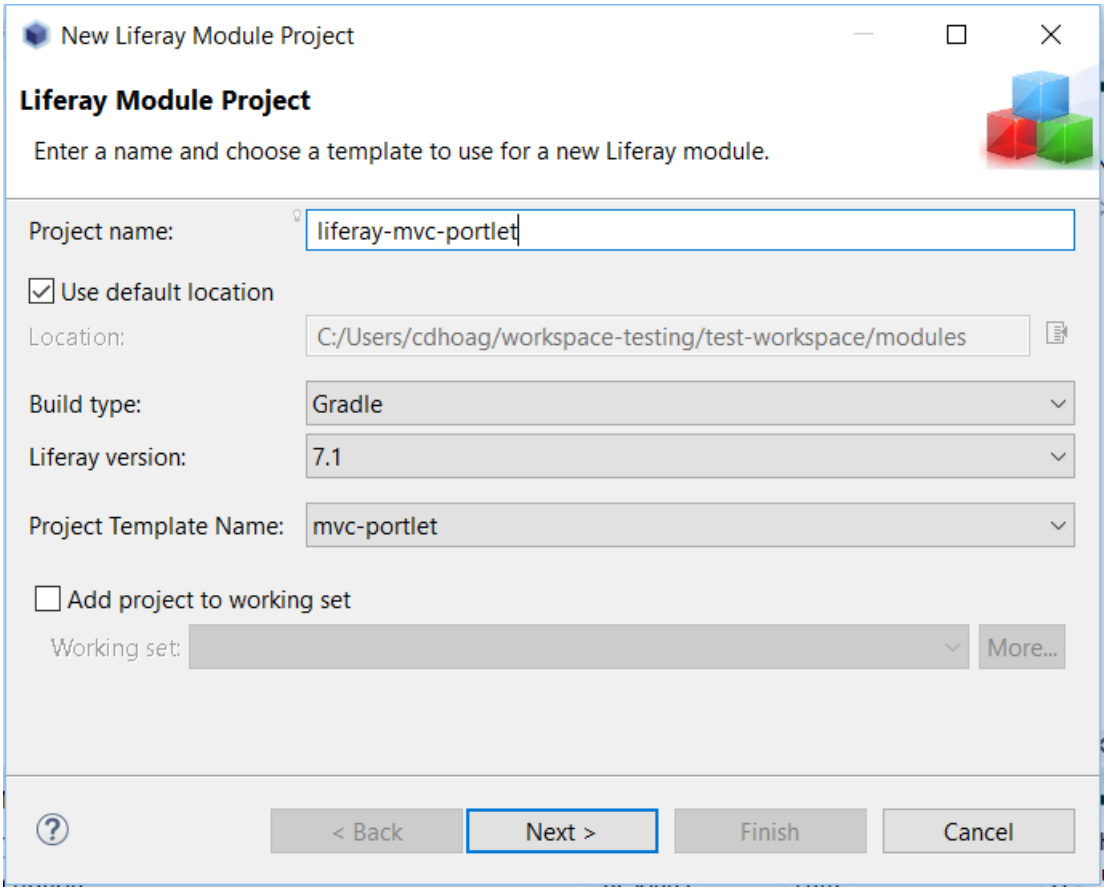

Figure 842.1: The New Liferay Module Project wizard offers project templates for JAR and WAR-based projects.

3. 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.

```
**Note:** You can also create a new component class for a pre-existing
module project. Navigate to *File* &rarr; *New* &rarr; *Liferay Component
Class*. This is a similar wizard to the previous component class wizard,
except you can select a component class template.
```

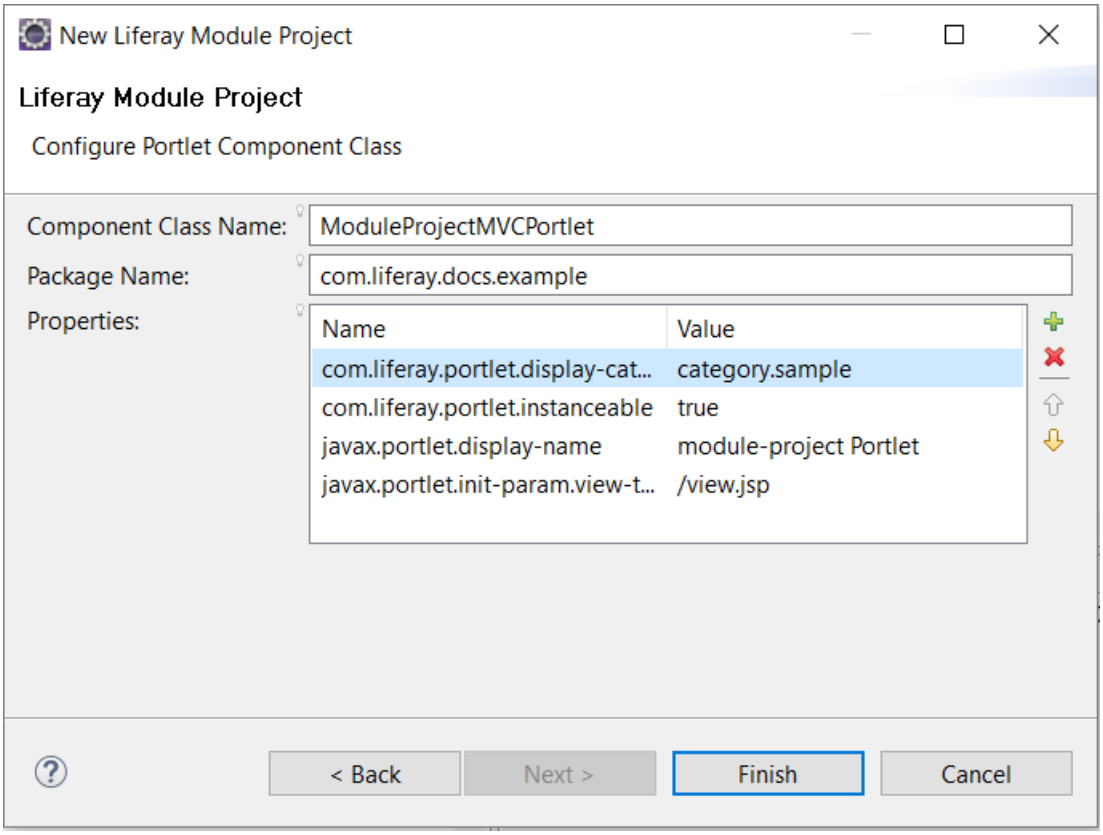

Figure 842.2: Specify your component class's details in the Portlet Component Class Wizard.

4. Click *Finish* to create your project.

842.3 Liferay IntelliJ Plugin

1. Navigate to *File* → *New* → *Liferay Module*.

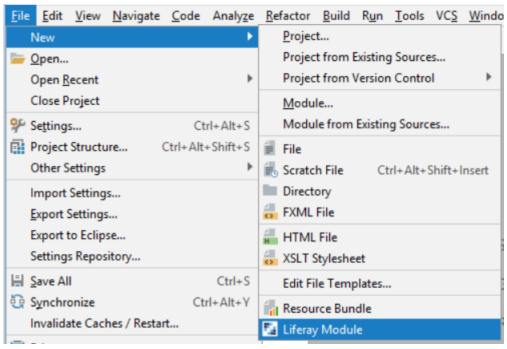

Figure 842.3: Selecting *Liferay Module* opens the New Liferay Modules wizard.

2. Select the project you want to create.

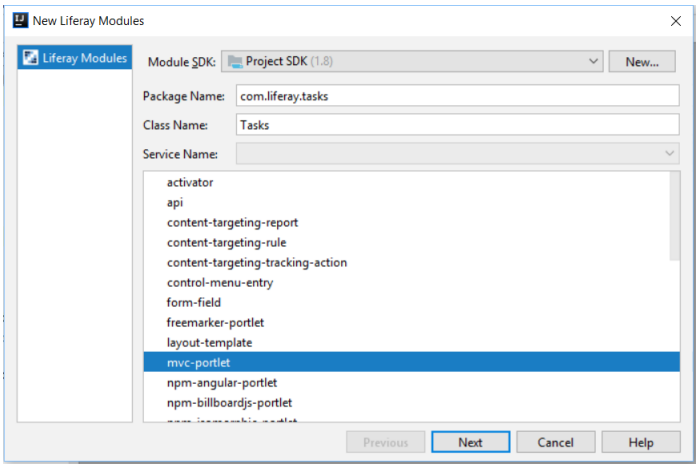

Figure 842.4: Choose the project template to create your module.

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*.

842.4 Maven

1. Execute the following Maven command:

```
mvn archetype:generate -Dfilter=liferay
```

2. Select the archetype you want to leverage and proceed through the configuration prompts.

Note: Maven projects can also be generated using Blade CLI. Follow Blade CLI's project creation instructions and insert the `-b maven` parameter in the Blade command.

Archetypes prefixed with `com.liferay.project.templates.[TYPE]` or `com.liferay.faces.archetype:[TYPE]` are compatible with 7.0. All other Liferay archetypes are legacy archetypes targeted for previous versions of Liferay DXP.

See Maven's Archetype Generation documentation for further details on how to modify the Maven archetype generation process.

CHAPTER 843

DEPLOYING A PROJECT

Deploying a project to Liferay DXP can be completed using your tool of choice. The following tools are preconfigured (or can be easily configured) for Liferay project generation:

- Blade CLI
- Gradle
- Liferay Dev Studio
- Liferay IntelliJ Plugin
- Maven

The deployment process is the same across all tools; the deployment command/action builds and 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. The Liferay Home folder is preconfigured in most cases; if it's not, ways to configure it are included below. All tools support JAR and WAR-style project deployment.

It's recommended to deploy Liferay projects within a Liferay Workspace. Most tools, however, support deploying projects in a standalone environment (except for IntelliJ). Visit the appropriate section to learn how to deploy a project with the highlighted tool.

843.1 Blade CLI

This is the recommended way to deploy Gradle and Maven projects in a Liferay Workspace via command line. Blade CLI is leveraged by Dev Studio and IntelliJ too.

Run this command to deploy your project:

```
blade deploy
```

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:

```
blade deploy -l
```

843.2 Gradle

Deploying with pure Gradle is not recommended unless you prefer to develop outside of a Liferay Workspace. Blade CLI is a better tool for deploying Liferay Gradle projects in most cases.

1. Apply the Liferay Gradle plugin in your project's `build.gradle` file:

   ```
   apply plugin: "com.liferay.plugin"
   ```

2. 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")
   }
   ```

3. Run this command to deploy your project:

   ```
   ./gradlew deploy
   ```

843.3 Liferay Dev Studio

These steps assume you've installed a Liferay server in Dev Studio.

1. Right-click the server from the Servers window and select *Add and Remove...*.
2. Add the project(s) you'd like to deploy from the Available window to the Configured window. Then click *Finish*.
3. Verify your project builds, deploys, and starts successfully by viewing the results in the Console window.

Dev Studio's deployment mechanism executes the `watch` task behind the scenes. For more information on what to expect from the `watch` task, see this article.

843.4 Liferay IntelliJ Plugin

These steps assume you've installed a Liferay server in IntelliJ. A configured Liferay Workspace is required to create and deploy Liferay projects in IntelliJ.

1. Right-click your project from within the Liferay Workspace folder structure and select *Liferay → Deploy*.

   This automatically loads a build progress window viewable at the bottom of your IntelliJ instance.

2. Verify that your project builds successfully from the build progress window. Then navigate back to your server's window and confirm it starts in your configured Liferay DXP instance.

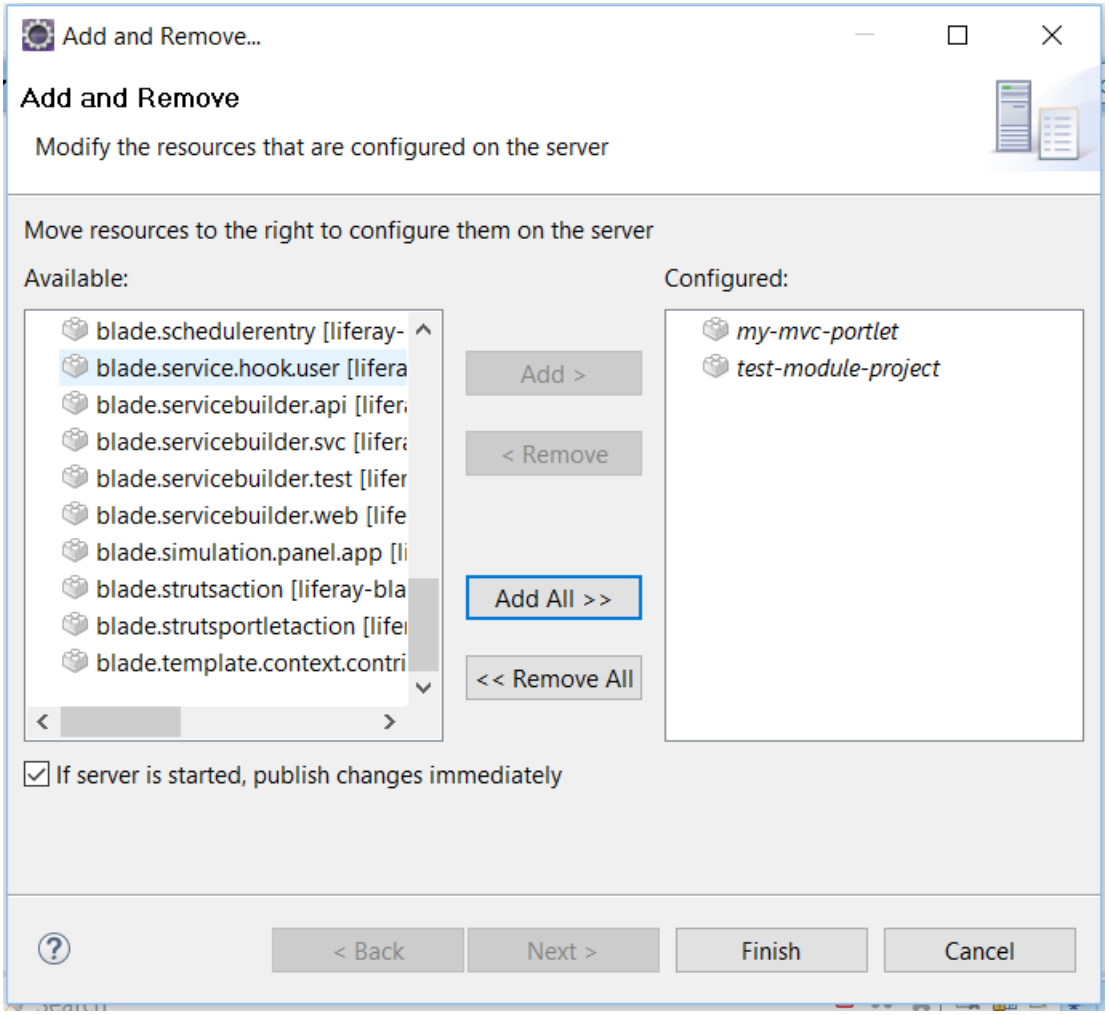

Figure 843.1: Using the this deployment method is convenient when deploying multiple projects.

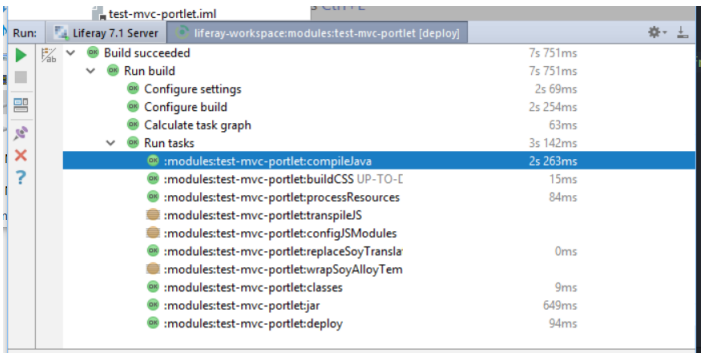

Figure 843.2: Verify that your project built successfully.

843.5 Maven

If you're developing your project in a Liferay Workspace, skip to step 3.

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>3.4.1</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. This plugin is applied in Liferay Workspace by default. The Bundle Support configuration 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. 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>C:/liferay/liferay-ce-portal-7.1-ga1</liferayHome>
</configuration>
```

3. Run this command to deploy your project:

```
mvn verify
```

CHAPTER 844

BLADE CLI

Blade CLI is a command line tool that makes it easy for Liferay developers to create, manage, and deploy Liferay projects (Gradle or Maven). Blade CLI can

- Create Liferay projects usable in any IDE or development environment
- Create/manage Liferay DXP instances
- Deploy Liferay projects
- And more

The table below describes all Blade CLI commands for the latest Blade CLI release.

Command | Description `convert` | Converts a Plugins SDK plugin project to a Gradle Workspace project. See the Running the Migration Command command for details. `create` | Creates a new Liferay project from available templates. See the Creating a Project section for Blade CLI for more information. `deploy` | Builds and deploys projects to Liferay DXP. See the Deploying a Project section for Blade CLI for more information. `extension install` | Installs an extension into Blade CLI. `extension uninstall` | Uninstalls an extension from Blade CLI. `gw` | Executes a Gradle command using the Gradle Wrapper, if detected (e.g., `blade gw tasks`). `help` | Provides information for Blade CLI's commands. `init` | Initializes a new Liferay Workspace. See the Creating a Liferay Workspace article for more information. `samples` | Generates a sample project. See the Generating Project Samples with Blade CLI article for more information. `server init` | Initializes the Liferay server configured in Liferay Workspace's `gradle.properties` file. Set the `liferay.workspace.bundle.url` property to configure the server to initialize. `server start` | Starts the Liferay server in the background. You can add the `-d` flag to start the server in debug mode. 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. See the Managing Your Liferay Server with Blade CLI article for more information. `server stop` | Stops the Liferay server. `server run` | Starts the Liferay server in the foreground. See the `server start` property for more information. `sh` | Connects to Liferay DXP, executes succeeding Gogo command, and returns output. For example, `blade sh lb` lists Liferay DXP's bundles using the Gogo shell. See the Managing Your Liferay Server with Blade CLI article for more information. `update` | Updates Blade CLI to the latest version. See the Updating Blade CLI article for details. `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. `watch` | Watches for changes to a deployed project and automatically redeploys it when changes are detected. This

command does not rebuild your project and copy it to Portal every time a change is detected, but rather, installs it 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 `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. `version` | Displays version information about Blade CLI.

For information on command options, run the command with the `--help` flag (e.g., `blade samples --help`).

Continue on to learn about leveraging Blade CLI to create and test Liferay DXP instances and projects.

CHAPTER 845

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. Do not use the Liferay Project SDK installer to update Blade CLI; instead, follow the instructions for updating Blade CLI.

If you must configure proxy settings for Blade CLI, follow the Installing Blade CLI with Proxy Requirements instructions instead.

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 without 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.

3. Select the Java Runtime to use with Blade CLI. Then click *OK*.

4. Click *Next* to step through the installer's introduction.

5. If you'd like to initialize a Liferay Workspace, you can set its location.

   Select *Don't initialize Liferay Workspace directory* if you only want to install Blade CLI. Then click *Next*.

6. If you initialized a Liferay Workspace folder, an additional option appears for selecting the Liferay product type to use with your workspace. Choose the product type and click *Next*.

7. 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.

If Blade CLI doesn't work properly on your machine, visit the Common Errors with Blade CLI article for solutions to common problems.

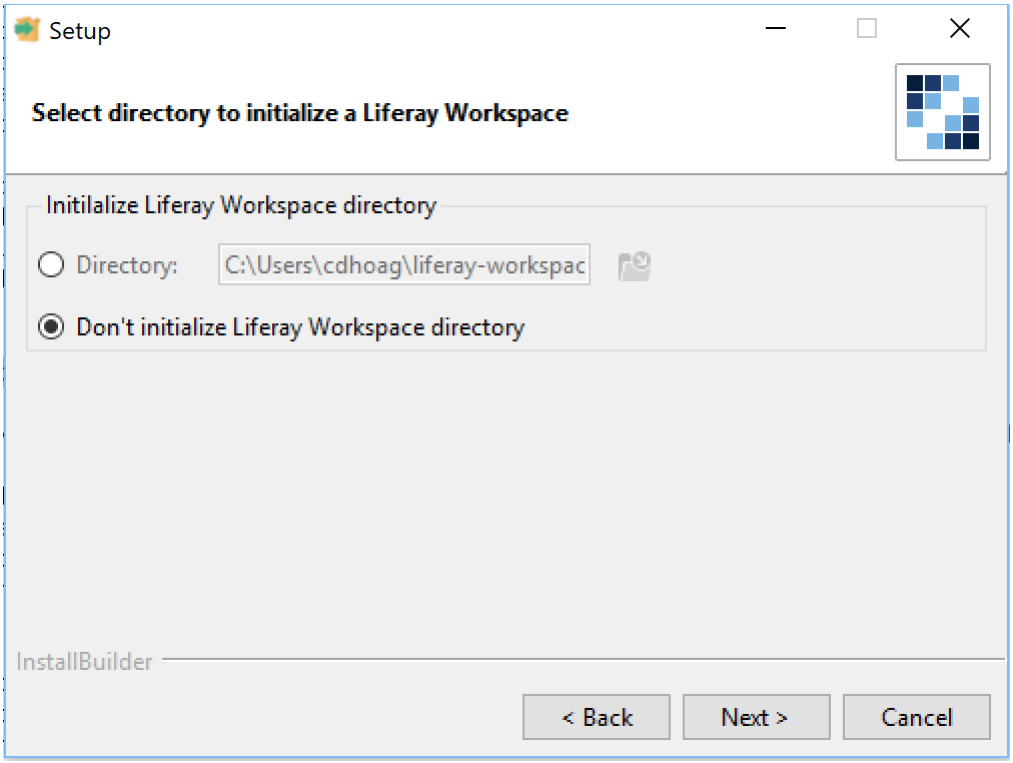

Figure 845.1: Determine where your Liferay Workspace should reside, if you want one.

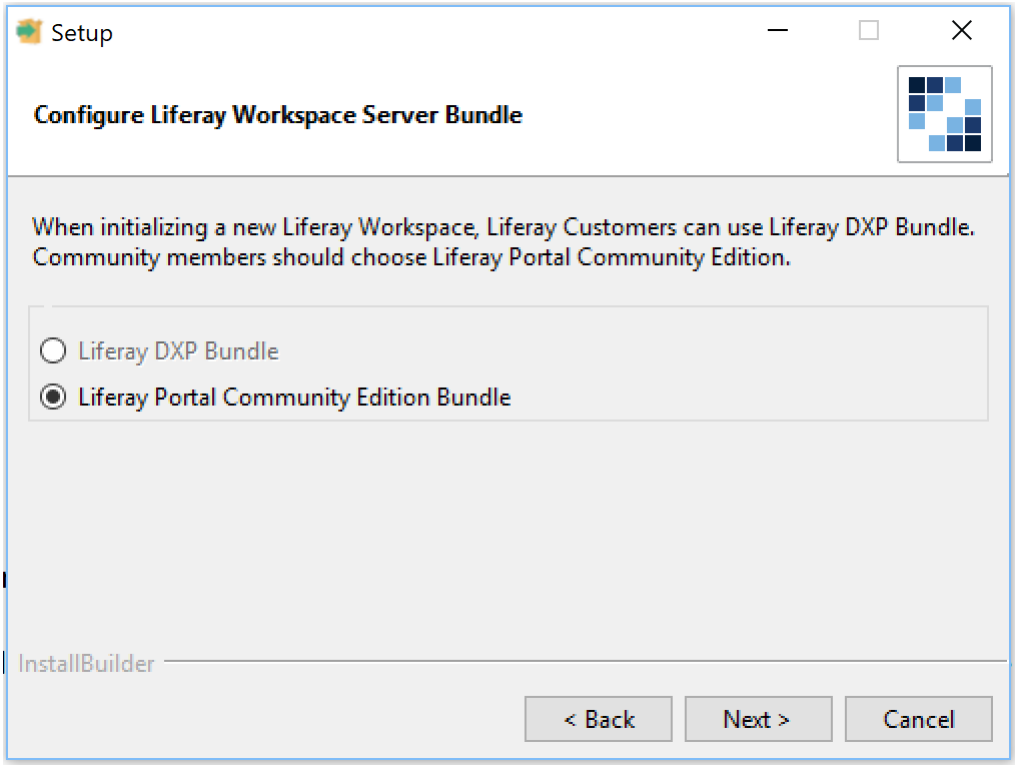

Figure 845.2: Select the product version you'll use with your Liferay Workspace.

CHAPTER 846

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:

1. Install JPM and Blade CLI using the Liferay Project SDK installer. Read the Installing Blade CLI tutorial for more details.
2. 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.

CHAPTER 847

MANAGING YOUR LIFERAY SERVER WITH BLADE CLI

You can manage a Liferay server using Blade CLI. Managing a server with Blade CLI should be done in a Liferay Workspace.

Blade CLI has commands for installing, starting, stopping, inspecting, and modifying a Liferay server:

1. Make sure you've created a Liferay Workspace. See the Creating a Liferay Workspace article for more information.

2. Initialize a Liferay server by running

   ```
   blade server init
   ```

   This downloads the Liferay DXP bundle set in your workspace's `gradle.propeties` file. See the Adding a Liferay Bundle to Workspace article for more information.

   You can initialize a server based on a defined environment by running the following command:

   ```
   blade server init --environment [ENVIRONMENT]
   ```

   For example, you could pass in the `uat` variable to generate a bundle with the configs set in the `configs/uat` workspace folder.

3. Start your Liferay server (Tomcat or Wildfly/JBoss) by running

   ```
   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. See the Blade CLI article for additional flags you can set when starting your Liferay server.

4. Examine your server's OSGi container by using Blade CLI's `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 847.1: 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 workspace by default. See the Using the Felix Gogo Shell section for more information on this tool.

5. Once you're finished modifying your Liferay bundle, you can package it as a sharable file by running this command:

   ```
   blade gw distBundle[Zip|Tar]
   ```

   This lets you create a ZIP or TAR file to share with others. This option is only available with Gradle at this time. The above command leverages Blade CLI's `gw` option, which executes the project's Gradle wrapper.

```
**Note:** You can avoid deploying a module inside your workspace's
`modules/` folder when `distBundle[Zip|Tar]` is called by adding the
following snippet to your workspace's `build.gradle` file:

```groovy
distBundle {
    exclude "com.liferay.jsp.spy*.jar"
}
```
```

```
You can replace the JAR name above with the module JAR you want to exclude.
This is useful for those who want to have a module in their workspace that
is used for development or debug purposes only, and it should not be deployed
to production. This works for Gradle builds only at this time.
```

```
<!-- TODO: Add way for producing a distributable workspace using Blade, when
available. It can only be done currently with ./gradlew distBundle[Zip|Tar].
-->
```

6. Turn off your Liferay server:

    ```
    blade server stop
    ```

To reference all of Blade CLI's available options, see the Blade CLI article.
Awesome! You learned how to interact with Liferay DXP using Blade CLI.

CHAPTER 848

GENERATING PROJECT SAMPLES WITH BLADE CLI

Liferay provides many useful sample projects for those interested in learning best practices for Liferay DXP projects. You can learn more about these samples by visiting Sample Projects.

Rather than cloning the repository, you can generate these samples using Blade CLI for convenience.

1. List the available sample projects:

   ```
   blade samples
   ```

   Note the sample project you want to generate; you'll use it in the next step.

2. Run the following command to generate a sample project:

   ```
   blade samples <NAME>
   ```

   For example, to generate the portlet-ds sample, execute

   ```
   blade samples ds-portlet
   ```

   The sample is generated in the current folder.

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
```

Awesome! You've successfully generated a Liferay sample project using Blade CLI!

CHAPTER 849

UPDATING BLADE CLI

Blade CLI is updated frequently, so you should 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`.

To update your Blade CLI installation to the latest stable version, run

```
blade update
```

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.

CHAPTER 850

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/7-2/reference/-/knowledge_base/r/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/7-2/reference/-/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 7.2](/docs/7-2/tutorials/-/knowledge_base/t/upgrading-6-2-themes-to-7-2).
```

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 851

EXTENDING BLADE CLI

You can extend Blade in three different ways:

- Creating Custom Commands for Blade CLI
- Creating Custom Project Templates for Blade CLI
- Installing New Extensions for Blade CLI

There are a few use cases to consider when extending Blade CLI. For example, if you only want to add a new command that applies globally to all types of workspaces, you can create and install a new custom command as explained in the links above.

Alternatively, you may want a set of custom commands that only apply to a specific workspace environment. Normally, Liferay developers who use Blade CLI run a series of Blade commands that make sense in the *default* Liferay Workspace. What if the workspace, however, should support a containerized environment (e.g., Docker) or some other specialized environment? The commands used in the development workflow must complete the workflow differently.

To customize Blade CLI's development workflow, you must create a Blade *profile*. Blade profiles *override* existing Blade commands or add *new* commands in a preserved environment that can be applied to any Liferay Workspace. For example, `blade init` for a profile `myprofile` would override the default `init` command to do something before/after the normal `init` command. For more information, see Creating a Blade Profile.

Note: Blade CLI leverages the profile system internally for Maven support. The Maven specific code is stored in an extension JAR and embedded inside the default Blade JAR.

Continue on to learn more!

CHAPTER 852

CREATING CUSTOM COMMANDS FOR BLADE CLI

To create a custom command for Blade CLI, follow these steps:

Note: This article creates a Gradle-based command project. These steps can be completed for a Maven-based project too.

1. Create a generic OSGi module.
2. You'll leverage JCommander and the Blade CLI API to create your custom command. Add these dependencies in your build file. For example, a `build.gradle` file's `dependencies` block looks like this:

```
dependencies {
    compileOnly group: "com.beust", name: "jcommander", version: "1.72"
    compileOnly group: "com.liferay.blade", name: "com.liferay.blade.cli", version: "latest.release"
}
```

3. Build a Command class by extending the `BaseCommand` class:

```
import com.liferay.blade.cli.command.BaseCommand;

public class Hello extends BaseCommand<HelloArgs> {

    @Override
    public void execute() throws Exception {
        HelloArgs helloArgs = getArgs();

        getBladeCLI().out("Hello " + helloArgs.getName());
    }

    @Override
    public Class<HelloArgs> getArgsClass() {
        return HelloArgs.class;
    }

}
```

   This registers your new command with Blade. You must define the `execute()` command for all classes extending `BaseCommand`. The `BaseCommand` class expects an arguments class as its parameter. You'll create this next.

4. Create a class that holds your command's arguments:

```
import com.beust.jcommander.Parameter;
import com.beust.jcommander.Parameters;

import com.liferay.blade.cli.command.BaseArgs;

@Parameters(commandDescription = "Executes a hello command", commandNames = "hello")
public class HelloArgs extends BaseArgs {

    public String getName() {
        return _name;
    }

    @Parameter(description = "The name to say hello to", names = "--name", required = true)
    private String _name;

}
```

   This class extends the `BaseArgs` class. Notice that the class declaration has the `@Parameters` JCommander annotation. This sets your command's description and name. The `@Parameter` annotation applied to the private string `_name` sets how the command's parameter is called and whether it's required.

5. Since Blade looks for custom commands using the `com.liferay.blade.cli.command.BaseCommand` service interface, you must use a standard JRE service loader mechanism to finish registering your new command with Blade CLI.

   Create a file named `com.liferay.blade.cli.command.BaseCommand` in the `src/main/resources/META-INF/services/` folder. This class should list all of your custom commands' fully qualified class names:

```
com.liferay.extensions.sample.command.Hello
```

```
**Note:** Java's Service Loader Interface (SPI) is used to load the
fully qualified classes in the `META-INF/services` folder. You can learn
more about SPIs
[here](https://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html).
```

---

6. Generate the extension's JAR file (e.g., `gradlew build`).

Awesome! You've created a custom command! You can deploy multiple custom commands in a single JAR, so you can continue adding custom command projects to this module, if desired. See the Installing New Extensions article to install the command (JAR) to Blade CLI.

You can examine a working custom command project here.

CHAPTER 853

CREATING CUSTOM PROJECT TEMPLATES FOR BLADE CLI

Blade comes with 32+ project templates, but many times you may feel that those are too simple or don't fit the need for your development team. You can create new custom project templates that fit your team's workflow and have Blade use them instead.

Important: An extension JAR can only contain one template. If you have multiple templates, each must be deployed in a separate JAR.

Implementing a custom project template should mimic that of a Maven archetype. The best way to illustrate this is by visualizing a sample template's folder structure:

- `src/`
    - `main/resources/`
        - `META-INF`
            - `maven`
            - `archetype-metadata.xml`
            - `archetype-post-generate.groovy` (optional; only invoked by Maven projects)
        - `archetype-resources`
            - Folder structure to be generated
- `bnd.bnd`
- `[build.gradle|pom.xml]`

You can read more about Maven archetypes and their features and capabilities here.
To create a custom project template that can be generated using Blade CLI, follow these steps:

1. Create a generic Maven archetype following the folder structure outlined above. Follow Maven's documentation to configure the archetype project appropriately.

2. Open the template's `bnd.bnd` file and ensure it sets the following configurations:

   ```
   Bundle-Description: TEMPLATE_DESCRIPTION
   Bundle-Name: TEMPLATE_NAME
   Bundle-SymbolicName: SYMBOLIC_NAME
   Bundle-Version: TEMPLATE_VERSION
   Liferay-Versions: LIFERAY_VERSION_RANGE
   -removeheaders:\
       Import-Package,\
       Private-Package,\
       Require-Capability
   ```

   For example, a template's `bnd.bnd` could look like this:

   ```
   Bundle-Description: Creates a Sample as a module project.
   Bundle-Name: Liferay Project Templates Sample
   Bundle-SymbolicName: com.liferay.project.templates.sample
   Bundle-Version: 1.0.0
   Liferay-Versions: [7,8)
   -removeheaders:\
       Import-Package,\
       Private-Package,\
       Require-Capability
   ```

   The `Bundle-SymbolicName` of your template JAR must have the pattern `*.project.templates.<name>.*`. The `-removeheaders` definition is a packaging requirement for all project templates. For more information on Bnd versioning, visit Bnd's official docs.

3. Generate the extension's JAR file (e.g., `gradlew build`).

It's that easy! You've created a custom project template. See the Installing New Extensions article to install the project template (JAR) to Blade CLI.

You can examine a working custom project template here.

CHAPTER 854

INSTALLING NEW EXTENSIONS FOR BLADE CLI

After you've created a new extension for Blade CLI, you must install it so it's available for use. You can learn how to create custom commands and custom project templates in their respective articles.

When Blade CLI starts, it looks in the user's `${user.home}/.blade/extensions` folder for JAR files. All JAR files are searched to see if they contain valid Blade extensions. You'll learn how to install new extensions next.

854.1 Installing a New Extension

To install an extension, you must move the extension JAR to the user's `${user.home}/.blade/extensions` folder. You can do this automatically from Blade CLI by running

