

2014

Liferay 6.2 Developer Guide

<https://www.liferay.com/documentation/liferay-portal/6.2/development/>

The Liferay Portal 6.2 release is a major release, containing many additional features over the previous release. Liferay recommends that all users upgrade to this release so they can take advantage of these features. This release addresses several usability issues, provides enhancements to existing functionality, and gives developers an easier way to work with mobile clients. What follows is a summarized list of the important enhancement highlights.

Ashok Felix

11/18/2014



Table of Contents

1.	Realizing the Benefits of Liferay's Development Platform	14
1.1.	Developing Applications for Liferay.....	14
1.1.1.	Portlets	14
1.1.2.	OpenSocial Gadgets.....	15
1.1.3.	Reusing Existing Web Applications.....	15
1.1.4.	Supported Technology Frameworks	16
1.2.	Extending and Customizing Liferay	17
1.2.1.	Customizing the Look and Feel: Themes	17
1.2.2.	Adding New Predefined Page Layouts: Layout Templates	17
1.2.3.	Customizing or Extending the Out-of-Box Functionality: Hook Plugins	17
1.2.4.	Advanced Customization: Ext Plugins	17
1.3.	Choosing Your Development Tools.....	18
2.	Working with Liferay's Developer Tools	19
2.1.	Developing Apps with Liferay IDE	19
2.1.1.	Installing Liferay IDE	20
2.1.2.	Setting Up Liferay IDE	23
2.1.3.	Launching and Testing Your Liferay Server	26
2.1.4.	Creating New Liferay Projects.....	27
2.1.5.	Deploying New Liferay Projects to a Liferay Server	30
2.1.6.	Creating Plugins.....	31
2.1.7.	Using the Service Builder Graphical Editor	35
2.1.8.	Importing Existing Liferay Projects from a Plugins SDK	37
2.1.9.	Converting Existing Eclipse Projects into Liferay IDE Projects	39
2.1.10.	Verifying Successful Project Import	40
2.1.11.	Using Liferay IDE's Remote Server Adapter	41
2.2.	Leveraging the Plugins SDK.....	47
2.2.1.	Installing the SDK.....	47
2.2.2.	Structure of the SDK.....	49
2.2.3.	Creating Plugins with Liferay SDK	50
2.2.4.	Best Practices	52
2.3.	Developing Plugins Using Maven	52

2.3.1.	Installing Maven	54
2.3.2.	Using Maven Repositories.....	54
2.3.3.	Installing Required Liferay Artifacts	58
2.3.4.	Using Liferay IDE with Maven	66
2.3.5.	Using a Parent Plugin Project	71
2.3.6.	Creating Liferay Plugins with Maven.....	74
2.3.7.	Deploying Liferay Plugins with Maven.....	78
2.3.8.	Liferay Plugin Types to Develop with Maven	81
2.4.	Deploying Your Plugins: Hot Deploy vs. Auto Deploy	88
2.4.1.	Using Hot Deployment	88
2.4.2.	Using Auto Deployment	89
2.5.	Summary	90
3.	Developing Portlet Applications	91
3.1.	Creating a Portlet Project	92
3.1.1.	Deploying the Portlet	94
3.2.	Anatomy of a Portlet Project.....	96
3.2.1.	A Closer Look at the My Greeting Portlet.....	97
3.3.	Writing the My Greeting Portlet	104
3.4.	Understanding the Two Phases of Portlet Execution	109
3.5.	Passing Information from the Action Phase to the Render Phase	112
3.5.1.	Using Portlet Namespacing.....	115
3.6.	Developing a Portlet with Multiple Actions.....	116
3.7.	Adding Friendly URL Mapping to the Portlet	117
3.8.	Localizing Your Portlets	118
3.8.1.	Using Liferay's Language Keys.....	119
3.8.2.	Sharing Language Keys Between Your Portlets	120
3.8.3.	Generating Language Properties File and Automated Translations.....	122
3.8.4.	Localizing Control Panel Portlets	124
3.9.	Implementing Configurable Portlet Preferences	127
3.9.1.	Creating a Default Setup Tab in the Portlet's Configuration Page	127
3.9.2.	Step 1: Specify a Configuration JSP in the <code>portlet.xml</code>	128
3.9.3.	Step 2: Create the Configuration JSP for Displaying the Portlet Preference Options	
	128	

3.9.4. Step 3: Create a Configuration Action Implementation Class for Processing the Portlet Preference Value.....	130
3.9.5. Step 4: Modify the View JSP to Respond to the Current Portlet Preference Value 131	
3.10. Creating Plugins to Extend Plugins	135
3.11. Creating Plugins to Share Templates, Structures, and More	136
3.12. Summary.....	144
4. Developing JSF Portlets with Liferay Faces.....	145
4.1. Developing JSF Portlets	147
4.1.1. Creating a JSF Portlet Project	147
4.1.2. Deploying JSF Portlets	150
4.1.3. Specifying the portlet.xml for Your JSF Portlet.....	151
4.1.4. Using Portlet Preferences with JSF.....	152
4.1.5. Accessing the Portlet API with ExternalContext.....	153
4.1.6. Internationalizing JSF Portlets	154
4.1.7. Using IPC with JSF Portlets	154
4.1.8. Developing JSF Portlets with CDI.....	158
4.1.9. Using Liferay Faces Bridge JSF Component Tags	162
4.1.10. Dynamically Adding JSF Portlets to Liferay Portal (Runtime Portlets)	166
4.1.11. Extending Liferay Faces Bridge Using Factory Wrappers.....	167
4.2. Understanding Liferay Faces Bridge.....	170
4.2.1. Configuring Liferay Faces Bridge	172
4.3. Leveraging Liferay Utilities with Liferay Faces Portal	177
4.3.1. Using the LiferayFacesContext.....	178
4.3.2. Leveraging the Current Theme	178
4.3.3. Giving Feedback to Users with Validation Messages	178
4.3.4. Leveraging the Portal User's Locale	179
4.4. Using Liferay Portal UIComponent and Composite Component-Tags	179
4.4.1. The liferay-ui:input-editor tag.....	179
4.5. Using Liferay Composite Component Tags	180
4.5.1. The liferay-ui:ice-info-data-paginator tag.....	180
4.5.2. The liferay-ui:ice-nav-data-paginator tag	181
4.5.3. The liferay-ui:icon tag.....	181

4.5.4. The liferay-security:permissionsURL tag	182
4.6. Leveraging AlloyUI Components with Liferay Faces Alloy	182
4.7. Understanding the Liferay Faces Version Scheme.....	183
4.8. Migrating to Liferay Faces.....	184
4.8.1. Migrating BridgeRequestAttributeListener	184
4.8.2. Migrating Configuration Option Names.....	185
4.8.3. Migrating File Upload.....	185
4.8.4. Migrating Facelet Tag Library Namespaces	186
4.8.5. Migrating GenericFacesPortlet	186
4.8.6. Migrating LiferayFacesContext.....	187
4.8.7. Migrating Logging	187
4.8.8. Migrating Portlet Preferences	187
4.9. Migrating From Liferay Faces 3.1 to Liferay Faces 3.2/4.2	188
4.9.1. Migrating Liferay Faces Alloy 3.1 Tags to Liferay Faces Alloy 3.2/4.2 Tags	188
4.9.2. Migrating the liferay-portlet.xml File for Liferay Faces 3.2/4.2	189
4.10. Building Liferay Faces From Source.....	190
4.10.1. Installing the liferay-faces Project.....	190
4.10.2. Building Liferay Faces with Maven.....	191
4.11. Summary.....	192
5. Generating Your Service Layer.....	192
5.1. What is Service Builder?.....	193
5.2. Defining Your Object-Relational Map	195
5.2.1. Creating the service.xml File	197
5.2.2. Defining Global Service Information	198
5.2.3. Defining Service Entities	199
5.2.4. Defining the Columns (Attributes) for Each Service Entity	201
5.2.5. Defining Relationships Between Service Entities	202
5.2.6. Defining Ordering of Service Entity Instances	203
5.2.7. Defining Service Entity Finder Methods	203
5.3. Generating Services.....	206
5.4. Writing Local Service Classes	207
5.5. Calling Local Services	213

5.6.	Understanding the Service Builder-generated Code	218
5.7.	Creating User Interfaces for Service Builder Portlets	224
5.8.	Calling Liferay Services.....	233
5.9.	Using Model Hints	234
5.10.	Writing Remote Service Classes	238
5.10.1.	Calling Remote Services	240
5.11.	Developing Custom SQL Queries	243
5.11.1.	Step 1: Specify Your Custom SQL.....	244
5.11.2.	Step 2: Implement Your Finder Method.....	245
5.11.3.	Step 3: Access Your Finder Method from Your Service	247
5.12.	Configuring service.properties	247
5.13.	Summary.....	248
6.	Accessing Services Remotely	248
6.1.	Finding Services	249
6.1.1.	Finding Portal Services	249
6.1.2.	Finding Portlet Services	249
6.2.	Invoking the API Remotely.....	249
6.3.	Service Security Layers.....	250
6.4.	SOAP Web Services	252
6.4.1.	SOAP Java Client.....	254
6.4.2.	SOAP PHP Client	259
6.5.	JSON Web Services.....	259
6.5.1.	Registering JSON Web Services	260
6.5.2.	Portal Configuration of JSON Web Services	264
6.5.3.	Invoking JSON Web Services	265
6.5.4.	Returned Values	270
6.5.5.	Common JSON Web Service Errors	270
6.5.6.	JSON Web Services Invoker.....	271
6.6.	Authorizing Access to Services with OAuth.....	274
6.6.1.	Selecting an OAuth Client Library	275
6.6.2.	Configuring OAuth's Service Implementation	275
6.6.3.	Creating a User Interface for Authentication	278

6.7.	Summary	280
7.	Using Liferay Frameworks	280
7.1.	ServiceContext	281
7.1.1.	Service Context Fields	281
7.1.2.	Creating and Populating a Service Context	282
7.1.3.	Accessing Service Context Data.....	283
7.2.	Security and Permissions	284
7.2.1.	JSR Portlet Security	284
7.2.2.	Liferay's Permission System.....	286
7.2.3.	Adding a Resource	291
7.2.4.	Adding Permissions	292
7.2.5.	Checking Permissions.....	293
7.2.6.	Creating Helper Classes for Permission Checking	294
7.3.	Creating Portlet URLs in JavaScript	295
7.4.	Asset Framework.....	298
7.4.1.	Adding, Updating, and Deleting Assets	299
7.4.2.	Entering and Displaying Tags and Categories	301
7.4.3.	More JSP Tags for Assets.....	302
7.4.4.	Publishing Assets with Asset Publisher	303
7.5.	Implementing the Recycle Bin in Your App	308
7.5.1.	Moving Entries to the Recycle Bin	309
7.5.2.	Restoring Entries from the Recycle Bin	315
7.5.3.	Implementing the Undo Functionality	318
7.5.4.	Moving/Restoring Parent Entities.....	320
7.5.5.	Resolving Conflicts.....	323
7.6.	Using Message Bus	327
7.6.1.	The Message Bus System	327
7.6.2.	Example Use Case--Procurement Process	329
7.6.3.	Synchronous Messaging	330
7.6.4.	Asynchronous Messaging with Callbacks	336
7.6.5.	Asynchronous "Send and Forget"	340
7.7.	Device Detection	342

7.7.1.	Using the Device API.....	342
7.7.2.	Device Capabilities	343
7.8.	Summary	343
8.	Designing Workflow with Kaleo	343
8.1.	Installing Kaleo Designer for Java	346
8.2.	Creating a Workflow	348
8.2.1.	Palette and Floating Palette.....	350
8.2.2.	Workflow Diagram Features	357
8.2.3.	Properties View and Outline View	359
8.3.	Using Workflow Scripts	360
8.4.	Leveraging Template Editors for Notifications	365
8.4.1.	Creating Notifications	365
8.4.2.	Workflow Context and Service Context Variables	370
8.5.	Viewing Workflow Definition XML Source	373
8.6.	Publishing Workflows to the Server.....	375
8.7.	Using Workflows in Liferay Portal	376
8.8.	Using Dynamic Data Lists (DDLs) with Workflows	376
8.8.1.	Using Kaleo Forms to Run Workflows.....	377
8.9.	Summary	378
9.	Creating Mobile Apps that Use Liferay	379
9.1.	Setting Up the Mobile SDK	381
9.2.	Creating the Liferay Android Sample Project	383
9.3.	Calling Liferay Services in your Android App	385
9.4.	Using Custom Portlet Services in your Android App	385
9.5.	Using the Android SDK	388
9.5.1.	Manually Setting Up the Android SDK.....	388
9.5.2.	Invoking Liferay Services in Your Android App	388
9.5.3.	Invoking Services Asynchronously from Your Android App	390
9.5.4.	Sending Your Android App's Requests Using Batch Processing	392
9.6.	Using the iOS SDK	393
9.6.1.	Setting Up the iOS SDK	393
9.6.2.	Invoking Liferay Services in Your iOS App	393