```
blade extension install /path/to/my_extension.JAR
```

You can verify that the extension is available by running the following commands, depending on extension type:

Custom Command:

```
blade help
```

Custom Project Template:

```
blade create -l
```

Great! You've installed a new extension!

854.2 Uninstalling an Extension

You can uninstall a Blade extension by running this:

```
blade extension uninstall EXTENSION_NAME.jar
```

This removes the extension JAR from the user's `${user.home}/.blade/extensions` folder.

CHAPTER 855

CREATING A BLADE PROFILE

There are two steps to follow when adding a new Blade profile:

- Creating a new profile
- Setting the profile in Liferay Workspace

You'll learn how to create a profile first.

855.1 Creating a New Profile

To create a new Blade profile, follow these steps:

1. Create a generic OSGi module.
2. To create a new command, create the command and arguments classes extending `BaseCommand` and `BaseArgs`, respectively, as described in the Creating Custom Commands article. These classes should reside in the profile module's `src/main/java/PACKAGE_NAME` folder. These classes register your command and arguments to Blade CLI.
3. To override a default command, follow the same steps outlined here:

    - Create a command class
    - Create an arguments class
    - Define your commands' fully qualified class names for the service loader

    Instead of extending the `BaseCommand` and `BaseArgs`, classes, however, extend the command/arguments classes defined for the command you intend to override. Make sure to also set the `@Parameters` annotation's `commandNames` argument to the command to override.

    For example, if you intend to override the default `deploy` command, your arguments class declaration would look like this:

    ```
    @Parameters(commandDescription = "Overridden Deploy Command", commandNames = "deploy")
    public class OverriddenArgs extends DeployArgs {

    }
    ```

The corresponding command class override would look like this:

```
public class OverriddenCommand extends BaseCommand<OverriddenArgs> {

    @Override
    public void execute() throws Exception {
            OverriddenArgs args = getArgs();

        getBladeCLI().out("OverriddenCommand says " + args.isWatch());
    }

    @Override
    public Class<OverriddenArgs> getArgsClass() {
        return OverriddenArgs.class;
    }
}
```

You can search for the default command/arguments classes here.

4. To associate a command to your new profile, set the `BladeProfile` annotation in your command class:

   ```
   @BladeProfile("myprofile")
   public class NewCommand extends BaseCommand<NewArgs> {

   }
   ```

   The annotation's parameter should specify the profile you want to associate the command with (e.g., `myprofile`).

Excellent! You've created a new Blade profile and learned how to add new commands or override default commands by leveraging the profile.

Note: Command classes spanning multiple JARs can use/share the same profile name. For example, if you want to extend the internal (provided) `maven` profile extension with new commands, you can do it externally the same way you'd create a new profile.

You can reference the sample profile project to examine a new command and overridden command's setup in a custom profile.

Next, you'll learn how to set your new profile for use in a Liferay Workspace.

855.2 Setting a Profile

To set your new Blade profile in a Liferay Workspace, open the `${workspaceDir}/.blade.properties` file and set the `profile.name` property to your profile name:

```
profile.name=myprofile
```

This specifies which Blade profile is active and uses its defined commands. The default setting is `gradle`. You can also set this property to `maven` out-of-the-box, which is applied for Maven-based workspaces. You can only set one profile for a workspace.

You can specify the Blade profile for a workspace when initializing it too. This is done by passing the profile name as an argument when creating the workspace:

```
blade init -P [profile-name] [workspace-name]
```

For example, if you execute the following command:

```
blade init -P myprofile my-new-custom-workspace
```

Your `my-new-custom-workspace` has the following properties set in its `${workspaceDir}/.blade.properties` file:

```
liferay.version.default=7.2
profile.name=myprofile
```

Note: The `-P` profile parameter can be used for any command to specify the profile to use for that command. This is helpful if you want to run a command not associated with the workspace's current profile.

Awesome! You've set your new Blade profile!

CHAPTER 856

COMMON ERRORS WITH BLADE CLI

If Blade CLI isn't working as expected, you may find answers here.

The following issues are documented:

- The blade command is not available in my CLI
- I can't update my Blade CLI version

Visit the appropriate section to learn more.

856.1 The blade command is not available in my CLI

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 (e.g., Korn, csh) or you've customized your CLI via `.profile` or some other configuration file, you must add JPM to your path manually.

To add JPM's 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 open a new shell, the `blade` command should be available.

856.2 I can't update my Blade CLI version

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.

CHAPTER 857

LIFERAY DEV STUDIO

Liferay Dev Studio is an extension for the Eclipse platform for developing Liferay DXP plugins. It works with build tools such as Gradle and Maven and configuration tools like BndTools.

Dev Studio makes Liferay development easier by providing an intuitive GUI. It contains

- Liferay-specific wizards for creating/developing Liferay DXP projects.
- Liferay server management and project testing capabilities.
- Editors for Service Builder files, workflow definitions, POM files, and more.
- Snippets for tag libraries
- etc.

Here you'll learn how to

- Install Dev Studio
- Set Proxy Requirements
- Install a Liferay server
- Import a Liferay project
- Use the Gogo Shell
- Search Liferay DXP source
- Debug Liferay DXP source
- Update Dev Studio

You can also find Gradle and Maven reference articles that highlight popular use cases for those respective build tools in Dev Studio.

For help creating and deploying projects with Dev Studio, or creating a Liferay Workspace, visit their respective articles.

Let Dev Studio aid in your conquest for Liferay DXP development. Continue on to learn how!

CHAPTER 858

INSTALLING LIFERAY DEV STUDIO

Liferay Dev Studio is a plugin for Eclipse that brings many Liferay-specific features to the table. You can install it into your existing Eclipse environment, or Liferay provides a bundled version included in its Project SDK. Here you'll learn the different methods available for installing 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 Photon or newer. For instructions on upgrading to Photon, 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., Oxygen).

858.1 Install the Dev Studio Bundle

1. Download and install Java. Liferay runs on Java, so you'll need it to run everything else. Because you'll be developing apps for Liferay DXP in 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 Project SDK with Dev Studio. Go to the Downloads page in Liferay's Help Center. Select *Developer Tools* in the Product drop-down and *Developer Studio* for the file type. Then select the executable that correlates to your operating system.

   The Project SDK includes Dev Studio DXP, Liferay Workspace, and Blade CLI.

3. Run the Project SDK executable and step through the installer to install everything to your machine. For help with setting up proxy settings (if necessary), see the Dev Studio Proxy Settings and Liferay Workspace Proxy Settings articles for more information.

Congratulations! You've installed Liferay Dev Studio! It's now available in the Project SDK folder's `liferay-developer-studio`. To run Dev Studio, execute the `DeveloperStudio` executable. A Liferay Workspace has also been initialized in that same folder.

858.2 Install Dev Studio into Eclipse

If you already have an Eclipse environment that you're using for other things, it's easy to add Dev Studio to your existing Eclipse installation.

1. In Eclipse, select *Help → Install New Software*.
2. In the *Work with* field, copy in the following URL: https://releases.liferay.com/tools/ide/latest/stable/
3. You'll see the Liferay Dev Studio components in the list below. Check them off and click *Next*.
4. Accept the terms of the agreements and click *Next*, and Dev Studio is installed. Like other Eclipse plugins you'll have to restart Eclipse to enable it.

858.3 Install Dev Studio into Eclipse from a ZIP File

To install Liferay Dev Studio into Eclipse from a Zip file, follow these steps:

1. Go to the Dev Studio downloads page. Under *Other Downloads*, select the *Liferay IDE [version] Archived Update-site* option to download it.
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 Dev Studio Zip file.
4. You'll see the Dev Studio components in the list below. Check them off and click *Next*.
5. Accept the terms of the agreements and click *Next*, and Liferay Dev Studio is installed. Like other Eclipse plugins you'll have to restart Eclipse to enable it.

Awesome! You've installed Liferay Dev Studio. Now you can begin Liferay development using a popular and supported IDE.

CHAPTER 859

SETTING PROXY REQUIREMENTS FOR 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*.
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). Do not leave whitespace at the end of your proxy host or port settings.
5. Once you've configured your proxy entry, click *Apply and Close*.

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 article for more details.

Awesome! You've successfully configured Dev Studio's proxy settings!

859.1 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
```

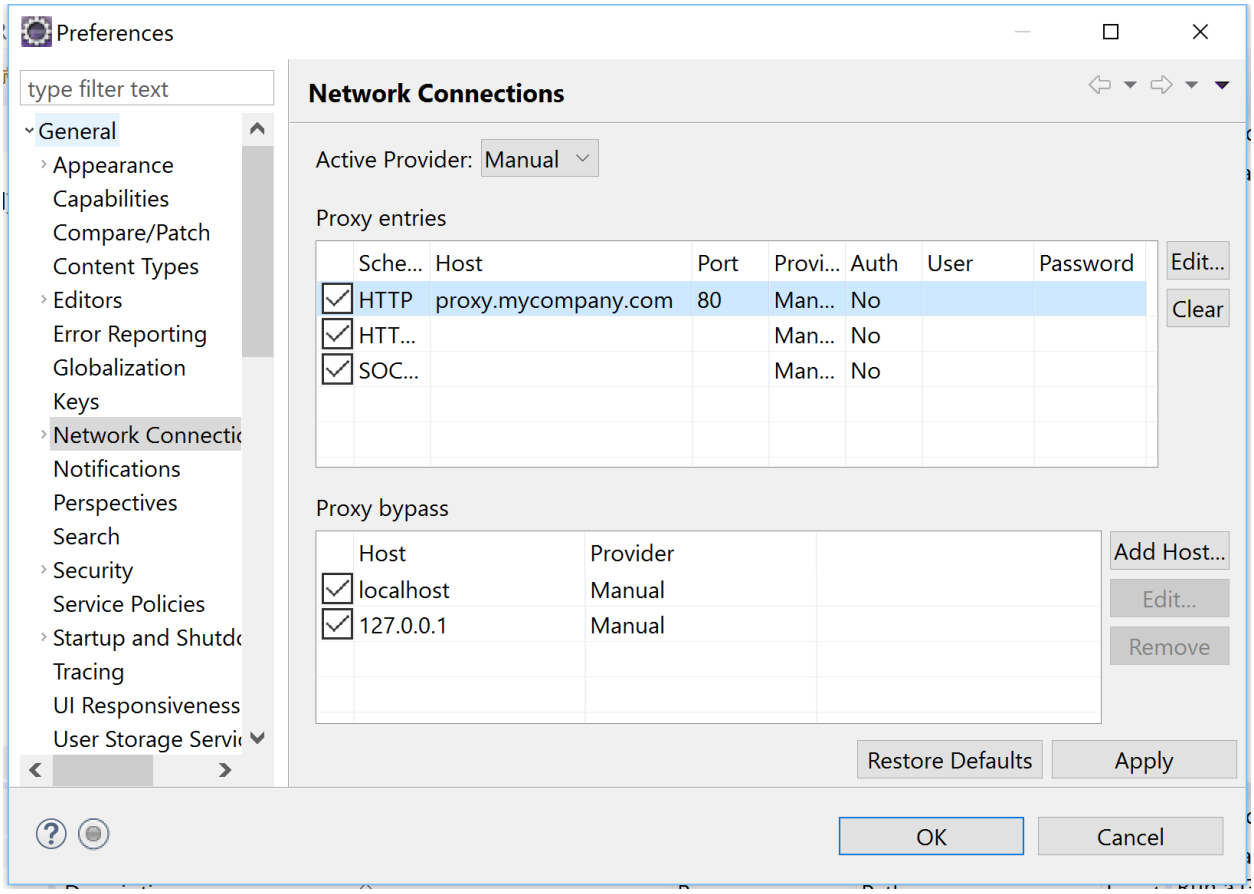

Figure 859.1: You can configure your proxy settings in Dev Studio's Network Connections menu.

```
-Dhttps.proxyUser=userId
-Dhttps.proxyPassword=somePassword
```

After saving the file, restart Eclipse. Now your additional proxy settings are applied!

CHAPTER 860

INSTALLING A LIFERAY SERVER IN DEV STUDIO

Dev Studio offers a single GUI for managing a Liferay server and its deployed projects. A server is installed and managed from the Servers view (lower left corner in Eclipse).

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 Dev Studio article.

Follow these steps to install your server. Note you must have already downloaded and decompressed the server bundle:

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*.

   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-7.2.0-m2\tomcat-9.0.10`.

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

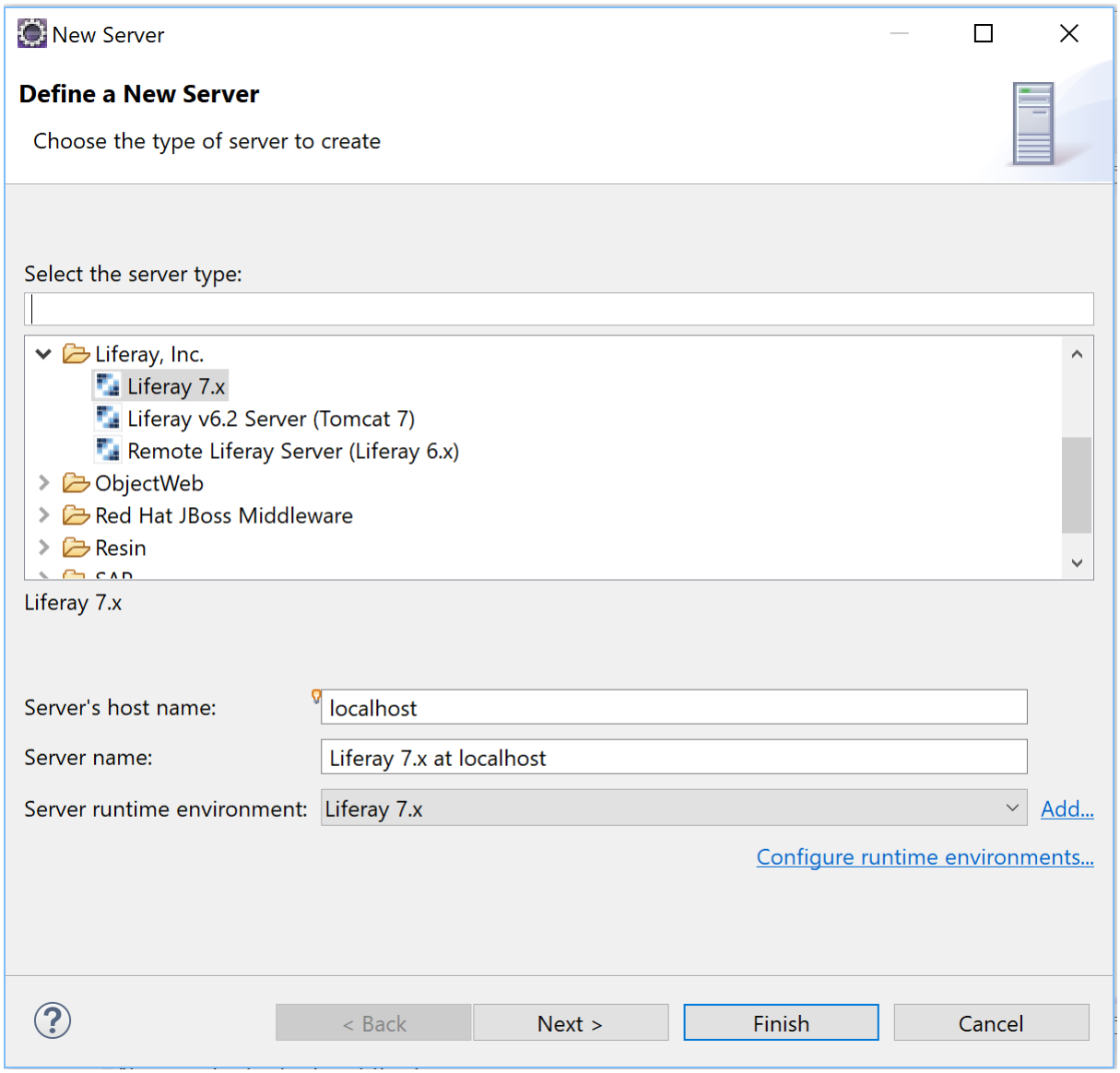

Figure 860.1: Choose the type of server you want to create.

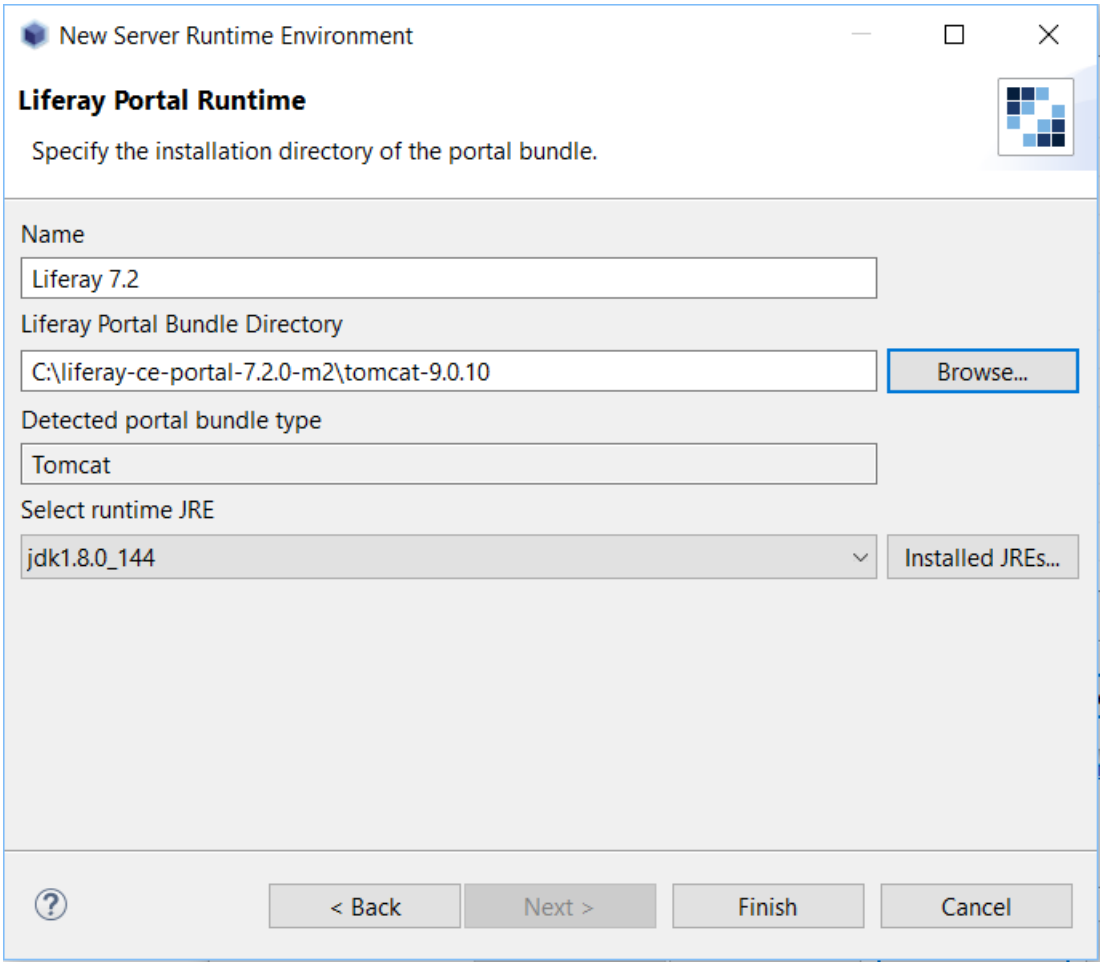

Figure 860.2: Specify the installation folder of the bundle.

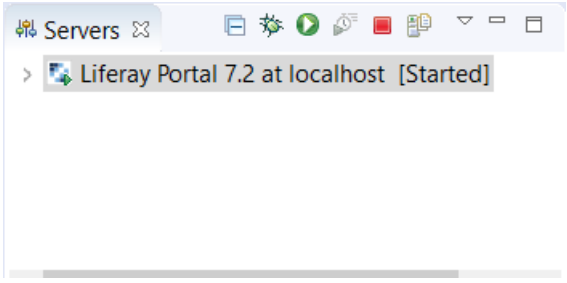

Figure 860.3: Your new server appears under the *Servers* view.

Congratulations! Your server is now available in Liferay Dev Studio!

CHAPTER 861

IMPORTING PROJECTS IN DEV STUDIO

There are two types of Liferay projects you can import into Dev Studio:

- Liferay Module Project (this also includes WAR-style projects)
- Liferay Workspace Project

You cannot import Liferay projects individually that reside in a Liferay Workspace. You can either import a non-workspace Liferay project (or group of projects if the parent folder is specified) or an entire workspace project with all its Liferay projects.

To import a pre-existing Liferay project into Dev Studio, follow the steps outlined below:

1. Right-click in the Project Explorer and select *Import → Liferay Module Project*. If you're interested in importing an entire Liferay Workspace, select the *Liferay Workspace Project* instead.

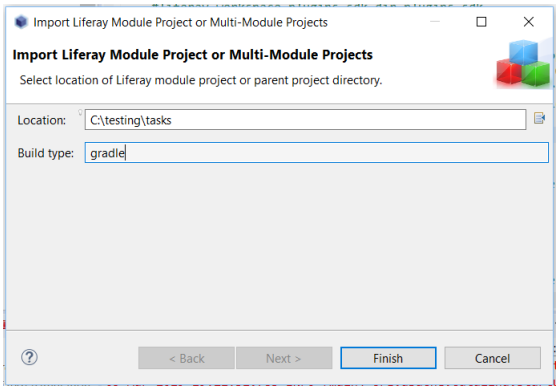

Figure 861.1: You can import a single project or folder of projects.

Once you've selected your project(s), the project build type is displayed.

2. Click *Finish*.

Now your Liferay project is available from the Package Explorer.

CHAPTER 862

USING THE GOGO SHELL IN DEV STUDIO

If you're using Dev Studio to develop and deploy your projects, you may be interested in managing them after they're deployed with Dev Studio too. You can do this with the Dev Studio's Gogo Shell feature.

1. Right-click your started portal instance in the Servers view.
2. Select *Open Gogo Shell*.

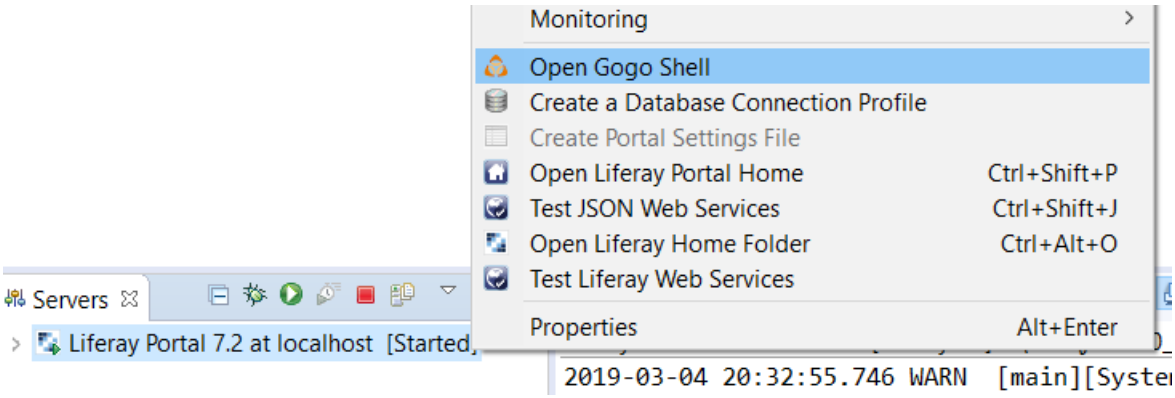

Figure 862.1: Select *Open Gogo Shell* to open a terminal window in Dev Studio using Gogo shell.

   A Gogo shell terminal appears, allowing you to enter Gogo commands to inspect your Liferay instance and the projects deployed to it.

3. A common use case for the Gogo Shell is verifying successful project deployment. Enter the `lb` command to view a list of deployed bundles. If the project status is active, then it deployed successfully.

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

Excellent! You've learned how to manage your deployed projects with Dev Studio's Gogo Shell integration.

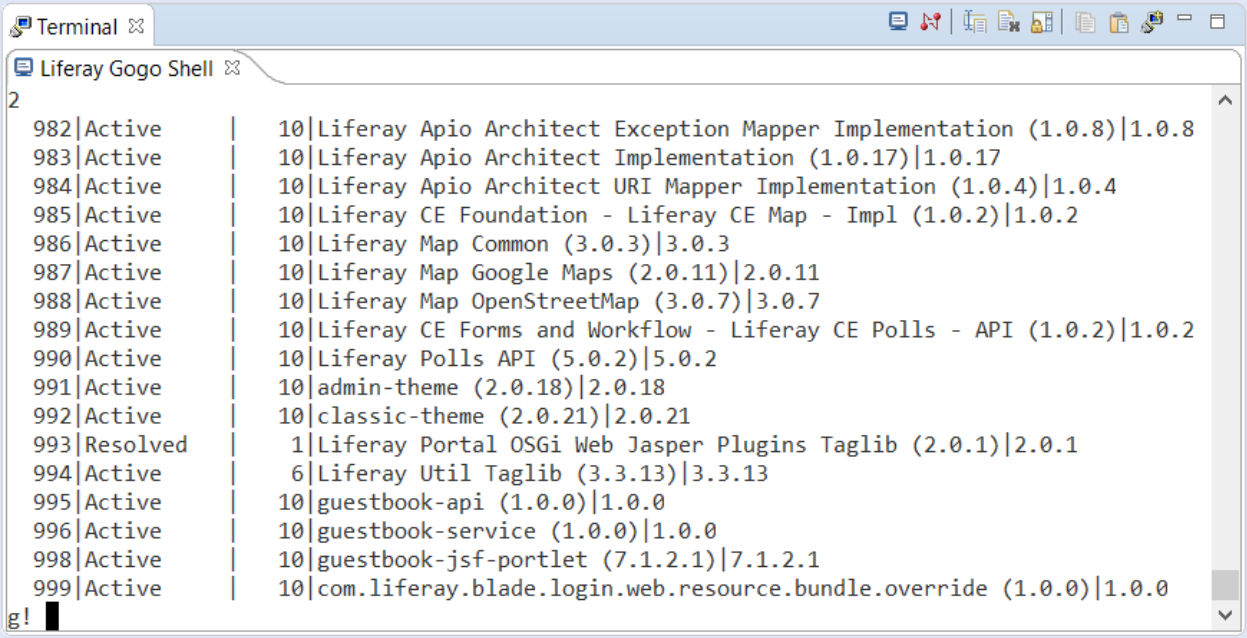

Figure 862.2: You can check to see if your project deployed successfully to Liferay using the Gogo shell.

CHAPTER 863

SEARCHING LIFERAY DXP SOURCE IN 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 an application 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.

863.1 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 can search classes that extend that same class in Dev Studio. Complete the steps below for a simple example:

1. Right-click the `MVCPortlet` declaration.
2. Select *Open Type Hierarchy*.

This opens a window that lets you inspect all classes residing in the target platform that extend `MVCPortlet`.

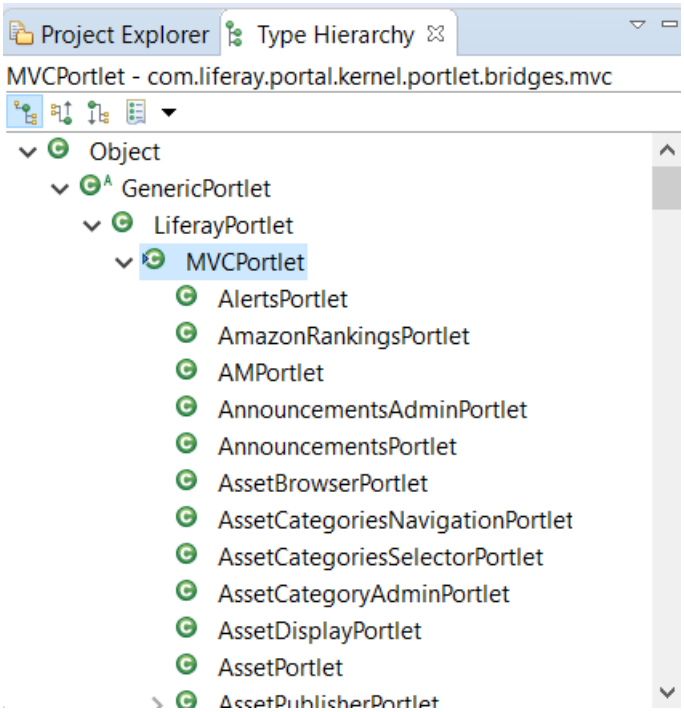

Figure 863.1: Browse the Type Hierarchy window and open the provided classes for examples on how to extend a class.

Great! Now you can search for all extensions and implementations of a class/interface to aid in your quest for developing the perfect app.

863.2 Search Method Declarations

Sometimes you want a search to be more granular, exploring the declarations of a specific method provided by a class/interface. 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 want to search for declarations of its `doView` method you're overriding. Here's how to do it:

1. Right-click the `doView` method declaration in your custom app's class.
2. Select *Declarations → Workspace*.

The rendered Search window displays the other occurrences in the target platform where that method was overridden.

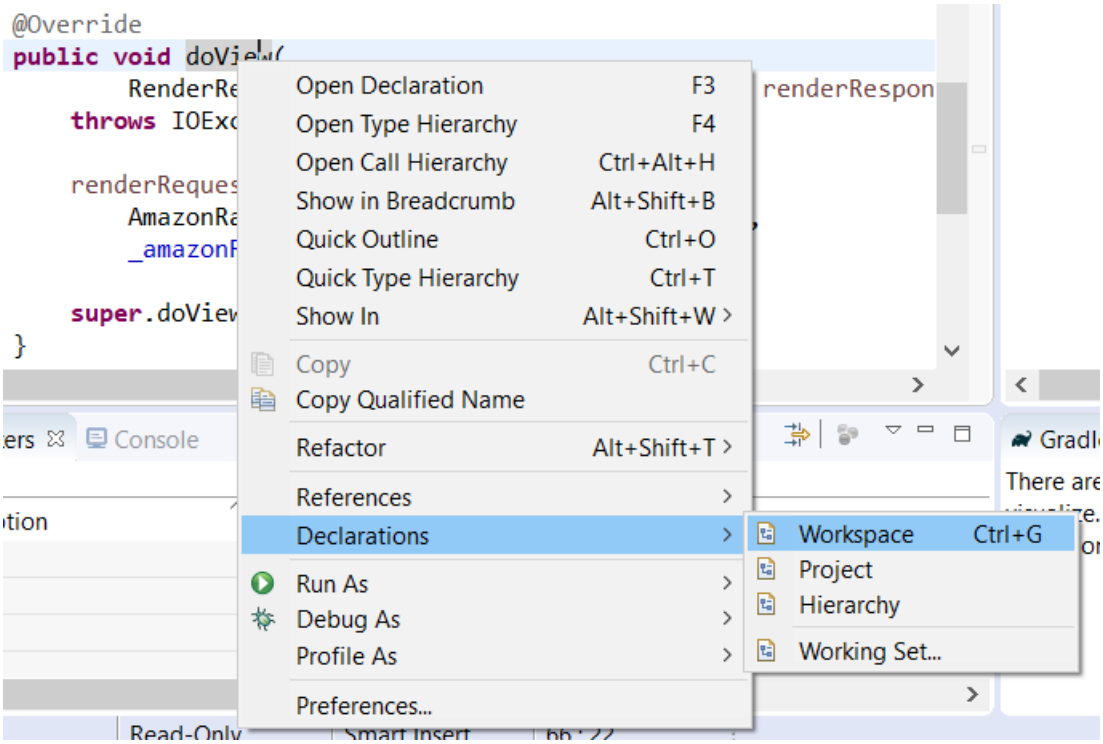

Figure 863.2: All declarations of the method are returned in the Search window.

863.3 Search Annotation References

Annotations used in Liferay DXP's source code can sometimes be cryptic. You can find out how they can be used in your own application by searching for where these types of annotations exist in Liferay's target platform.

For example, you may find some 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 can complete this search like this:

1. Right-click the `@Reference` annotation in a class.
2. Select *References → Workspace*.

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!

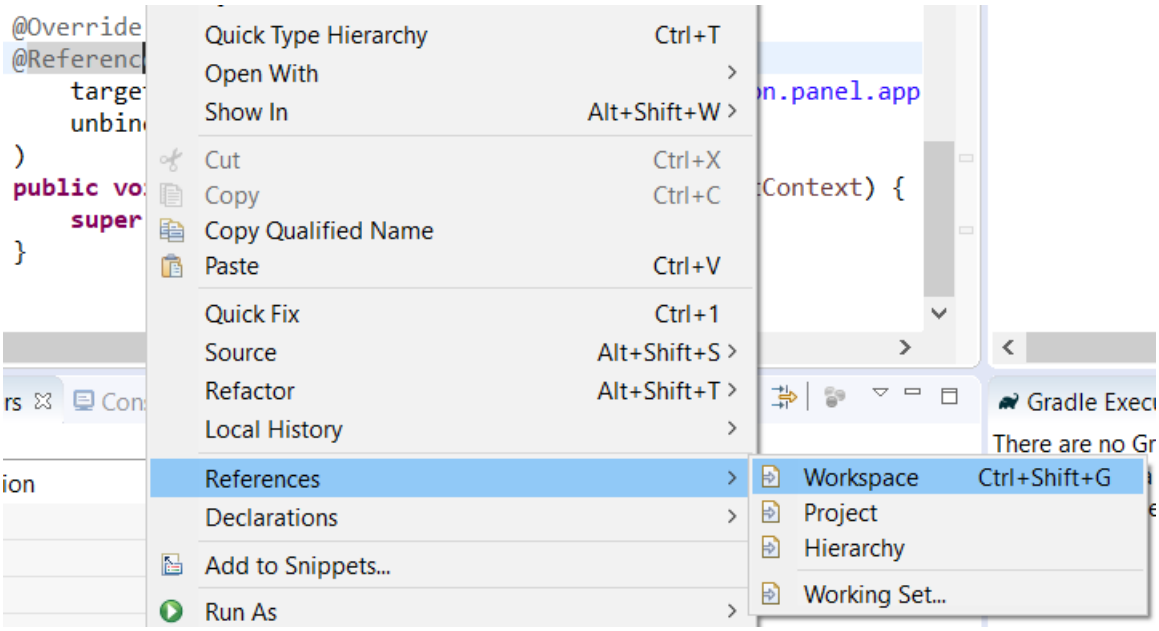

Figure 863.3: All matching annotations are displayed in the Search window.

CHAPTER 864

DEBUGGING LIFERAY DXP SOURCE IN DEV STUDIO

You can debug Liferay DXP source code in Dev Studio to help resolve errors. Debugging Liferay DXP code follows most of the same techniques associated with debugging in Eclipse. If you need 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.

864.1 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. Liferay 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+).

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.

864.2 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 Dev Studio for more details.

2. Start the server in debug mode. To do this, click the debug button in the Servers pane of Dev Studio.

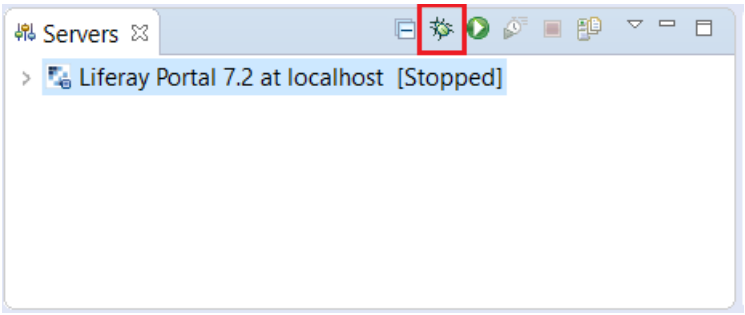

Figure 864.1: The red box in this screenshot highlights the debug button. Click this button to start the server in debug mode.

Awesome! You're now equipped to begin debugging in Liferay Dev Studio!

CHAPTER 865

UPDATING LIFERAY DEV STUDIO

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

1. In Dev Studio, go to *Help → Install New Software...*.
2. In the *Work with* field, copy in the URL http://releases.liferay.com/tools/ide/latest/stable/.
3. You'll see the Dev Studio components in the list below. Check them off and click *Next*.

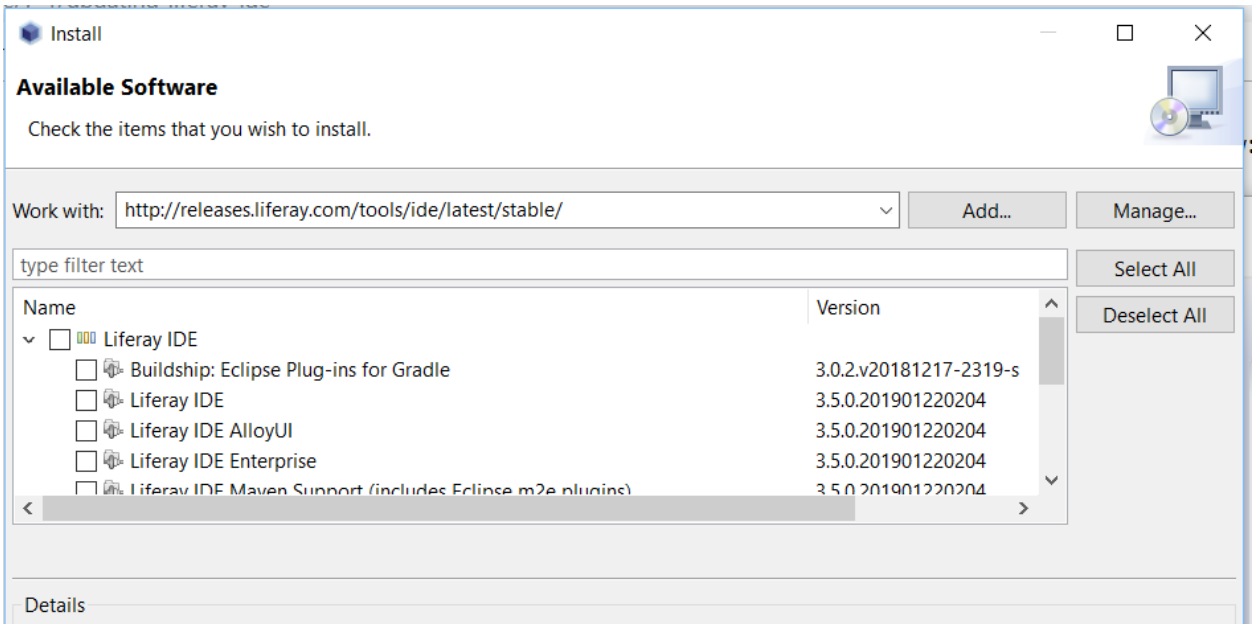

Figure 865.1: Make sure to check all the Dev Studio components you wish to install.

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!

CHAPTER 866

GRADLE IN DEV STUDIO

Gradle is a popular open source build automation system. You can take full advantage of Gradle in Liferay Dev Studio through Buildship, a collection of Eclipse plugins that provide support for building software using Gradle with Liferay Dev Studio. Buildship is bundled with Liferay Dev Studio versions 3.0 and higher.

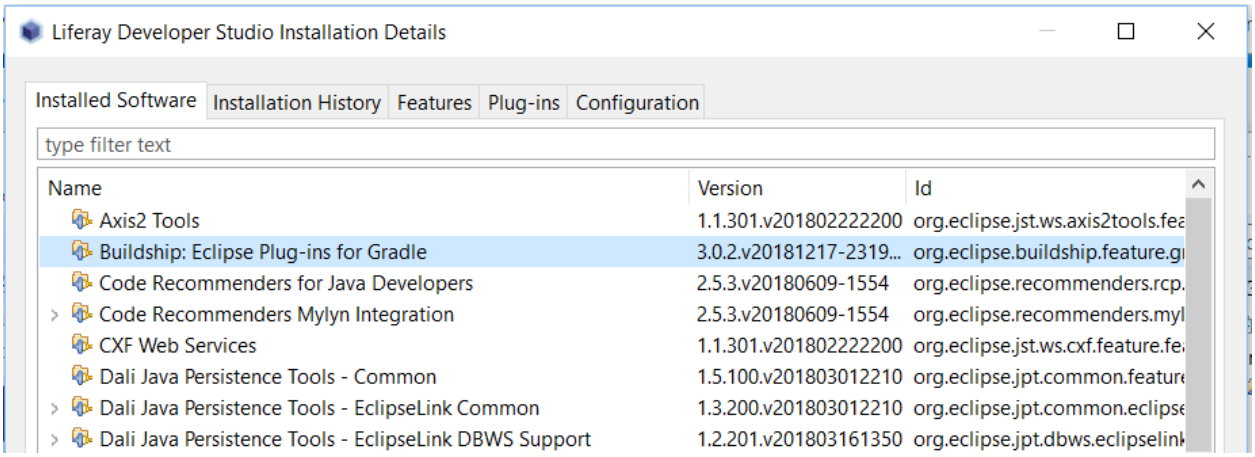

Figure 866.1: Navigate to *Help → Installation Details* to view plugins included in Dev Studio.

This reference article highlights some useful tips for leveraging Gradle in Dev Studio.

- Creating Pure Gradle Projects
- Importing Pure Gradle Projects
- Gradle Tasks and Executions

Note that creating Liferay Gradle projects and deploying Gradle projects with Dev Studio are covered in their respective articles.

The first thing you'll learn about in this tutorial is creating pure Gradle projects in Dev Studio.

866.1 Creating Pure Gradle Projects

Most of Dev Studio's wizards rely on your usage of Liferay Workspace. This is for good reason; it's the recommended developer environment for Liferay projects. You can, however, create pure Gradle projects and manually configure them to be deployable to Liferay DXP.

You can create a pure Gradle project by using the Gradle Project wizard.

1. Navigate to *File → New → Project...*.
2. Select *Gradle → Gradle Project*. Then click *Next → Next*.
3. Enter a valid project name. You can also specify your project location and working sets.
4. Optionally, you can navigate to the next page and specify your Gradle distribution and other advanced options. Once you're finished, select *Finish*.

Excellent! You've created a pure Gradle project using Dev Studio.

866.2 Importing Pure Gradle Projects

You can also import existing pure Gradle projects in Dev Studio.

1. Go to *File → Import...*.
2. Select *Gradle → Existing Gradle Project → Next → Next*.
3. Click the *Browse...* button to choose a Gradle project.
4. Optionally, you can navigate to the next page and specify your Gradle distribution and other advanced options. Once you're finished, click *Next* again to review the import configuration. Select *Finish* once you've confirmed your Gradle project import.

Next you'll learn about Gradle tasks and executions, and learn how to run and display them in Dev Studio.

866.3 Gradle Tasks and Executions

Dev Studio provides two views to enhance your development 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.

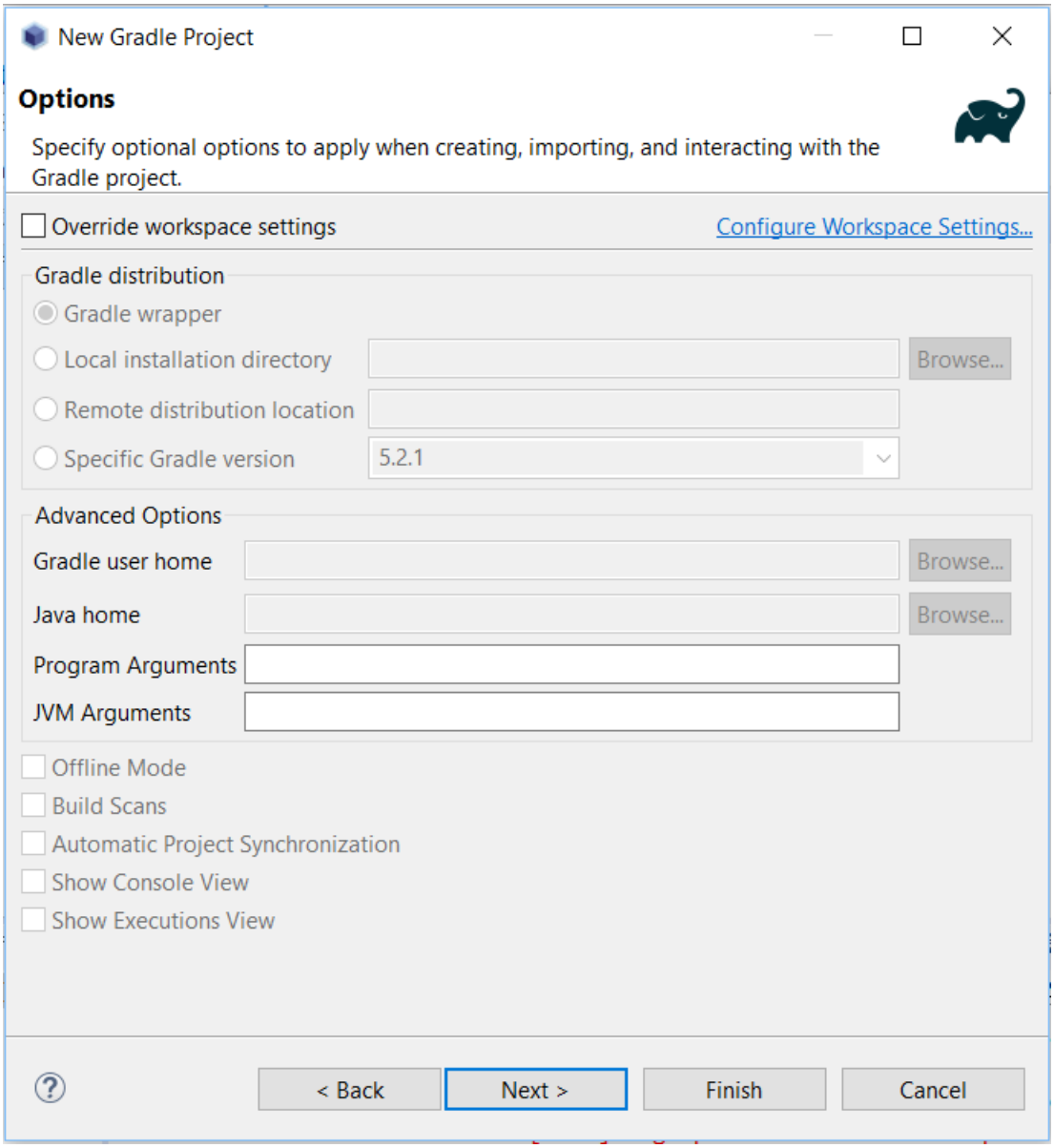

Figure 866.2: You can specify your Gradle distribution and advanced options such as home directories, JVM options, and program arguments.

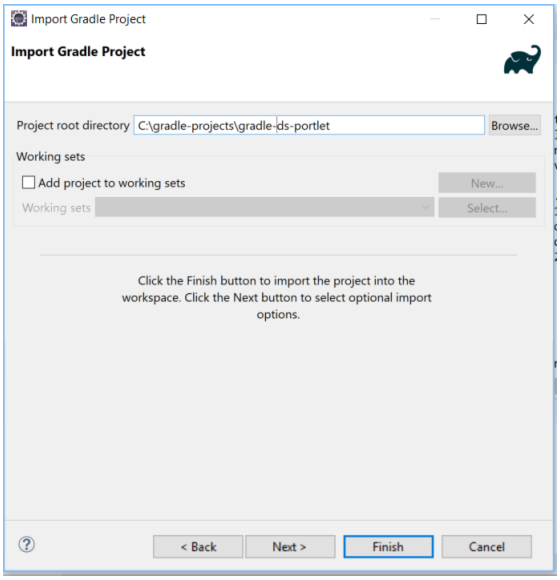

Figure 866.3: You can specify what Gradle project to import from the *Import Gradle Project* wizard.

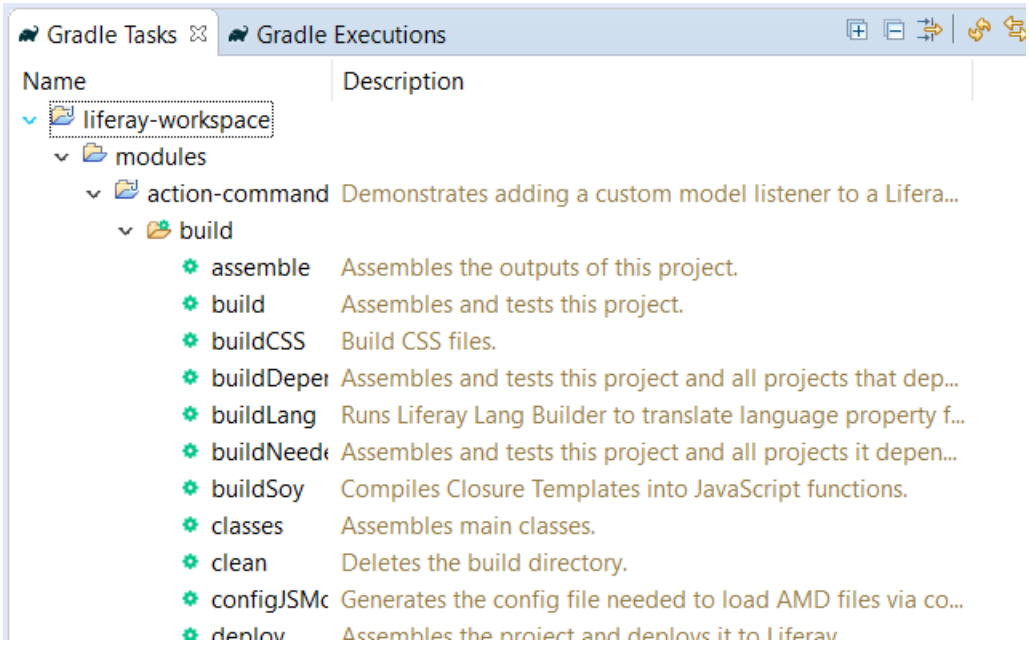

Figure 866.4: Navigate into your preferred Gradle project to view its available Gradle tasks.

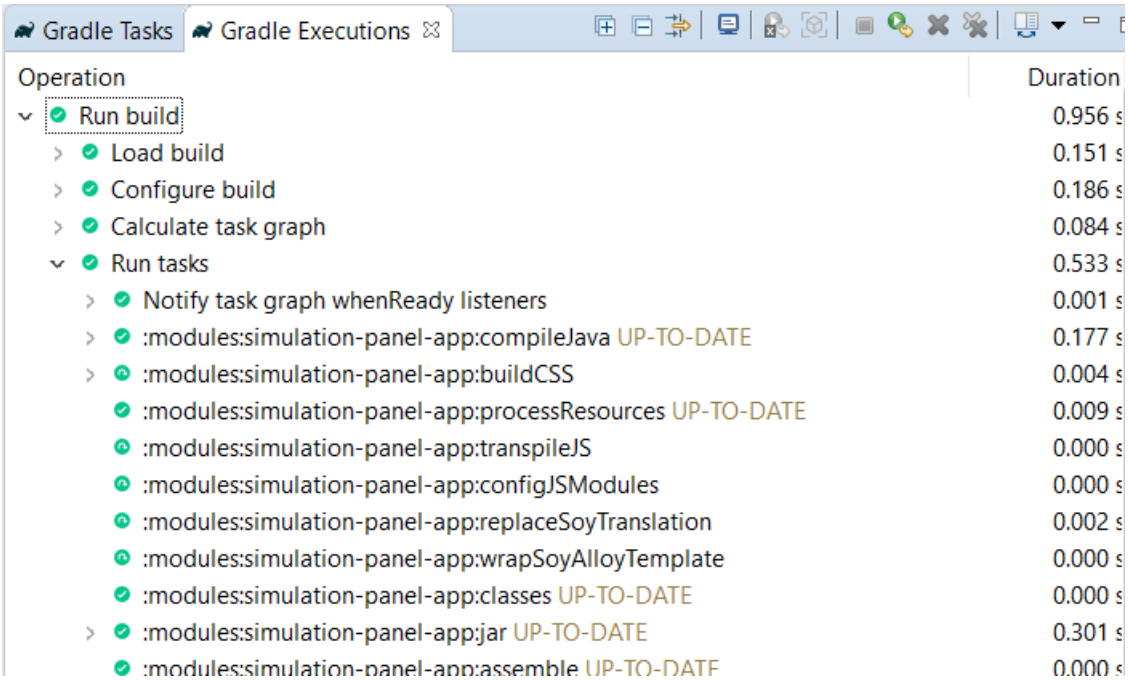

Figure 866.5: The Gradle Executions view helps you visualize the Gradle build process.

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*.

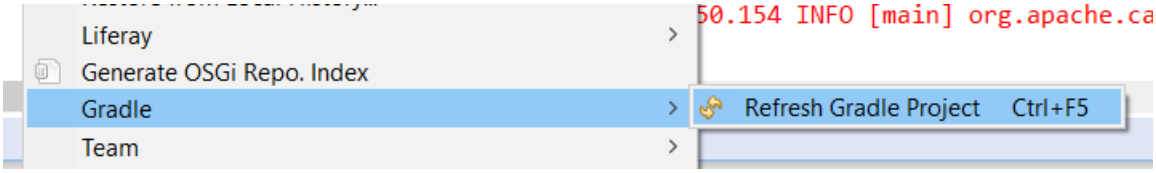

Figure 866.6: Make sure to always refresh your Gradle project in Dev Studio after build script edits.

If you prefer Eclipse refresh your Gradle projects automatically, navigate to *Window → Preferences → Gradle* and enable the *Automatic Project Synchronization* checkbox. If you'd like to enable Gradle's automatic synchronization for just one Gradle project, you can right-click a Gradle project and select *Properties → Gradle* and enable auto sync that way.

Excellent! You're now equipped with the knowledge to add, import, and build your Gradle projects in Liferay Dev Studio!

CHAPTER 867

MAVEN IN DEV STUDIO

You can take full advantage of Maven in Liferay Dev Studio with its built-in Maven support. In this article, you'll learn about the following topics:

- Installing Maven Plugins for Liferay Dev Studio
- Importing Maven Projects
- Using the POM Graphic Editor

Note that creating and deploying Maven projects with Dev Studio are covered in their respective articles.

First you'll install the necessary Maven plugins for Dev Studio.

867.1 Installing Maven Plugins for Dev Studio

To support Maven projects in Dev Studio properly, you first need a mechanism to recognize Maven projects as Dev Studio projects. 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 Dev Studio, the installation startup screen asks if you want 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:

```
http://releases.liferay.com/tools/ide/latest/stable/
```

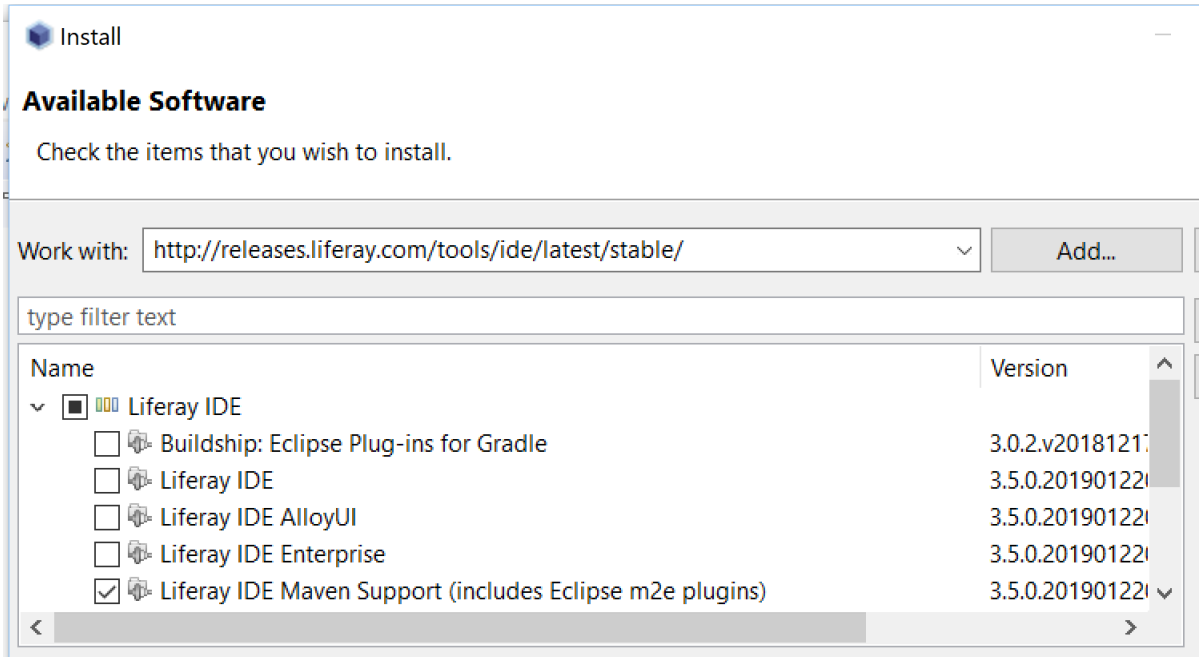

Figure 867.1: You can install all the necessary Maven plugins for Dev Studio by installing the *Liferay IDE Maven Support* option.

2. Check the *Liferay IDE Maven Support* option. This bundles all the required Maven plugins you need to begin developing Maven projects for Liferay DXP.

   If the *Liferay IDE Maven Support* option does not appear, then it's already installed. To verify that it's installed, uncheck the *Hide items that are already installed* checkbox and look for *Liferay IDE Maven Support* in the list of installed plugins. Also, if you want 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*.

Note: Both Maven and Eclipse have their own standard build project lifecycles that are independent from each other. For both to work together and run seamlessly within Dev Studio, a lifecycle mapping is required to link both lifecycles into one combined lifecycle. Normally, this would have to be done manually by the user. Fortunately, the m2e-liferay plugin combines the lifecycle metadata mapping and Eclipse build lifecycles, to provide a seamless user experience. The lifecycle mappings for your project can be viewed by right-clicking your project and selecting *Properties → Maven → Lifecycle Mapping*.

Awesome! Your Dev Studio is ready to develop Maven projects for Liferay DXP!
You'll learn about importing Maven projects in Dev Studio next.

867.2 Importing Maven Projects

To import a pre-existing, non-Liferay Maven project into Dev Studio, follow the steps outlined below:

1. Navigate to *File → Import → Maven → Existing Maven Projects* and click *Next*.

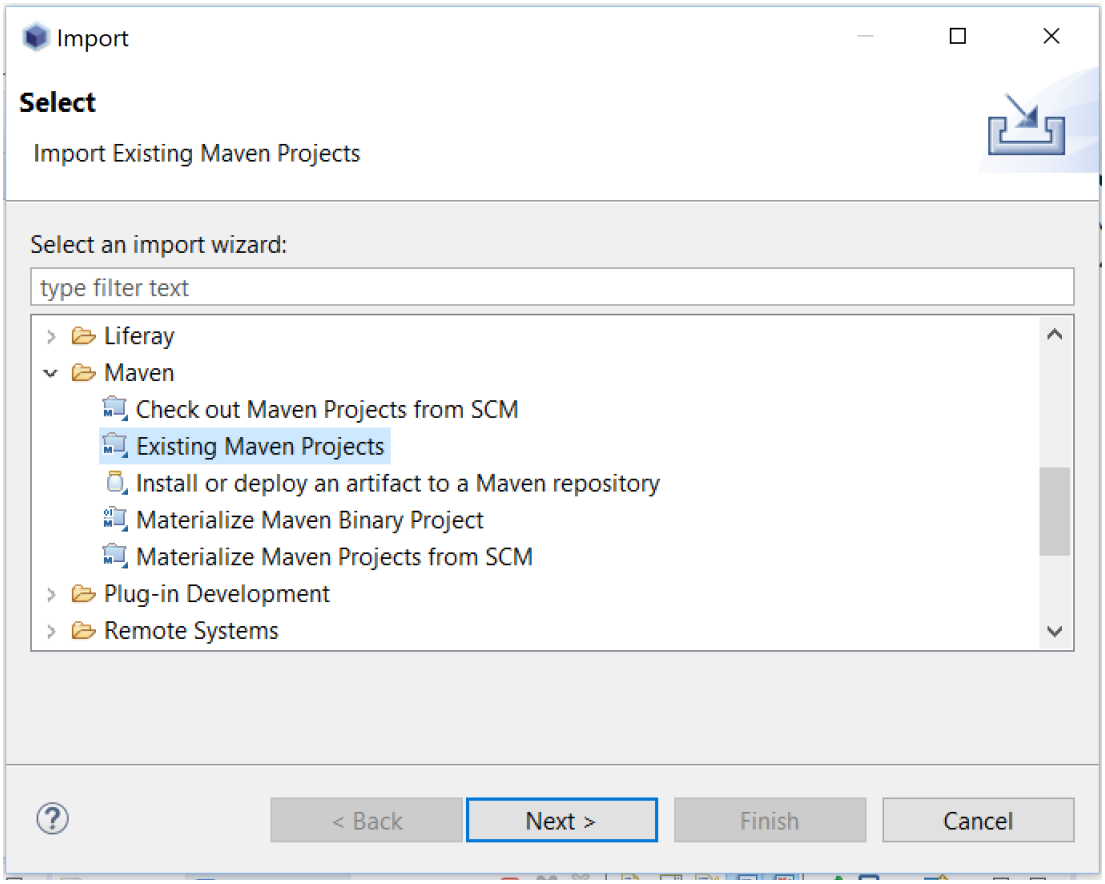

Figure 867.2: Dev Studio offers the Maven folder in the Import wizard.

2. Click *Browse...* and select the root folder for your Maven project. Once you've selected it, the `pom.xml` for that project should be visible in the Projects menu.

3. Click *Finish*.

Now your Maven project is available from the Package Explorer. To import a Liferay project built with Maven, see the Importing Projects in Dev Studio Next you'll learn about Dev Studio's POM graphical editor.

867.3 Using the POM Graphic Editor

You're provided a graphical POM 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:

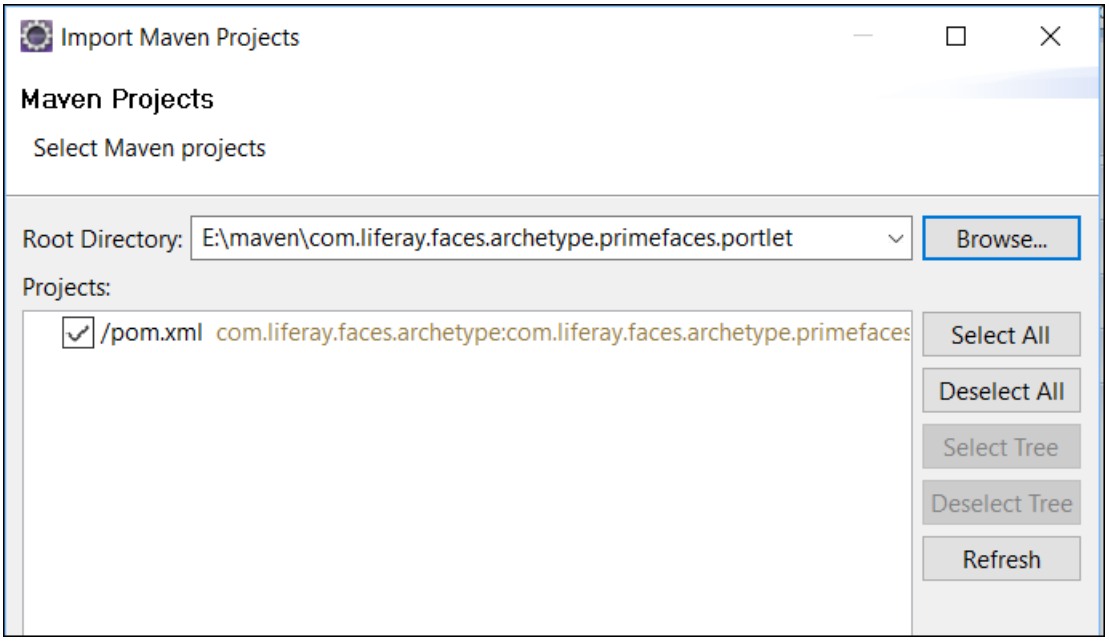

Figure 867.3: Use the Import Maven Projects wizard to import your pre-existing project.

- **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.

By taking advantage of these interactive modes, Dev Studio makes modifying and organizing your POM and its dependencies a snap!

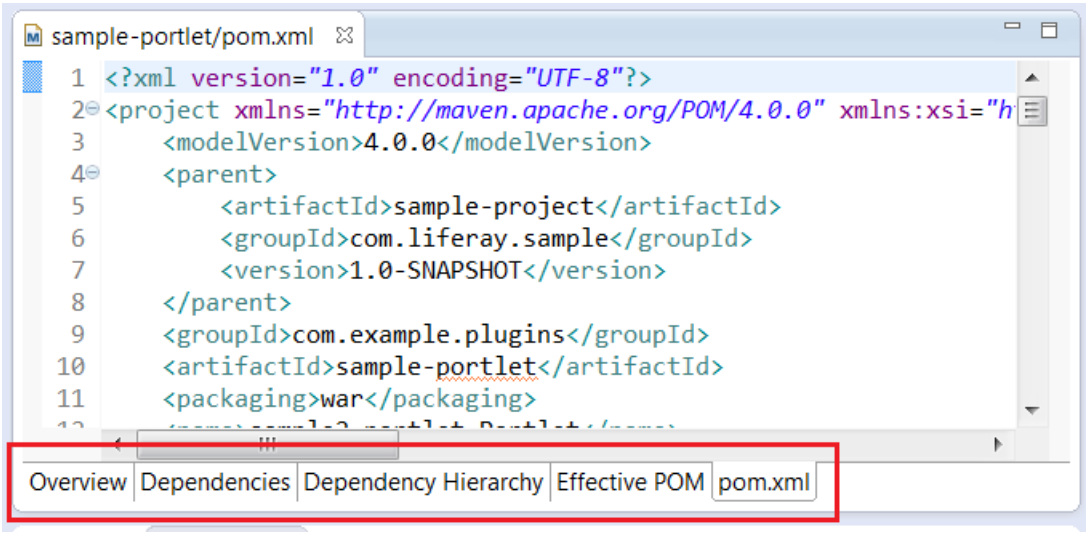

Figure 867.4: Liferay Dev Studio provides five interactive modes to help you edit and organize your POM..

CHAPTER 868

IntelliJ

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/Wildfly 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 articles, you'll learn how to install the Liferay IntelliJ plugin and leverage its features to improve Liferay development with IntelliJ IDEA.

CHAPTER 869

INSTALLING THE LIFERAY INTELLIJ PLUGIN

There are two ways to install the Liferay IntelliJ plugin:

- via IntelliJ Marketplace
- via Zip file

Follow the steps pertaining to your preferred installation process.

869.1 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.

869.2 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. Select the Liferay IntelliJ plugin and click *OK*.
5. 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!

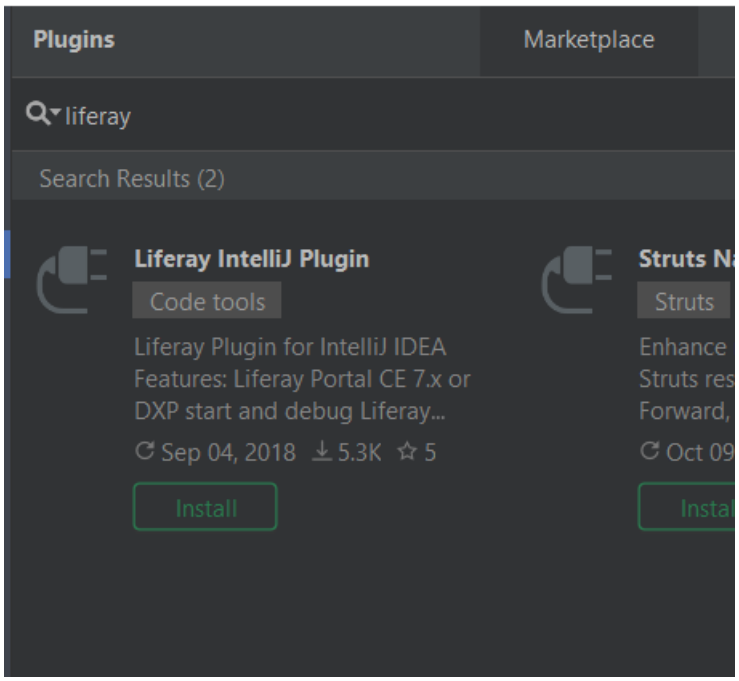

Figure 869.1: IntelliJ Marketplace offers a streamlined way to install plugins.

CHAPTER 870

INSTALLING A SERVER IN INTELLIJ

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.

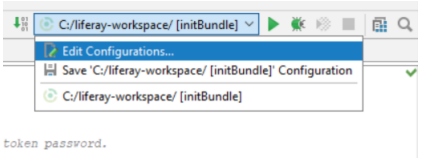

Figure 870.1: You have several options to choose from the server dropdown menu.

3. Click the *Add New Configuration* button (+) and select *Liferay Server*.

4. Give your server a better name and modify any other configurations, if necessary. Then select *OK* .

Your server is now available in IntelliJ! Make sure to select it in the Configurations dropdown before executing the configuration buttons (below).