9.6.3.	Invoking Services Asynchronously from Your iOS App	395
9.6.4.	Sending Your iOS App's Requests Using Batch Processing	396
9.7.	Summary	396
10.	Creating and Integrating with OpenSocial Gadgets.....	397
10.1.	OpenSocial Gadget Basics.....	397
10.1.1.	Gadget Metadata	399
10.1.2.	Gadget Content.....	400
10.2.	Accessing Third-Party Applications from Your Gadget	401
10.3.	Gadget/Portlet Communication with PubSub.....	402
10.3.1.	Gadget to Gadget.....	404
10.3.2.	Communicating Between Portlets and Gadgets	408
10.4.	Using the Gadget Editor	410
10.5.	Summary.....	412
11.	Creating Liferay Themes and Layout Templates	412
11.1.	Creating Liferay Themes	413
11.1.1.	Creating a Theme Project	413
11.1.2.	Setting a Base Theme	416
11.1.3.	Deploying the Theme	416
11.1.4.	Anatomy of a Theme Project	417
11.2.	Using Developer Mode with Themes	419
11.3.	Creating a Theme Thumbnail	421
11.4.	Designing a Look and Feel	421
11.4.1.	Making Themes Configurable with Settings.....	421
11.4.2.	Specifying Color Schemes	424
11.4.3.	Leveraging Portal Predefined Settings.....	425
11.5.	Understanding Your Theme's JavaScript Callbacks in main.js	427
11.6.	Importing Resources with Your Themes	428
11.7.	Creating Liferay Layout Templates	437
11.7.1.	Creating a Layout Template Project.....	437
11.7.2.	Anatomy of a Layout Template Project	439
11.7.3.	Layout Template Files	439
11.7.4.	Liferay Configuration Files.....	440

11.7.5.	Deploying Layout Templates	440
11.7.6.	Designing a Layout Template.....	440
11.8.	Embedding Portlets in a Layout Template.....	444
11.9.	Variables Available to Layout a Template	446
11.10.	Summary.....	447
12.	Customizing and Extending Functionality with Hooks	447
12.1.	Creating a Hook.....	448
12.1.1.	Deploying the Hook	450
12.1.2.	Anatomy of the Hook.....	451
12.2.	Overriding Web Resources	451
12.3.	Customizing JSPs by Extending the Original.....	454
12.4.	Customizing Sites and Site Templates with Application Adapters.....	455
12.4.1.	Including an Original JSP	456
12.4.2.	Creating an Application Adapter	456
12.5.	Performing a Custom Action	458
12.6.	Extending and Overriding portal.properties	459
12.7.	Overriding and Adding Struts Actions.....	459
12.8.	Overriding a Portal Service	463
12.9.	Overriding a Language.properties File	465
12.10.	Extending the Indexer Post Processor	465
12.11.	Supporting Right-to-Left Languages in Plugins.....	468
12.11.1.	Applying the RTL Support to Custom Plugins	472
12.11.2.	Defining Custom CSS for RTL Languages	473
12.12.	Other Hooks.....	474
12.13.	Summary.....	474
13.	Designing User Interfaces with AlloyUI.....	474
13.1.	A simple AlloyUI example	475
13.2.	Using an AlloyUI Carousel in Your Portlet	478
13.2.1.	Adding a Carousel to a Portlet	478
13.2.2.	Customizing the AUI-Carousel	480
13.3.	Working with the AlloyUI project	486
13.3.1.	Working with an AlloyUI Project Release Zip File	487

13.3.2.	Working with the AlloyUI Project Source	488
13.4.	Summary.....	491
14.	Liferay Marketplace.....	491
14.1.	Marketplace Basics.....	492
14.1.1.	What is an App?	492
14.1.2.	What is a Version?.....	492
14.1.3.	What is a Package?.....	493
14.1.4.	How Do Apps Relate to Users and Companies?.....	493
14.1.5.	What Are the Requirements for Publishing Apps?.....	493
14.1.6.	Things You Need Before You Can Publish	494
14.1.7.	Image and Naming Requirements	494
14.1.8.	What Kind of Validations Are Performed by Liferay?.....	495
14.1.9.	What Versions of Liferay Should I Target?	495
14.2.	Developing and Publishing Apps	496
14.2.1.	Develop a Sample App.....	496
14.2.2.	Specify App Packaging Directives	496
14.2.3.	Establish a Marketplace Account	498
14.2.4.	Upload (Publish) Your App	501
14.2.5.	The Review Process	511
14.3.	Making Changes to Published Apps	511
14.3.1.	Editing Your App Details	511
14.3.2.	Editing App Prices.....	512
14.3.3.	Adding Support for New Versions of Liferay Portal.....	512
14.3.4.	Releasing a New Version of your App	512
14.3.5.	Deactivating Your App	513
14.4.	Tracking App Performance	513
14.4.1.	Views.....	515
14.4.2.	Downloads.....	515
14.4.3.	Purchases	515
14.5.	Understanding Plugin Security Management	515
14.6.	Developing Plugins with Security in Mind	516
14.6.1.	Consider Common Security Pitfalls	516

14.6.2.	Develop Your Plugin	519
14.6.3.	Build Your Plugin's PACL.....	519
14.6.4.	Test the Plugin with the Security Manager Enabled	521
14.6.5.	Using a Java Security Policy File.....	521
14.6.6.	Convert PACL Absolute File Paths into Relative Paths	522
14.7.	Enabling the Security Manager	524
14.8.	Portal Access Control List (PACL) Properties	524
14.9.	Summary.....	525
15.	Advanced Customization with Ext Plugins	525
15.1.	Creating an Ext Plugin.....	526
15.1.1.	Using Developer Studio	526
15.1.2.	Using the Terminal	527
15.1.3.	Anatomy of the Ext Plugin.....	528
15.2.	Developing an Ext Plugin.....	530
15.2.1.	Set Up.....	530
15.2.2.	Initial Deployment.....	531
15.2.3.	Redeployment.....	537
15.2.4.	Distribution.....	540
15.2.5.	Ext Plugin Packaging Requirements for JBoss 7	542
15.2.6.	Advanced Customization Techniques	542
15.3.	Deploying in Production.....	547
15.3.1.	Method 1: Redeploying Liferay's Web Application	547
15.3.2.	Method 2: Generate an Aggregated WAR File.....	547
15.4.	Migrating Old Extension Environments.....	548
15.5.	Summary.....	549
16.	What's New in Liferay 6.2 APIs?.....	549
16.1.	Application Display Templates.....	550
16.1.1.	Using the Application Display Templates API.....	550
16.1.2.	Recommendations	555
16.2.	AlloyUI 2.0 / Bootstrap Migration	556
16.2.1.	Removal of the "aui-" Prefix from All Classes	556
16.2.2.	AlloyUI Module Deprecations	557

16.2.3.	CSS Classes Replaced with Bootstrap Equivalents	558
16.2.4.	Component Output and Markup Changes	558
16.2.5.	Icon Removals, in Favor of Using Bootstrap Icons	559
16.2.6.	Upgrading Plugins with the Liferay AlloyUI Upgrade Tool	559
16.2.7.	Example: Upgrading the Microblogs Portlet to AlloyUI 2.0	560
16.3.	Liferay's Deprecation Policy	563
16.4.	Summary.....	563
17.	Conclusions.....	563

1. Realizing the Benefits of Liferay's Development Platform

Welcome to the Developer's Guide, Liferay's official guide for developers. If you're interested in developing applications on Liferay portal or customizing it, you're in the right place. This guide assumes you already know what a portal is and how to use Liferay from an end-user perspective. If you don't, please read the [What is a Portal?](#) article on [liferay.com](#) and the [What is Liferay?](#) chapter in *Using Liferay Portal 6.2*.

This chapter summarizes how to develop applications for Liferay and how to customize Liferay's built-in applications, themes, and settings. You will develop Liferay plugins to encapsulate these applications and customizations. Finally, we'll talk about technologies and tools available to use as you develop your plugins.

This chapter covers the following:

- **Developing Applications for Liferay:** Ways to develop new applications and reuse existing applications
- **Extending and Customizing Liferay:** Options for extending functionality and customizing your portal applications, themes, and templates
- **Choosing Your Development Tools:** Comparison of tools available for developing applications for Liferay

Let's talk about developing applications for Liferay.

1.1. *Developing Applications for Liferay*

According to Wikipedia "A web application is an application that is accessed over a network such as the Internet or an intranet." A portal application is a web application that can coexist with other applications. Portal applications leverage functionality provided by the portal platform to reduce development time and deliver a more consistent experience to end users.

As a developer wanting to run your own applications on top of Liferay Portal, you probably want to know *what's the best and quickest way to do it?* Liferay supports two main, standards-based technologies for incorporating your applications into Liferay: **Portlets** and **OpenSocial gadgets**.

1.1.1. **Portlets**

Portlets are small web applications written in Java that run in a portion of a web page. The heart of any portal implementation is its portlets, because they contain the actual functionality. The portlet container just aggregates the set of portlets to appear on each page.

Since they're entirely self-contained, portlets are the least invasive mechanism for extending Liferay, and are also the most forward compatible development option. They are hot-deployed as plugins into Liferay instances, resulting in zero downtime. A single plugin can contain multiple portlets, allowing you to split up your functionality into several smaller pieces that can be arranged dynamically on a page. Portlets can be written using any of the Java web frameworks that support portlet development, including Liferay's specific frameworks: MVC Portlet and Alloy Portlet. Portlets can be used to build complex applications since they can leverage the full suite of technologies and libraries for the Java platform.

1.1.2. OpenSocial Gadgets

OpenSocial gadgets are usually small applications, written using browser-side technologies such as HTML and JavaScript. Like portlets, OpenSocial gadgets provide a standard way to develop applications for a portal environment. From a technology perspective, one key difference is that they don't mandate a specific back-end technology, such as Java EE, PHP, Ruby, Python, etc. Another difference is that they have been designed specifically to implement social applications, while portlets were designed for any type of application. Because of this, OpenSocial gadgets not only provide a set of technologies to develop and run applications, but also a set of APIs that allow the application to obtain information from the social environment such as information about a user's profile, his activities, and his friends.

An OpenSocial gadget is deployed in Liferay as one of the following types:

- **Remote gadget:** is executed in a remote server but presented in a given page as if it were another platform application. Remote gadget deployment is simple, but the portal depends on the remote server for the gadget to work. Deployment as a remote gadget is not a viable option in some Intranet environments that lack full access to the Internet.
- **Local gadget:** is deployed in the Liferay server in a similar manner to portlets. Since a gadget is defined in an XML file, uploading this file is all that's necessary to deploy the gadget.

Once you've saved your new gadget, it appears as an application that administrators can add to their site's pages.

Liferay lets you expose portlets to the outside world as OpenSocial gadgets. That is, you can develop a portlet and then let anyone with access to your portlet add it as a remote gadget to pages on other portals or social networks.

1.1.3. Reusing Existing Web Applications

What if you already have an existing application that has not been implemented as a portlet or OpenSocial gadget? You have many options, including:

- Rewrite the application as a portlet.
- Create simple portlets that interact with the application (possibly using Web Services) and offer that functionality to end-users.
- Create an OpenSocial gadget as a wrapper for the application. The gadget can use an IFrame to show part of the application in the portal page.
- Create a portlet that integrates the remote application either using an IFrame or an HTTP proxy (e.g., using Liferay's WebProxy portlet). This requires implementing single sign-on between the portal and the application.
- If the application is implemented using Struts 1.x, it can be converted to a portlet application with only a few changes.
- If the application is implemented using JSF, it can be converted to a portlet application with only a few changes.

There are many more options, each with its own merits. Reviewing them all is out of the scope of this guide; however, the above options are worth considering.

Next let's consider some of the technology frameworks Liferay supports.

1.1.4. Supported Technology Frameworks

Liferay, as a platform, strives to provide compatibility with any Java technology you may want to use to develop your applications. Thanks to the portlet and Java EE specifications, each portlet application can use its own set of libraries and technologies, whether they are used by Liferay or not. This section refers mainly to portlet plugins; other plugin types are more restricted. For example, Ext plugins can only use libraries that are compatible with the ones used by the core Liferay code.

Since the choice of available frameworks and technologies is very broad, choosing the appropriate one can be daunting. We'll provide some advice to help you choose the best frameworks for your needs, summarized as follows:

1. *Use what you know:* If you already know a framework, that can be your first option (Struts 2, Spring MVC, PHP, Ruby, etc).
2. *Adapt to your real needs:* Component-based frameworks, such as JavaServer™ Faces (JSF), Vaadin, and Google Web Toolkit (GWT), are especially good for desktop-like applications. MVC frameworks, on the other hand, provide more flexibility.
3. *When in doubt, pick the simpler solution:* Portlet applications are often more simple to implement than standalone web applications. When in doubt, use the simpler framework (e.g., Liferay's MVC Portlet or Alloy Portlet).

Some of the frameworks mentioned above include their own JavaScript code to provide a high degree of interaction. That is the case with GWT, Vaadin, and JSF implementations (e.g. ICEfaces or Rich Faces). You may prefer to write your own JavaScript code and leverage one of the JavaScript libraries available. You can use any JavaScript library with Liferay, including jQuery, Dojo, YUI, Sencha (previously known as ExtJs), and Sproutcore.

Since version 6, however, Liferay has its own library called *AlloyUI* which is based on YUI 3. AlloyUI has a large set of components specifically designed for modern user interfaces. Liferay's core portlets make use of AlloyUI. You can use AlloyUI for your custom portlets or use another JavaScript library, as long as the library does not conflict with libraries referenced by other portlets deployed in the same portal.

Liferay's Service Builder automates creating interfaces and classes for database persistence and service layers. It generates most of the common code that implements database access, letting you focus on higher level aspects of service design. You implement the local interface with your business logic, and the remote interface with your permission checks. Objects on the portal instance interact with the local interface, while objects outside interact with the remote interface via JSON, SOAP, and Java RMI.

In addition to those mentioned above, there are thousands more frameworks and libraries available to you for handling persistence, caching, connections to remote services, and much more. Liferay does not impose specific requirements on the use of any of those frameworks. You, the portal developer, choose the best tools for your projects.

1.2. *Extending and Customizing Liferay*

Liferay provides many out-of-the-box features, including a fully featured content management system, a social collaboration suite, and several productivity tools. For most installations, these features are exactly what you need; but sometimes you'll want to extend these features or customize their behavior and appearance.