For reference, here's how the IntelliJ configuration buttons work with your Liferay DXP instance:

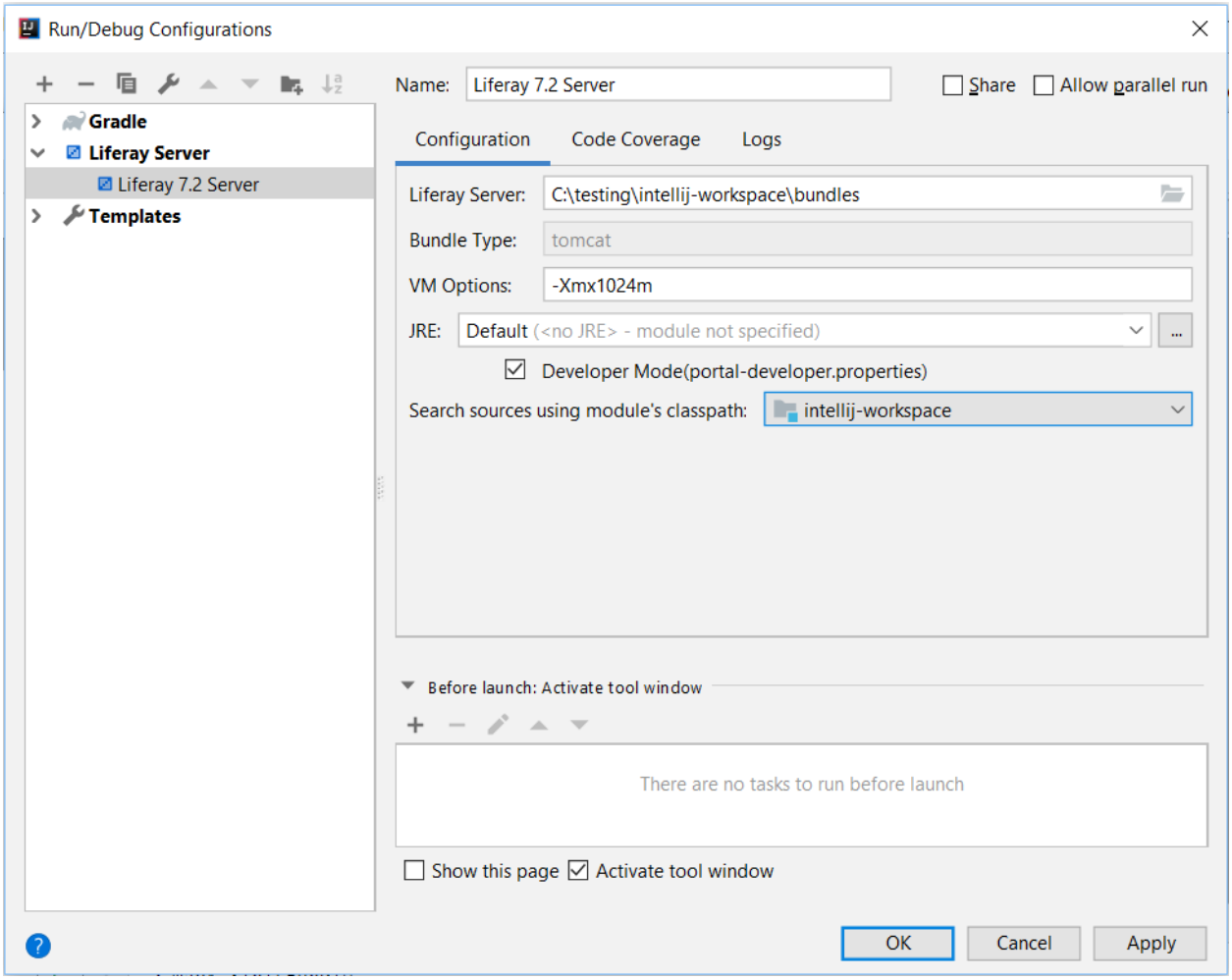

Figure 870.2: Set your Liferay server's configurations in the Run/Debug Configurations menu.

- *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!

CHAPTER 871

UPDATING LIFERAY INTELLIJ PLUGIN

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

1. In IntelliJ, navigate to *File → Settings → Plugins*.
2. Click the *Updates* tab. If the Liferay IntelliJ plugin is outdated, it will be listed as an available update.

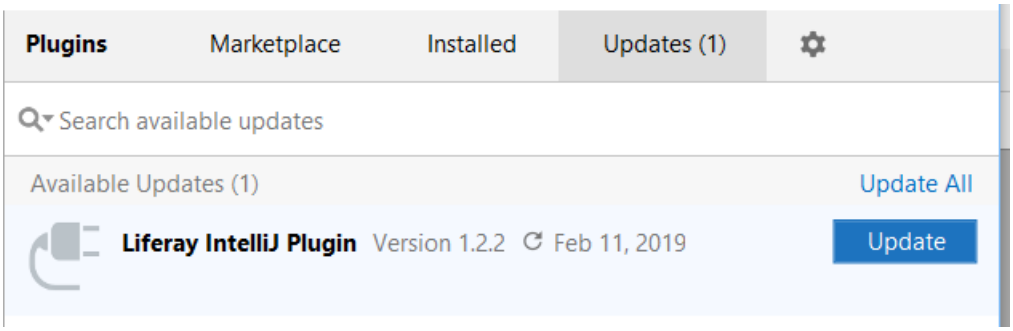

Figure 871.1: Check for updates periodically to ensure you're leveraging the latest features.

3. Click the *Update* button (if available) to update the Liferay IntelliJ plugin.
4. Once it's downloaded, click the *Restart IDE* button.

Once IntelliJ restarts, the Liferay IntelliJ plugin is updated and ready for use.

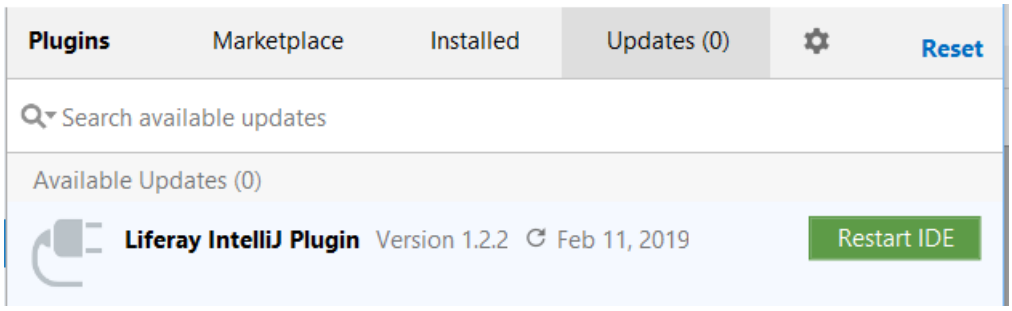

Figure 871.2: The Available Updates prompt also prints the plugin version to which you're updating.

CHAPTER 872

LIFERAY JS GENERATOR

The Liferay JS Generator generates JavaScript widgets using pure JavaScript tooling. You don't have to have a deep understanding of Java to write a widget for Liferay DXP. See the Liferay JS Generator developer documentation for more information on configuring generated JavaScript widgets. This section covers these reference topics for the Liferay JS Generator:

- How to install the Liferay JS Generator and use it to create a JS widget
- An explanation of JS Portlet Extender's method signature
- A reference list of the available configuration options for system settings and instance settings

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.

The available commands for bundles created with the Liferay JS Generator are listed below:

Command	Description
`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 application to the configured 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 the need to deploy. Note, however, that because it's outside a Liferay instance, you don't have access to Liferay's APIs.

Command	Description
`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.

Note: By default, the webpack server uses port 8080, which conflicts with the port used by Tomcat. You can point the webpack server to a different port by setting the `port` key in `.npmbuildrc`:

```
"webpack": {
  "port": 2070
}
```

Read this section to learn how to install the Liferay JS Generator and understand its configuration.

CHAPTER 873

INSTALLING THE JS GENERATOR AND GENERATING A BUNDLE

This document has been replaced by an article on Liferay Learn and is no longer maintained here. The Liferay CLI tool is used for creating JavaScript application projects for Liferay versions 7.1+.

Here you'll learn how to install the Liferay JS Generator and use it to create JavaScript widgets. See the Angular Application, React Application, and Vue Application articles to learn how to use your existing Angular, React, and Vue apps in Liferay DXP.

Note: To use the Liferay JS Generator, you must have the Liferay JS Portlet Extender activated in your Liferay DXP instance. It's activated by default. You can confirm this by opening the Control Menu, navigating to the *App Manager*, and searching for `com.liferay.frontend.js.portlet.extender`.

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
```

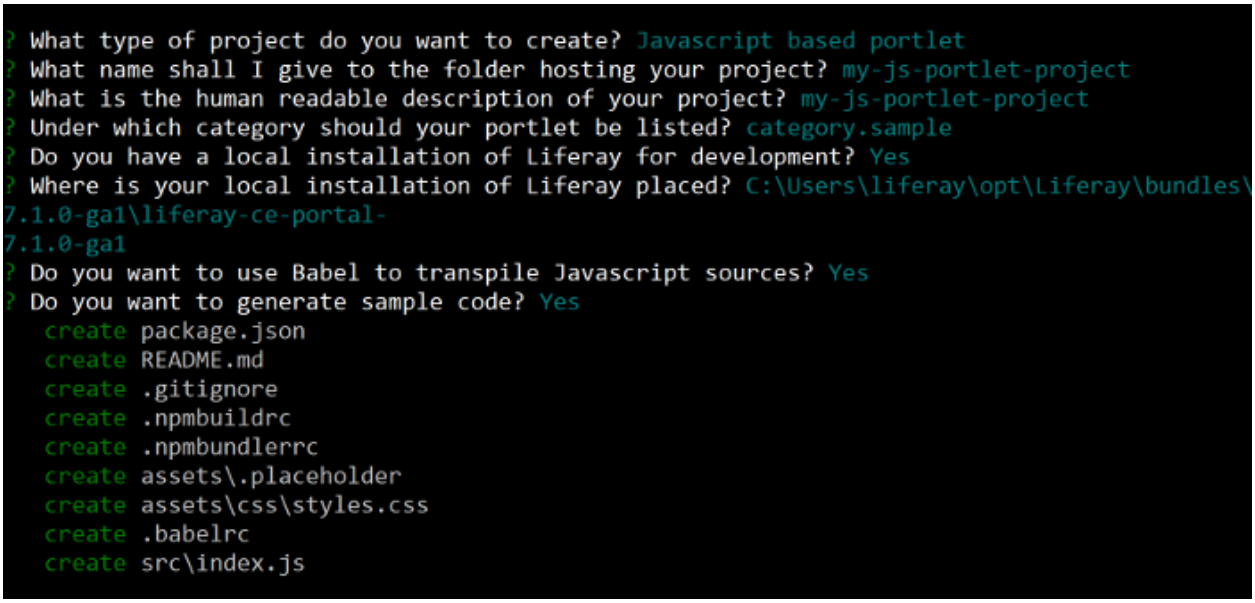

Figure 873.1: The liferay-js generator prompts you for widget options.

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.

CHAPTER 874

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.

874.1 Manifest Header

The OSGi bundle contains the MANIFEST header shown below, which specifies a dependency on the JS Portlet Extender:

```
Require-Capability: osgi.extender;filter:="(osgi.extender=liferay.npm.portlet)"
```

874.2 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 widget*:

```
"main": "index.js"
```

CHAPTER 875

CONFIGURATION JSON AVAILABLE OPTIONS

If you've created an OSGi bundle with the Liferay JS Generator and want to provide system settings or instance 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.

875.1 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 876

ADAPTING EXISTING APPS TO RUN ON LIFERAY DXP

There are two ways to get your existing front-end applications running on Liferay DXP:

1. Migrate your project to a Liferay JS Toolkit project.
2. Since v2.15.0 of the Liferay JS Toolkit, create projects normally, as you would with create-react-app, Angular CLI (any project containing `@angular/cli` as a dependency or devDependency), and Vue CLI (any project containing `@vue/cli-service` as a dependency or devDependency), and adapt them to run on Liferay DXP.

Only adapt your project if you intend it to be platform-agnostic. If you want to integrate with Liferay DXP fully and have access to all the features and benefits that it provides, migrate your project to a true Liferay JS Toolkit project instead.

The reason for this is some of Liferay DXP's features may not be available because the native frameworks expect certain things. For example, Angular assumes that it controls a whole Single Page Application as opposed to the small portion of the page that it controls in a portlet-based platform such as Liferay DXP. Since webpack bundles all JavaScript in a single file to consume per app, if there are five widgets on a page, you have five copies of the framework in the JavaScript interpreter. To prevent this, migrate your project to a true Liferay JS Toolkit project instead.

To adapt your project, it must have the structure shown below:

- **Angular CLI projects** must use `app-root` as the application's Dom selector.
- **creact-react-app projects** must use `ReactDom.render()` call in your entry point with a `document.getElementById()` parameter.
- **Vue CLI projects** must use `#app` as the application's DOM selector.

When your project meets the requirements, you can follow these steps to use the Liferay JS Generator to adapt it:

1. Open the command line and navigate to your project's folder.
2. Run the Liferay JS Generator's `adapt` subtarget:

```
yo liferay-js:adapt
```

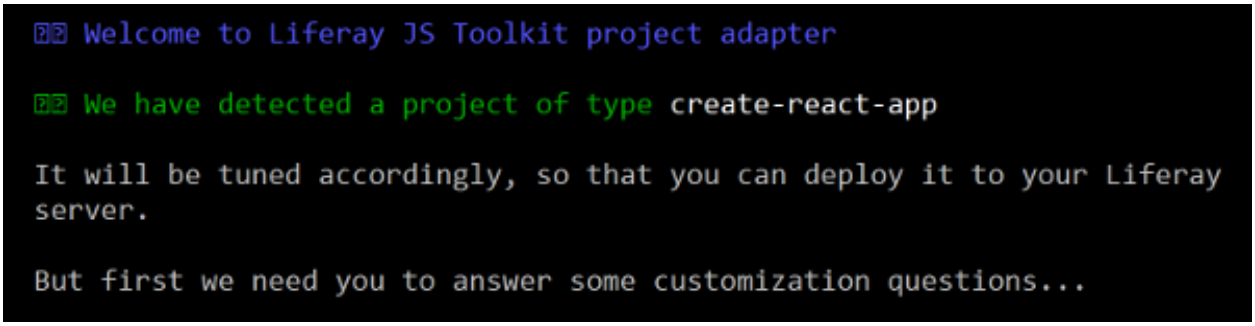

Figure 876.1: You can run the adapt subtarget of the Liferay JS Generator to adapt your existing apps for Liferay.

3. Answer the prompts. An example configuration appears below:

```
? Under which category should your widget be listed? category.sample
? Do you have a local installation of Liferay for development? Yes
? Where is your local installation of Liferay placed? /home/user/liferay
```

   Your project is adapted to use the Liferay JS Toolkit and run on Liferay DXP!

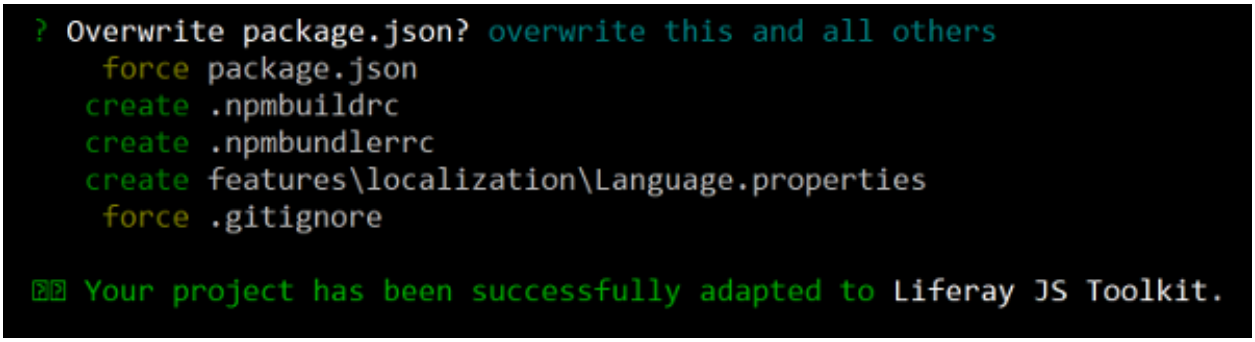

Figure 876.2: You can run the adapt subtarget of the Liferay JS Generator to adapt your existing apps for Liferay.

4. The adapt process automatically adds a few npm scripts to the project's `package.json` so you can build and deploy your project to Liferay DXP. Note that you can swap `npm` for `yarn` below if you prefer to use yarn instead.

   Run the command below to add a deployable JAR to the `build.liferay` folder in your project. For example, if you want to build the JAR and copy it to another app server, you can run this command:

```
npm run build:liferay
```

   Run the command below to deploy the adapted app to your Liferay DXP instance:

```
npm run deploy:liferay
```

Great! Now you know how to use the Liferay JS Generator to adapt your existing apps to run on Liferay DXP. See the React Guestbook App for a working example of an adapted app.

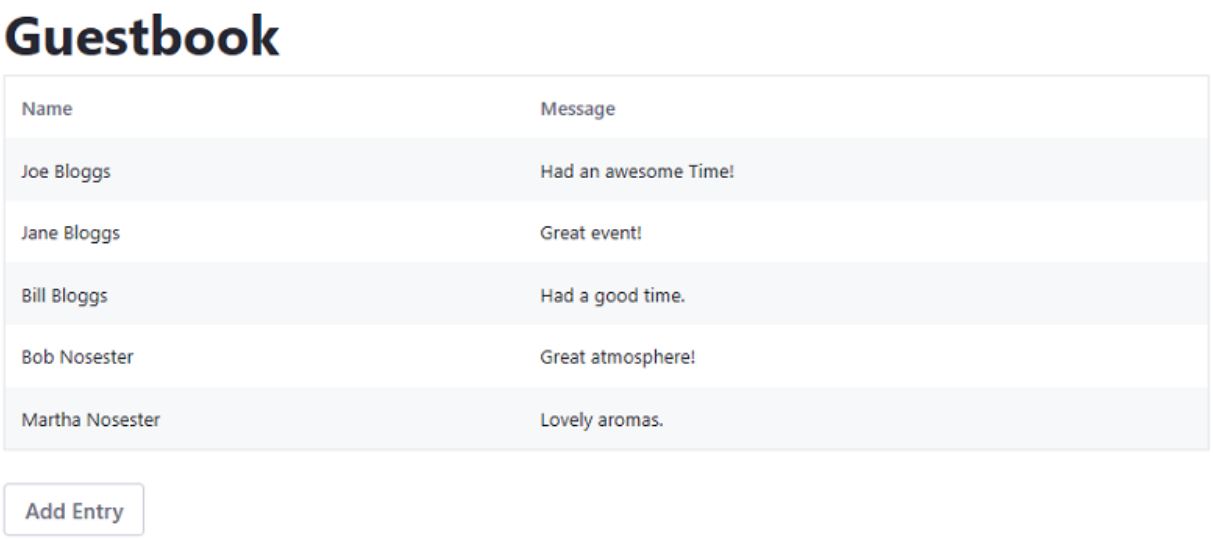

Figure 876.3: Your adapted app runs in Liferay in no time.

CHAPTER 877

LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 build scripts and configured properties. 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.

Liferay Workspace is the official way to create/manage 7.0 projects using Gradle. Do you prefer Maven over Gradle? You can also generate a Maven-based workspace.

You'll cover the following topics in this section:

- Installing Liferay Workspace
- Creating a Liferay Workspace
- Importing a Liferay Workspace
- Setting Proxy Requirements
- Adding a Bundle
- Setting Environment Configurations
- Building Node.js Themes
- Building Gradle/Maven Themes
- Managing the Target Platform
- Validating Modules Against the Target Platform
- Leveraging Docker
- Updating Liferay Workspace
- Updating Default Plugins Provided by Liferay Workspace

Liferay Workspaces can be used in many different development environments, which makes it flexible and applicable to many different developers. For example, a Liferay Workspace easily integrates with Eclipse and IntelliJ, providing a seamless development experience. See how to install and create a Liferay Workspace for more information.

You'll learn about workspace's anatomy and development lifecycle next.

877.1 Workspace Anatomy

A Liferay Workspace offers a development environment that can be configured to fit your development needs. Properties are available to help manage default and optional folders. This provides you the power to customize your workspace's folder structure any way you'd like. 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 Projects 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 front-end portlets created with the Liferay JS Toolkit.
- `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 workspace generated for Maven projects, your folder hierarchy is the same, except the Gradle build files are swapped out for a `pom.xml` file.

Visit your workspace's `gradle.properties` file for a list of properties (with descriptions) you can define to adapt your workspace. For a Maven-based workspace, see the Bundle Support Plugin article for info on adapting your Maven workspace.

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.

Next, you'll learn about workspace's development lifecycle.

877.2 Development Lifecycle

Liferay Workspaces offer a full development lifecycle for your projects to make your Liferay development easier than ever. The development lifecycle includes

- Creating projects
- Building projects
- Deploying projects

- Testing projects
- Releasing projects
- Test

You'll learn about each lifecycle option next.

877.3 Creating Projects

Workspace provides a slew of project templates that you can use to create many different types of Liferay projects. Workspace also provides development support for front-end portlets generated with the Liferay JS Toolkit. They're stored in the `modules` folder by default.

You can also configure where to generate certain projects (modules, themes, WARs, etc.). These settings are documented in the `gradle.properties` file. See the Creating a Project article for more information.

Liferay Workspace manages theme projects in two separate folders based on how they're created:

- Liferay Theme Generator (Node.js-based themes that use the Liferay JS Theme Toolkit)
- Project template/archetype (Gradle/Maven-based themes)

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.

877.4 Building Projects

Liferay Workspace abstracts many build requirements away so you can focus on developing projects instead of worrying about how to build them. This is done by incorporating a slew of plugins under the hood to allow for easily accessible tooling. See the Gradle Plugins and Maven Plugins sections for information on some of the plugins provided by workspace.

Gradle-based workspaces also include 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. For Maven-based workspaces, Maven build commands are supported (e.g., `package`, `verify`, `deploy`, etc.).

Liferay Workspace lets you build your projects out-of-the-box without the hassle of manual build configurations.

877.5 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, visit the Deploying a Project article.

877.6 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 six subfolders:

`common`: holds a common configuration that you want applied to all environments.

`dev`: holds the development configuration.

`docker`: holds the configuration for a Docker container.

`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`, `test`, 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.

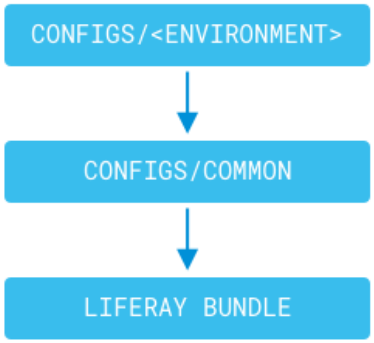

Figure 877.1: The `configs/common` and `configs/[environment]` overlay you Liferay DXP bundle when it's generated.

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`, etc.) is applied on top of any existing configurations from the `common` folder.

See the Setting Environment Configurations for Liferay Workspace article for more information.

877.7 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 articles.

CHAPTER 878

INSTALLING LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 covered in the Installing Blade CLI article.

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 folder where your Liferay Workspace should be initialized.

   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 Liferay Workspace](/docs/7-2/reference/-/knowledge_base/r/adding-a-liferay-bundle-to-liferay-
workspace)
for more information on this topic.
```

5. Click *Next* to begin installing Liferay Workspace on your machine.

That's it! Liferay Workspace is now installed on your machine!

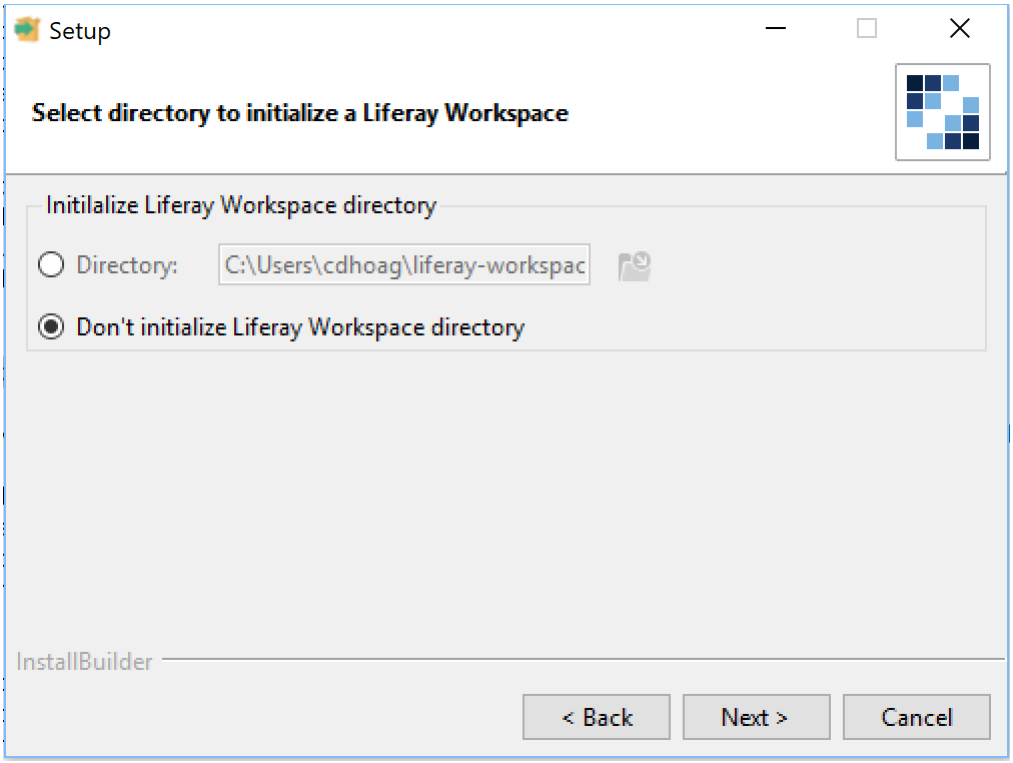

Figure 878.1: Determine where your Liferay Workspace should reside.

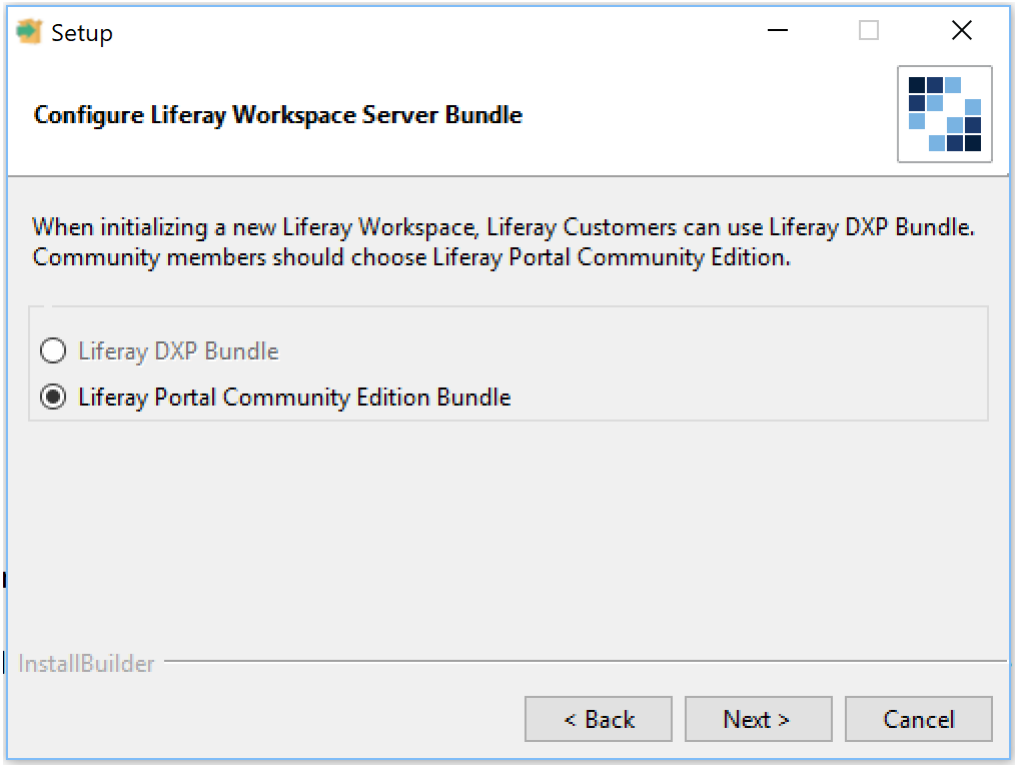

Figure 878.2: Select the product version you'll use with your Liferay Workspace.

CHAPTER 879

CREATING A LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

You can create a Liferay Workspace using the following tools:

- Blade CLI
- Dev Studio
- IntelliJ
- Maven

Visit the appropriate section to learn how to create a workspace with the highlighted tool.

879.1 Blade CLI

1. Navigate to the folder where you want your workspace generated.
2. Run the following command to build a Gradle-based workspace:

```
blade init -v 7.2 [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,

```bash
blade init -v 7.0 [WORKSPACE_NAME]
```
```

You can also create a Maven-based workspace with Blade CLI. See the Maven section for more information.

1. Select *File → New → Liferay Workspace Project*.

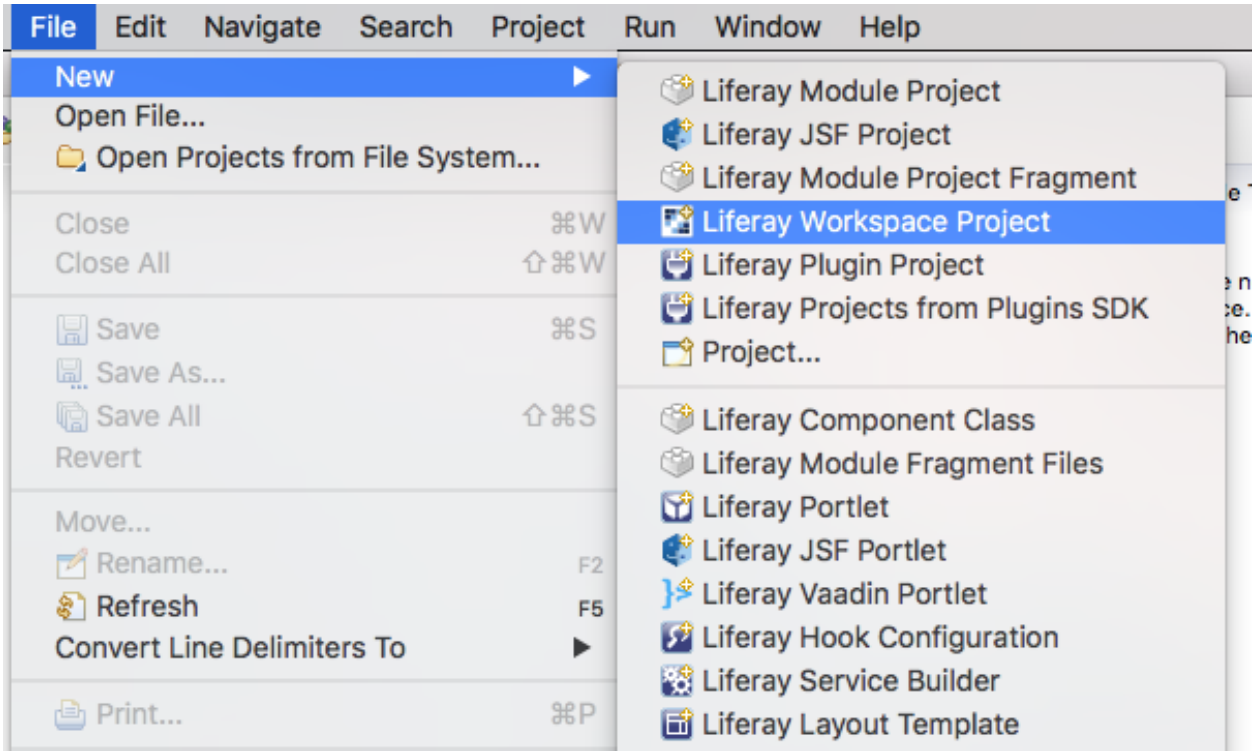

Figure 879.1: By selecting *Liferay Workspace Project*, you begin the process of creating a new workspace for your Liferay projects.

A New Liferay Workspace dialog appears, presenting several configuration options.

2. Give your workspace project a name.
3. 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.
4. Select the build tool you want your workspace to be built with (i.e., Gradle or Maven).
5. Choose the Liferay Portal version you plan to develop for (i.e., 7.2, 7.1, or 7.0).
6. Select the specific target platform version corresponding to the GA release you're developing for (e.g., 7.2.0 → 7.2 GA1). For more information on target platform benefits, see the Managing the Target Platform articles.
7. 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.

```
**Note:** You can configure a pre-existing Liferay bundle in your
workspace by creating a folder for the bundle in your workspace and
configuring it in the workspace's `gradle.properties` file by setting the
`liferay.workspace.home.dir` property.
```

---

8. Check the *Add project to working set* checkbox if you want your 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.

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

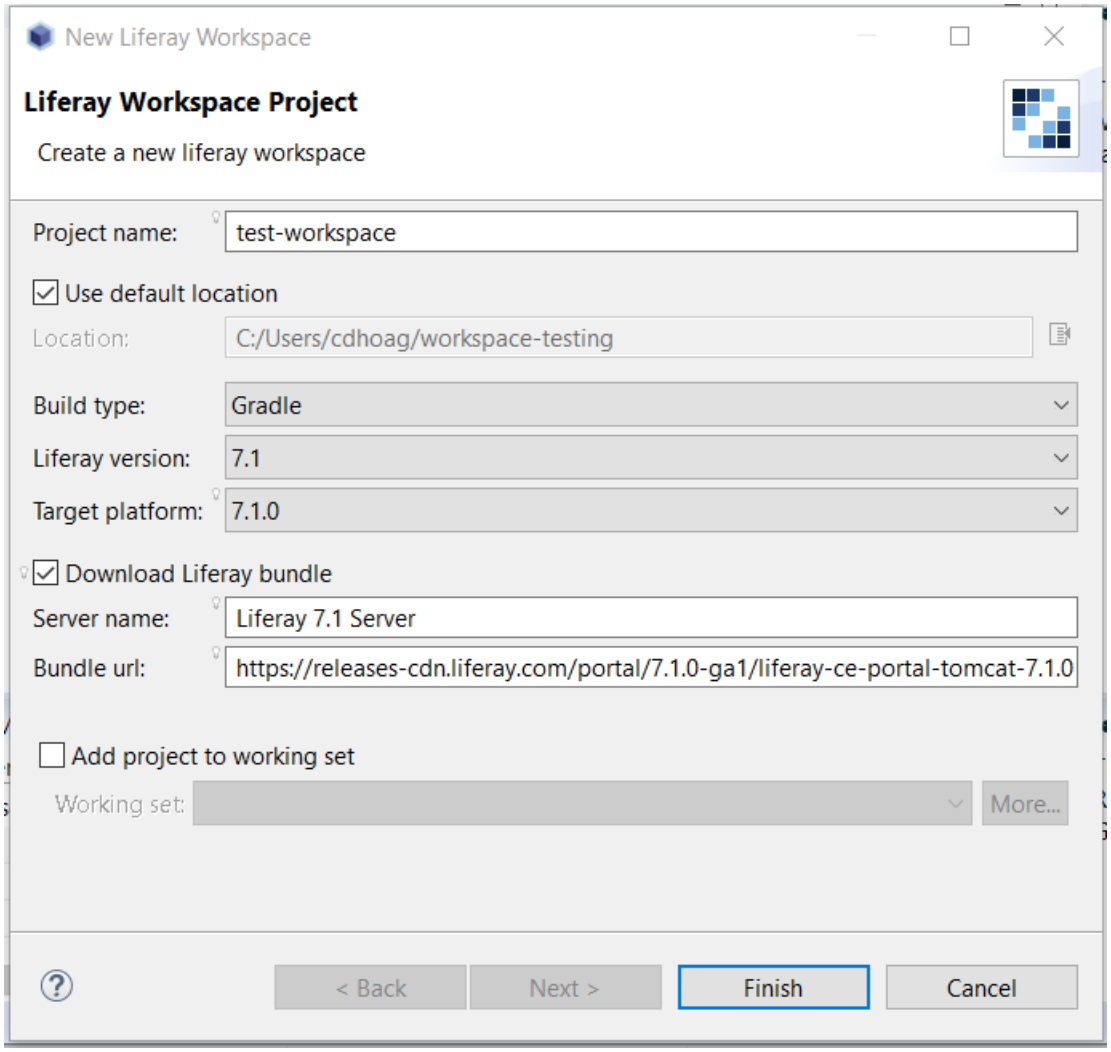

Figure 879.2: Dev Studio provides an easy-to-follow menu to create your Liferay Workspace.

A dialog appears prompting you to open the Liferay Workspace perspective. Click *Yes*, and your perspective will switch to Liferay Workspace.

879.3 IntelliJ

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*.

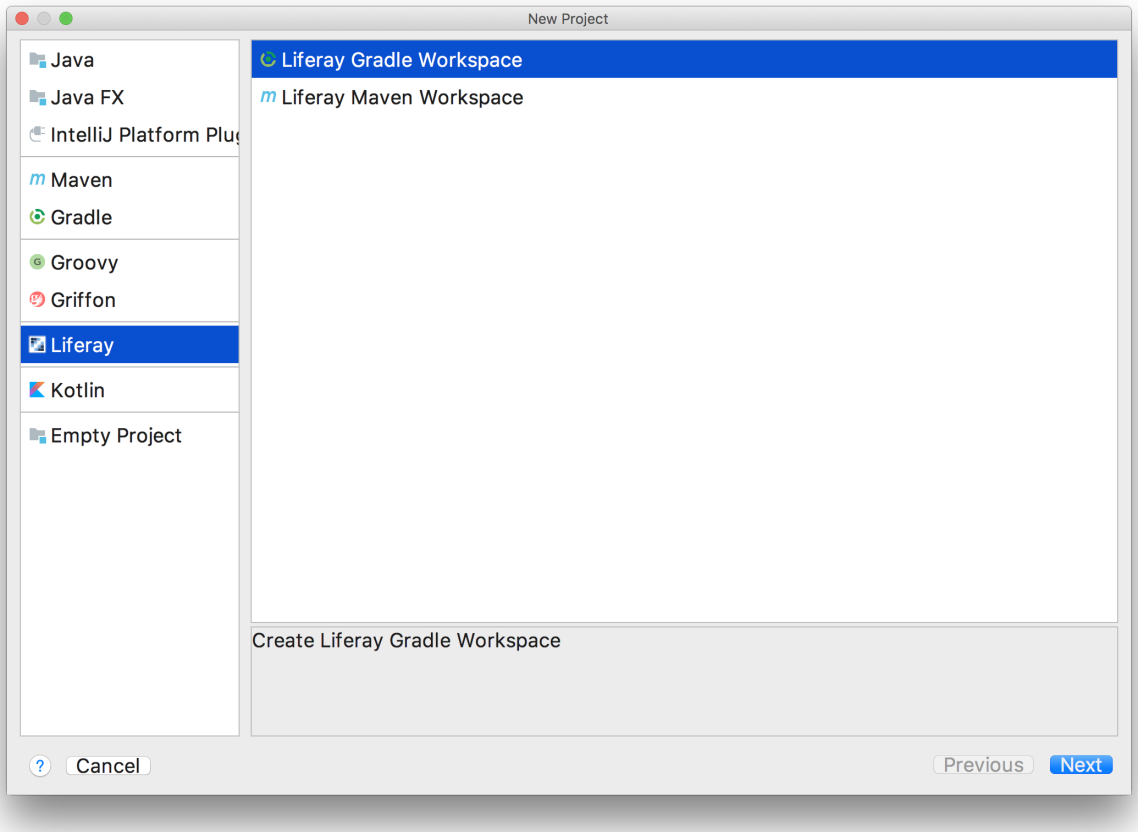

Figure 879.3: Choose *Liferay Gradle Workspace* or *Liferay Maven Workspace*, depending on the build you prefer.

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*.

879.4 Maven

1. Execute the following Maven command:

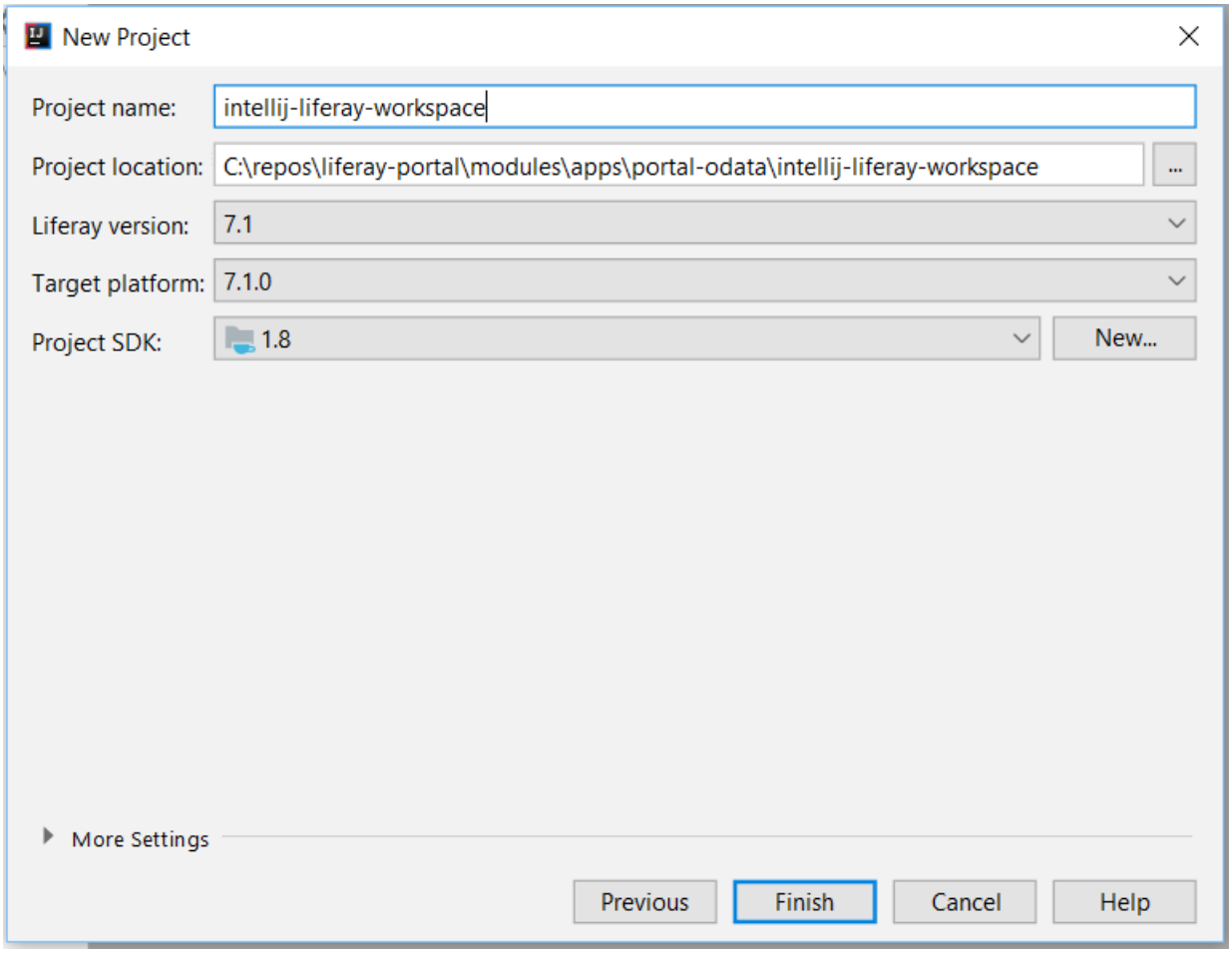

Figure 879.4: Specify your workspace's configurations.

```
mvn archetype:generate -Dfilter=liferay
```

2. Select the `com.liferay.project.templates.workspace` archetype to generate.
3. Step through the remaining prompts to generate the workspace project.

A Maven-based Liferay Workspace can also be generated using Blade CLI. Follow Blade CLI's workspace creation instructions and insert the `-b maven` parameter in the Blade command.

CHAPTER 880

IMPORTING A LIFERAY WORKSPACE INTO AN IDE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Liferay supports two IDEs with preconfigured Liferay Workspace wizards and functionalities

- Dev Studio
- IntelliJ

These aren't the only IDEs you can leverage to use Liferay Workspace, but they are the ones with out-of-the-box support for it.

Visit the appropriate section to learn how to import a workspace with the highlighted tool.

880.1 Dev Studio

1. Navigate to *File → Import → Liferay → Liferay Workspace Project*.
2. Click *Next* and browse for your workspace project.
3. Once you've selected you workspace, click *Finish*.

880.2 IntelliJ

1. Select *File → New → Project from Existing Sources...*.
2. Select the workspace you want to import. Then click *OK*.
3. Click the *Import project from external model* radio button and select the build tool your workspace uses (e.g., Gradle or Maven).
4. Configure the project import (if necessary) and then click *Finish*. See the Import a Project section of IntelliJ's official documentation for more information.
5. Step through the remaining import prompts and then open your imported workspace as you desire (i.e., in the current window or a new window).

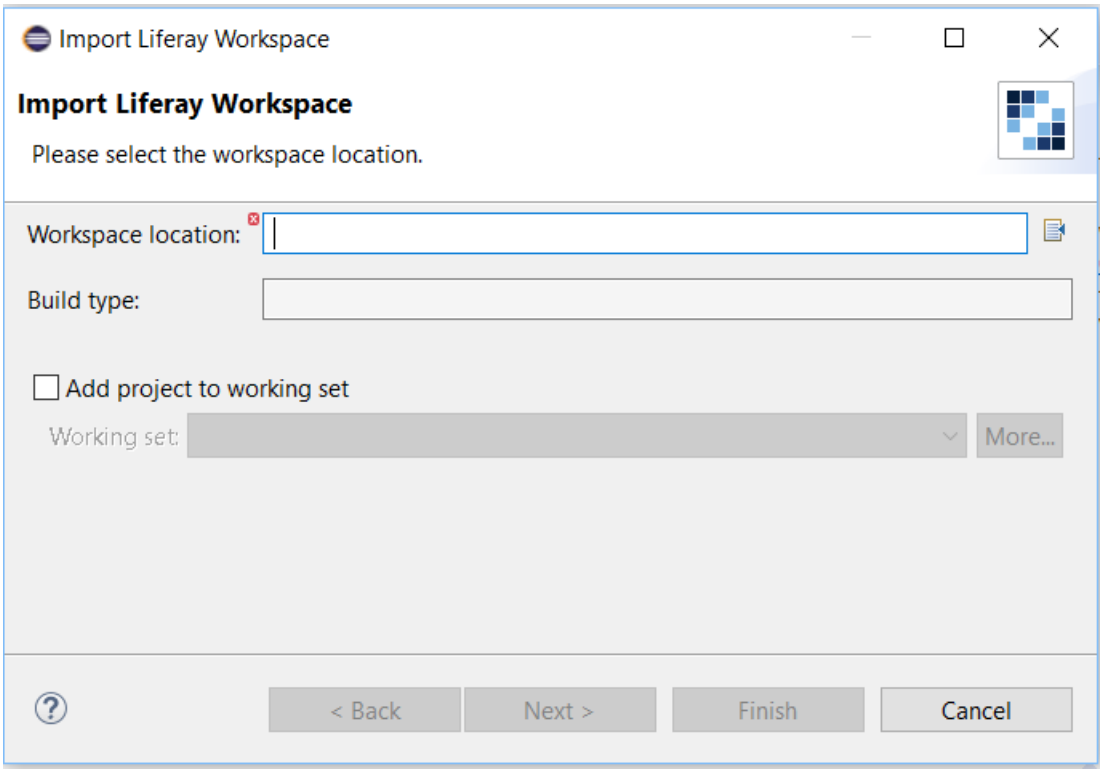

Figure 880.1: You can import an existing Liferay Workspace into your current Dev Studio session.

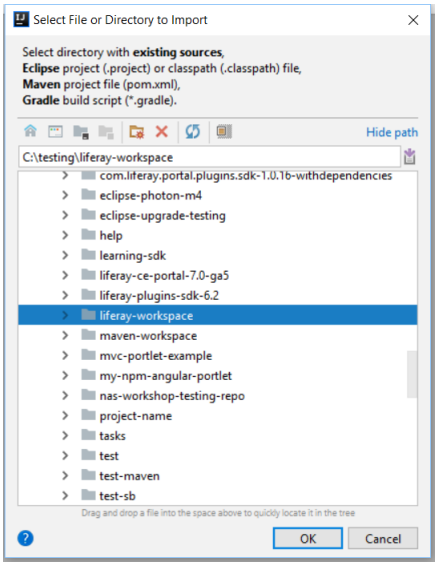

Figure 880.2: Specify your workspace's configurations.

CHAPTER 881

SETTING PROXY REQUIREMENTS FOR LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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.

881.1 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.

881.2 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.

CHAPTER 882

ADDING A LIFERAY BUNDLE TO LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Liferay Workspaces can generate and hold a Liferay Server. This lets you build/test your workspace's plugins against a running Liferay instance. Follow the instructions below to get started.

1. Open your workspace's root `gradle.properties` file.
2. Set the `liferay.workspace.bundle.url` property to the bundle's download URL you want to generate and install. For example,

   ```
   liferay.workspace.bundle.url=https://releases-cdn.liferay.com/portal/7.2.0-ga1/liferay-ce-portal-tomcat-7.2.0-ga1-20190531153709761.7z
   ```

   For DXP subscribers, it would look like this:

   ```
   liferay.workspace.bundle.url=https://api.liferay.com/downloads/portal/7.2.10/liferay-dxp-tomcat-7.2.10-ga1-20190531140450482.7z
   ```

```
**Note:** The DXP download URL must be set to the bundle hosted on
*api.liferay.com*. It can be tricky to find the fully qualified bundle
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 from
Liferay's Customer Portal, take note of the file name, and append it to
`https://api.liferay.com/downloads/portal/`.
```

```
DXP subscribers must also set the `liferay.workspace.bundle.token.download`
property to `true` to allow your workspace to access Liferay's API site.
```

3. Navigate to your workspace's root folder and run

   ```
   blade server init
   ```

4. Verify your bundle was downloaded. The bundle is generated in the `bundles` folder by default. You can change this by setting the `gradle.properties` file's `liferay.workspace.home.dir` property to a different 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 Projects section.

You're all set to develop projects for a nested Liferay DXP bundle.

CHAPTER 883

SETTING ENVIRONMENT CONFIGURATIONS FOR LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Liferay Workspace offers the `configs` folder, which provides a way to organize multiple environment settings and generate a Liferay bundle for each environment configuration.

To simulate using the `configs` folder, you'll 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:

   ```
   blade server init --environment uat
   ```

5. To generate a distributable Liferay DXP installation of the environment to the workspace's `/build` folder, run

```
./gradlew distBundle[Zip|Tar] -Pliferay.workspace.environment=uat
```

```
**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 environment variable inside the
`[USER_HOME]/.gradle/gradle.properties` file.

```properties
liferay.workspace.environment=local
```

The variable is set to `local` by default.
```

```
You've successfully configured two environments and generated one of them.
```

Awesome! You can now test various Liferay DXP bundle environments using Liferay Workspace.

CHAPTER 884

BUILDING NODE.JS THEMES IN LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 for this folder. 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.

   "'bash 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. For more information on building Gradle/Maven-based themes in workspace, see its dedicated article.

CHAPTER 885

BUILDING GRADLE/MAVEN THEMES IN LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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.

Follow the steps below to build a Gradle/Maven theme in workspace's `wars` folder:

1. Follow the Creating a Project article to generate a project based on a project template or archetype. Make sure to select the `theme` template.

   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.

2. Navigate into your new theme and run

   ```
   ./gradlew build
   ```

   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 article for more information.

Awesome! You know how WAR-style themes are built in workspace and where they should reside.

CHAPTER 886

MANAGING THE TARGET PLATFORM

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Note: The Target Platform articles currently assume you're using Gradle as a build tool. If your projects are built with Maven, you can still leverage the Target Platform features, but it is not built into Liferay Workspace *yet* (LPS-90524). See the Targeting a Platform with Maven article to set the Target Platform for Maven-based projects.

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.

Next, you'll discover how all of this is possible.

886.1 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 (see this article for details).

Note: The Target Platform feature is only supported for Gradle projects at this time.

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. To view a BOM file and its mapping of artifacts and versions, visit repository.liferay.com and search for the BOM artifacts (e.g., release.portal.bom and release.dxp.bom).

Pretty cool, right? Next, you'll learn how to leverage platform targeting in Dev Studio.

886.2 Leveraging Target Platform in Dev Studio

Liferay Dev Studio 3.2+ helps you streamline targeting a specific version even more. Dev Studio can index the configured Liferay DXP source code to

- provide advanced Java search (Open Type and Reference Searching) (article)
- debug Liferay DXP sources (article)

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+). See the Updating Liferay Workspace article for instructions on how to update your workspace.

These options in Dev Studio 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 article for more info on applying the Target Platform Gradle plugin.

Continue on to learn how to set the target platform.

CHAPTER 887

SETTING THE TARGET PLATFORM

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Setting the target platform 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=7.2.0
    ```

```
 **Note:** You must explicitly uncomment the property in your workspace's
 `gradle.properties` file to set it. Target Platform is not enabled by
 default.
```

```
If you're using Liferay DXP, you can set the property like this:

```properties
liferay.workspace.target.platform.version=7.2.10
```

The versions following a GA1 release of DXP follow fix pack versions (e.g.,
`7.2.10.fp1`, `7.2.10.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 articles for more info.

Note: 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 article to do this.

You've configured your target platform in workspace. You're all set!

CHAPTER 888

TARGETING A PLATFORM OUTSIDE OF WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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, follow the steps below.

1. Open your project's `build.gradle` file and add this:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.target.platform", version: "2.0.0"
    }
    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}
```

   This sets the dependency on the Target Platform Gradle plugin and configures the repository that provides the necessary artifacts for your project build.

2. Apply Liferay's Target Platform Gradle plugin to the `build.xml` file:

```
apply plugin: "com.liferay.target.platform"
```

3. Set the Target Platform plugin's dependencies:

```
dependencies {
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom", version: "7.2.0"
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom.compile.only", version: "7.2.0"
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom.third.party", version: "7.2.0"
}
```

   These dependencies are described below:

   `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`).

`release.portal.bom.third.party`: provides all third party artifacts that make up the Liferay Portal bundle.

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.bom.third.party` → `release.dxp.bom.third.party`
- `7.2.0` → `7.2.10`

4. If you're interested in advanced search and/or debugging Liferay DXP's source using Liferay Dev Studio, 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.

Now you can define your target platform!

CHAPTER 889

TARGETING A PLATFORM WITH MAVEN

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Although a Maven-based Liferay Workspace does not offer a configurable property to set the target platform, you can still leverage the Target Platform framework by adding a few dependencies to your project.

1. Open your workspace's root `pom.xml` file and add the following dependencies:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.liferay.portal</groupId>
            <artifactId>release.portal.bom</artifactId>
            <version>7.2.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>com.liferay.portal</groupId>
            <artifactId>release.portal.bom.compile.only</artifactId>
            <version>7.2.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>com.liferay.portal</groupId>
            <artifactId>release.portal.bom.third.party</artifactId>
            <version>7.2.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

   These dependencies are described below:

   `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`).

   `release.portal.bom.third.party`: provides all third party artifacts that make up the Liferay Portal bundle.

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.bom.third.party`
- `7.2.0` → `7.2.10`

2. Go through the remaining POMs in your workspace and remove `<version>` tags for all Liferay-specific artifacts. These versions are now being provided by the Target Platform framework.

Great! You can now target a platform in your Maven-based workspace.

CHAPTER 890

VALIDATING MODULES AGAINST THE TARGET PLATFORM

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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.

You'll cover the following topics in this section:

- Resolving your modules.
- Modifying the target platform's capabilities.
- Including the resolver in your Gradle build.

Continue on to learn how this works.

890.1 Resolving Your Modules

You can resolve 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=7.2.0
```

If you're using Liferay DXP, you can set the property like this:

```
liferay.workspace.target.platform.version=7.2.10
```

The versions following a GA1 release of DXP follow fix pack versions (e.g., `7.2.10.fp1`, `7.2.10.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! If the resolver throws errors, see the article on how to resolve common output errors reported by the `resolve` task. Sometimes, you must modify the `resolve` task's default behavior to successfully validate your app. See the next section for more information.

890.2 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.

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.

890.3 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

Note: You should only embed a third party library in your module if it's the only module that depends on it. You should not bypass the resolver failure this way if more than one project in the OSGi container depends on that library.

For help resolving third party dependency errors, see the Resolving Third Party Library Package Dependencies tutorial.

890.4 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 Depending on a Customized Distribution of Liferay DXP article.

890.5 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.

You can learn how to include the resolver in your Gradle build by visiting this article.

Continue on for various step-by-step instructions for configuring/manipulating the resolver task.

CHAPTER 891

ADDING A THIRD PARTY LIBRARY'S CAPABILITIES TO THE RESOLVER'S CAPABILITIES

This document has been updated and ported to Liferay Learn and is no longer maintained here.

You can add your third party dependencies to the target platform's default list of capabilities by listing them as provided modules.

1. Open your workspace's root `build.gradle` file.
2. Add a code snippet similar to this:

```
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`).

CHAPTER 892

SKIPPING THE RESOLVING PROCESS FOR A MODULE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

It may be easiest to skip validating a particular module during the resolve process.

1. Open your workspace's root `build.gradle` file.
2. 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`).

3. (Optional) 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.

CHAPTER 893

DEPENDING ON A CUSTOMIZED DISTRIBUTION OF LIFERAY DXP

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 7.2.0
   ```

   Liferay DXP users must replace the `release.portal.distro` artifact name with `release.dxp.distro` and use the `7.2.10` 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.

CHAPTER 894

INCLUDING THE RESOLVER IN YOUR GRADLE BUILD

This document has been updated and ported to Liferay Learn and is no longer maintained here.

To call the `resolve` task during Gradle's `check` task automatically, follow the instructions below:

1. Open your workspace's root `build.gradle` file.
2. Add the following directive:

   ```
   check.dependsOn resolve
   ```

   The `resolve` task is now called during the `check` task.

   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`.
3. (Optional) 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.

CHAPTER 895

How to Resolve Common Output Errors Reported by the Resolve Task

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 general help with OSGi related issues, visit the Troubleshooting FAQ 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.

895.1 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.

895.2 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.

895.3 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.

CHAPTER 896

VALIDATING MODULES OUTSIDE OF WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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. Follow the instructions below to do this.

1. Open your project's root `build.gradle` file.
2. Add the `targetPlatformDistro` dependency to the list of dependencies. It should look like this:

```
dependencies {
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom", version: "7.2.0"
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom.compile.only", version: "7.2.0"
    targetPlatformDistro group: "com.liferay.portal", name "release.portal.distro", version: "7.2.0"
}
```

   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`
   - `7.2.0` → `7.2.10`

Now you can validate your non-workspace modules against a target platform!

CHAPTER 897

LEVERAGING DOCKER

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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.

The following Docker commands (Gradle-based) are available in Liferay Workspace:

Command | Description `buildDockerImage` | Builds the Docker image with all modules/configurations deployed. `createDockerContainer` | Creates a Docker container from the Liferay DXP image and mounts the workspace's `/build/docker` folder to the container's `/etc/liferay` folder. `createDockerfile` | Creates a `Dockerfile` to build the Docker image. `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. `logsDockerContainer` | Prints the portal runtime's logs. You can exit log tracking mode while maintaining a running container (e.g., [Ctrl|Command] + C). `pullDockerImage` | Pulls the Docker image. `removeDockerContainer` | Removes the container from Docker's system. `startDockerContainer` | Starts the Docker container. `stopDockerContainer` | Stops the Docker container.

Note: Leveraging Docker in Liferay Workspace is only available for Gradle projects at this time.

In this section, you'll learn how to

- Create a Docker container based on a provided Liferay DXP image.

- Configure the container.
- Build a custom image.

Continue on to learn more.

CHAPTER 898

CREATING A LIFERAY DXP DOCKER CONTAINER

This document has been updated and ported to Liferay Learn and is no longer maintained here.

To create a Liferay DXP Docker container in Liferay Workspace, complete the steps below.

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 7.2 GA1 image, you would set this property:

   ```
   liferay.workspace.docker.image.liferay=liferay/portal:7.2.0-ga1
   ```

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.

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.

Once your container is created, you can configure it.

CHAPTER 899

CONFIGURING A DOCKER CONTAINER

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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.

As an example, you'll enable the Gogo shell for your container.

1. Add a `portal-ext.properties` file to your workspace's `configs/docker` folder.
2. Add the following property to the `portal-ext.properties` file:

   ```
   module.framework.properties.osgi.console=0.0.0.0:11311
   ```

   This lets you access your container using Gogo shell via telnet session.
3. Start the container.

   Once the container is started, the configurations stored in `configs/common` 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.
```

You can now apply configurations to your Liferay DXP Docker container.

CHAPTER 900

BUILDING A CUSTOM DOCKER IMAGE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

You can preserve your container's configuration by building it as an image.

1. Build your custom Liferay DXP image by running

   ```
   ./gradlew buildDockerImage
   ```

   A `Dockerfile` is generated for your container when building your image. 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.

   You can generate a `Dockerfile` manually at any time by running

   ```
   ./gradlew createDockerfile
   ```

2. Run `docker image ls` to verify the image's availability.

You can now build Liferay DXP Docker images in Liferay Workspace!

CHAPTER 901

UPDATING LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 DXP development experience. Updating your Liferay Workspace is easy; you'll learn how to do it for Gradle and Maven-based workspaces next.

901.1 Gradle

1. Find the latest Liferay Workspace version. To do this, view the Workspace Gradle plugin's released versions on Liferay's repository. Copy the version to which you want to upgrade.
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 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've upgraded your Gradle-based Liferay Workspace!

901.2 Maven

1. Find the latest Liferay Workspace version. To do this, view the Bundle Support plugin's released versions on Liferay's repository. Copy the version to which you want to upgrade.
2. Open your Liferay Workspace's root `pom.xml` file.
3. Within the `plugin` tags, you'll find code similar to below:

```
<plugin>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.portal.tools.bundle.support</artifactId>
    <version>3.4.2</version>
    ...
</plugin>
```

   Update the `com.liferay.portal.tools.bundle.support` artifact's `version` to the version number you copied in step 1.

4. Execute any Maven command to initiate the update process for your Workspace (e.g., `mvn verify`).

Awesome! You've upgraded your Maven-based Liferay Workspace!

CHAPTER 902

UPDATING DEFAULT PLUGINS PROVIDED BY LIFERAY WORKSPACE

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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 `build.gradle` 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=1.0.819
   ```

   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 903

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 for generating and managing your project. Liferay also provides easy to obtain Maven artifacts that are required for Liferay Maven module development. Here, you'll learn how to

- Install Liferay Maven artifacts
- Create/Manage a Maven Repository
- Apply Maven plugins

Because Liferay DXP is tool-agnostic, Maven is fully supported for Liferay DXP development. Read on for details about these topics.

903.1 Installing Liferay Maven Artifacts

To create Liferay projects using Maven, you'll need its dependencies. This isn't a problem—Liferay provides them as Maven artifacts. You can retrieve them from a remote repository.

There are two remote 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. Using the Central Repository to install Liferay Maven artifacts only requires that you specify your module's dependencies in its `pom.xml` file.

When packaging your module, the automatic Maven artifact installation process only downloads the artifacts necessary for that module from the Central Repository.

The Central Repository *usually* offers the latest Liferay Maven artifacts, but the Liferay Repository *guarantees* access to the latest artifacts released by Liferay. Other than a slight delay in artifact releases, the two repositories are identical. When the Liferay repository is configured in your `settings.xml` file, archetypes are generated based on that repository's contents. The Liferay Maven repository offers a good alternative for those who want the most up-to-date Maven artifacts produced by Liferay.

Note: If you've configured the Liferay Nexus repository to access Liferay Maven artifacts and you've already been syncing from the Central Repository, you might have to clear out parts of your

local repository to force Maven to re-download the newer artifacts. Also, don't leave the Liferay repository enabled when publishing artifacts to Maven Central. You must comment out the Liferay Repository credentials when publishing your artifacts.

Next, you'll learn about managing your Maven artifacts.

903.2 Managing Maven Artifacts in a Repository

You can share Liferay artifacts and modules with teammates or manage your repositories using a GUI by using Sonatype Nexus. It's a Maven repository management server for 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 section focuses on Nexus.

You'll learn how to

- Create a repository
- Configure a repository
- Deploy artifacts to a repository

Before using 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.

Note: You must only configure a repository server if you're sharing artifacts (e.g., Liferay artifacts and/or your modules) with others. If you're 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 Remote Liferay Maven Artifacts article.

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.

Once you've created a deployable artifact, you can 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.

903.3 Applying Maven Plugins

There are several important Maven plugins that provide important functionality to Liferay Maven projects. The available Liferay Maven plugins are available in the Maven Plugins section.

The following tasks are covered in this section:

- Building an OSGi module JAR
- Building themes
- Compiling Sass files

- Using Service Builder

Read on to learn more!

CHAPTER 904

INSTALLING LIFERAY MAVEN ARTIFACTS

If you haven't configured your project to retrieve artifacts from a custom Maven repository, your project will leverage the artifacts from the Central Repository. You can view these artifacts from the Maven Central Repository site. Use the Latest Version column as a guide to see what's available for the version of Liferay DXP you're developing for.

If you'd like to access Liferay's latest released Maven artifacts, configure Maven to use Liferay's Nexus repository instead. To do this, open your project's parent `pom.xml` and add this:

```
<repositories>
    <repository>
        <id>liferay-public-releases</id>
        <name>Liferay Public Releases</name>
        <url>https://repository-cdn.liferay.com/nexus/content/repositories/liferay-public-releases</url>
    </repository>
</repositories>

<pluginRepositories>
    <pluginRepository>
        <id>liferay-public-releases</id>
        <url>https://repository-cdn.liferay.com/nexus/content/repositories/liferay-public-releases/</url>
    </pluginRepository>
</pluginRepositories>
```

The above configuration retrieves artifacts from Liferay's release repository.

If you're most interested in retrieving Liferay's latest snapshot artifacts, follow the instructions below to configure Liferay's Nexus repository to access them.

1. Open your project's parent `pom.xml` and add this:

    ```
    <repositories>
        <repository>
            <id>liferay-public-snapshots</id>
            <name>Liferay Public Snapshots</name>
            <url>https://repository-cdn.liferay.com/nexus/content/repositories/liferay-public-snapshots</url>
        </repository>
    </repositories>

    <pluginRepositories>
        <pluginRepository>
            <id>liferay-public-snapshots</id>
            <url>https://repository-cdn.liferay.com/nexus/content/repositories/liferay-public-snapshots/</url>
        </pluginRepository>
    </pluginRepositories>
    ```

2. Enable your project to access snapshot artifacts by adding this code to your parent project's `pom.xml`:

```
<snapshots>
    <enabled>true</enabled>
</snapshots>
```

You're now equipped to access Liferay's Maven artifacts via the

- Central Repository
- Liferay Repository (releases)
- Liferay repository (snapshots)

Great job!

CHAPTER 905

CREATING A MAVEN REPOSITORY

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*.

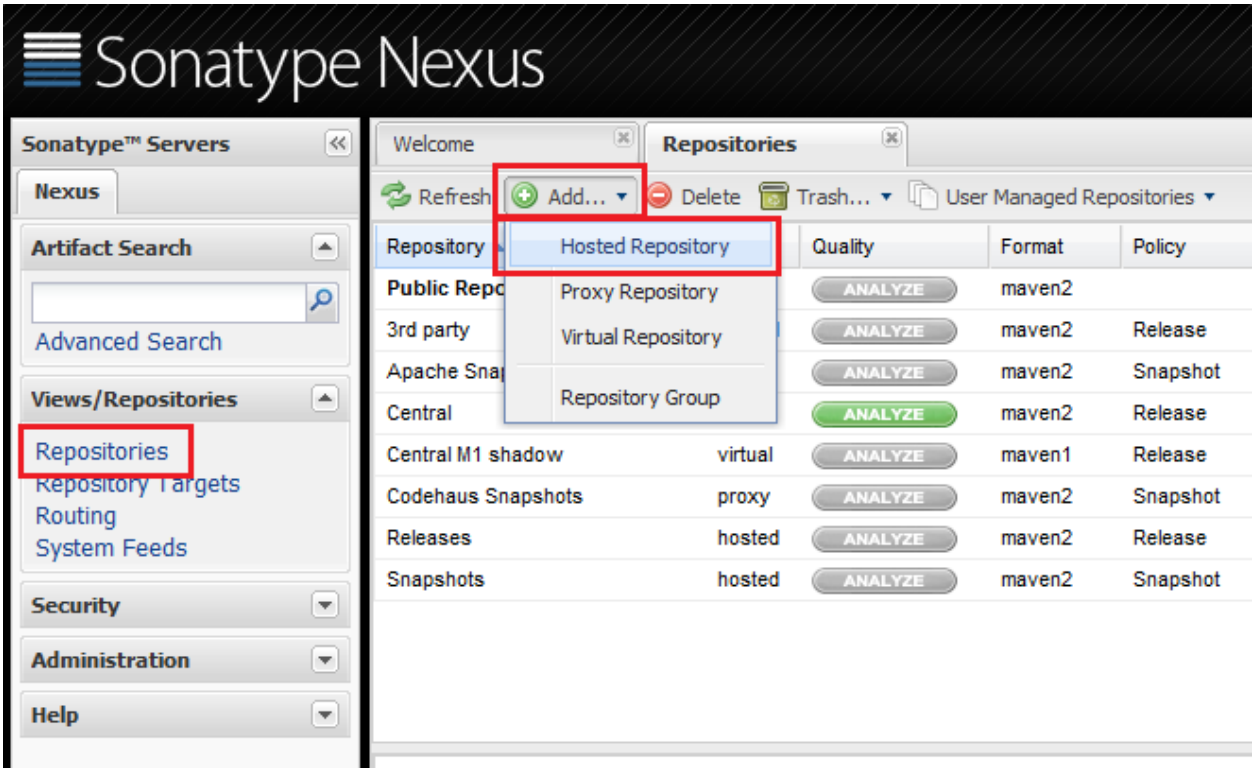

Figure 905.1: Adding a repository to hold your Liferay artifacts is easy with Nexus.

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*

    To create a snapshot repository, choose *Snapshot* for the Repository Policy and update the ID and name accordingly.

4. Click *Save*.

Voila! You've created a repository for your Liferay releases (i.e., `liferay-releases`) and/or 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.

See the Configuring Local Maven Settings to Access Repositories to configure your new repository servers in your Maven settings to install artifacts to them.

CHAPTER 906

CONFIGURING LOCAL MAVEN SETTINGS TO ACCESS REPOSITORIES

To configure your Maven environment to access your repository servers, do the following:

1. Navigate to your `[USER_HOME]/.m2/settings.xml` file. Create it if it doesn't yet exist.
2. Configure your repository server settings. 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!

CHAPTER 907

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, create one. Also make sure your `[USER_HOME]/.m2/settings.xml` file specifies your remote repository's ID, user name, and password (configuration instructions here).

To follow this article, 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.

Now it's time to deploy a Maven artifact to your Nexus repository.

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. This repository should also be specified in your [USER_HOME]/.m2/settings.xml. 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.

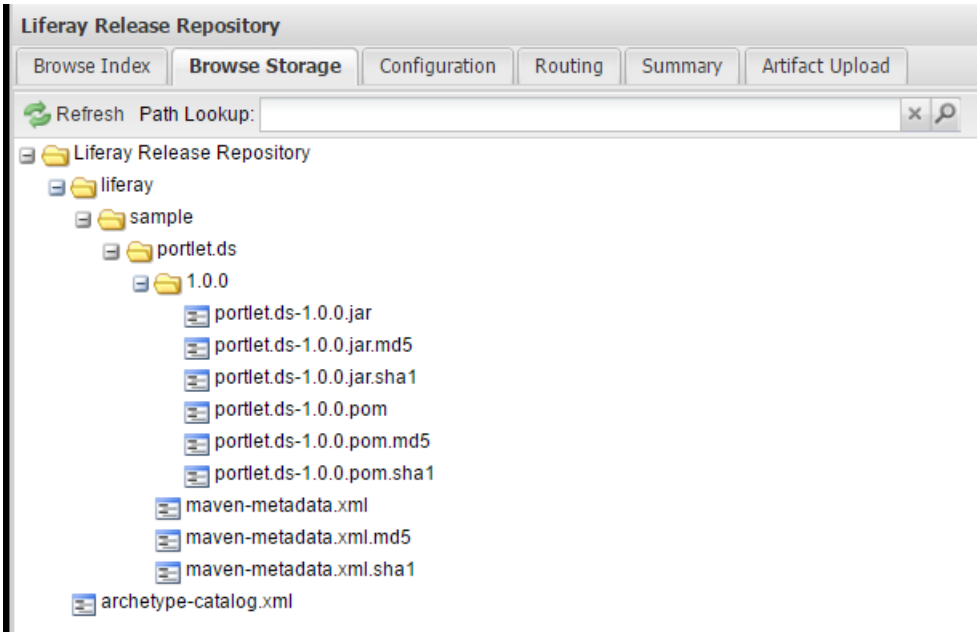

Figure 907.1: Your repository server now provides access to your Liferay Maven artifacts.

Awesome! You can now share your Liferay module projects with anyone by deploying them as artifacts to your remote repository!

CHAPTER 908

BUILDING AN OSGI MODULE JAR WITH 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/7-2/customization/-/knowledge_base/c/deploying-wars-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. Add 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 article.

Fantastic! You've configured your Liferay Maven project to package itself into a deployable OSGi module.

CHAPTER 909

BUILDING A THEME WITH MAVEN

Liferay's Theme Builder is 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.

The easiest way to create a Liferay theme with Maven is to create a new Maven project using Liferay's provided Theme archetype; Theme Builder is configured in the new project by default. 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 leverage Theme Builder. 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>1.1.7</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 plugin and 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. For more info on these properties, see the Theme Builder Plugin article.

2. Apply the CSS Builder plugin, which is required to use Theme Builder:

```
<plugin>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.css.builder</artifactId>
    <version>2.1.3</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>2.1.3</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>1.1.7</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 with Theme Builder!