Liferay is designed to be customized. Multiple plugins and plugin types can be combined into a single WAR file. Let's take a look at these plugin types and how they can be used.

1.2.1. *Customizing the Look and Feel: Themes*

Themes let you dictate your site's look and feel. You can specify color schemes and commonly used images. You'll apply styling for UI elements such as fonts, links, navigation elements, page headers, and page footers, using a combination of CSS and Velocity or FreeMarker templates. With Liferay's AlloyUI API framework, you use a consistent interface to common UI elements that make up your page. This makes it easy to create sites that respond well to the window widths of your users' desktop, tablet, and mobile devices. Most importantly, themes let you focus on designing your site's UI, while leaving its functionality to the portlets.

1.2.2. *Adding New Predefined Page Layouts: Layout Templates*

Layouts are similar to themes, except they specify the *arrangement* of portlets on a page rather than their look and feel. You can create custom layout templates to arrange portlets just the way you like them. And you can even embed commonly used portlets. Like themes, layout templates are also written in Velocity and are hot-deployable.

1.2.3. *Customizing or Extending the Out-of-Box Functionality: Hook Plugins*

Hook plugins are how you customize the core functionality of Liferay at many predefined extension points. Hook plugins are used to modify portal properties or to perform custom actions on startup, shutdown, login, logout, session creation, and session destruction. Using service wrappers, a hook plugin can replace any of the core Liferay services with a custom implementation. Hook plugins can also replace the JSP templates used by any of the default portlets. Best of all, hooks are hot-deployable plugins just like portlets.

1.2.4. *Advanced Customization: Ext Plugins*

Ext plugins provide the largest degree of flexibility in modifying the Liferay core, allowing you to replace essentially any class with a custom implementation. However, it is highly unlikely that an Ext plugin written for one version of Liferay will continue to work in the next version without modification. For this reason, Ext plugins are only recommended for cases in which an advanced customization is truly necessary, and there is no alternative. Make sure you are familiar with the Liferay core so your Ext plugin doesn't negatively effect existing functionality. Even though Ext plugins are deployed as plugins, the server must be restarted for their customizations to take

effect.



Note: If you have developed for Liferay 5.2 or prior releases, you may be familiar with what was known as the *Extension Environment*. Ext plugins were introduced in Liferay 6.0 to replace the extension environment in order to simplify development. For instructions on converting an existing Extension Environment into a plugin, see the section on migrating old extension environments in Advanced Customization with Ext Plugins.

Now that you're familiar with the best options for developing applications on Liferay and customizing Liferay, let's consider some of the tools you'll be using.

1.3. Choosing Your Development Tools

The Java ecosystem is known for providing a variety of options for almost any type of software development. This is advantageous because you can find the tool that best fits your needs and the way you work. Naturally, when you get comfortable with a tool, you want to keep using it.

If you're a newcomer, the wide variety of tools available can be overwhelming. Throughout this guide, we'll give you the best of both worlds showing you how to develop plugins in two environments that use open source technologies 1) A command-line environment that integrates with a wide variety of tools. 2) An easy-to-use IDE that minimizes your learning curve while giving you powerful development features. Here are those two environments:

Apache Ant and the Plugins SDK: Liferay provides a development environment called the *Plugins SDK* that lets you develop plugins of all types by executing a set of predefined commands (known as *targets*, in Ant's nomenclature). You can use the Plugins SDK directly from the command-line and use file editors like Emacs, Vim, EditPlus, or even Notepad. You can also integrate the Plugins SDK with your favorite IDE, since most IDEs provide support for Apache Ant. The next chapter includes a section on how to use the Plugins SDK.

Eclipse and the Liferay IDE: Eclipse is the most popular and well known Java IDE and it provides a wide variety of features. *Liferay IDE* is a plugin for Eclipse that extends its functionality to facilitate developing all types of Liferay plugins. Liferay IDE uses the Plugins SDK underneath, but you don't need to know the SDK unless you're performing an advanced operation not directly supported by Liferay IDE. To develop applications for Liferay Portal Enterprise Edition (EE), use Liferay Developer Studio which extends Liferay IDE, providing additional integration plugins such as the Kaleo Designer for Java.

This guide shows you how to develop for Liferay using both the Plugins SDK and Liferay IDE, to benefit you and other developers even if you don't like IDEs or don't use Eclipse. If you use Eclipse, you'll be happy to know that we'll show you how to develop apps using Liferay IDE in the next chapter.

What about if I don't like Apache Ant and I prefer to use Maven? Many developers prefer other command-line alternatives to Apache Ant. The most popular of these alternatives is Maven. To support developers that want to use Maven we have *mavenized* Liferay artifacts for referencing in your Maven projects. See the next chapter for an in-depth look at developing

plugins in Maven.

What if I don't like Eclipse and prefer to use NetBeans, IntelliJ IDEA or other another IDE? There are many IDEs available, and each one has its strengths. We built Liferay IDE on top of Eclipse because it's the most popular open source option. We also want to make sure you can use the IDE of your choice. In fact, many core developers use NetBeans and IntelliJ IDEA. Both of these IDEs have support for integration with Apache Ant, so you can use the Plugins SDK with them. Additionally, there is an extension to NetBeans called the *Portal Pack* that is explicitly designed for developing plugins for Liferay. You can find more about the Portal Pack at <http://contrib.netbeans.org/portalpack>.

That's it for the introduction. Let's get started with real development work!

2. Working with Liferay's Developer Tools

If you're anything like Liferay Portal's developers, you don't want to be forced to work with one development technology. Our developers build Liferay with the tools they prefer. That's why we strive to provide you with as much flexibility as possible. You can develop your Liferay-based portal with tools ranging in complexity from IDEs like Eclipse, Netbeans, or IntelliJ Idea, to text editors like Notepad, Vim, or Emacs. You can write your persistence layer directly using SQL and JDBC, or use advanced object-relational mapping libraries like Hibernate or iBATIS. You get the idea.

In this chapter, we'll explain how to set up a streamlined development environment specifically designed for developing your Liferay Portal. Then we'll consider how to develop plugins with other tools. We'll cover the following topics along the way:

- Developing Apps with Liferay IDE
- Leveraging the Plugins SDK
- Developing Plugins Using Maven
- Deploying Your Plugins: Hot Deploy vs. Auto Deploy

Liferay's tool-agnosticism is great for experienced developers who understand the strengths and weaknesses of different development technologies; it can be overwhelming for newcomers, though. So we removed some of the options, narrowing down your choices and forcing you to use a tool we like, right? No! We actually added to the list of technologies you can use by developing specific tools that soften the learning curve for Liferay plugin development, and providing ways for you to use alternative tools. The most significant Liferay-specific tool is Liferay IDE, a fully featured Integrated Development Environment (IDE) based on Eclipse. There's also the Plugins Software Development Kit (SDK), which is based on Apache Ant and can be used with any editor or Integrated Development Environment you'd like. If you'd like, you can also use Apache Maven archetypes; there are plenty of Liferay archetypes you can use to develop your plugins.

First let's consider the most robust tool for Liferay development, Liferay IDE.

2.1. **Developing Apps with Liferay IDE**

Even if you're a grizzled veteran Java developer, if you're going to be doing a lot of development

for your Liferay Portal instance, consider using Liferay IDE. When Liferay IDE is paired with the Plugins SDK or Maven and a Liferay runtime environment, you have a one stop development environment where you can develop your Liferay plugins, build them, and deploy them to your Liferay instance.

Liferay IDE is an extension for Eclipse IDE and supports development of plugin projects for the Liferay Portal platform. You can install Liferay IDE as a set of Eclipse plugins from an update site. The latest version of Liferay IDE supports development of portlets, hooks, layout templates, themes, and Ext plugins. To use Liferay IDE, you need the Eclipse Java EE developer package using Indigo or a later version.

In this section we'll show you how to install Liferay IDE, set up projects for your applications, and deploy them to your portal. We'll get you started with the basics of developing your Liferay application in Liferay IDE. The guide has other chapters geared to each specific plugin type (e.g., the *portlets* chapter covers portlet development, the *hooks* chapter covers hook development, etc.). But, as we create a Liferay portlet project in this chapter, you'll get the gist of how Liferay IDE helps you create all types of plugins easily.

We'll also introduce you to Liferay's Service Builder. It helps you leverage Hibernate's Object-Relational Mapping capabilities and gives you the capability to automatically generate code to access Liferay object data. We'll point out the various editor modes Liferay IDE provides for creating your data entities, relating them, and building services around them. This section gives you a quick tour, but we've dedicated an entire chapter later in this guide to give Liferay's Service Builder the attention it deserves; check it out, we think you'll be impressed.

To install and set up Liferay IDE, follow the instructions in the first two subsections below. If you're already using *Liferay Developer Studio* (the king of Liferay's development tools), which comes with Liferay Portal Enterprise Edition, skip to the section titled *Testing and Launching your Liferay Server*--Liferay IDE comes preconfigured in Developer Studio.

2.1.1. Installing Liferay IDE

Liferay IDE is an extension of the Eclipse IDE. You can install Eclipse bundled with Liferay IDE or you can add it to an Eclipse instance.

Liferay IDE requires the following software:

- Java 6.0 JRE or greater.
- One of the following Eclipse releases:
 - Eclipse Kepler Java EE (4.3.x)
 - Eclipse Juno Java EE (4.2.x)
 - Eclipse Indigo Java EE (3.7.x)

If you don't already have Eclipse installed, you can install Liferay IDE bundled with Eclipse. You can alternatively install Liferay IDE onto an existing supported Eclipse installation. All Liferay IDE installation options are explained in this section.

2.1.1.1. Installing Liferay IDE Bundled with Eclipse

Installing Liferay IDE and Eclipse from the same bundle is convenient and easy to do.

1. Go to the Liferay IDE page. Under *Other Downloads*, select the Eclipse bundle for your operating system and click *Download*.
2. Install Eclipse bundled with Liferay IDE by extracting its contents to a local folder.
3. To start Eclipse, execute the Eclipse executable file (e.g., `eclipse.exe`) from the installation folder.
4. Select *Window → Open Perspective → Other ... → Liferay* to use Liferay IDE.

You've installed Eclipse and Liferay IDE together!

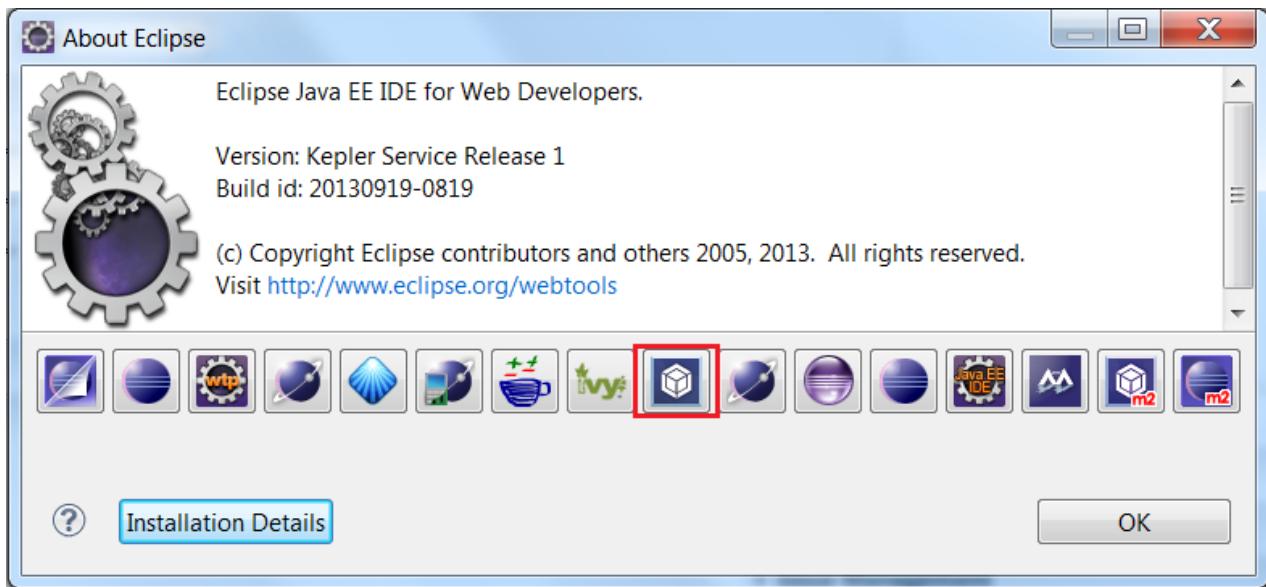
You can alternatively install Liferay IDE onto an existing supported Eclipse installation. Supported versions of Eclipse are available from the Eclipse website.

To install Liferay IDE onto Eclipse, you can either access the Liferay IDE update site from Eclipse or download a Liferay IDE archive file to access from Eclipse. Installing Liferay IDE from the update site is the easiest way to add it to Eclipse.

2.1.1.2. *Installing Liferay IDE from the Update Site onto Eclipse*

To install Liferay IDE and specify an Eclipse update URL, follow these steps:

1. Start Eclipse.
2. When Eclipse opens, go to *Help → Install New Software....*
3. In your browser, go to the Liferay IDE downloads page. Copy the URL of the update site you're interested in (stable or milestone).
4. In the *Work with* field, paste in the Liferay IDE update site URL and press *Enter*.
5. Make sure the Liferay IDE features are selected, then click *Next*.
6. After calculating dependencies, click *Next*, accept the license agreement, and click *Finish* to complete the installation.
7. Restart Eclipse to verify that Liferay IDE is properly installed.
8. After restarting Eclipse, go to *Help → About Eclipse*; if you see a Liferay IDE icon badge as in the screenshot below, it's properly installed.