CHAPTER 910

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 CSS Builder plugin, which 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 and 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. For more info on these properties, see the CSS Builder Plugin article.

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.

CHAPTER 911

USING SERVICE BUILDER IN A MAVEN PROJECT

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; it's configured in the new project by default. 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.

For these cases, you can configure Service Builder in your project manually. Follow the instructions below to configure Service Builder in your Maven project!

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>1.0.276</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>
```

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.

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>1.0.10</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 required to generate your project's remote services.

See the WSDD Builder Plugin article for more information on the configuration properties.

Terrific! You've successfully configured your Maven project to use Service Builder by applying the Service Builder and WSDD Builder plugins in your project's POM.

CHAPTER 912

UPGRADING YOUR MAVEN BUILD ENVIRONMENT

Note: This upgrade article only applies to projects residing in a pre Liferay Portal 7.0 Maven environment that are not upgrading to Liferay Workspace. If you're interested in upgrading to a Maven-based Liferay Workspace (recommended), see the Upgrading Code to 7.0 tutorials for more information.

If you're an avid Maven user and have been using it for Liferay Portal 6.2 project development or older, 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 section.

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

You'll start off by upgrading your Maven environment's Liferay Maven plugins.

912.1 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>2.1.3</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 articles explaining how to configure them:

Common Liferay Maven Plugins

Name	Artifact ID	Tutorial
Bundle Support	com.liferay.portal.tools.bundle.support	Bundle Support Plugin
CSS Builder	com.liferay.css.builder	CSS Builder Plugin
DB Support	com.liferay.portal.tools.db.support	DB Support Plugin
Deployment Helper	com.liferay.deployment.helper	Deployment Helper Plugin
Javadoc Formatter	com.liferay.javadoc.formatter	Javadoc Formatter Plugin
Lang Builder	com.liferay.lang.builder	Lang Builder Plugin
REST Builder	com.liferay.portal.tools.rest.builder	REST Builder Plugin
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	TLD Formatter Plugin
WSDD Builder	com.liferay.portal.tools.wsdd.builder	WSDD Builder Plugin
XML Formatter	com.liferay.xml.formatter	XML Formatter Plugin

Note: When upgrading to a Liferay Workspace built with Maven, many of these plugins are applied to the workspace by default.

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.

912.2 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 article.

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.

With these updates, you can easily upgrade your Liferay Maven build so you can begin developing projects for 7.0.

CHAPTER 913

THEME GENERATOR

The Liferay Theme Generator generates themes for Liferay DXP. It is just one of Liferay JS Theme Toolkit's tools.

A couple 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`
7.2	9.x.x	`npm install -g generator-liferay-theme@^9.x.x`

See Version Compatibility Matrix for more information.

913.1 Sub-generators

The Liferay Theme Generator includes the sub-generators listed in the table below:

Sub-generator	Command to run	Description
Layouts	`yo liferay-theme:layout`	Generate layout templates with an interactive VIM.
Themelets	`yo liferay-theme:themelet`	Create small, reusable, pieces of CSS and HTML for your themes.

913.2 Layouts Sub-generator

The Layouts sub-generator provides the controls to create a layout template that meets your needs. You can add and remove rows and columns on-the-fly as you require. See Generating Layout Templates for more information.

913.3 Themelets Sub-generator

Themelets are small, extendable, and reusable pieces of code. They only require the CSS and/or JavaScript files that you want to add to your theme. This modular approach reduces the need for duplicated code across themes and makes it easy for developers to share code snippets with other developers. Themelets are applicable for changes both small and large, from modifying the appearance of an individual piece of UI to adding new features. If there is something you have to manually code for every theme you create, it's a good candidate for a themelet. See Generating Themelets for more information.

While you can create themes using the tools you prefer, the Liferay Theme Generator is designed with theme development for Liferay DXP in mind. Its toolset and features help streamline theme development.

Chapter 914

Installing the Theme Generator and Creating a Theme

The steps below show how to install the Liferay Theme Generator and generate a theme.

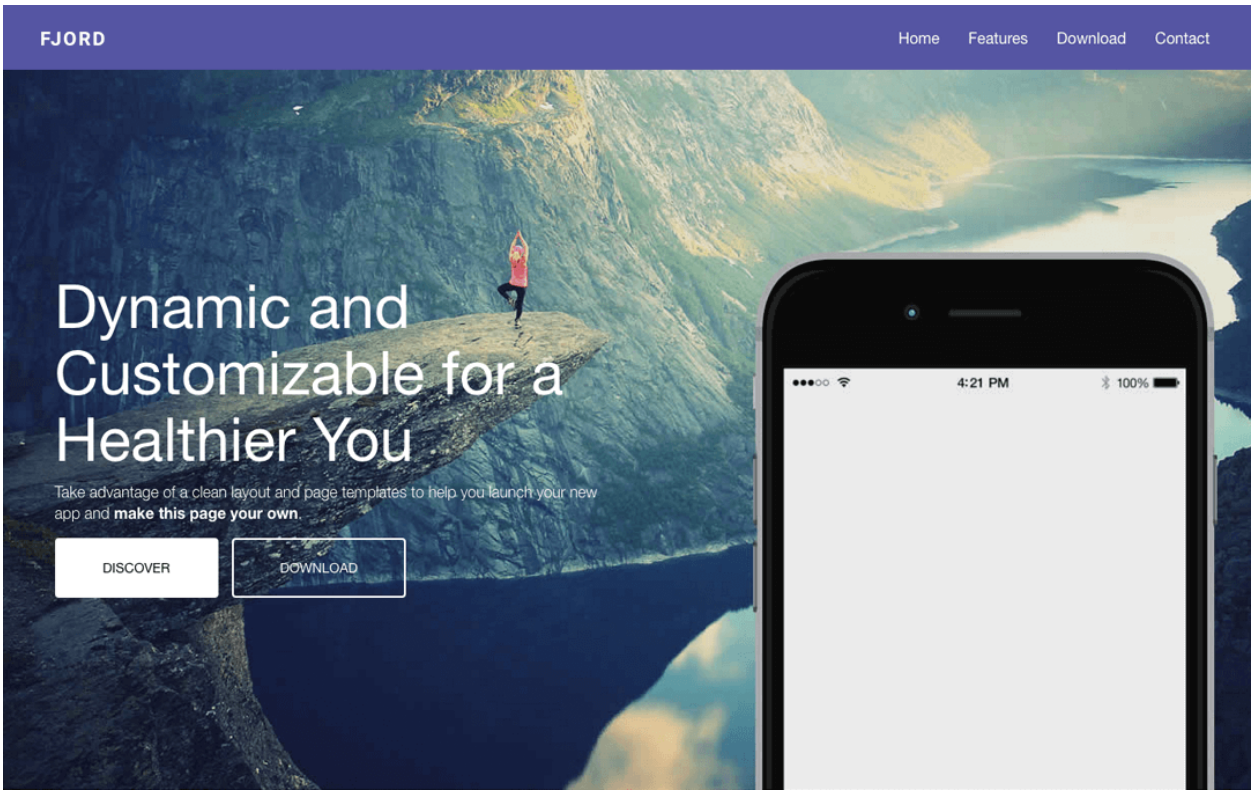

Figure 914.1: The tools are in your hands to create any theme you can imagine.

Your first step in generating a theme is installing NodeJS (along with Node Package Manager(npm)) if it's not already installed. We recommend installing v10.15.1, which is the version Liferay Portal 7.2 supports (See the compatibility matrix). Once NodeJS is installed and you've set

up your npm environment, you can follow these steps to install the Liferay Theme Generator and generate a theme:

1. Use npm to install the Yeoman dependency:

   ```
   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.
```

---

2. Install the Liferay Theme Generator with the command below:

   ```
   npm install -g generator-liferay-theme
   ```

   If you're on Windows, follow the instructions in step 3 to install Sass, otherwise you can skip to step 4.

3. The generator uses node-sass. If you're on Windows, you must also install node-gyp and Python.

4. Run the generator and follow the prompts to create your theme:

   ```
   yo liferay-theme
   ```

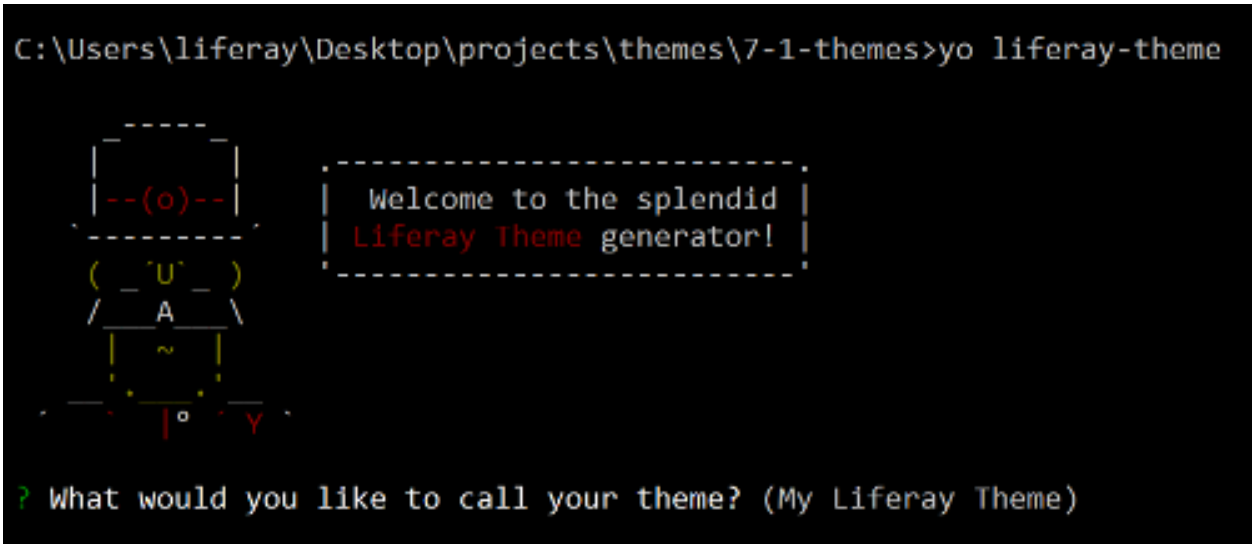

Figure 914.2: You can generate a theme by answering just a few configuration questions.

```
**Note:** Since Liferay DXP Fix Pack 2 and Liferay Portal 7.2 CE GA2, Font
Awesome is available globally as a system setting, which is enabled by
default. If you're using Font Awesome icons in your theme, answer yes (y)
to the Font Awesome question to include Font Awesome imports in your
theme. This ensures that your icons won't break if a Site Administrator
disables the global setting.
```

---

5. 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!

CHAPTER 915

GENERATING LAYOUT TEMPLATES WITH THE THEME GENERATOR

This article shows how to use the Liferay Theme Generator's Layouts sub-generator to create a layout template.

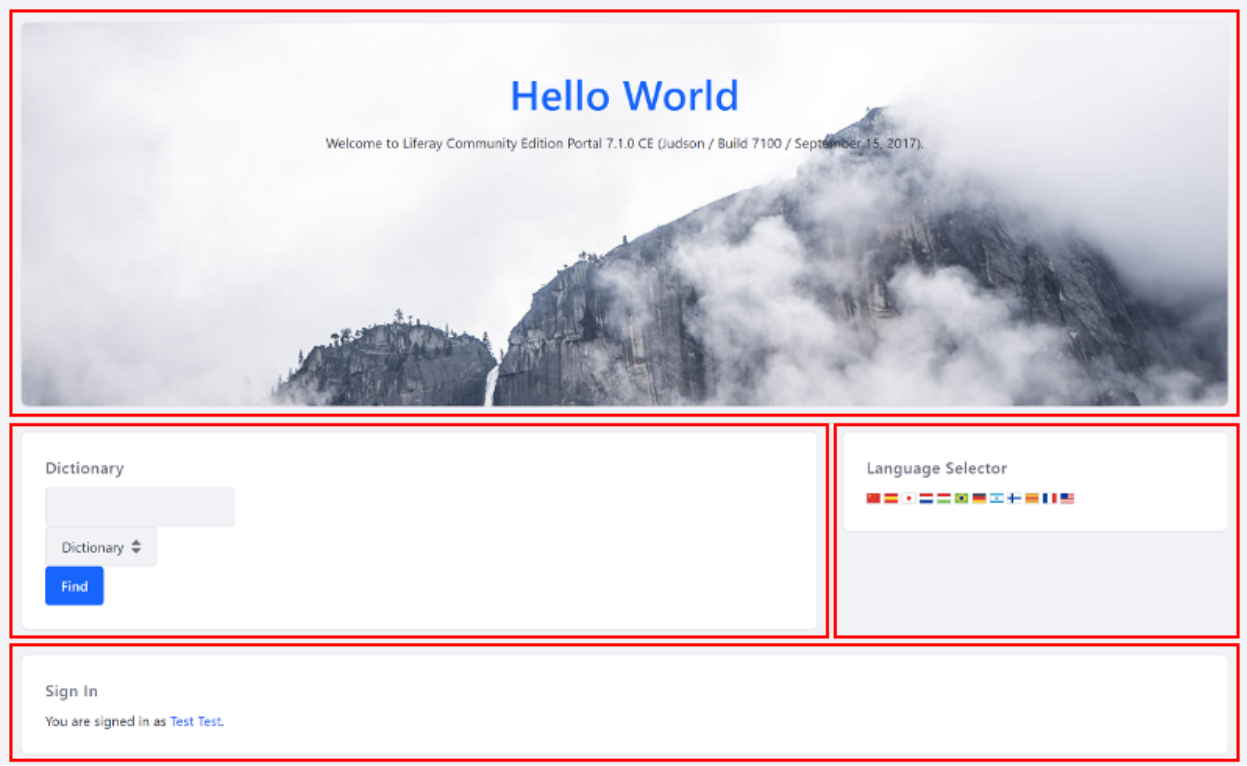

Figure 915.1: The *1-2-1 Columns* page layout creates a nice flow for your content.

Your first step in creating a layout template with the Liferay Theme Generator's Layouts sub-generator is installing the Liferay Theme Generator if it's not already installed. Once the generator

is installed, you can follow these steps to create a layout template:

1. Open the Command Line and navigate to the folder where you want to create your layout template.

```
**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.
```

---

2. Run The Layouts sub-generator with the command below, and use the options listed below to create your layout:

    ```
    yo liferay-theme:layout
    ```

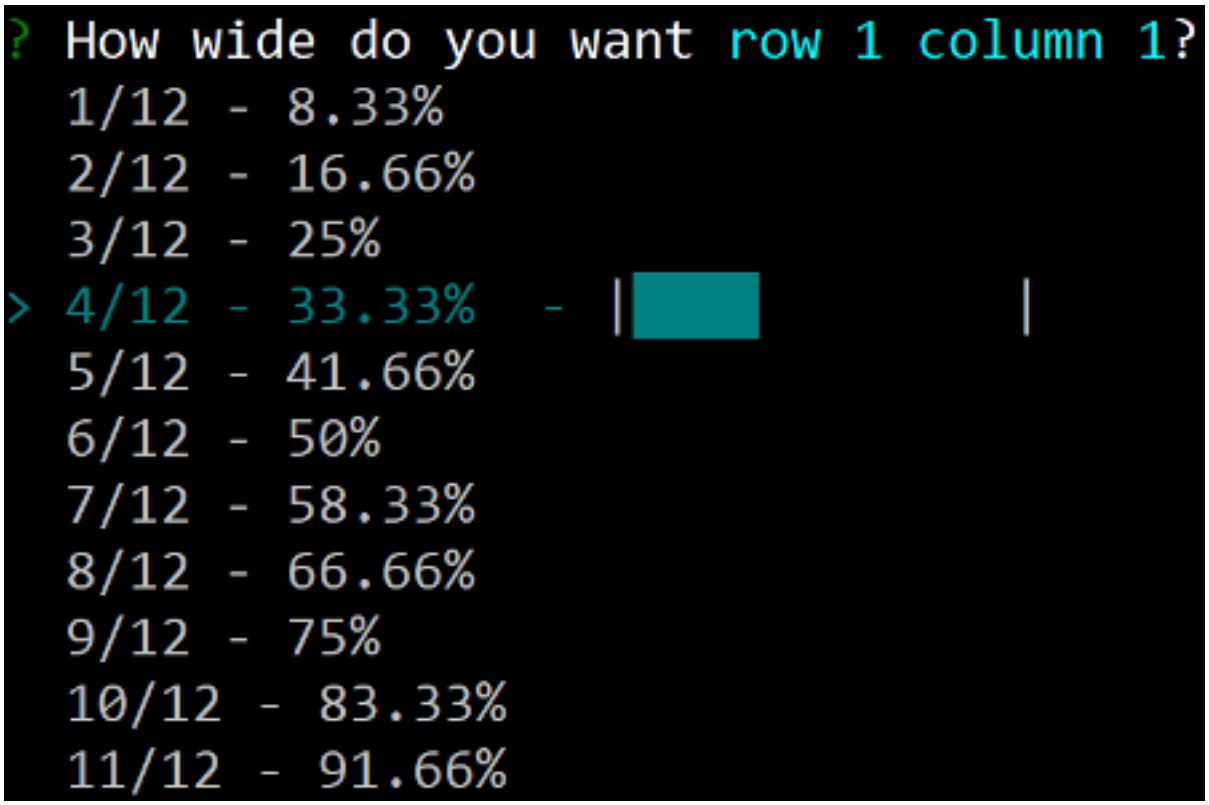

Figure 915.2: You must specify the width for each column in the row.

- **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.

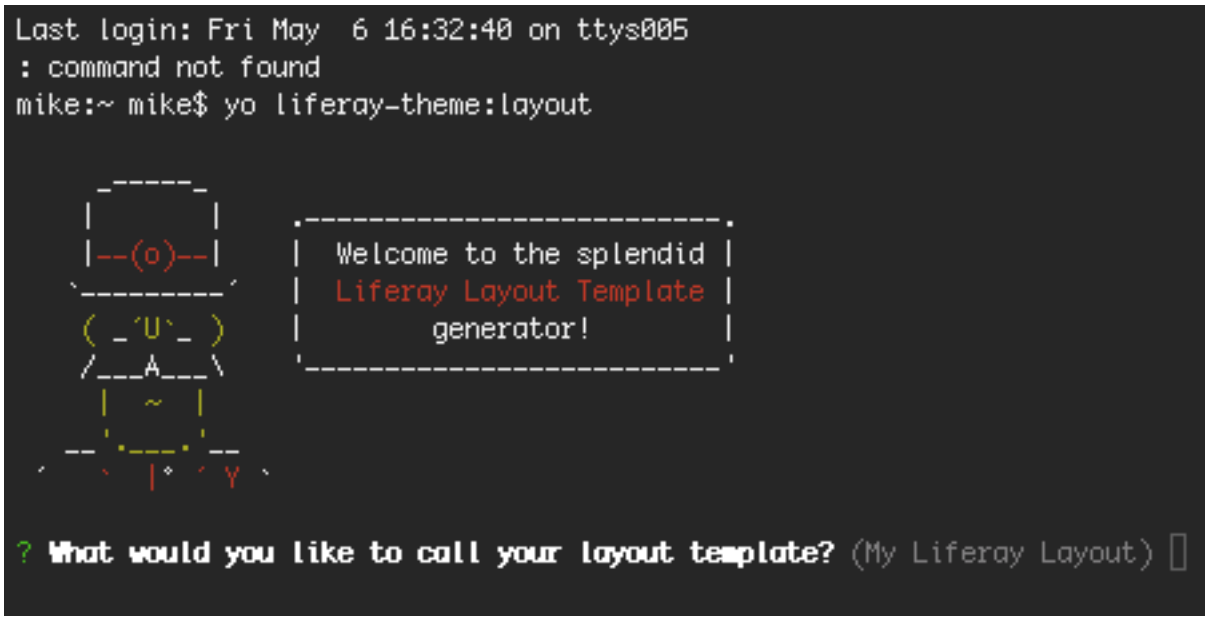

Figure 915.3: The Layouts sub-generator automates the layout creation process.

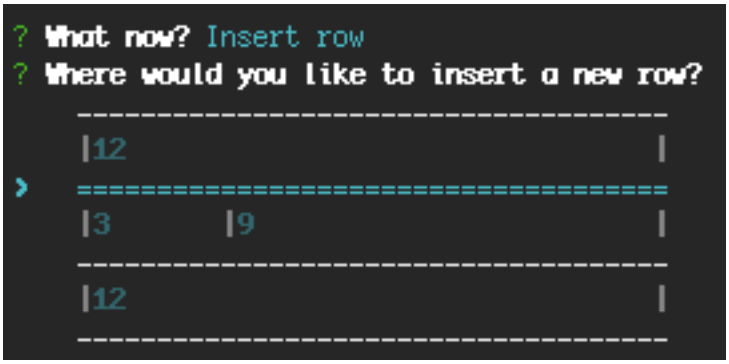

Figure 915.4: Rows can be inserted using the layout vi.

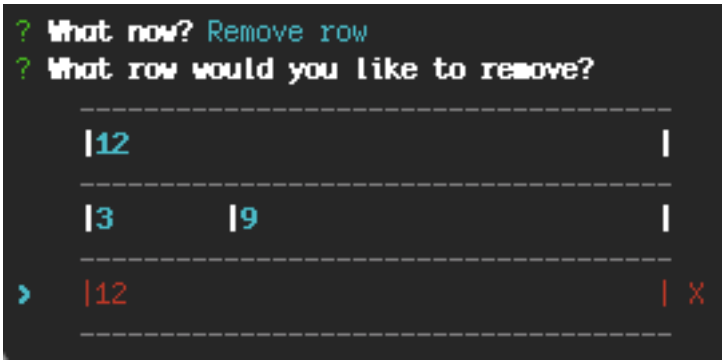

Figure 915.5: Rows are removed using the layout vi.

- **Remove row:** Displays a vi to remove your row. Use your arrow keys to select the row you want to remove, highlighted in red, then press Enter to remove the row.

- **Finish Layout:** Complete the layout template.

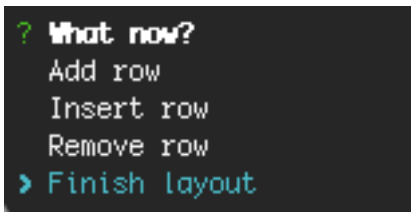

Figure 915.6: Select the *Finish layout* option to complete your design.

3. Run `gulp deploy` to deploy your layout template to the server you specified, or deploy your theme if the layout is bundled with it.

```
**Note:** Gulp is included as a local dependency of the generator, 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.
```

Awesome! You just created a layout template with the Theme Generator's Layouts sub-generator. Your layout template project should have a file structure similar to the one below:

```
- my-liferay-layout-layouttpl/
    - docroot/
        * WEB-INF/
            · liferay-layout-templates.xml
            · liferay-plugin-package.properties
        * my_liferay_layout.png
        * my_liferay_layout.tpl
    - node_modules/
        * (lots of packages)
    - gulpfile.js
    - liferay-plugin.json
    - package-lock.json
    - package.json
```

Chapter 916

Generating Themelets with the Theme Generator

This steps below show how to use the Liferay Theme Generator's Themelets sub-generator to create a themelet.

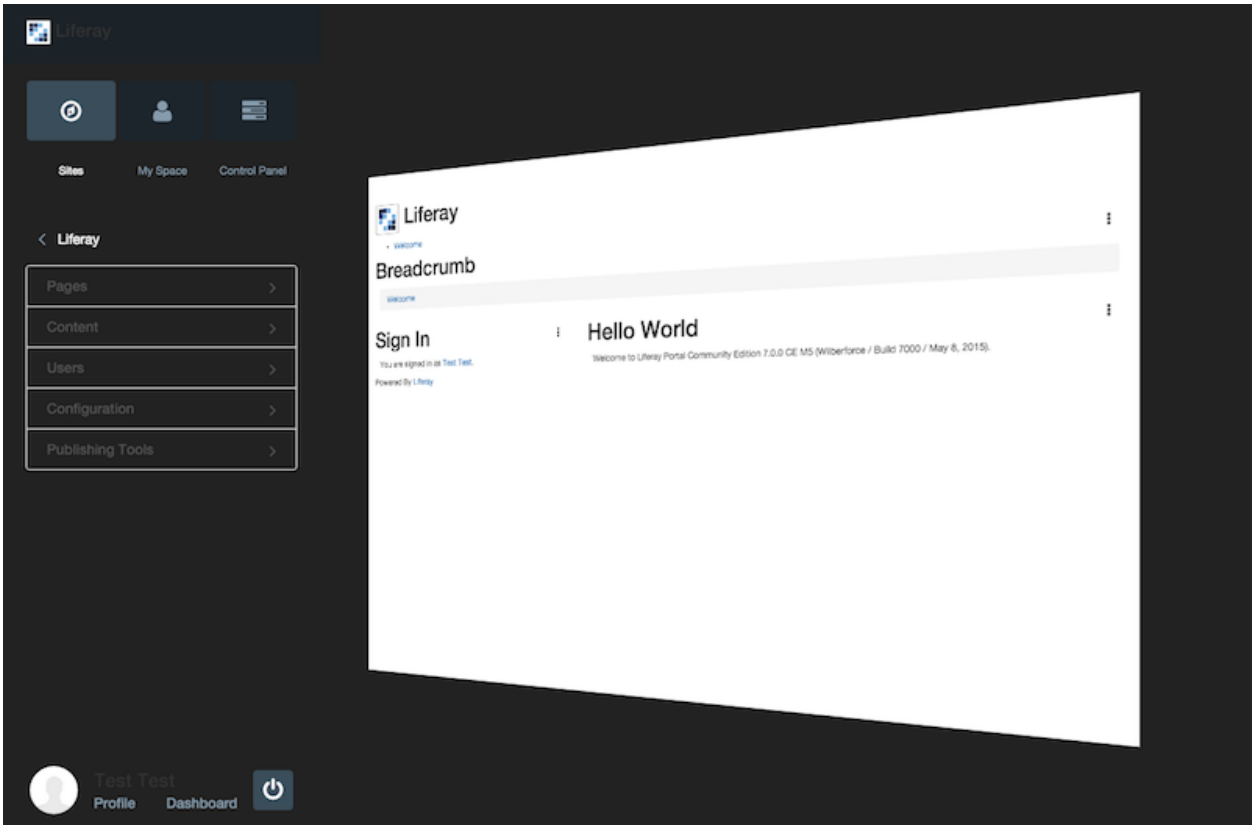

Figure 916.1: Themelets can be used to modify one aspect of the UI, that you can then reuse in your other themes.

Your first step in creating a themelet is installing the Liferay Theme Generator if it's not already

installed. Once the generator is installed, you can 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.

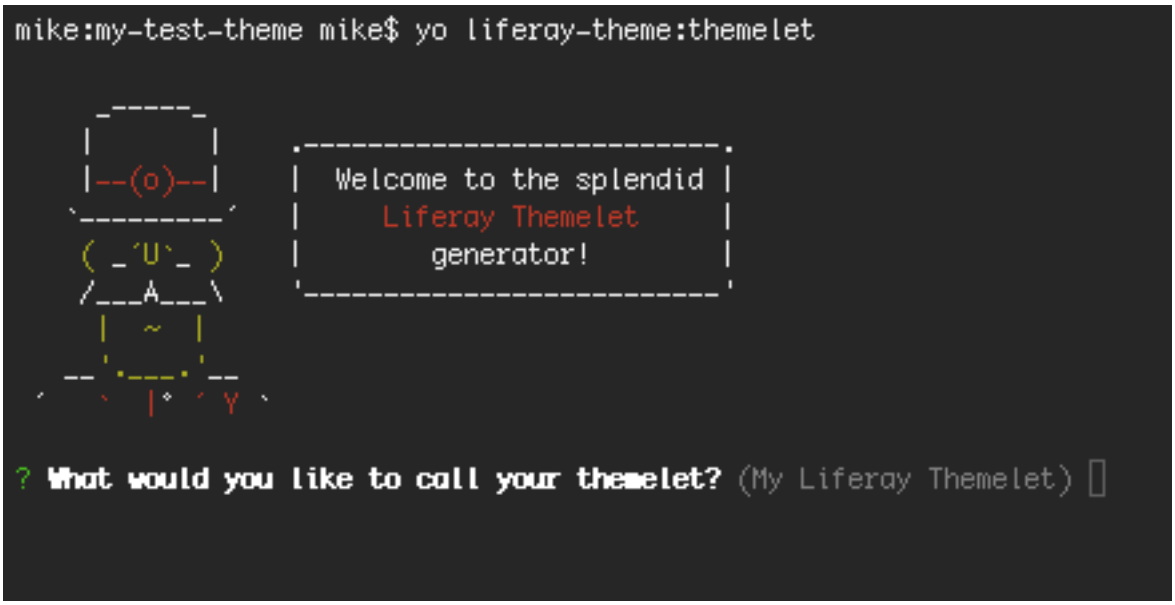

Figure 916.2: The Themelet sub-generator automates the themelet creation process, making it quick and easy.

   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.

3. 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.

Awesome! You just created a themelet with the Theme Generator's Themelets sub-generator.

CHAPTER 917

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

For step-by-step instructions for following the two upgrade paths, see the following documentation:

- Data Upgrade
- Code Upgrade

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*.

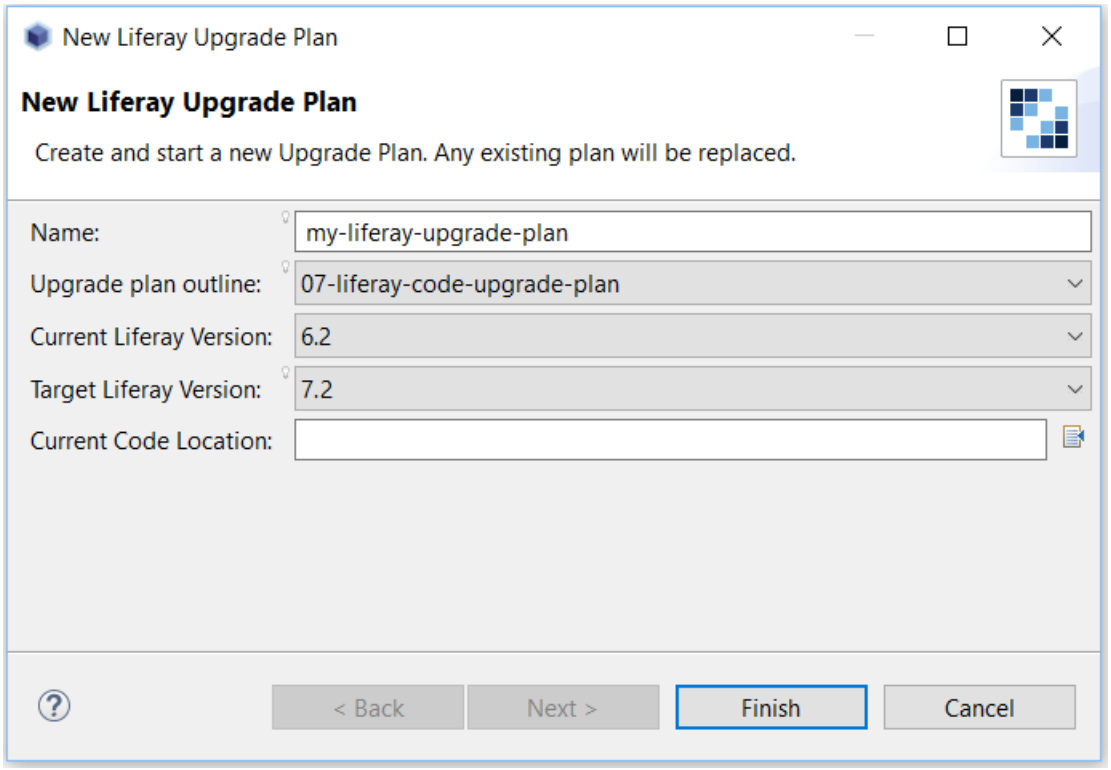

Figure 917.1: Configure your upgrade plan before beginning the upgrade process.

Switch to the new Liferay Upgrade Planner perspective (prompted automatically). You're now offered several windows in the UI:

- *Project Explorer:* displays your legacy plugin environment and new development environment. It also displays your upgrade problems that are detected during the *Fix Upgrade Problems* step.
- *Liferay Upgrade Plan:* outlines the upgrade plan's steps and step summaries.
- *Liferay Upgrade Plan Info:* shows official documentation that describes the upgrade step.

To progress through your upgrade plan, click the steps outlined in the Liferay Upgrade Plan window. Each step can have several options:

- *Click to preview:* previews what an automated step will perform.
- *Click to perform:* executes an automated process provided with the step. This is only offered for steps where the Upgrade Planner can assist.
- *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.

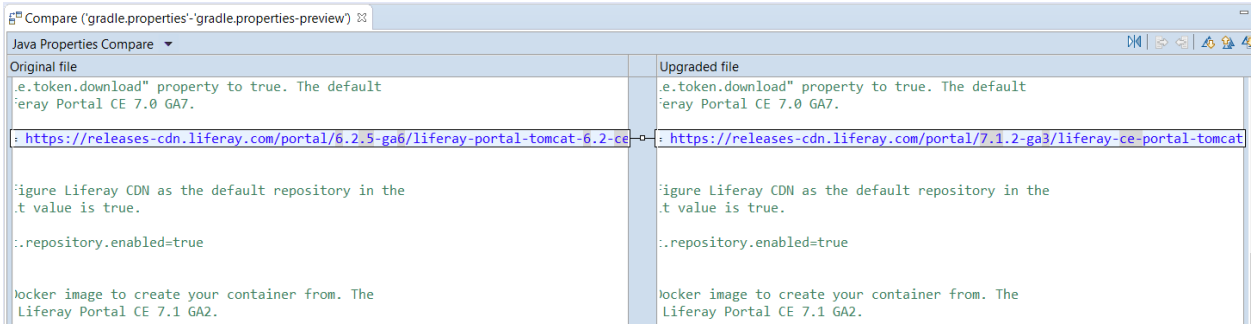

Figure 917.2: You can preview the Upgrade Planner's automated updates before you perform them.

Great! You now have a good understanding of the Liferay Upgrade Planner's UI and how to get started. Visit the Data Upgrade and Code Upgrade sections for more information on those upgrade processes.

CHAPTER 918

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!

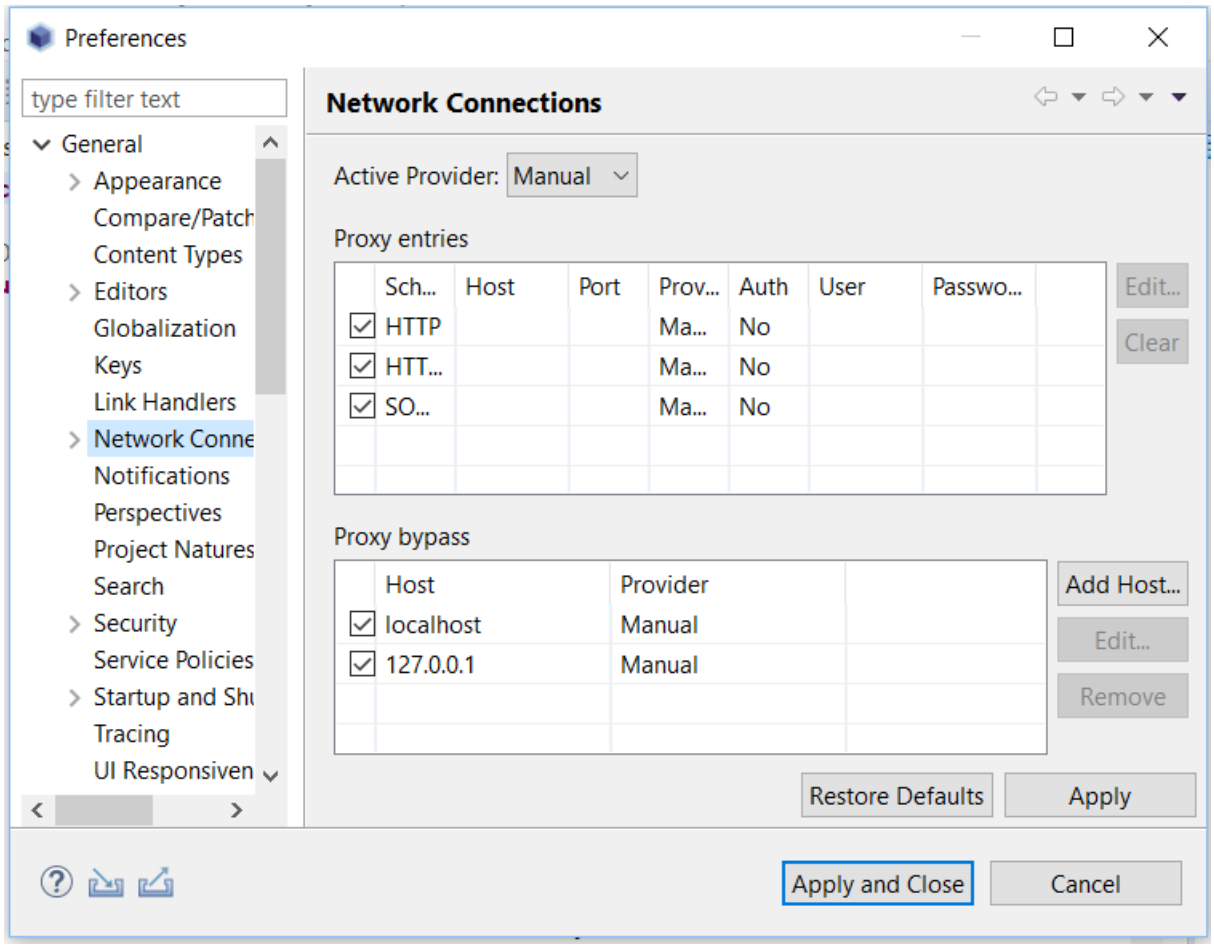

Figure 918.1: You can configure your proxy settings in Dev Studio's Network Connections menu.

CHAPTER 919

WEB EXPERIENCE MANAGEMENT REFERENCE

Browse this section's reference articles for additional information on the Web Experience Management framework.

CHAPTER 920

FRAGMENT SPECIFIC TAGS

There are Liferay-specific tags for creating editable, text, image, and link fields, and for embedding widgets.

920.1 Making Text Editable

You can make 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>
```

If you need formatting options like text or color styles, use `rich-text`:

```
<lfr-editable id="unique-id" type="rich-text">
   This is editable text that I can make bold or italic!
</lfr-editable>
```

The `lfr-editable` tag doesn't render without a unique `id`.

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.

920.2 Making Images Editable

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 to select an image or configure content mapping for the image.

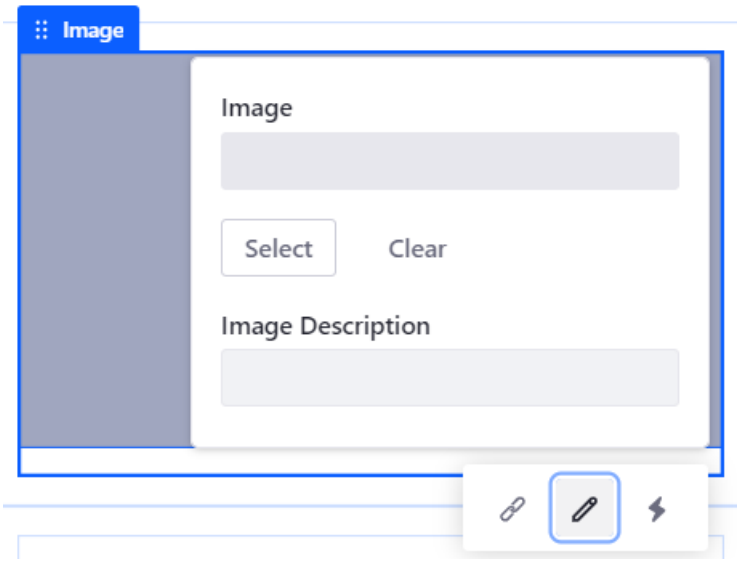

Figure 920.1: You have several options for defining an image on a Content Page.

Most images can be handled like this, but to add an editable background image you must add an additional property to set the background image ID, `data-lfr-background-image-id`. The background image ID is set in the main `div` for the Fragment and is the same as your editable image ID.

```
<div data-lfr-background-image-id="unique-id">
   <lfr-editable id="unique-id" type="image">
      <img src="...">
   </lfr-editable>
</div>
```

Content mapping connects editable fields in your Fragment with fields from an Asset type like Web Content or Blogs. For example, you can map an image field to display a preview image for a Web Content Article. For more information on mapping fields, see Editable Elements.

920.3 Creating Editable Links

There is also a specific syntax for creating editable link elements:

```
<lfr-editable id="unique-id" type="link">
    <a href="default-target-url-goes-here">Link text goes here</a>
</lfr-editable>
```

Marketers can edit the link text, target URL, and basic link styling—primary button, secondary button, link.

For more information on editable links, see Editable Links.

920.4 Including Widgets Within A Fragment

To include a widget, you must know its registered name. For example, the Site Navigation Menu portlet is registered as `nav`. Each registered portlet has an `lfr-widget-[name]` tag that's used to

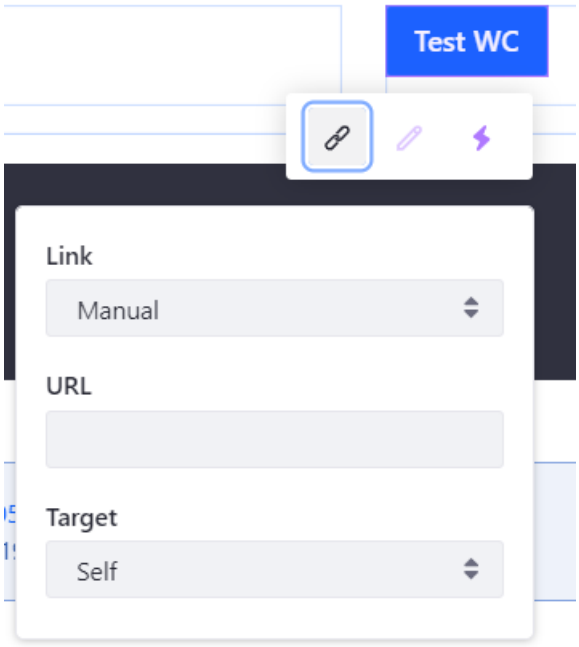

Figure 920.2: You have several options for defining a link's appearance and behavior.

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>
```

These are the widgets that can be embedded and their accompanying tags:

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>`

Widget Name	Tag
Iframe	`<lfr-widget-iframe>`

920.5 Enabling Embedding for Your Widget

If you have a custom widget that you want to embed in a fragment, you can configure that widget to be embeddable. To embed your widget, it must be an OSGi Component. Inside the `@Component` annotation for the portlet class 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.

CHAPTER 921

FRAGMENT CONFIGURATION TYPES

There are four configurable Fragment types available to implement:

- `checkbox`
- `colorPalette`
- `select`
- `text`

For more information on how to make a Fragment configurable, see Making a Fragment Configurable.

In this article, you'll explore JSON examples of each Fragment type.

921.1 Checkbox Configuration

The following JSON configuration creates a checkbox you can implement for cases where a boolean value selection is necessary:

```
{
    "fieldSets": [
        {
            "fields": [
                {
                    "name": "hideBody",
                    "label": "Hide Body",
                    "description": "hide-body",
                    "type": "checkbox",
                    "defaultValue": false
                }
                ...
            ]
        }
    ]
}
```

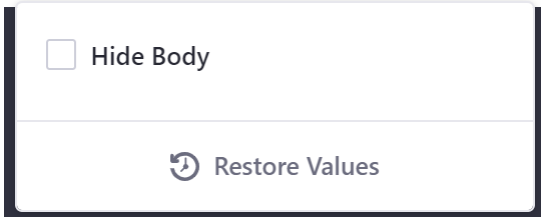

Figure 921.1: The checkbox configuration is useful when a boolean selection is necessary.

921.2 Color Palette Configuration

The following JSON configuration creates a color selector you can implement for cases where you must select a color:

```
{
    "fieldSets": [
        {
            "fields": [
                {
                    "name": "textColor",
                    "label": "Text color",
                    "type": "colorPalette",
                    "dataType": "object",
                    "defaultValue": {
                        "cssClass": "white",
                        "rgbValue": "rgb(255,255,255)"
                    }
                }
                ...
            ]
        }
    ]
}
```

The `colorPalette` type stores an object with two values: `cssClass` and `rgbValue`.

For example, if you implement the snippet above, you can use it in the FreeMarker context like this:

```
<h3 class="text-${configuration.textColor.cssClass}">Example</h3>
```

If you were to choose the color white, the `h3` tag heading would have the class `text-white'`.

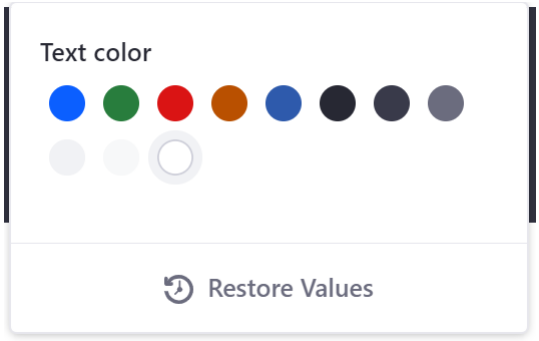

Figure 921.2: The `colorPalette` configuration is useful when a color selection is necessary.

921.3 Select Configuration

The following JSON configuration creates a selector you can implement for cases where you must select a predefined option:

```
{
    "fieldSets": [
        {
            "fields": [
                {
                    "name": "numberOfFeatures",
                    "label": "Number Of Features",
                    "description": "number-of-features",
                    "type": "select",
                    "dataType": "int",
                    "typeOptions": {
                        "validValues": [
                            {"value": "1"},
                            {"value": "2"},
                            {"value": "3"}
                        ]
                    },
                    "defaultValue": "3"
                }
                ...
            ]
        }
    ]
}
```

Configuration values inserted into the FreeMarker context honor the defined `datatype` value specified in the JSON file.

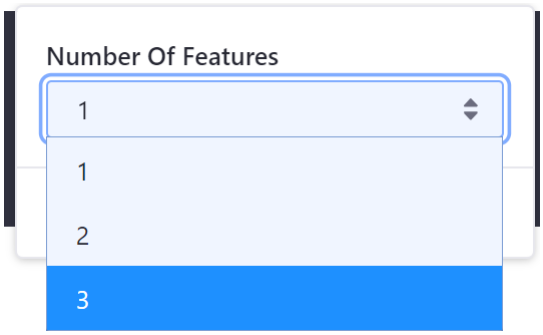

Figure 921.3: The `select` configuration is useful when an option choice is necessary.

921.4 Text Configuration

The following JSON configuration creates an input text field you can implement for cases where you must manually input a text option:

```
{
    "fieldSets": [
        {
            "fields": [
```

```
            {
                "name": "buttonText",
                "label": "Button Text",
                "description": "button-text",
                "type": "text",
                "typeOptions": {
                    "placeholder": "Placeholder"
                },
                "dataType": "string",
                "defaultValue": "Go Somewhere"
            }
            ...
        ]
    }
  ]
}
```

Configuration values inserted into the FreeMarker context honor the defined `datatype` value specified in the JSON file.

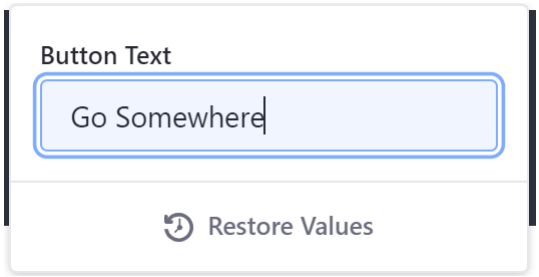

Figure 921.4: The `text` configuration is useful when an input text option is necessary.

CHAPTER 922

ESCAPING FRAGMENT CONFIGURATION TEXT VALUES

When you define text configuration and other options for a Fragment, Fragment developers can declare any text value they want. With this freedom comes risk; malicious code could be inserted into the text field, wreaking havoc for other users of the Fragment.

In this article, you'll learn how to escape Fragment text values so Fragment authors are protected from XSS attacks.

922.1 Escaping Values in HTML/FreeMarker

You must take special care when adding a text value in your Fragment's HTML. For example, if a user includes malicious code within `<script>` tags, it runs when the page is rendered.

To solve this problem, a utility is available in the FreeMarker context via the `htmlUtil` class.

For generic cases, an `escape` method is available:

```
<div class="fragment_38816">
    ${htmlUtil.escape(configuration.text)}"
</div>
```

This escapes your Fragment configuration's `text` field, preventing malicious code from affecting Fragment authors.

For more information on escaping methods, see the HtmlUtil class.

922.2 Escaping Values in JavaScript

When including JavaScript in your Fragment, you must be aware of potential attack vectors, like setting an attribute or appending HTML children. To prevent these attacks, you should use the `Liferay.Util.escapeHTML` function. You can call it from your Fragment's JavaScript like this:

```
function (fragmentElement, configuration) {
    const escapedValue = Liferay.Util.escapeHTML(configuration.text)
}
```

This escapes your Fragment configuration's text field, preventing malicious code in your Fragment's JavaScript code.

CHAPTER 923

ASSET DISPLAY PAGE TAGLIB

Once you have created an Asset Display Page associated with your Asset type, you can add the option to select an Asset Display Page in the form where you create the Asset. The `<liferay-asset:select-asset-display-page />` taglib renders a form field for selecting an Asset Display Page for your asset.

There are three options when selecting a display page:

- The default display page for the asset type if one has been configured.
- Any other selectable display page.
- None

If you select no default display page, a display page is not defined.

923.1 Display Page Attributes

When you use the display page selector taglib, you can define the following attributes:

`classNameId (long)` (required): a class name ID of the asset type to select an asset display page for.

`classPK (long)`: a primary key of the asset entry to select an asset display page for.

`classTypeId (long)`: a class type ID of the asset type to select an asset display page for.

`eventName (String)`: event name which fires when a user selects a display page using the item selector.

`groupId (long)` (required): the entity's group ID to select an asset display page for.

`showPortletLayouts (boolean)`: allow selection of pages that have Asset Publisher configured as a default Asset Publisher for the page.

The attribute `showPortletLayout` provides backwards compatibility for display pages created for Journal Articles in older versions. When `showPortletLayouts` is set to true, you can select any public or private pages with an Asset Publisher widget on it configured as the *Default Asset Publisher for the page*.

When submitting a form with the taglib, it populates the request with the following parameters:

`displayPageType (int)`: 1 = Default, 2 = Specific, 3 = None.

`assetDisplayPageId (long)`: ID of selected Asset Display Page.
`layoutUuid`: Layout UUID in case of a portlet page with default Asset Publisher.

CHAPTER 924

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.

924.1 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 extract the definitions for yourself,

1. Navigate to

   ```
   [Liferay Home]/osgi/marketplace
   ```

2. Using an archive manager, open the LPKG

   ```
   Liferay CE Forms and Workflow.lpkg
   ```

3. Open the JAR file named

   ```
   com.liferay.portal.workflow.kaleo.runtime.impl-[version].jar
   ```

4. In the JAR file, navigate to

   ```
   META-INF/definitions/
   ```

5. 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.

924.2 Schema

The XML structure of a workflow definition is defined in an XSD file:

```
liferay-worklow-definition-7_2_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_7.2.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:liferay.com:liferay-workflow_7.2.0
        http://www.liferay.com/dtd/liferay-workflow-definition_7_2_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.

924.3 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.

CHAPTER 925

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.

925.1 State Nodes

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>
```

925.2 Conditions

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).

925.3 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>
```

925.4 Task Nodes

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.

CHAPTER 926

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.

926.1 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.

926.2 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.

926.3 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.

CHAPTER 927

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.

927.1 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.

Description The `<description>` element contains the subject of the notification if the notification type is `email`. The syntax is determined by the template language you're using.

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.

CHAPTER 928

BREAKING CHANGES

This document has been updated and ported to Liferay Learn and is no longer maintained here.

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, 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.

928.1 Breaking Changes List

928.2 Removed Support for JSP Templates in Themes

- **Date:** 2018-Nov-14
- **JIRA Ticket:** LPS-87064

What changed?

Themes can no longer leverage JSP templates. Also, related logic has 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 JSP templates or is using the removed methods.

How should I update my code?

If you have a theme using JSP templates, consider migrating it to FreeMarker.

Why was this change made?

JSP is not a real template engine and is rarely used. FreeMarker is the recommended template engine moving forward.

The removal of JSP templates allows for an increased focus on existing and new template engines.

928.3 Lodash Is No Longer Included by Default

- **Date:** 2018-Nov-27
- **JIRA Ticket:** LPS-87677

What changed?

Previously, Lodash was included in every page by default and made available through the global `window._` and scoped `AUI._` variables. Lodash is no longer included by default and those variables are now undefined.

Who is affected?

This affects any developer who used the `AUI._` or `window._` variables in their custom scripts.

How should I update my code?

You should provide your own Lodash version for your custom developments to use following any of the possible strategies to add third party libraries.

As a temporary measure, you can bring back the old behavior by setting the *Enable Lodash* property in Liferay Portal's *Control Panel → Configuration → System Settings → Third Party → Lodash* to `true`.

Why was this change made?

This change was made to avoid bundling and serving additional library code on every page that was mostly unused and redundant.

928.4 Moved Two Staging Properties to OSGi Configuration

- **Date:** 2018-Dec-12
- **JIRA Ticket:** LPS-88018

What changed?

Two Staging properties have been moved from `portal.properties` to an OSGi configuration named `ExportImportServiceConfiguration.java` in the `export-import-service` module.

Who is affected?

This affects anyone using the following portal properties:

- `staging.delete.temp.lar.on.failure`
- `staging.delete.temp.lar.on.success`

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 → Infrastructure → Export/Import* and editing the settings there.

If you would like to include the new configuration in your application, follow the instructions for making applications configurable.

Why was this change made?

This change was made as part of the modularization efforts to ease portal configuration changes.

928.5 Remove Link Application URLs to Page Functionality

- **Date:** 2018-Dec-14
- **JIRA Ticket:** LPS-85948

What changed?

The *Link Portlet URLs to Page* option in the Look and Feel portlet was marked as deprecated in Liferay Portal 7.1, allowing the user to show and hide the option through a configuration property. In Liferay Portal 7.2, this has been removed and can no longer be configured.

Who is affected?

This affects administrators who used the option in the UI and developers who leveraged the option in the portlet.

How should I update my code?

You should update any portlets leveraging this feature, since any preconfigured reference to the property is ignored in the portal.

Why was this change made?

A limited number of portlets use this property; there are better ways to achieve the same results.

928.6 Moved TermsOfUseContentProvider out of kernel.util

- **Date:** 2019-Jan-07
- **JIRA Ticket:** LPS-88869

What changed?

The `TermsOfUseContentProvider` interface's package changed:
`com.liferay.portal.kernel.util` → `com.liferay.portal.kernel.term.of.use`
The `TermsOfUseContentProviderRegistryUtil` class' name and package changed:
`TermsOfUseContentProviderRegistryUtil` → `TermsOfUseContentProviderUtil`
and `com.liferay.portal.kernel.util` → `com.liferay.portal.internal.terms.of.use`
The logic of getting `TermsOfUseContentProvider` was also changed. Instead of always returning the first service registered, which is random and depends on the order of registered services, the `TermsOfUseContentProvider` service is tracked and updated with `com.liferay.portal.kernel.util.ServiceProxyFactory`. As a result, the `TermsOfUseContentProvider` now respects service ranking.

Who is affected?

This affects anyone who used `com.liferay.portal.kernel.util.TermsOfUseContentProviderRegistryUtil` to lookup the `com.liferay.portal.kernel.util.TermsOfUseContentProvider` service.

How should I update my code?

If `com.liferay.portal.kernel.util.TermsOfUseContentProvider` is used, update the import package name. If there is any usage in `portal-web`, update `com.liferay.portal.kernel.util.TermsOfUseContentProviderRegistr` to `com.liferay.portal.kernel.term.of.use.TermsOfUseContentProviderUtil`. Remove usages of `com.liferay.portal.kernel.util.TermsOfUseContentProviderRegistryUtil` in modules and use the `@Reference` annotation to fetch the `com.liferay.portal.kernel.term.of.use.TermsOfUseContentProvider` service instead.

Why was this change made?

This is one of several steps to clean up kernel provider interfaces to reduce the chance of package version lock down.

928.7 Removed HibernateConfigurationConverter and Converter

- **Date:** 2019-Jan-07
- **JIRA Ticket:** LPS-88870

What changed?

The interface `com.liferay.portal.kernel.util.Converter` and its implementation `com.liferay.portal.spring.hiberna` were removed.

Who is affected?

This removes the support of generating customized `portlet-hbm.xml` files implemented by `HibernateConfigurationConverter`. Refer to LPS-5363 for more information.

How should I update my code?

You should remove usages of `HibernateConfigurationConverter`. Make sure the generated `portlet-hbm.xml` is accurate.

Why was this change made?

This is one of several steps to clean up kernel provider interfaces to reduce the chance of package version lock down.

928.8 Switched to Use JDK Function and Supplier

- **Date:** 2019-Jan-08
- **JIRA Ticket:** LPS-88911

What changed?

The `Function` and `Supplier` interfaces in package `com.liferay.portal.kernel.util` were removed. Their usages were replaced with `java.util.function.Function` and `java.util.function.Supplier`.

Who is affected?

This affects anyone who implemented the `Function` and `Supplier` interfaces in package `com.liferay.portal.kernel.util`.

How should I update my code?

You should replace usages of `com.liferay.portal.kernel.util.Function` and `com.liferay.portal.kernel.util.Supplie` with `java.util.function.Function` and `java.util.function.Supplier`, respectively.

Why was this change made?

This is one of several steps to clean up kernel provider interfaces to reduce the chance of package version lock down.

928.9 Deprecated com.liferay.portal.service.InvokableService Interface

- **Date:** 2019-Jan-08
- **JIRA Ticket:** LPS-88912

What changed?

The `InvokableService` and `InvokableLocalService` interfaces in package `com.liferay.portal.kernel.service` were removed.

Who is affected?

This affects anyone who used `InvokableService` and `InvokableLocalService` in package `com.liferay.portal.kernel.ser`

How should I update my code?

You should remove usages of `InvokableService` and `InvokableLocalService`. Make sure to use the latest version of Service Builder to generate implementations for services in case there is any compile errors after removal.

Why was this change made?

This is one of several steps to clean up kernel provider interfaces to reduce the chance of package version lock down.

928.10 Dropped Support of ServiceLoaderCondition

- **Date:** 2019-Jan-08
- **JIRA Ticket:** LPS-88913

What changed?

The interface `ServiceLoaderCondition` and its implementation `DefaultServiceLoaderCondition` in package `com.liferay.portal.kernel.util` were removed.

Who is affected?

This affects anyone using `ServiceLoaderCondition` and `DefaultServiceLoaderCondition`.

How should I update my code?

You should remove usages of `ServiceLoaderCondition`. Update usages of `load` methods in `com.liferay.portal.kernel.util.ServiceLoader` according to the updated method signatures.

Why was this change made?

This is one of several steps to clean up kernel provider interfaces to reduce the chance of package version lock down.

928.11 Switched to Use JDK Predicate

- **Date:** 2019-Jan-14
- **JIRA Ticket:** LPS-89139

What changed?

The interface `com.liferay.portal.kernel.util.PredicateFilter` was removed and replaced with `java.util.function.Predicate`. As a result, the following implementations were removed:

- `com.liferay.portal.kernel.util.AggregatePredicateFilter`
- `com.liferay.portal.kernel.util.PrefixPredicateFilter`
- `com.liferay.portal.kernel.portlet.JavaScriptPortletResourcePredicateFilter`
- `com.liferay.dynamic.data.mapping.form.values.query.internal.model.DDMFormFieldValuePredicateFilter`

The `com.liferay.portal.kernel.util.ArrayUtil_IW` class was regenerated.

Who is affected?

This affects anyone who used `PredicateFilter`, `AggregatePredicateFilter`, `PrefixPredicateFilter`, `JavaScriptPortletResourcePredicateFilter`, and `DDMFormFieldValuePredicateFilter`.

How should I update my code?

You should replace usages of `com.liferay.portal.kernel.util.PredicateFilter` with `java.util.function.Predicate`. Additionally, remove usages of `AggregatePredicateFilter`, `PrefixPredicateFilter`, `JavaScriptPortletResourcePredicat` and `DDMFormFieldValuePredicateFilter`.

Why was this change made?

This is one of several steps to clean up kernel provider interfaces to reduce the chance of package version lock down.

928.12 Removed Unsafe Functional Interfaces in Package com.liferay.portal.kernel.util

- **Date:** 2019-Jan-15
- **JIRA Ticket:** LPS-89223

What changed?

The `com.liferay.portal.osgi.util.test.OSGiServiceUtil` class was removed. Also, the following interfaces were removed from the `com.liferay.portal.kernel.util` package:

- `UnsafeConsumer`
- `UnsafeFunction`
- `UnsafeRunnable`

Who is affected?

This affects anyone using the class/interfaces mentioned above.

How should I update my code?

The `com.liferay.portal.osgi.util.test.OSGiServiceUtil` class has been deprecated since Liferay Portal 7.1. If usages for this class still exist, replace it with its direct replacement: `com.liferay.osgi.util.service.OSGiServiceUtil`. Replace usages of `UnsafeConsumer`, `UnsafeFunction` and `UnsafeRunnable` with their corresponding interfaces in package `com.liferay.petra.function`.

Why was this change made?

This is one of several steps to clean up kernel provider interfaces to reduce the chance of package version lock down.

928.13 Deprecated NTLM in Portal Distribution

- **Date:** 2019-Jan-21
- **JIRA Ticket:** LPS-88300

What changed?

NTLM modules have been moved from the `portal-security-sso` project to a new project named `portal-security-sso-ntlm`. This new project is deprecated and available to download from Liferay Marketplace.

Who is affected?

This affects anyone using NTLM as an authentication system.

How should I update my code?

If you want to continue using NTLM as an authentication system, you must download the corresponding modules from Liferay Marketplace. Alternatively, you can migrate to Kerberos (recommended), which requires no changes and is compatible with Liferay Portal 7.0+.

Why was this change made?

This change was made to avoid using an old proprietary solution (NTLM). Kerberos is now recommended, which is a standard protocol and a more secure method of authentication compared to NTLM.

928.14 Deprecated OpenID in Portal Distribution

- **Date:** 2019-Jan-21
- **JIRA Ticket:** LPS-88906

What changed?

OpenID modules have been moved to a new project named `portal-security-sso-openid`. This new project is deprecated and available to download from Liferay Marketplace.

Who is affected?

This affects anyone using OpenID as an authentication system.

How should I update my code?

If you want to continue using OpenID as an authentication system, you must download the corresponding module from Liferay Marketplace. Alternatively, you should migrate to OpenID Connect, available on Liferay Portal Distribution.

Why was this change made?

This change was made to avoid using a deprecated solution (OpenID). OpenID Connect is now recommended, which is a more secure method of authentication since it runs on top of OAuth.

928.15 Deprecated Google SSO in Portal Distribution

- **Date:** 2019-Jan-21
- **JIRA Ticket:** LPS-88905

What changed?

Google SSO modules have been moved from the `portal-security-sso` project to a new project named `portal-security-sso-google`. This new project is deprecated and available to download from Liferay Marketplace.

Who is affected?

This affects anyone using Google SSO as an authentication system.

How should I update my code?

If you want to continue using Google SSO as an authentication system, you must download the corresponding module from Liferay Marketplace. Alternatively, you can use OpenID Connect.

Why was this change made?

This change was made to avoid using an old solution for authentication (Google SSO). OpenID Connect is the recommended specification to use Google implementation for authentication.

928.16 Updated AlloyEditor v2.0 Includes New Major Version of React

- **Date:** 2019-Feb-04
- **JIRA Ticket:** LPS-90079

What changed?

AlloyEditor was upgraded to version 2.0.0, which includes a major upgrade from React v15 to v16.

The `React.createClass` was deprecated in React v15.5.0 (April 2017) and removed in React v16.0.0 (September 2017). All the buttons bundled with AlloyEditor have been updated to use the ES6 class syntax instead of `React.createClass`.

Who is affected?

This affects anyone who built their own buttons using `React.createClass`. The `createClass` function is no longer available, and attempts to access it at runtime will trigger an error.

How should I update my code?

You should update your code in one of two ways:

- Port custom buttons from the `React.createClass` API to use the ES6 `class` API, as described in the React documentation. For example, see the changes made in moving to an ES6 class-based button from the previous `createClass`-based implementation.
- Provide a compatibility adapter. The create-react-class package (described here) can be injected into the page to restore the `createClass` API.

Why was this change made?

This change was made to use a newer major version of React, which brings performance and compatibility improvements and reduces the bundle size by removing deprecated APIs.

928.17 Deprecated dl.tabs.visible property

- **Date:** 2019-Apr-10
- **JIRA Ticket:** LPS-93948

What changed?

The `dl.tabs.visible` property let users toggle the visibility of a Documents and Media widget's navigation tabs when placed on a widget page. This configuration option has been removed, so the navigation tab will never appear on widget pages.

Who is affected?

This affects anyone who set the `dl.tabs.visible` property to `true`.

How should I update my code?

No code changes are necessary.

Why was this change made?

Documents & Media has been reviewed from a UX perspective, and removing the navigation tabs in widget pages was part of a UI clean up process.

928.18 Move the User Menu out of the Product Menu

- **Date:** 2019-Apr-19
- **JIRA Ticket:** LPS-87868

What changed?