9. Select *Window* → *Open Perspective* → *Other ...* → *Liferay* to use Liferay IDE.
Alternatively, you can install Liferay IDE from a downloaded archive file.

2.1.1.3. Installing Liferay IDE from an Archive File onto Eclipse

To install Liferay IDE from an archive file, follow these steps:

1. Go to the Liferay IDE downloads page. Under *Other Downloads*, select the *Liferay IDE [version] Archived Update-site* option and click *Download*.
2. Start Eclipse.
3. When Eclipse opens, go to *Help* → *Install New Software....*
4. In the *Add Site* dialog, click the *Archive* button and browse to the location of the downloaded Liferay IDE archive file.
5. Make sure the Liferay IDE features are selected, then click *Next*.
6. After calculating dependencies, click *Next*, accept the license agreement, and click *Finish* to complete the installation.
7. Restart Eclipse to verify that Liferay IDE is properly installed.
8. Select *Window* → *Open Perspective* → *Other ...* → *Liferay* to use Liferay IDE.

After restarting Eclipse, you can verify that Liferay IDE is installed by going to *Help* → *About Eclipse* and finding the Liferay IDE icon badge.

Congratulations on installing Liferay IDE!

Let's set up Liferay IDE now that you have it installed.

2.1.2. Setting Up Liferay IDE

Now that you have Liferay IDE installed, either from a downloaded zip file or from the update site appropriate for your Eclipse version, you need to perform some basic setup. This section describes the setup steps to perform so you can develop your Liferay portal and test your customizations.

Before setting up Liferay IDE, let's make sure you have all the appropriate software packages installed.

2.1.2.1. Requirements

Before setting up Liferay IDE, you need to have appropriate versions of Liferay Portal, Liferay Plugins SDK and/or Maven, and Eclipse. Make sure you satisfy these requirements before proceeding:

1. Liferay Portal 6.0.5 or greater is downloaded and unzipped.
2. Liferay Plugins SDK 6.0.5 or greater is downloaded and unzipped, and/or any version of Maven is installed. If you're using the Plugins SDK, make sure the Plugins SDK version matches the Liferay Portal version.
3. You've installed an appropriate Eclipse IDE version for Java EE Development, and the Liferay IDE extension--see the *Installation* section if you haven't already done this.



Note: Earlier versions of Liferay (e.g., 5.2.x) are not supported by the Liferay IDE.

Let's set up your Liferay Plugins SDK.

2.1.2.2. Setting Up the Liferay Plugins SDK

Before you begin creating new Liferay plugin projects, a supported Liferay Plugins SDK and/or Maven installation and Liferay Portal must be installed and configured in your Liferay IDE. If you're thinking, "Wait a second, buster! You told me earlier that the Plugins SDK and Maven could be used without Liferay IDE!", then you're right. In the second half of this chapter, we'll explain how to use the Plugins SDK and Maven on its own, with a text editor. Here, we explain the easiest way to use the Plugins SDK: by running it from Liferay IDE.

1. In Eclipse, open the *Installed Plugin SDKs* dialog box--from your *Windows* dropdown menu, click *Preferences* → *Liferay* → *Installed Plugin SDKs*.
2. Click *Add* to bring up the *Add SDK Dialog*.
3. Browse to your Plugins SDK installation. The default name is the directory name; you can change it if you want.
4. Select *OK* and verify that your SDK was added to the list of *Installed Liferay Plugin SDKs*.



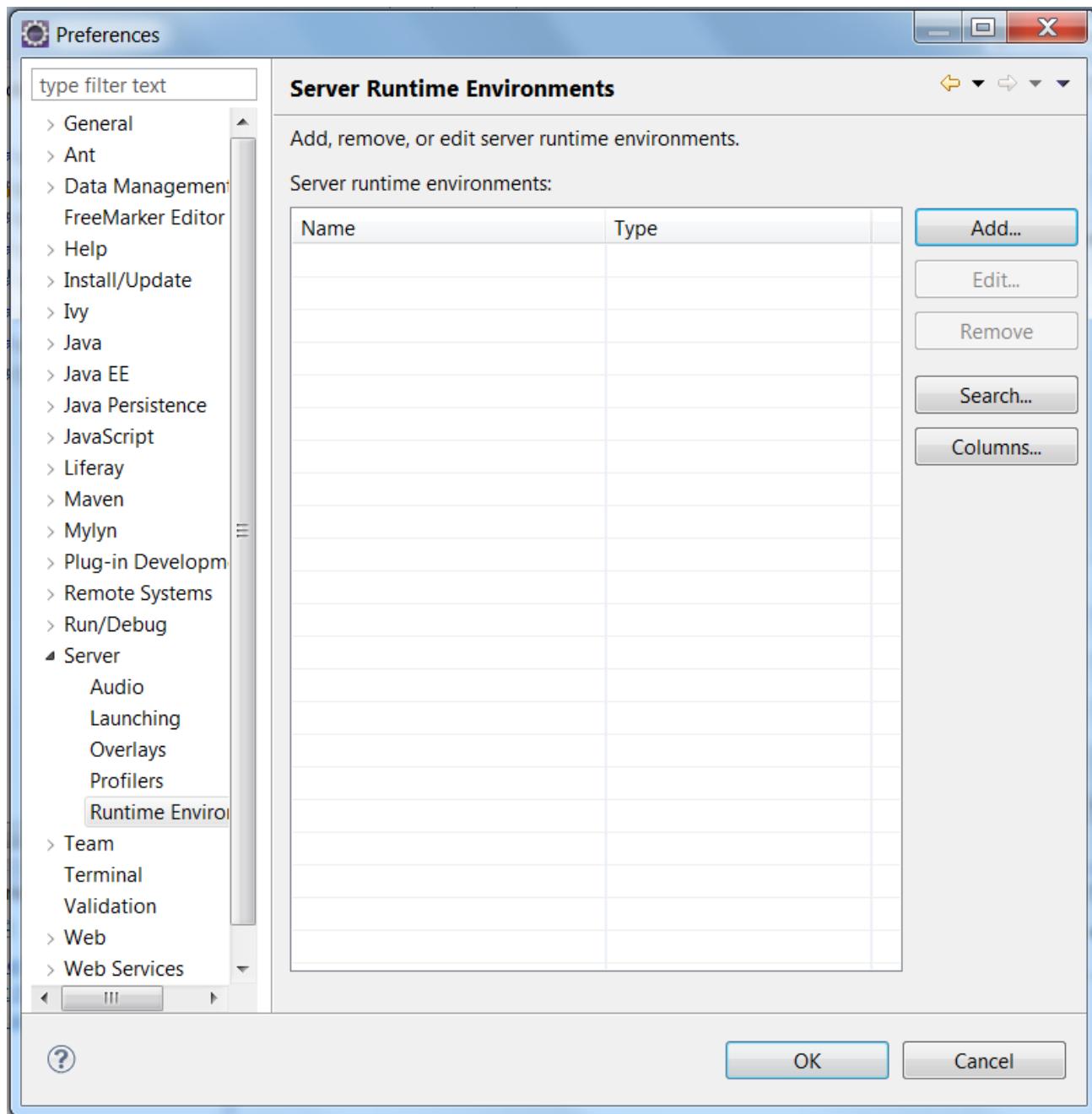
Note: You can have multiple Plugins SDKs configured. You can set the default Plugins SDK by checking its box in the list of *Installed Liferay Plugin SDKs*.

Let's set up your Liferay Portal Tomcat runtime and server.

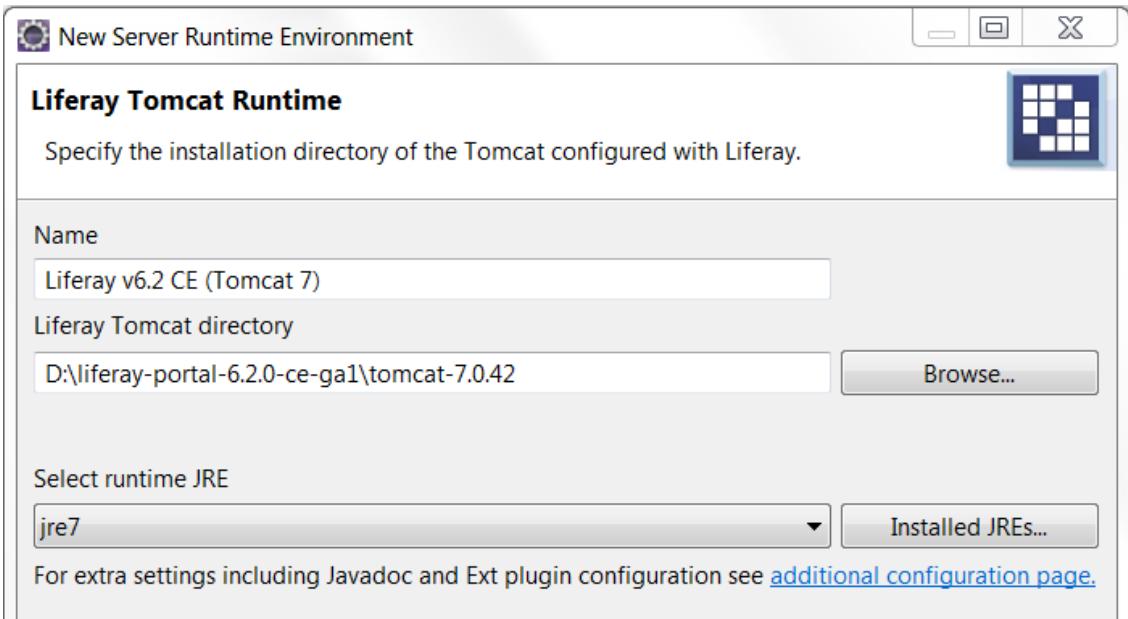
2.1.2.3. Liferay Portal Runtime and Server Setup

You can run Liferay on any application server supported by Liferay Portal. Here, for demonstration purposes, we'll set up our Liferay runtime on the Tomcat application server. The steps you'd follow for any other supported application server would be similar. For a list of Liferay bundles with other application servers, please visit Liferay's Downloads page. For instructions on installing Liferay manually on other application servers, please refer to the Installation and Setup chapter of *Using Liferay Portal 6.2*.

1. In Eclipse, open the *Server Runtime Environments* dialog box--go to *Window → Preferences → Server → Runtime Environments*.



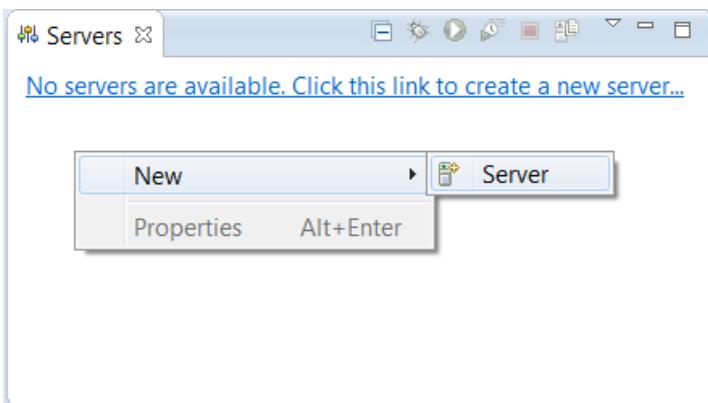
2. Click *Add* to add a new Liferay runtime; find *Liferay v6.2 (Tomcat 7)* under the *Liferay, Inc.* category and click *Next*.
 3. Click *Browse* and select your `liferay-portal-6.2.x/tomcat-7.x` directory.
 4. If you've selected the Liferay portal directory and a bundle JRE is present, it is automatically selected as the server's launch JRE. If no JRE bundle is present, then you must select the JRE to use for launch by clicking *Installed JREs*....



5. Click **Finish**; you should see your Liferay portal runtime listed in **Prefere nces** → **Server**

→ *Runtime Environments*.

6. Click **OK** to save your runtime preferences.
7. If you haven't created a server, create one now from the *Servers* view in Liferay IDE; then you can test the server. Note that you need to be in the Liferay perspective of Eclipse to see the Servers view. You can get there by selecting *Window* → *Open Perspective* → *Other...* and then selecting *Liferay* from the list.



8. Scroll to the *Liferay, Inc* folder and select *Liferay v6.2... Server*. Choose the *Liferay v6.2...* runtime environment that you just created.

Now your server is set up. Let's launch it and perform some tests!

2.1.3. Launching and Testing Your Liferay

Server

Once your Liferay Portal Server is set up, you can launch it from the Servers tab in Eclipse. You have a few options for launching and stopping the server once it's selected in the Servers tab.

From the *Servers* tab:

- Click on the green *Start the Server* button to launch it (or use *Ctrl+Alt+R*).
- Click on the red *Stop the Server* button to stop it (or use *Ctrl+Alt+S*). You'll only see this button if the server is running.

- Right click the server and select *Start*.
- Right click on the server and select *Stop*.

Once the server is launched, you can open Liferay portal home from the *Servers* tab by right clicking your Liferay Tomcat server and selecting *Open Liferay Portal Home*.

Next, you'll learn to create new Liferay projects in Liferay IDE.

2.1.4. Creating New Liferay Projects

Plugins for Liferay Portal must be created inside of a Liferay project. A Liferay project is essentially a root directory with a standardized structure containing the project's (and each of its plugins') necessary files. Since each plugin type requires a different folder and file structure, let's create a project to illustrate the process. Have you heard of the hip new social networking site for noses, *Nose-ster*? Harold Schnozz, the site's founder, wants to capitalize on the site's popularity by providing users with the ability to organize local meetings and events. For instance, there's a really active group of noses in Minneapolis, MN, who'd like to schedule a regional dance in January, which they're calling the Frozen Boogie. Why does this concern us? Mr. Schnozz has hired us to develop the necessary portlets to allow users to create and view events on the Nose-ster portal.