The User Menu was removed from the Product Menu, and the user menu entries were moved to the new Personal Menu, a dropdown menu triggered by the user avatar.

Who is affected?

This affects anyone who has customized the User Menu section of the Product Menu.

How should I update my code?

If you would like to keep your custom user menu entries and have them available in the Personal Menu, you need to implement the `PersonalMenuEntry` interface. All panel apps registered with the `PanelCategoryKeys.USER`, `PanelCategoryKeys.USER_MY_ACCOUNT`, and `PanelCategoryKeys.USER_SIGN_OUT` panel category keys should be converted to `PersonalMenuEntry`.

Why was this change made?

Product navigation has been reviewed from a UX perspective, and removing the User Menu from the Product Menu and splitting the menu to its own provides a better user experience.

928.19 Removed Hong Kong and Macau from the List of Countries

- **Date:** 2019-Apr-26
- **JIRA Ticket:** LPS-82203

What changed?

Hong Kong and Macau have been removed from the list of countries and listed as regions of China as Xianggang (region code: CN-91) and Aomen (region code: CN-92), respectively.

Who is affected?

This affects anyone who used Hong Kong or Macau in their addresses.

How should I update my code?

No code changes are necessary. However, if you have hardcoded the `countryId` of Hong Kong and Macau in your code, they should be updated to China's `countryId`. References to Hong Kong and Macau should be done with their corresponding `regionId`.

Why was this change made?

After the handover of Hong Kong in 1997 and of Macau in 1999, Hong Kong and Macau are now the special administrative regions of China.

928.20 JGroups Was Upgraded From 3.6.16 to 4.1.1

- **Date:** 2019-Aug-15
- **JIRA Ticket:** LPS-97897

What changed?

JGroups was upgraded from version 3.6.16 to 4.1.1.

Who is affected?

This affects anyone using Cluster Link.

How should I update my code?

The `cluster.link.channel.properties.*` property in `portal.properties` no longer accepts a connection string as a value; it now requires a file path to a configuration XML file. Some of the protocol properties from 3.6.16 are removed and no longer parsed by 4.1.1; you should update the protocol properties accordingly.

Why was this change made?

This upgrade was made to fix a security issue.

928.21 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.

928.22 Auto Tagging Must Be Reconfigured Manually

- **Date: 2019-Oct-2**
- **JIRA Ticket:** LPS-97123

What changed?

Auto Tagging configurations were renamed and reorganized. There's no longer an automatic upgrade process, so you must reconfigure Auto Tagging manually.

Who is affected?

This affects DXP 7.2 installations that are upgraded to SP1 and have Auto Tagging configured and enabled.

How should I update my code?

You must reconfigure Auto Tagging through System Settings (please see the official documentation for details). Any code referencing the old configuration interfaces must be updated to use the new ones.

Why was this change made?

This change unifies the previously split configuration interfaces, improving the user experience.

928.23 Blogs Image Properties Were Moved to System Settings

- **Date: 2019-Oct-2**
- **JIRA Ticket:** LPS-95298

What changed?

Blogs image configuration was moved from `portal.properties` to System Settings. There's no automatic upgrade process, so custom Blogs image properties must be reconfigured manually.

Who is affected?

This affects DXP 7.2 installations that are upgraded to SP1 and have custom values for the `blogs.image.max.size` and `blogs.image.extensions` properties.

How should I update my code?

If you would like to keep your custom Blogs image property values, you must reconfigure them through the System Settings under *Configuration → Blogs → File Uploads*. Any code referencing the old properties must be updated to use the new configuration interfaces.

Why was this change made?

This change was made so Blogs image properties can be configured without a restart.

928.24 Removed Cache Bootstrap Feature

- **Date:** 2020-Jan-8
- **JIRA Ticket:** LPS-96563

What changed?

The cache bootstrap feature has been removed. These properties can no longer be used to enable/configure cache bootstrap:

`ehcache.bootstrap.cache.loader.enabled`, `ehcache.bootstrap.cache.loader.properties.default`, `ehcache.bootstrap.cache.loader.properties.${specific.cache.name}`.

Who is affected?

This affects anyone using the properties listed above.

How should I update my code?

There's no direct replacement for the removed feature. If you have code that depends on it, you must implement it yourself.

Why was this change made?

This change was made to avoid security issues.

CHAPTER 929

CDI Portlet Predefined Beans

Liferay DXP provides injectable portlet artifacts for CDI called Portlet Predefined Beans, as specified by JSR 362. There are two types of predefined beans:

- Portlet Request Scoped Beans (`@PortletRequestScoped`)
- Dependent Scoped Beans (`@Dependent` scoped)

The table below describes these attributes for each bean:
Artifact: The bean's type.
Bean EL Name: Expression Language (EL) name for accessing the bean in a JSP or JSF page.
Qualifier: Annotation applied to the bean for defining and selecting a bean implementation.
Valid during (phase): The portlet phases in which the bean is valid.

929.1 Portlet Request Scoped Beans

These beans have the `@PortletRequestScoped` annotation. Here are their artifact types, bean EL names, and annotation qualifiers, along with their valid portlet phases.

Table 1: Portlet Request Scoped Beans[1]

Artifact	Bean EL Name	Qualifier	Valid during
`PortletConfig`	`portletConfig`	-	all
`PortletRequest`	`portletRequest`	-	all
`PortletResponse`	`portletResponse`	-	all
`ActionRequest`	`actionRequest`	-	action
`ActionResponse`	`actionResponse`	-	action
`HeaderRequest`	`headerRequest`	-	header
`HeaderResponse`	`headerResponse`	-	header
`RenderRequest`	`renderRequest`	-	render

[1] Martin Scott Nicklous, Java™ Portlet Specification 3.0, page 122.

Artifact	Bean EL Name	Qualifier	Valid during
`RenderResponse`	`renderResponse`	-	render
`EventRequest`	`eventRequest`	-	event
`EventResponse`	`eventResponse`	-	event
`ResourceRequest`	`resourceRequest`	-	resource
`ResourceResponse`	`resourceResponse`	-	resource
`StateAwareResponse`	`stateAwareResponse`	-	action, event
`MimeResponse`	`mimeResponse`	-	header, render, resource
`ClientDataRequest`	`clientDataRequest`	-	action, resource
`RenderParameters`	`renderParams`	-	all
`MutableRenderParameters`	`mutableRenderParams`	-	action, event
`ActionParameters`	`actionParams`	-	action
`ResourceParameters`	`resourceParams`	-	resource
`PortletContext`	`portletContext`	-	all
`PortletMode`	`portletMode`	-	all
`WindowState`	`windowState`	-	all
`PortletPreferences`	`portletPreferences`	-	all
`Cookies(List<Cookie>)`	`cookies`	-	all
`PortletSession`	`portletSession`	-	all
`Locales(List<Locale>)`	`locales`	-	all

929.2 Dependent Scoped Beans

These beans use the `@Dependent` scope. They're of type `java.lang.String`, which is `final`. This disqualifies them from being proxied. To prevent using dependent scoped beans in a scope broader than their original scope, you should only inject them into `@PortletRequestScoped` beans.

Table 2: Dependent Scoped Beans[2]

Artifact	Bean EL Name	Qualifier	Valid during
`Namespace` (String)	`namespace`	`@Namespace`	all
`ContextPath` (String)	`contextPath`	`@ContextPath`	all
`WindowID` (String)	`windowId`	`@WindowId`	all
`Portlet name` (String)	`portletName`	`@PortletName`	all

929.3 Related Topics

CDI Dependency Injection

[2]Martin Scott Nicklous, Java™ Portlet Specification 3.0, page 123.

CHAPTER 930

ITEM SELECTOR CRITERION AND RETURN TYPES

Liferay DXP contains Item Selector criterion (`ItemSelectorCriterion`) and return type (`ItemSelectorReturnType`) classes that developers can use in Item Selectors. The following sections in this document list the available classes:

- Criterion Classes
- Return Type Classes

If there isn't a criterion or return type for your needs, you can create your own by following the instructions in Creating Custom Criterion and Return Types. For more information on Item Selectors in general, including definitions of criterion and return types, see the Item Selector introduction.

930.1 Item Selector Criterion Classes

Assets:
AssetDisplayPageSelectorCriterion: Asset display page.
Blogs:
BlogsItemSelectorCriterion: Blogs item.
Page Fragments:
FragmentItemSelectorCriterion: Page fragment.
Item Selector:
AudioItemSelectorCriterion: Audio file.
FileItemSelectorCriterion: Document Library file.
ImageItemSelectorCriterion: Image file.
UploadItemSelectorCriterion: Uploadable file.
URLItemSelectorCriterion: URL.
VideoItemSelectorCriterion: Video file.
Journal (Web Content):
JournalItemSelectorCriterion: Web content article.
Knowledge Base:
KBAttachmentItemSelectorCriterion: Knowledge base attachment.

Layout:
LayoutItemSelectorCriterion: Page layout.
Organizations:
OrganizationItemSelectorCriterion: Organization.
Roles:
RoleItemSelectorCriterion: Role.
Site Navigation:
SiteNavigationMenuItemItemSelectorCriterion: Site navigation menu item.
SiteNavigationMenuItemSelectorCriterion: Site navigation menu.
Sites:
SiteItemSelectorCriterion: Site.
User Groups Admin:
UserGroupItemSelectorCriterion: User group.
Users Admin:
UserItemSelectorCriterion: User.
Wiki:
WikiAttachmentItemSelectorCriterion: Wiki attachment.
WikiPageItemSelectorCriterion: Wiki page.

930.2 Item Selector Return Type Classes

Adaptive Media:
AMImageFileEntryItemSelectorReturnType: Adaptive Media image file.
AMImageURLItemSelectorReturnType: Adaptive Media image URL.
Item Selector:
Base64ItemSelectorReturnType: The entity's Base64 encoding as a `String`.
DownloadURLItemSelectorReturnType: The entity's download URL as a `String`.
FileEntryItemSelectorReturnType: File entry information as a JSON object.
URLItemSelectorReturnType: The entity's URL as a `String`.
UUIDItemSelectorReturnType: The entity's universally unique identifier (UUID) as a `String`.
Site:
SiteItemSelectorReturnType: The Site's information as a JSON object.
Wiki:
WikiPageTitleItemSelectorReturnType: The wiki page's title.
WikiPageURLItemSelectorReturnType: The wiki page's URL.

CHAPTER 931

JAVA APIS

Here you'll find Javadoc for Liferay DXP and Liferay DXP apps. Note that each link to the Javadoc listed here opens in a new window.

For help finding module attributes and configuring dependencies, see Configuring Dependencies.

931.1 7.0 Java APIs

This table contains links to the Javadoc for 7.0 API modules. The root location for these modules' Javadoc is here.

Core

com.liferay.portal.kernel (portal-kernel): for developing applications on Liferay DXP

com.liferay.util.bridges (util-bridges): for using various non-proprietary computing languages, frameworks, and utilities on Liferay DXP

com.liferay.util.java (util-java): for using various Java-related frameworks and utilities on Liferay DXP

com.liferay.util.slf4j (util-slf4j): for using the Simple Logging Facade for Java (SLF4J)

com.liferay.portal.impl (portal-impl): 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

931.2 Liferay DXP App Java APIs

The tables in this section link to the API modules for apps in these categories:

- Collaboration
- Forms and Workflow
- Foundation
- Web Experience

Note that the root location for these modules' Javadoc is [https://docs.liferay.com/dxp/apps](## Collaboration

Announcements
com.liferay.announcements.api
Blogs
com.liferay.blogs.api
com.liferay.blogs.item.selector.api
com.liferay.blogs.recent.bloggers.api
Comment
com.liferay.comment.api
Document Library
com.liferay.document.library.api
com.liferay.document.library.content.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
Flags
com.liferay.flags.api
Invitation
com.liferay.invitation.invite.members.api
Item Selector
com.liferay.item.selector.api
com.liferay.item.selector.criteria.api
Mentions
com.liferay.mentions.api
Message Boards
com.liferay.message.boards.api
Ratings
com.liferay.ratings.api
Reading Time
com.liferay.reading.time.api
Social
com.liferay.social.activities.api
com.liferay.social.activity.api
com.liferay.social.bookmarks.api
com.liferay.social.user.statistics.api
Subscription
com.liferay.subscription.api
Upload
com.liferay.upload.api
Wiki
com.liferay.wiki.api

931.3 Forms and Workflow

Calendar

com.liferay.calendar.api
Dynamic Data Lists
com.liferay.dynamic.data.lists.api
Dynamic Data Mapping
com.liferay.dynamic.data.mapping.api
Polls
com.liferay.polls.api
Portal Reports Engine
com.liferay.portal.reports.engine.api
Portal Rules Engine
com.liferay.portal.rules.engine.api
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

931.4 Foundation

Captcha
com.liferay.captcha.api
Configuration Admin
com.liferay.configuration.admin.api
Contacts
com.liferay.contacts.api
Friendly URL
com.liferay.friendly.url.api
Frontend Editor
com.liferay.frontend.editor.api
Frontend Image Editor
com.liferay.frontend.image.editor.api
Frontend JS
com.liferay.frontend.js.loader.modules.extender.api
Map
com.liferay.map.api
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.api
Portal Background Task
com.liferay.portal.background.task.api
Portal Cache
com.liferay.portal.cache.api

Portal Configuration
com.liferay.portal.configuration.upgrade.api
Portal Instances
com.liferay.portal.instances.api
Portal Lock
com.liferay.portal.lock.api
Portal Remote
com.liferay.portal.remote.soap.extender.api
Portal Scripting
com.liferay.portal.scripting.api
Portal Search
com.liferay.portal.search.api
com.liferay.portal.search.engine.adapter.api
com.liferay.portal.search.web.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.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 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 SSO Google
com.liferay.portal.security.sso.google.api
Portal Settings
com.liferay.portal.settings.api
Portal Template
com.liferay.portal.template.soy.api
Portal URL Builder
com.liferay.portal.url.builder.api
Portal
com.liferay.portal.custom.jsp.bag.api
com.liferay.portal.dao.orm.custom.sql.api
com.liferay.portal.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

Roles
com.liferay.roles.admin.api
com.liferay.roles.item.selector.api
Text Localizer
com.liferay.text.localizer.address.api
User-associated Data
com.liferay.user.associated.data.api
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.item.selector.api
XStream
com.liferay.xstream.configurator.api

931.5 Web Experience

Application List
com.liferay.application.list.api
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.tags.api
com.liferay.asset.tags.navigation.api
Export Import
com.liferay.exportimport.api
com.liferay.exportimport.changeset.api
Fragment
com.liferay.fragment.api
com.liferay.fragment.item.selector.api
HTML Preview
com.liferay.html.preview.api
Journal
com.liferay.journal.api
com.liferay.journal.content.asset.addon.entry.api
com.liferay.journal.item.selector.api
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
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
RSS
com.liferay.rss.api
Site Navigation
com.liferay.site.navigation.api
com.liferay.site.navigation.admin.api
com.liferay.site.navigation.item.selector.api
com.liferay.site.navigation.language.api
Site
com.liferay.site.api
com.liferay.site.item.selector.api
Staging
com.liferay.staging.api
Trash
com.liferay.trash.api

931.6 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 Lexicon.

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: AlloyUI and all of its JavaScript APIs are available within portlets, templates, and themes.

931.7 Descriptor Definitions

DTDs: Describes the XML files used in configuring Liferay DXP apps, 7.0 plugins, and Liferay DXP 7.2.

CHAPTER 932

MEANINGFUL SCHEMA VERSIONING

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 schema version defines the status of the database schema and its data belonging to that module or Core in a certain moment. The concept of schema versioning is different from bundle versioning. The biggest concern in versioning a data schema is **backward-compatibility** between the new schema and the code that operates on the data.

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 previous version of the code.

MINOR: Contains schema/data changes compatible with the previous version of the code. The changes are required for the new version of the code to work (the application will fail without applying the schema/data changes)

MICRO: Contains schema/data changes that are compatible with the previous version of the code. The changes are optional.

If you're not sure what kind of schema version change represents your upgrade step, ask yourself these questions:

1. Will the previous code version (previous FP, SP, or GA) work with these schema/data changes?

    - If not, it is a major change.
    - If yes, continue.

2. Are the schema/data changes required for the application to work? (Obviously, all changes are intended to improve the application but in some cases the application is fully functional without them)

- If yes, it is a minor change.
- If not, it is a micro change.

Next are some concrete examples of micro, minor, and major changes.

932.1 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.

932.2 Minor change examples

Here are common minor changes:

- Adding a new DB field.
- Adding a new DB table.

Important: The changes above are major if they require modifying current existing data or extract information to populate the new field or table. In such cases, the data can become incorrect if you rolled back to the previous code version and then, after some time, installed the new code again.

932.3 Major change examples

Here are common major changes:

- Making data modifications that are not backward compatible.
- 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.
- Adding a new DB field or table that requires modifying current existing data or extracts information to populate the new field or table.

Now you can ascribe meaningful versions to your module's data schemas.

CHAPTER 933

PORTLET 3.0 API OPT IN

A portlet must specify version 3.0 to "opt in" to the Portlet 3.0 API. The 3.0 Portlet API version can be specified in the following ways.

933.1 Standard Portlet `@PortletApplication` Annotation

Standard portlets need only specify the `@PortletApplication` annotation.

```
@PortletApplication(version="3.0") // 3.0 is the default for this annotation attribute
@PortletConfiguration(portletName="myPortlet")
public class MyPortlet {
    ...
}
```

933.2 Liferay MVC Portlet `@Component` Annotation

Declarative Services portlets, such as `MVCPortlet`, can specify version 3.0 in their `@Component` annotation.

```
@Component(properties="javax.portlet.version=3.0", service=javax.portlet.Portlet.class)
public class MyDeclarativeServicesPortlet {
    ...
}
```

933.3 `portlet.xml` Descriptor

All portlets can specify version 3.0 in their `portlet.xml` descriptor.

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

<portlet-app xmlns="http://xmlns.jcp.org/xml/ns/portlet"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/portlet http://xmlns.jcp.org/xml/ns/portlet/portlet-app_3_0.xsd"
             version="3.0">
    ...
</portlet-app>
```

CHAPTER 934

PORTLET DESCRIPTOR TO OSGI SERVICE PROPERTY MAP

This document has been updated and ported to Liferay Learn and is no longer maintained here.

This article maps portlet XML descriptor values to OSGi service properties for publishing OSGi Portlets.

OSGi service definitions can use properties. OSGi service properties centralize and simplify portlet configuration. They are typically represented as key-value pairs or, more generally, as a Map-like object.

Portlet spec property keys are prefixed by

```
javax.portlet.
```

Liferay property keys are prefixed by

```
com.liferay.portlet.
```

The mappings essentially flatten what is found in the XML descriptor. The property names resemble the original descriptor names.

This article covers these descriptor mappings:

- Portlet descriptor mappings
- Liferay descriptor mappings
    - From `liferay-display.xml`
    - From `liferay-portlet.xml`

The standard portlet descriptor mappings are first.

934.1 Portlet Descriptor Mappings

Note: XPath notation derived from the **Portlet XSD** 4 is used in this document for simplicity.

portlet.xml XPath | OSGi Portlet Service Property| /portlet-app/container-runtime-option|not supported| /portlet-app/custom-portlet-mode|not supported| /portlet-app/custom-window-state|not supported| /portlet-app/default-namespace|javax.portlet.default-namespace=<String>| /portlet-app/event-definition|javax.portlet.event-definition=<QNameLocalPart>;<QNameURI>[;<PayloadType>][,<AliasQNameL
2| /portlet-app/filter/portlet-app/filter/init-param/name/portlet-app/filter-mapping|3javax.portlet.init-param.<name>=<value> 3, 93| /portlet-app/public-render-parameter|not supported| /portlet-app/resource-bundle|not supported| /portlet-app/security-constraint|not supported| /portlet-app/user-attribute|not supported| /portlet-app/version|javax.portlet.version=<value>| /portlet-app/portlet/async-supported|javax.portlet.async-supported=<boolean>| /portlet-app/portlet/cache-scope|not supported| /portlet-app/portlet/container-runtime-option|javax.portlet.container-runtime-option.<name>=<value> 2| /portlet-app/portlet/dependency|javax.portlet.dependency=<name>;<scope>;<versi
2, 6| /portlet-app/portlet/description|javax.portlet.description=<String>| /portlet-app/portlet/display-name|javax.portlet.display-name=<String>| /portlet-app/portlet/expiration-cache|javax.portlet.expiration-cache=<int>| /portlet-app/portlet/init-param/name|javax.portlet.init-param.<name>=<value>| /portlet-app/portlet/listener|javax.portlet.listener=<listener-class>;<ordinal> 2,8| /portlet-app/portlet/multipart-config/file-size-threshold|javax.portlet.multipart.file-size-threshold=<Integer>| /portlet-app/portlet/multipart-config/location|javax.portlet.multipart.location=<String>| /portlet-app/portlet/multipart-config/max-file-size|javax.portlet.multipart.max-file-size=<Long>| /portlet-app/portlet/multipart-config/max-request-size|javax.portlet.multipart.max-request-size=<Long>| /portlet-app/portlet/portlet-class|1| /portlet-app/portlet/portlet-info/keywords|javax.portlet.info
/portlet-app/portlet/portlet-info/short-title|javax.portlet.info.short-title=<String>| /portlet-app/portlet/portlet-info/title|javax.portlet.info.title=<String>| /portlet-app/portlet/portlet-name 10|javax.portlet.name=<String> 10| /portlet-app/portlet/portlet-preferences|javax.portlet.preferences=<Stri
/portlet-app/portlet/portlet-preferences/preferences-validator|javax.portlet.preferences-validator=<String> 1| /portlet-app/portlet/resource-bundle|javax.portlet.resource-bundle=<String>| /portlet-app/portlet/security-role-ref|javax.portlet.security-role-ref=<String>[,<String>]2| /portlet-app/portlet/supported-locale|javax.portlet.supported-locale=<String> 2| /portlet-app/portlet/supported-processing-event|javax.portlet.supported-processing-event=<QNameLocalPart>
OR javax.portlet.supported-processing-event=<QNameLocalPart>;<QNameURI> 2| /portlet-app/portlet/supported-public-render-parameter|javax.portlet.supported-public-render-parameter=<String>2| /portlet-app/portlet/supported-publishing-event|javax.portlet.supported-publishing-event=<QNameLocalPart>
OR javax.portlet.supported-publishing-event=<QNameLocalPart>;<QNameURI> 2| /portlet-app/portlet/supports/mime-type|javax.portlet.mime-type=<mime-type>| /portlet-app/portlet/supports/portlet-mode|javax.portlet.portlet-mode=<mime-type>;<portlet-mode>[,<portlet-mode>]*| /portlet-app/portlet/supports/window-state|javax.portlet.window-state=<mime-type>;<window-state>[,<window-state>]*|

934.2 Liferay Descriptor Mappings

934.3 Liferay Display

liferay-display.xml XPath | OSGi Portlet Service Property| /display/category[@name]|com.liferay.portlet.display
category=<value>|

934.4 Liferay Portlet

Note: XPath notation derived from **Liferay Portlet** 5 is used in this document for simplicity.

liferay-portlet.xml XPath | OSGi Liferay Portlet Service Property| /liferay-portlet-
app/portlet/action-timeout|com.liferay.portlet.action-timeout=<int>| /liferay-portlet-app/portlet/action-
url-redirect|com.liferay.portlet.action-url-redirect=<boolean>| /liferay-portlet-app/portlet/active|com.liferay
/liferay-portlet-app/portlet/add-default-resource|com.liferay.portlet.add-default-resource=<boolean>|
/liferay-portlet-app/portlet/ajaxable|com.liferay.portlet.ajaxable=<boolean>| /liferay-portlet-
app/portlet/asset-renderer-factory|3| /liferay-portlet-app/portlet/atom-collection-adapter|3|
/liferay-portlet-app/portlet/autopropagated-parameters|com.liferay.portlet.autopropagated-
parameters=<String>2| /liferay-portlet-app/portlet/configuration-action-class|3| /liferay-
portlet-app/portlet/configuration-path|com.liferay.portlet.configuration-path=<String>|
/liferay-portlet-app/portlet/control-panel-entry-category|com.liferay.portlet.control-panel-
entry-category=<String>| /liferay-portlet-app/portlet/control-panel-entry-class|3| /liferay-
portlet-app/portlet/control-panel-entry-weight|com.liferay.portlet.control-panel-entry-
weight=<double>| /liferay-portlet-app/portlet/css-class-wrapper|com.liferay.portlet.css-class-
wrapper=<String>| /liferay-portlet-app/portlet/custom-attributes-display|3| /liferay-portlet-
app/portlet/ddm-display|3| /liferay-portlet-app/portlet/facebook-integration|com.liferay.portlet.facebook-
integration=<String>| /liferay-portlet-app/portlet/footer-portal-css|com.liferay.portlet.footer-
portal-css=<String>2| /liferay-portlet-app/portlet/footer-portal-javascript|com.liferay.portlet.footer-
portal-javascript=<String>2| /liferay-portlet-app/portlet/footer-portlet-css|com.liferay.portlet.footer-
portlet-css=<String>2| /liferay-portlet-app/portlet/footer-portlet-javascript|com.liferay.portlet.footer-
portlet-javascript=<String>2| /liferay-portlet-app/portlet/friendly-url-mapper-class|3|
/liferay-portlet-app/portlet/friendly-url-mapping|com.liferay.portlet.friendly-url-mapping=<String>|
/liferay-portlet-app/portlet/friendly-url-routes|com.liferay.portlet.friendly-url-routes=<String>|
/liferay-portlet-app/portlet/header-portal-css|com.liferay.portlet.header-portal-css=<String>2|
/liferay-portlet-app/portlet/header-portal-javascript|com.liferay.portlet.header-portal-
javascript=<String>2| /liferay-portlet-app/portlet/header-portlet-css|com.liferay.portlet.header-
portlet-css=<String>2| /liferay-portlet-app/portlet/header-portlet-javascript|com.liferay.portlet.header-
portlet-javascript=<String>2| /liferay-portlet-app/portlet/header-request-attribute-prefix|com.liferay.portlet
request-attribute-prefix=<String> 7| /liferay-portlet-app/portlet/header-timeout|header-
timeout=<int>| /liferay-portlet-app/portlet/icon|com.liferay.portlet.icon=<String>| /liferay-
portlet-app/portlet/include|not supported| /liferay-portlet-app/portlet/indexer-class|3|
/liferay-portlet-app/portlet/instanceable|com.liferay.portlet.instanceable=<boolean>| /liferay-
portlet-app/portlet/layout-cacheable|com.liferay.portlet.layout-cacheable=<boolean>| /liferay-
portlet-app/portlet/maximize-edit|com.liferay.portlet.maximize-edit=<boolean>| /liferay-
portlet-app/portlet/maximize-help|com.liferay.portlet.maximize-help=<boolean>| /liferay-
portlet-app/portlet/open-search-class|3| /liferay-portlet-app/portlet/parent-struts-path|com.liferay.portlet.pa
struts-path=<String>| /liferay-portlet-app/portlet/permission-propagator|3| /liferay-portlet-

app/portlet/poller-processor-class|3| /liferay-portlet-app/portlet/pop-message-listener-class|3| /liferay-portlet-app/portlet/pop-up-print|com.liferay.portlet.pop-up-print=<boolean>| /liferay-portlet-app/portlet/portlet-data-handler-class|3| /liferay-portlet-app/portlet/portlet-layout-listener-class|3| /liferay-portlet-app/portlet/portlet-name|not supported| /liferay-portlet-app/portlet/portlet-url-class|3| /liferay-portlet-app/portlet/preferences-company-wide|com.liferay.portlet.preferences-company-wide=<boolean>| /liferay-portlet-app/portlet/preferences-owned-by-group|com.liferay.portlet.preferences-owned-by-group=<boolean>| /liferay-portlet-app/portlet/preferences-unique-per-layout|com.liferay.portlet.preferences-unique-per-layout=<boolean>| /liferay-portlet-app/portlet/private-request-attributes|com.liferay.portlet.private-request-attributes=<boolean>| /liferay-portlet-app/portlet/private-session-attributes|com.liferay.portlet.private-session-attributes=<boolean>| /liferay-portlet-app/portlet/remoteable|com.liferay.portlet.remoteable=<boolean> /liferay-portlet-app/portlet/render-timeout|com.liferay.portlet.render-timeout=<int>| /liferay-portlet-app/portlet/render-weight|com.liferay.portlet.render-weight=<int>| /liferay-portlet-app/portlet/requires-namespaced-parameters|com.liferay.portlet.requires-namespaced-parameters=<boolean>| /liferay-portlet-app/portlet/restore-current-view|com.liferay.portlet.restore-current-view=<boolean>| /liferay-portlet-app/portlet/scheduler-entry|3| /liferay-portlet-app/portlet/scopeable|com.liferay.portlet.scop /liferay-portlet-app/portlet/show-portlet-access-denied|com.liferay.portlet.show-portlet-access-denied=<boolean>| /liferay-portlet-app/portlet/show-portlet-inactive|com.liferay.portlet.show-portlet-inactive=<boolean>| /liferay-portlet-app/portlet/single-page-application|com.liferay.portlet.single-page-application=<boolean>| /liferay-portlet-app/portlet/social-activity-interpreter-class|3| /liferay-portlet-app/portlet/social-request-interpreter-class|3| /liferay-portlet-app/portlet/social-interactions-configuration|3| /liferay-portlet-app/portlet/staged-model-data-handler-class|3| /liferay-portlet-app/portlet/struts-path|com.liferay.portlet.struts-path=<String>| /liferay-portlet-app/portlet/system|com.liferay.portlet.system=<boolean>| /liferay-portlet-app/portlet/template-handler|3| /liferay-portlet-app/portlet/trash-handler|3| /liferay-portlet-app/portlet/url-encoder-class|3| /liferay-portlet-app/portlet/use-default-template|com.liferay.portlet.use-default-template=<boolean>| /liferay-portlet-app/portlet/user-notification-definitions|not supported| /liferay-portlet-app/portlet/user-notification-handler-class|3| /liferay-portlet-app/portlet/user-principal-strategy|com.liferay.portlet.user-principal-strategy=<String>| /liferay-portlet-app/portlet/virtual-path|com.liferay.portlet.virtual-path=<String>| /liferay-portlet-app/portlet/webdav-storage-class|3| /liferay-portlet-app/portlet/webdav-storage-token|not supported| /liferay-portlet-app/portlet/workflow-handler|3| /liferay-portlet-app/portlet/xml-rpc-method-class|3|

- [1] Portlets are registered as concrete objects.
- [2] Multiples of these properties may be used. This results in an array of values.
- [3] This type is registered as an OSGi service.
- [4] https://xmlns.jcp.org/xml/ns/portlet/portlet-app_3_0.xsd
- [5] http://www.liferay.com/dtd/liferay-portlet-app_7_2_0.dtd
- [6] Here's an example of using multiple `javax.portlet.dependency` properties.
  Old:

```
<portlet>
    ...
```

```
    <dependency>
        <name>jquery</name>
        <scope>com.jquery</scope>
        <version>2.1.1</version>
    </dependency>
    <dependency>
        <name>jsutil</name>
        <scope>com.mycompany</scope>
        <version>1.0.0</version>
    </dependency>
    ...
</portlet>
```

New:

```
@Component(
    immediate = true, property = {
        "javax.portlet.name=my_portlet",
        "javax.portlet.display-name=my-portlet",
        "javax.portlet.dependency=jquery;com.jquery;2.1.1",
        "javax.portlet.dependency=jsutil;com.mycompany;1.0.0"
    }, service = Portlet.class
)
public class MyPortlet extends GenericPortlet {
    ...
}
```

- [7] Here's an example for the `com.liferay.portlet.header-request-attribute-prefix` property.
  Old:

```
<portlet>
    ...
    <header-request-attribute-prefix>com.mycompany</header-request-attribute-prefix>
    ...
</portlet>
```

New:

```
@Component(
    immediate = true, property = {
        "javax.portlet.name=my_portlet",
        "javax.portlet.display-name=my-portlet",
        "javax.portlet.dependency=jquery;com.jquery;2.1.1",
        "javax.portlet.dependency=jsutil;com.mycompany;1.0.0",
        "com.liferay.portlet.header-request-attribute-prefix=com.mycompany"
    }, service = Portlet.class
)
public class MyPortlet extends GenericPortlet {
    ...
}
```

- [8] Here's an example for the `javax.portlet.listener` property.
  Old:

```
<portlet>
    ...
    <listener>
        <listener-class>com.mycompany.MyPortletURLGenerationListener</listener-class>
        <ordinal>1</ordinal>
    </listener>
    ...
</portlet>
```

New:

```
@Component(
    immediate = true,
    property = {"javax.portlet.name=myPortlet",
        "javax.portlet.listener=com.mycompany.MyPortletURLGenerationListener;1"
    }, service = Portlet.class
)
public class MyPortlet extends GenericPortlet {
    ...
}
```

- [9] An `javax.portlet.init-param` property can be declared like this:

```
@Component(
    immediate = true,
    property = {"javax.portlet.name=myPortlet",
        "javax.portlet.init-param.myInitParam=1234"},
    service = PortletFilter.class
)
public class MyFilter implements RenderFilter {
    ...
}
```

- [10] Liferay DXP creates each portlet's ID based on the portlet's name (i.e., the `portlet-name` descriptor in `liferay-portlet.xml` or the `javax.portlet.name` OSGi service property). Dashes, periods, and spaces are allowed in the portlet name, but they and all other JavaScript unsafe characters are stripped from the name value that's used for the portlet ID. Therefore, make your portlet name unique in light of the characters that are removed. Otherwise, if you try to deploy a portlet whose ID is the same as a portlet that's already deployed, your portlet deployment fails and Liferay DXP logs a message like this:

```
Portlet id [portletId] is already in use
```

CHAPTER 935

THIRD PARTY PACKAGES PORTAL EXPORTS

This document has been updated and ported to Liferay Learn and is no longer maintained here.

The `com.liferay.portal.bootstrap` module exports many third party Java packages that can cause problems if used improperly. If your WAR's Gradle file, for example, uses the `compile` scope for a dependency that Liferay's OSGi runtime already provides, the dependency JAR is included in the WAR's `WEB-INF/lib` and deployed in the resulting WAB, and two versions of dependency classes wind up on the classpath. This can cause weird errors that are hard to debug.

To find a list of the packages exported by `com.liferay.portal.bootstrap`, go to the source file `modules/core/portal-bootstrap/system.packages.extra.bnd`. If you don't have access to the source code, the same list (in a less user-friendly format) is in the `META-INF/system.packages.extra.mf` file in `[LIFERAY_HOME]/osgi/core/com.liferay.portal.bootstrap.jar`. These packages are installed and available in Liferay's OSGi runtime. If your module or WAR uses one of them, specify the corresponding dependency as being "provided" (provided by Liferay DXP). Here's how to specify a provided dependency:

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

Gradle: `providedCompile`

Now you can safely leverage third party packages Liferay DXP provides!