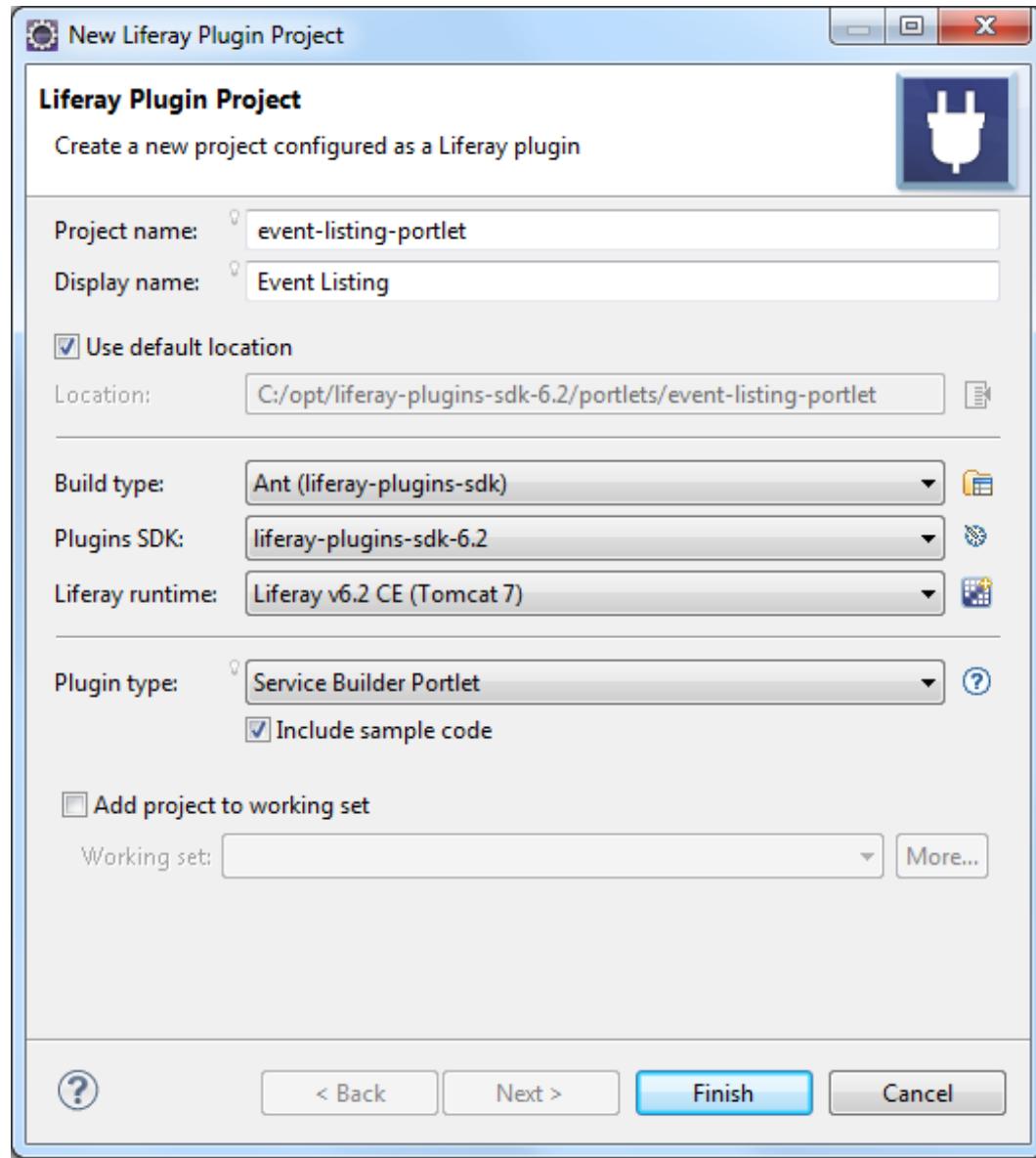
If you've been following our Liferay IDE configuration instructions, your Plugins SDK and Liferay portal server have already been configured in Liferay IDE. Now let's create a new Liferay plugin project in Liferay IDE.

1. Go to *File → New → Liferay Plugin Project*.
2. In the project creation wizard, you'll name and configure your project.

We'll create a plugin project that we'll use throughout this guide. First, we'll create a bare bones plugin project; then, we'll manually add an additional plugin to the project and add additional configurations.

- 2.1. Provide both a *Project Name*, which is used to name the project's directory, and a *Display Name*, which is used to identify the plugin when adding it to a page in Liferay Portal. Our demonstration project will have the project name *event-listing-portlet* and the display name *Event Listing*.
- 2.2. Leave the *Use default location* checkbox checked. By default, the default location is set to your Plugins SDK. If you'd like to change where your plugin project is saved in your file system, uncheck the box and specify your alternate location.
- 2.3. Select the *Ant (liferay-plugins-sdk)* option for your build type. If you'd like to use *Maven* for your build type, navigate to the *Developing Plugins Using Maven* section for details.
- 2.4. Your newly configured SDK and Liferay Runtime should already be selected. If you haven't yet pointed Liferay IDE to a Plugins SDK, click *Configure SDKs* to open the *Installed Plugin SDKs* management wizard. You can also access the *New Server Runtime Environment* wizard if you need to set up your runtime server; just click the *New Liferay Runtime* button next to the *Liferay Portal Runtime* dropdown menu.
- 2.5. Under *Plugin Type*, indicate which plugin type your project will hold by selecting one from

the list. You can choose from *Portlet*, *Service Builder Portlet*, *Hook*, *Layout Template*, *Theme*, *Ext*, or *Web*. Liferay IDE provides handy wizards for creating new Liferay projects. Our demonstration project will hold service builder portlets for the Nose-ster organization, so make sure *Service Builder Portlet* is selected.



Great!
You've
created a
Liferay
portlet
project!

You can find
more
information
on Liferay's
plugin
frameworks
in the
chapter on
portlet
developmen
t. In that
chapter,
we'll discuss
the plugin
creation
wizard in
more detail.

Note: We're
creating the
event-
listing-
portlet
project now
so that we
can

highlight how Liferay IDE simplifies project creation. For more information on creating portlets, please see the chapter of this guide on portlets. Similarly, for more information on themes, layout templates, hooks, or Ext plugins, please refer to the appropriate chapter of this guide.

Our *event-listing-portlet* plugin project should appear in the Eclipse package explorer. The project was created in the plugins SDK you configured, under the directory corresponding to the plugin type the project contains. Here's the generalized directory structure for portlet projects

created in Liferay IDE/Developer Studio:

- PROJECT-NAME/
 - docroot/WEB-INF/src
 - build.xml - **Common project file**
 - docroot/
 - css/
 - main.css
 - view.jsp
 - js/
 - main.js
 - META-INF/
 - MANIFEST.MF
 - WEB-INF/
 - lib/
 - tld/
 - aui.tld
 - liferay-portlet-ext.tld
 - liferay-portlet.tld
 - liferay-security.tld
 - liferay-theme.tld
 - liferay-ui.tld
 - liferay-util.tld
 - liferay-display.xml
 - liferay-plugin-package.properties - **Common project file**
 - liferay-portlet.xml
 - portlet.xml
 - service.xml
 - web.xml
 - icon.png
 - view.jsp

All projects, regardless of type, are created with a `build.xml` and a `liferay-plugin-package.properties` file--we've highlighted each of them with the note **Common project file** in the directory structure above. The `build.xml` file allows Liferay IDE to use Ant to automatically compile and deploy your plugins. Another default file is `liferay-plugin-package.properties`. This file contains important metadata for your project. Liferay IDE's `properties` view gives you a simple interface to inspect or specify the file's fields, including your project's dependencies and deployment context, display name, and Liferay version. If you publish your project as an app to Liferay Marketplace, the value of the `name` property in `liferay-plugin-package.properties` is used as the app's name. The value of the `liferay-versions` property is used on Liferay Marketplace to specify the versions of Liferay on which your application is intended to run. Next, you need to deploy your new plugin project to your Liferay Server.

2.1.5. Deploying New Liferay Projects to a Liferay Server

You have a plugin project, but you need to deploy it onto your Liferay Server. The easiest way to deploy a plugin project is to drag the project from the Package Explorer view onto your Liferay runtime in the Servers view. Alternatively, you can use the following method:

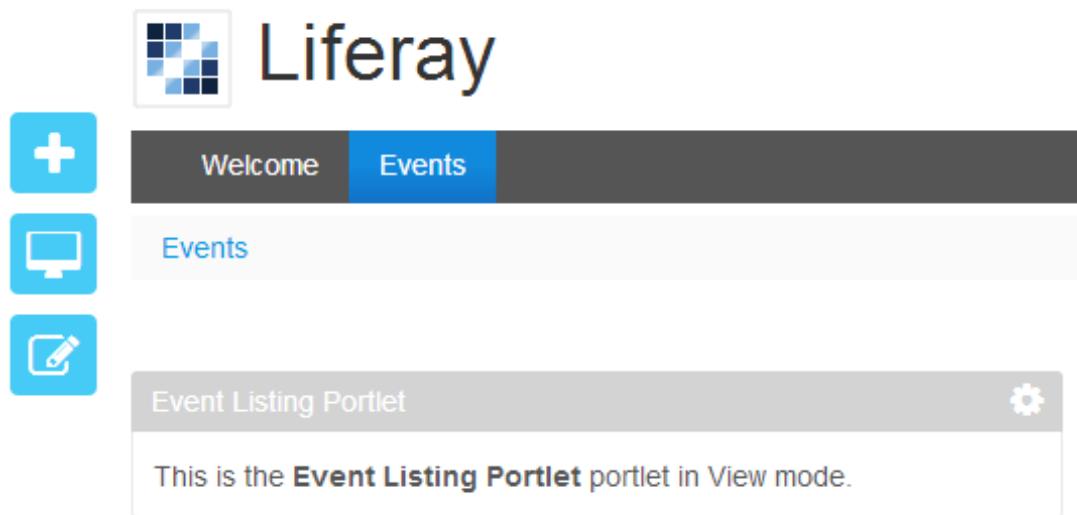
1. Select your new plugin project then right click the Liferay Server in the *Servers* tab.
2. Select *Add and Remove....*
3. Select your plugin project and click *Add* to deploy it to the server.
4. Click *Finish*.

Deploy your project. You should see the project get deployed to your Liferay server; in the console you'll see a message, like the one below, indicating that your new portlet is available for use.

```
INFO [localhost-startStop-2] [PortletHotDeployListener:490] 1 portlet for event-listing-portlet is available for use
```

Open *Liferay Portal Home* (<http://localhost:8080/> for a fresh Liferay installation) and log in with your administrator account. If this is your first time starting Liferay, follow the instructions in the setup wizard. For more setup wizard details, see the Using Liferay's Setup Wizard section of Chapter 15 in *Using Liferay Portal 6.2*.

Once you're logged in, click *Add → More*; expand the *Sample* category and click the *Add* link next to your Event Listing application. Your *Event Listing Portlet* shows on the page.



Liferay IDE. But before we do, let's clean out the bare-bones portlet from our event-listing-portlet project.

The portlet project wizard conveniently creates a default portlet named after the project. However, for demonstration purposes, we want to begin creating portlets with a clean slate in our project. Let's tear out the default Event Listing portlet by removing its descriptors and its JSP.

1. Open the portlet's `docroot/WEB-INF/liferay-display.xml` file and remove

the `<portlet id="event-listing" />` tag.

2. Open the `docroot/WEB-INF/liferay-portlet.xml` file and remove the `<portlet>...</portlet>` tags and code residing between those tags.
3. Navigate to the `docroot/WEB-INF/portlet.xml` file and remove the `<portlet>...</portlet>` tags and code residing between those tags.
4. Remove the `docroot/view.jsp` file.

Super! You've cleaned out the default portlet from the project. Now you're ready to start creating the example plugins.

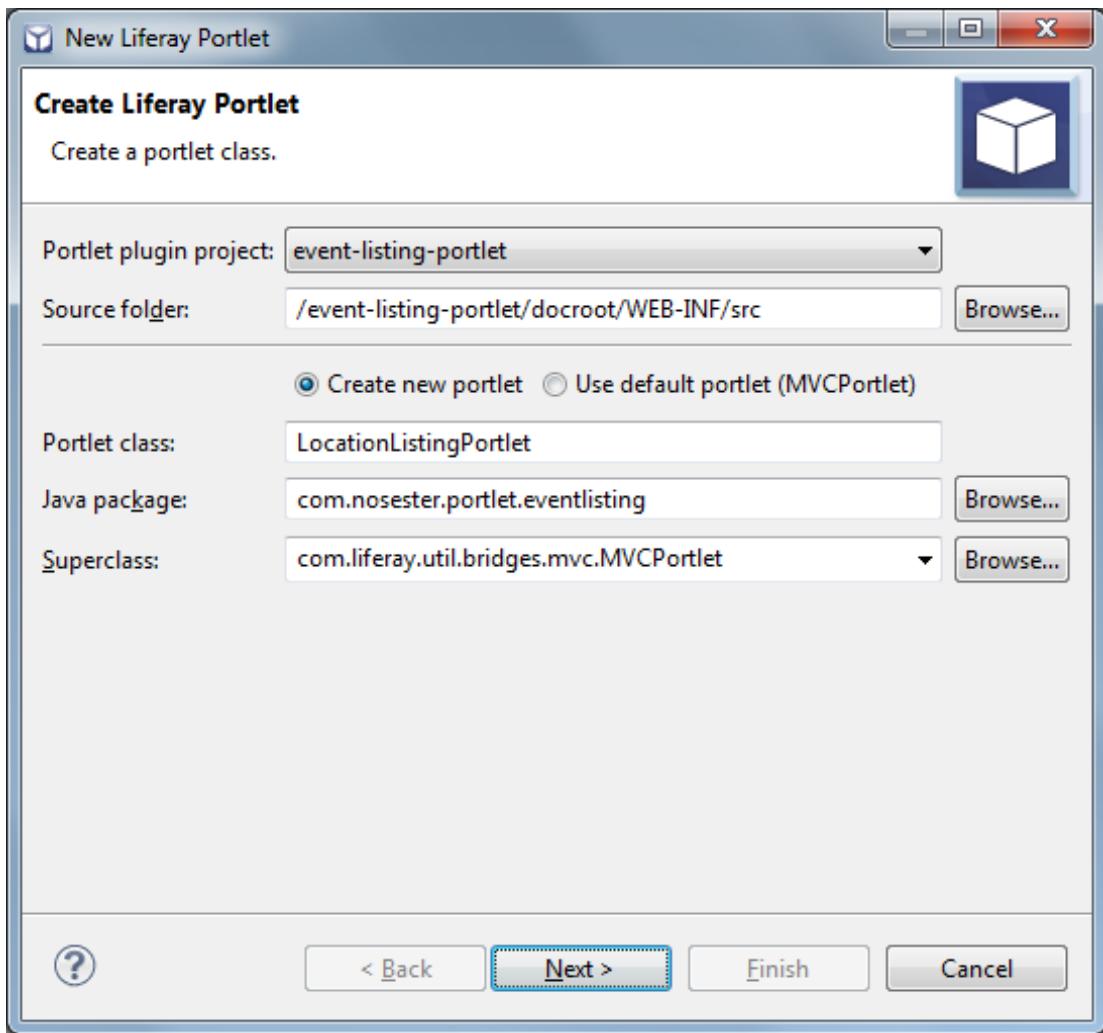
2.1.6. Creating Plugins

Liferay projects can contain multiple plugins. If you've followed the instructions from the earlier section on creating new Liferay projects, you should already have created the event-listing-portlet project. In this section we'll add two portlets to the event-listing-portlet project: the Location Listing portlet and the Event Listing portlet. This illustrates the general process for creating plugins inside of an existing Liferay project. Later in this guide, when we complete developing the Event Listing and Location Listing portlets, they'll allow users to add, edit, or remove events or locations, display lists of events or locations, search for particular events or locations, and view the details of individual events or locations. For now, we'll show you how to create both portlets in the event-listing-portlet project.

Your Liferay IDE's Package Explorer shows your Event Listing plugin project. Since it's a portlet type project it has a skeleton in place for supporting more portlet plugins. Let's start by creating the Location Listing portlet.

Use the following steps to create the Location Listing portlet:

1. Right click on your `event-listing-portlet` project in Liferay IDE's *Package Explorer* and select `New → Liferay Portlet`.
2. The *New Liferay Portlet* dialog box appears with your plugin project `event-listing-portlet` selected as the *Portlet plugin project* by default. It's a good idea to name your *Portlet class* after the name of your portlet. We'll name the class `LocationListingPortlet` in this example. Name your *Java package* after the plugin's parent project, so it will be `com.nosester.portlet.eventlisting`, and leave the *Superclass* as `com.liferay.util.bridges.mvc.MVCPortlet`. Alternatively, you could have selected `com.liferay.portal.kernel.portlet.LiferayPortlet` or `javax.portlet.GenericPortlet` for your superclass.



- **Portlet class:** *LocationListingPortlet*
- **Java package:** *com.nosester.portlet.eventlisting*
- **Superclass:** *com.liferay.util.bridges.mvc.MVCPortlet*

Click *Next*.

3. In the next window of the *New Liferay Portlet* wizard, you'll specify deployment descriptor details for your portlet. First enter the *Name* of your portlet--in our example, this will be *locationlisting*. Next, enter the portlet's *Display name* and *Title*; we'll specify both as *Location Listing Portlet*. In this window, you can also specify which portlet modes you'd like your portlet to have. *View* mode is automatically selected. There are also options for creating resources: you can specify the folder where JSP files will be created as well as whether or not a resource bundle file will be created. We'll leave the *Create JSP files* box flagged, specify *html/locationlisting* as the JSP folder and flag the *Create resource bundle file* box.

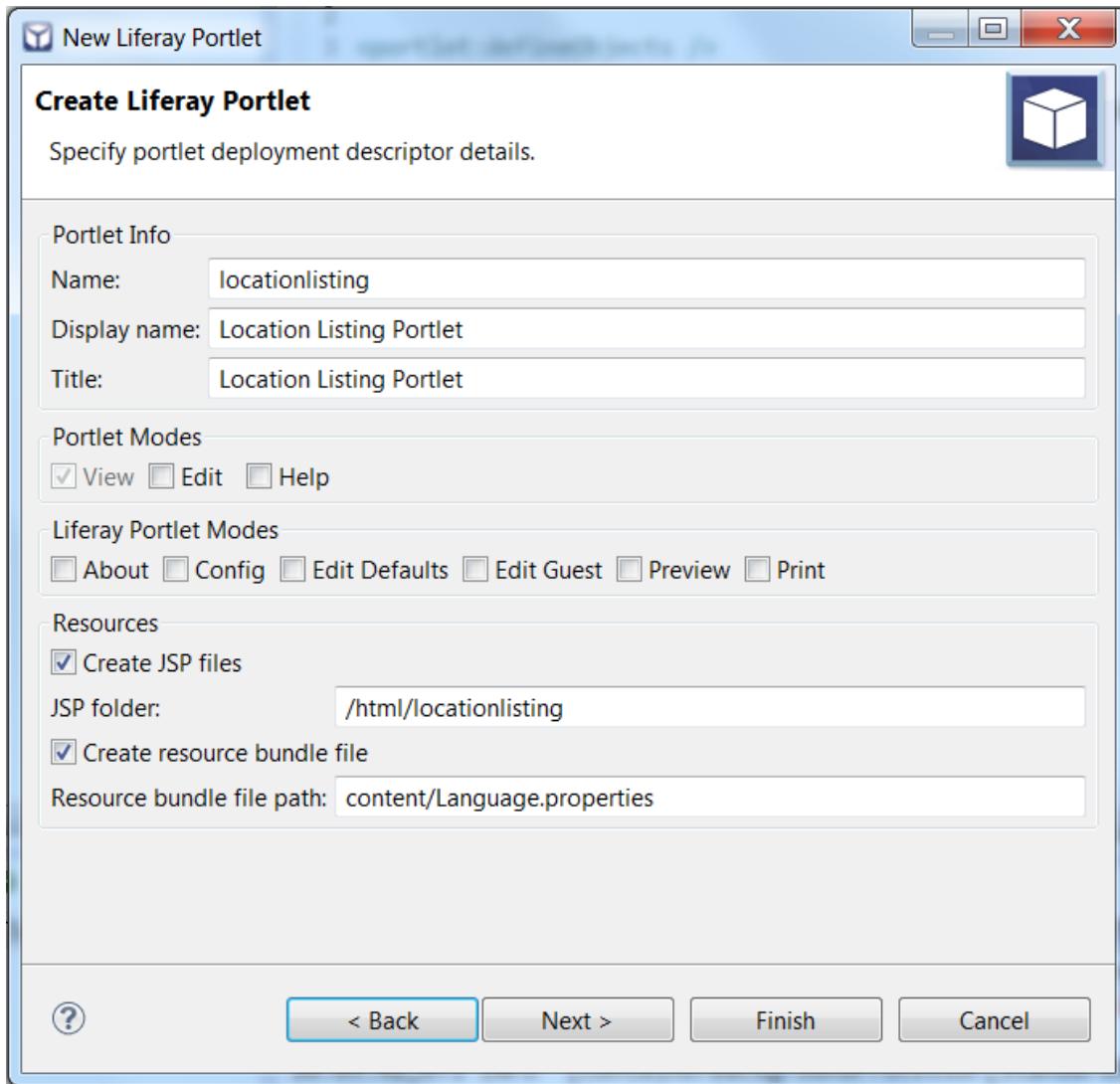
Here are the portlet deployment descriptor details to specify for the Location Listing portlet:

Here are the portlet class values to specify for the example Location Listing portlet:

P
ortlet
plugin
project:
event-
listing-
portlet
S
ource
folder:
/event-
listing-
portlet/d
ocroot/W
EB-
INF/src

- › **Name:** *locationlisting*
- › **Display name:** *Location Listing Portlet*
- › **Title:** *Location Listing Portlet*
- › **JSP folder:** *html/locationlisting*

Click *Next*.



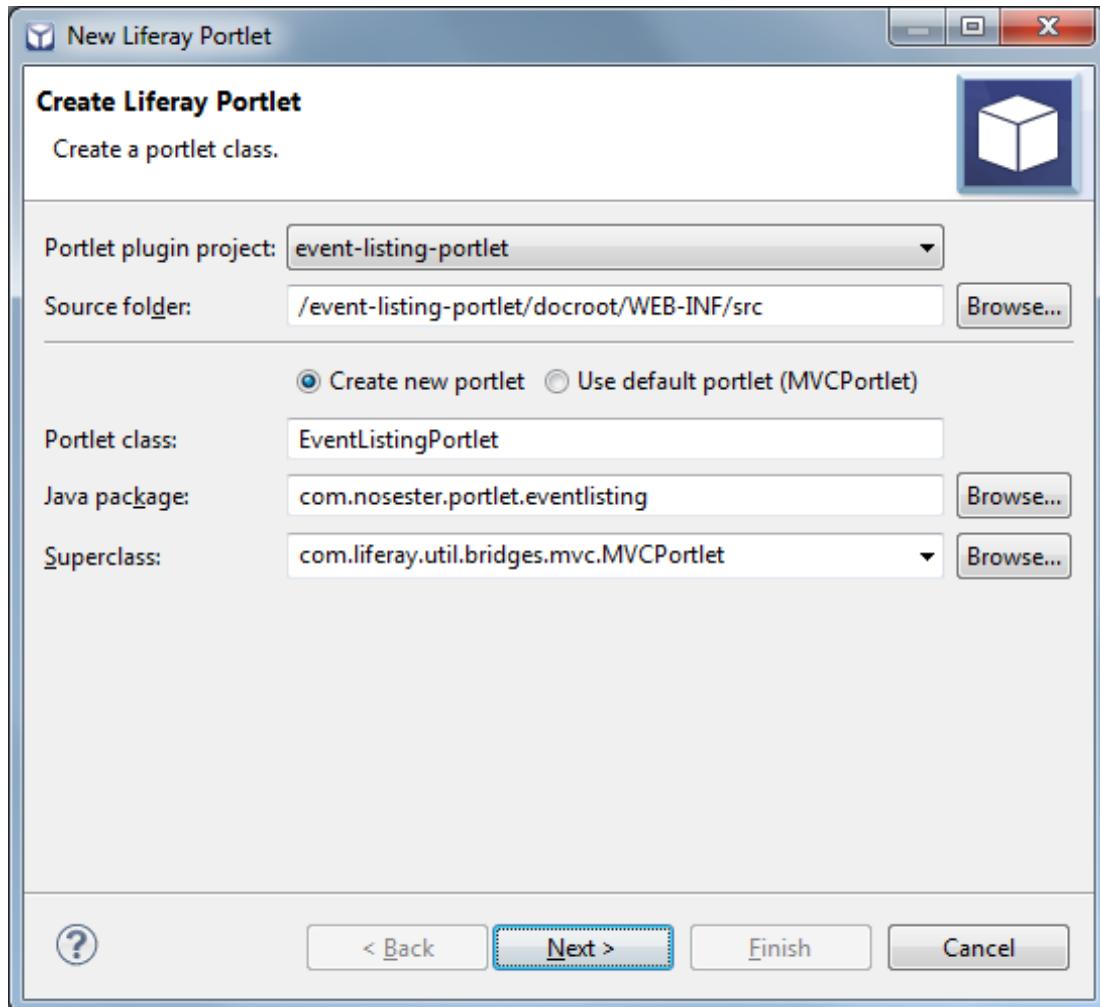
4. The next window lets you specify portlet deployment descriptor details that are specific to Liferay. You can set the file paths of your portlet's custom

icon, main CSS file, and main JavaScript file. You can also specify a CSS class wrapper. Next, you can also choose the category for your portlet (it's categorized under *Sample* by default) and choose whether or not to add it to the *Control Panel* of your Liferay Portal. Accept the default, leaving the *Add to Control Panel* box unflagged. Click *Next*.

5. The last step is to specify modifiers, interfaces, and method stubs to generate in the Portlet class. Accept the defaults and click *Finish*.

Use the following steps to create the Event Listing portlet:

1. Right-click your event-listing-portlet project → *New* → *Liferay Portlet*. Specify *EventListingPortlet* as the name of the portlet class, enter *com.nosester.portlet.eventlisting* as its Java package, and select *com.liferay.util.bridges.mvc.MVCPortlet* as its superclass.



Here are the portlet class values to specify for the example Event Listing portlet:

P

Portlet plugin project:
event-listing-portlet

S

Source folder:
/event-listing-portlet/docroot/WEB-INF/src

- INF/src
- › **Portlet class:** *EventListingPortlet*
- › **Java package:** *com.nosester.portlet.eventlisting*
- › **Superclass:** *com.liferay.util.bridges.mvc.MVCPortlet*

Click *Next*.

2. In this window we'll specify the portlet's deployment descriptor details.

Here are the portlet deployment descriptor details to specify for the Event Listing portlet:

- › **Name:** *eventlisting*
- › **Display name:** *Event Listing Portlet*
- › **Title:** *Event Listing Portlet*
- › **JSP folder:** *html/eventlisting*

Click *Next*.

3. This window lets you specify portlet deployment descriptor details that are specific to Liferay. You can set the file paths of your portlet's custom icon, main CSS file, and main JavaScript file. You can also specify a CSS class wrapper. In the *Liferay Display* section, you can choose the category for your portlet (it's categorized under *Sample* by default) and choose whether or not to add it to the *Control Panel* of your Liferay Portal. Accept the default, leaving the *Add to Control Panel* box unflagged and click *Next*.
4. The last step in creating your portlet with the wizard is to specify modifiers, interfaces, and method stubs to generate in the Portlet class. Accept the defaults and click *Finish*.

By default, new portlets use the MVCPortlet framework, a light framework that hides part of the complexity of portlets and makes the most common operations easier. The default MVCPortlet project uses separate JSPs for each portlet mode: each of the registered portlet modes has a corresponding JSP with the same name as the mode. For example, `edit.jsp` is for edit mode and `help.jsp` is for help mode.

Let's redeploy the plugin project to make our portlet plugins available in the portal. In the *Servers* tab, simply right click the *event-listing-portlet* project, then click *Redeploy*.

Now you've created and deployed the *Location Listing* portlet and the *Event Listing* portlet from the same project. Eventually, when the Location Listing portlet is complete it will allow users to enter viable event locations.

Next, we'll show you how our Service Builder tool helps you generate your model, persistence, and service layers.

2.1.7. Using the Service Builder Graphical Editor

Loose coupling is a great principle to use when developing your applications. By keeping all of your code for fetching data self contained in a service layer, separate from the business logic of your application, you can more easily swap out your entire service layer without disrupting the functionality of your application.

Service Builder is a model-driven code generation tool that lets you define custom object models called entities. Service Builder reads the contents of a file you create called `service.xml` and automatically creates your application's model, persistence, and service layers, freeing you to focus on the higher level aspects of your application's code.

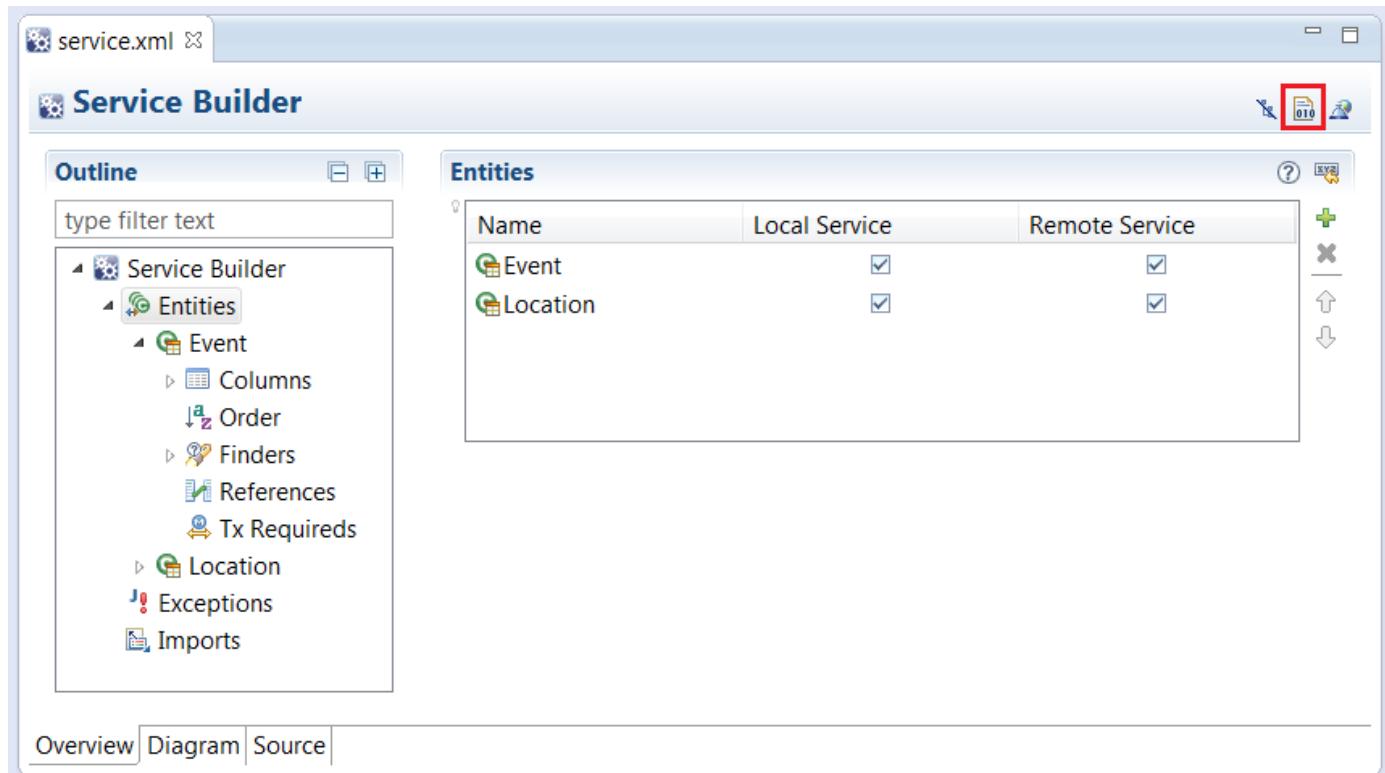
Why should you use Service Builder? Because it lets different portlets access the same data and application logic, creating an underlying framework that supports a portal environment. If your database access code is buried in a single application's code, it can't readily be shared with other applications, and your efforts will be duplicated with each application you write. Service Builder puts the generated code in a service JAR file inside of one plugin, but it can be easily shared among all portlets.

To allow you more than one way to view and edit the `service.xml` file, Service Builder gives you three modes to work in:

- Overview mode provides an easy to use graphical interface in Liferay IDE where you can

add to and edit the `service.xml` file. Overview mode also gives you a *Build Services* button to generate the service layer.

- Diagram mode gives you a visualization of the relationships between service entities; it's often helpful to create your entities using diagram mode.



- Source mode displays the raw XML content of the `service.xml` file.

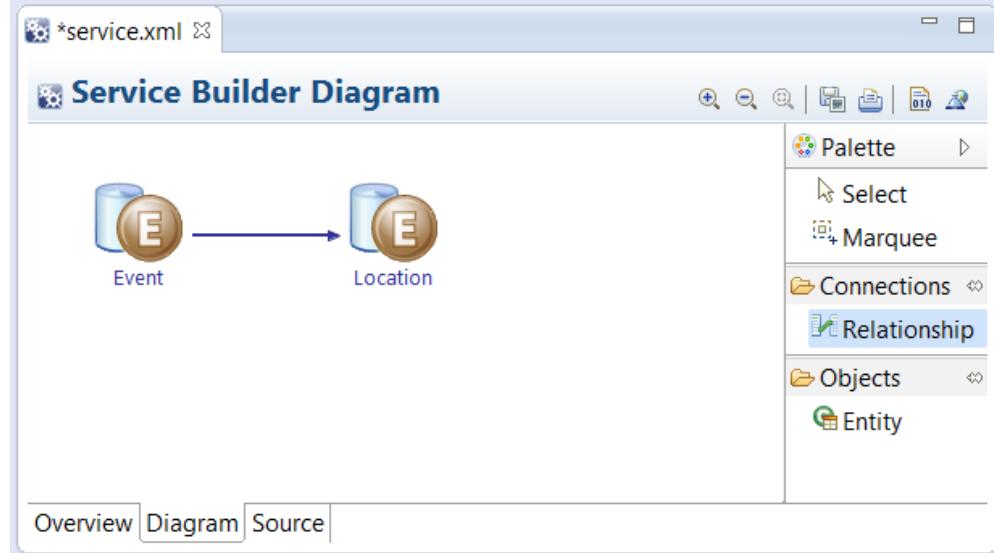
With Liferay IDE, generating your service layer is easy. First you'll create `service.xml`, by selecting your project in the Package Explorer and then selecting *File* → *New* → *Liferay Service Builder*. Service Builder creates a `service.xml` file in your `docroot/WEB-INF` folder and displays the file in overview mode. If you're following along with the event-listing-portlet, you already have the `service.xml` file because we created service builder portlet project during setup.

Our Service Builder chapter of this guide will lead you through filling out `service.xml` to define the following:

- Global service information
- Service entities
- The attributes for each service entity
- Relationships between service entities
- Ordering of service entities instances

- Service entity finder methods

In the Service Builder chapter of this guide, we'll show you how our two custom portlets, the Events Listing Portlet and the Location Listing Portlet, can be developed more efficiently and modularly by using Service Builder. We'll describe the contents of `service.xml` in detail and get you started using Service Builder to develop your custom applications using our code generation tool. And if *code generator* is a bad word to you, let us assure you that Liferay always gives you full control over all your code, including code generated by Service Builder.



Once you've generated your `service.xml`, you can build services. When viewing `service.xml` in overview mode, there's a button available at the top right hand corner of the window. The button looks like

a document with the numerical sequence *010* in front of it. Alternatively, right click on your project and select *Liferay → Build Services*. Once the process is finished, you'll see a plethora of new Java classes under `docroot/WEB-INF/src` in your project that Service Builder generated for you. Now you can get out there and write your business logic, but make sure to check out the Service Builder chapter of this guide for a thorough description of its capabilities.

Now you know how to create projects and plugins from scratch and you know about Service Builder's amazing time-saving capabilities. Let's learn how to import existing projects into Liferay IDE.

2.1.8. Importing Existing Liferay Projects from a Plugins SDK

Do you want to import one or more Liferay projects into your Liferay IDE workspace from a Liferay Plugins SDK? Liferay IDE makes it easy. Don't worry if the projects already contain `.project` or `.classpath` files, the process we'll show you will still import them into your workspace.

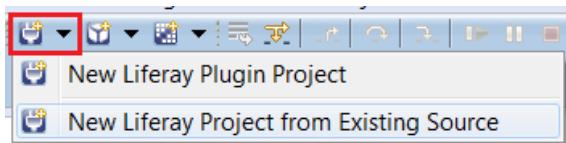


Note: This section assumes that you've created projects with the Plugins SDK and are familiar with the directory structure used by the Plugins SDK. If you need to, check out the *Plugins SDK* section of this chapter; it comes right after this section.

First, let's look at the steps for importing a single Liferay project from a Plugins SDK project into your workspace. For these steps, we'll assume you haven't yet configured your Plugins SDK in Liferay IDE:

1. In Liferay IDE, go to *File* → *New* → *Project...* → *Liferay* → *Liferay Project from Existing Source*.

You can invoke the same wizard from the Liferay shortcut bar; just click the *New* button and select *Liferay Project from Existing Source*.



2. In the *New Liferay Project* window, click the *Browse* button and navigate to the project folder of the plugin you'd like to import. It should be a subfolder of one of the SDK's plugin type folders (e.g., portlets, hooks, themes, etc) or you'll get an error message stating that your Liferay project location is invalid.

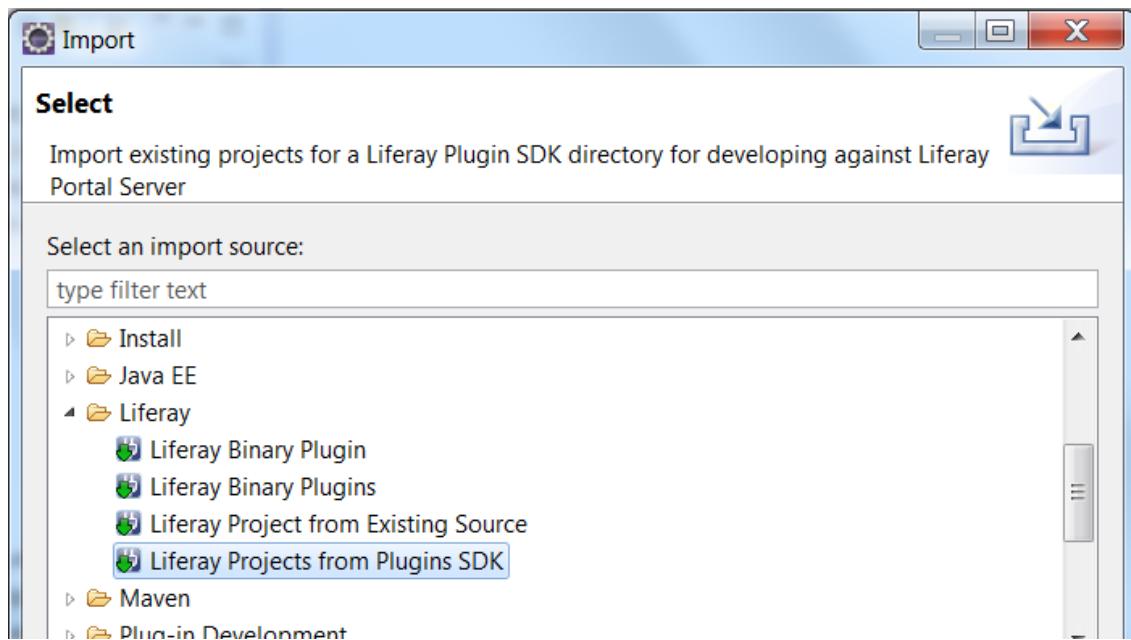
On selecting the plugin project folder, the *Liferay plugin type* and *Liferay plugin SDK version* values are updated. If your Plugins SDK is outdated or you entered an incorrect project type, its field gets marked with an error.

3. Select the *Liferay target runtime* for the plugin project. If you don't have a Liferay Portal Runtime, use the *New...* button to create one now. For more detailed instructions, see the section *Liferay Portal Runtime and Server Setup*, found earlier in this chapter.
4. Click *Finish* to complete the import.

Any time you import a project into Liferay IDE, you can verify that it was successfully configured as a Liferay IDE project by using the process outlined in the section *Verifying Successful Project Import*, found later in this chapter.

Next, let's import multiple projects from a Liferay Plugins SDK you've already set up in Liferay IDE. You can use these steps:

1. In Liferay IDE, go to *File* → *Import...* → *Liferay* → *Liferay Projects from Plugins SDK*.



Plugins SDK from which you're importing plugins.



Note: If your SDK isn't configured in Liferay IDE (i.e., it's not in the dropdown list of the *Import Projects* window), use the *Configure* link to add one. To configure a Plugins SDK from the Installed SDKs window, just click *Add* and then browse to the Plugins SDK's root directory.

3. Once you select your Plugins SDK in the combo box, the *Liferay Plugin SDK Location* and *Liferay Plugin SDK Version* fields are automatically filled in, as long as they're valid. Invalid entries are marked with an error.
4. The list of projects that are available for import are displayed in a list. Any projects already in the workspace are disabled. Projects available for import have an empty check box; select each project you'd like to import.
5. Select the Liferay runtime for the imported projects. If you don't have a Liferay runtime, can add one now with the *New...* button.
6. Click *Finish*.

You've imported your plugins into your workspace! Next, we'll discuss a different scenario; converting existing Eclipse projects into Liferay projects.

2.1.9. Converting Existing Eclipse Projects into Liferay IDE Projects

The steps outlined in the previous section are for importing Liferay projects that aren't already in your Eclipse workspace. You can also import a non-Liferay project in your Eclipse workspace

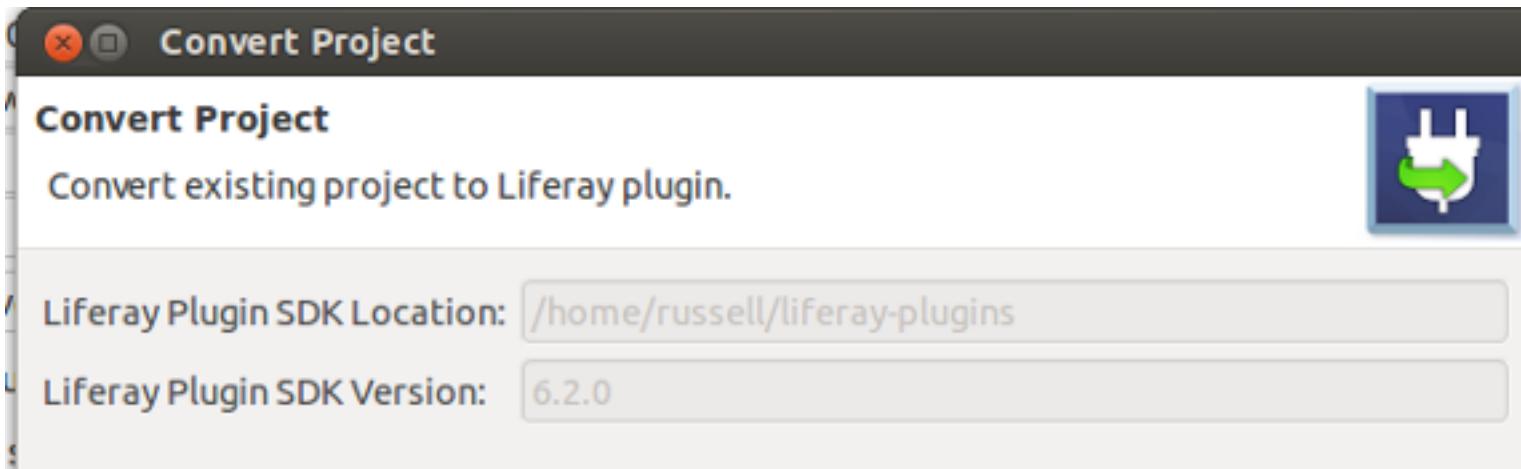
(i.e., you can see it in Eclipse's project explorer) and convert it to a Liferay project. Just follow the steps below.

1. Move the project into a Liferay Plugins SDK if it is not already in one. To import the project, select *File → Import...* and then follow the import instructions that appear.
2. In Eclipse's Project Explorer, right-click on the project and select *Liferay → Convert to Liferay plugin project*.



Note: If no convert action is available, either the project is already a Liferay IDE project or it is not faceted (i.e., Java and Dynamic Web project facets are not yet configured for it). For instructions on resolving these issues, see the section *Verifying Successful Project Import*, found later in this chapter.

3. In the *Convert Project* wizard, your project is selected and the SDK location and SDK version of your project is displayed.



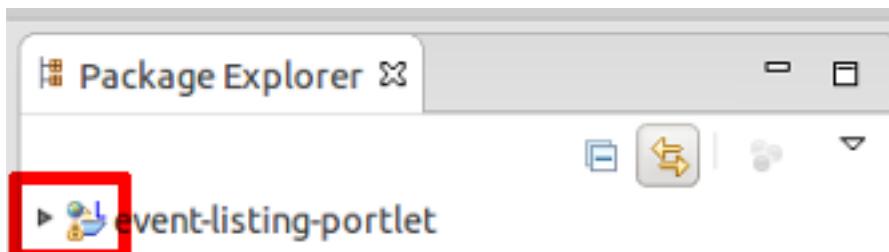
4. Select the Liferay runtime to use for the project. If you don't have a Liferay Runtime defined, define one now by clicking *New....*
5. Click *Finish*.

Let's verify the success of your imports and ensure that they're properly configured as Liferay IDE projects.

2.1.10. Verifying Successful Project Import

After importing projects into Liferay IDE, you'll want to verify that they imported successfully and that they're properly configured as Liferay IDE projects. Here's how you verify that your imports were successful:

1. Once the project is imported, you should see a new project inside Eclipse and it should have an "L" overlay image; the "L" is for Liferay!



Explorer, right click → *Properties* → *Targeted Runtimes*.

2.2. In the *Properties* window, click *Project Facets* and make sure that the Liferay plugin facets are properly configured.

The screenshot shows the "Project Facets" tab selected in the left sidebar of the Properties window. On the right, a list of facets is shown with their versions and descriptions. The "Liferay Portlet" facet is selected, indicated by a checked checkbox icon.

Facet	Version	Description
Liferay Plugins	6.0	Great! You've confirmed that your import was successful; you can now make revisions to your configured Liferay IDE project. Next, let's explore
Liferay Ext	6.0	
Liferay Hook	6.0	
Liferay Layout Template	6.0	
Liferay Portlet	6.0	
Liferay Theme	6.0	

Liferay IDE's Remote Server Adapter feature.

2.1.11. Using Liferay IDE's Remote Server Adapter

The *Remote Server Adapter* is a feature that lets you deploy your Liferay projects to a remote Liferay Portal server; it first became available in Liferay IDE 1.6.2. Let's talk about when to use the Remote Server Adapter, then we'll cover setting it up and using it in more detail.

Your remote Liferay Portal instance needs to satisfy two requirements to use a Remote Server Adapter:

- It is version 6.1 or later.
- It has the Remote IDE Connector application installed from Liferay Marketplace. The Remote IDE Connector contains the `server-manager-web` plugin for Liferay that provides an API for Liferay IDE's Remote Server Adapter to use for all its remote operations.

The Remote Server Adapter lets developers deploy local projects to a remote development server for testing purposes--this is its primary use case. If you're using Liferay IDE and want to deploy projects to a remote server, just make sure you have access to a remote server with the Remote IDE Connector application installed. It's possible to install the Remote IDE Connector application on a production server, but it creates an unnecessary security risk, so we don't recommend it. Clients shouldn't update, or hot-fix, remotely deployed plugins with the adapter; the portal system administrator should use normal mechanisms to apply plugin updates and fixes.

To start deploying projects to a remote server, you'll need to download and install the following resources on your local development machine:

- Download Liferay IDE from Liferay's downloads page or download Liferay Developer Studio from the Customer Portal.

2. Check the project's target runtime and facets to make sure it's configured as a *Liferay IDE* project:

2.1. In the *Package*

- Download Liferay Portal CE or EE to your local development machine.

You'll need to download Liferay Portal CE or EE to your remote (test) server as well.

Our demonstration uses the Remote Server Adapter on Liferay Portal bundled with Apache Tomcat, but you can use the adapter with Liferay Portal running on any application server Liferay Portal supports. Install Liferay Portal locally to compile the plugins you develop. Install Liferay Portal on your remote test server to host the plugins you'll deploy to it.



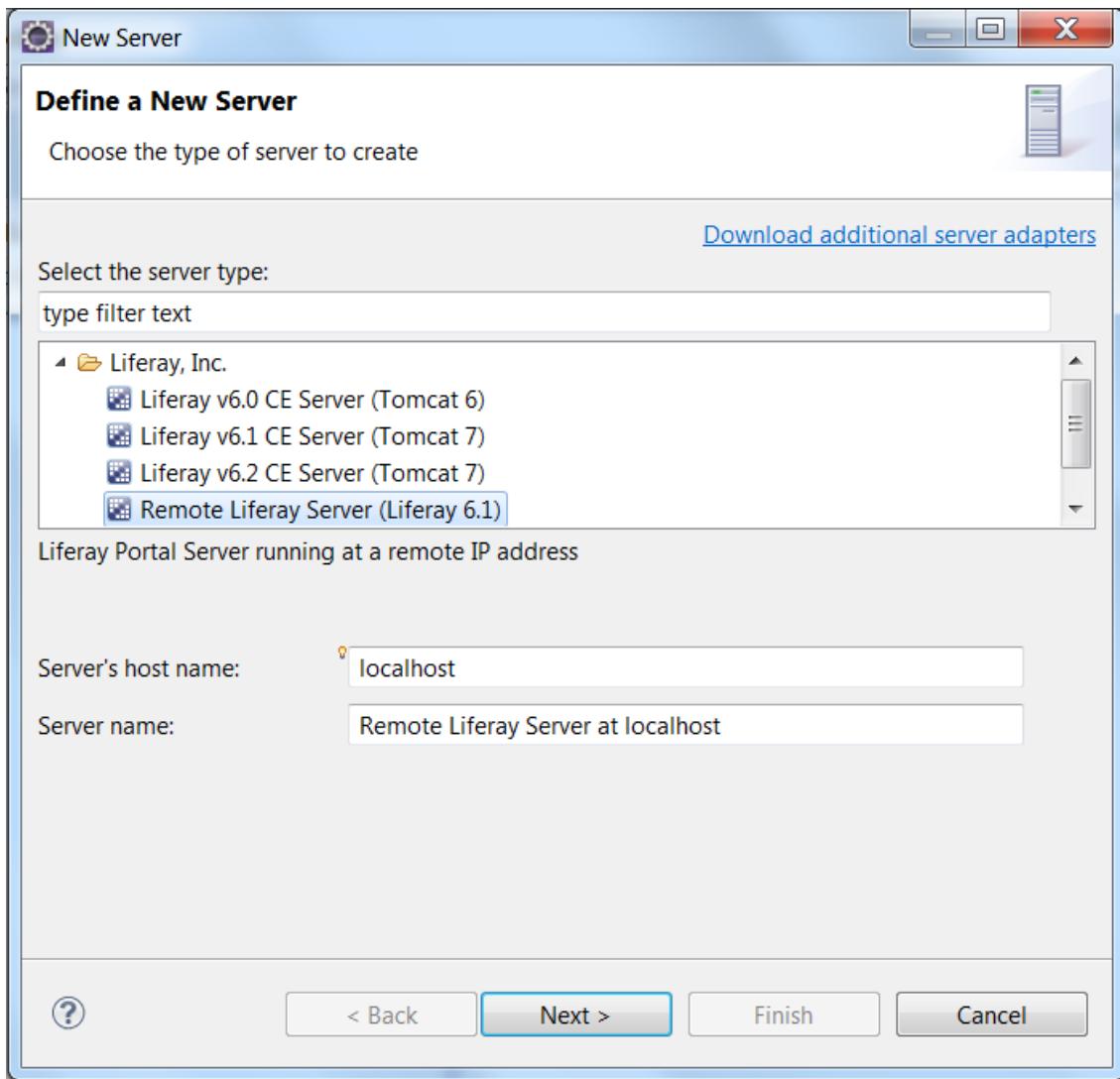
Important: Keep a record of your portal administrator login credentials (e.g., username/password) for your remote Liferay server; you'll need them to configure your connection from Liferay IDE to the remote Liferay server.

Let's start by configuring the Remote Server Adapter.

2.1.11.1. Configuring the Remote Server Adapter

You can use Liferay IDE's Remote Server wizard to configure the Remote Server Adapter and install the Remote IDE Connector to your remote Liferay instance. Alternatively, you can install the Remote IDE Connector to your remote Liferay instance before configuring Liferay IDE's Remote Server Adapter. To configure the Remote Server Adapter, use the following steps:

1. Start your remote Liferay Portal instance and verify that you can log in as an administrator.
2. Launch Liferay IDE and open the new server wizard by clicking *File* → *New* → *Other*; select *Server* in the Server category and click *Next*. Select *Remote Liferay Server (Liferay 6.2)* in the Liferay, Inc. category.
3. Enter the IP address of the machine with the remote Liferay Portal instance into the *Server's host name* field. For the *Server name*, enter *Liferay@[IP address]*, then click *Next*.



e stub for satisfying JAR dependencies needed to compile various Liferay projects. Select the *Liferay bundle type* based on the version of your local Liferay bundle, browse to the *Liferay bundle directory* and click *Next*.

5. On the next page of the wizard, configure your connection to your remote Liferay instance:
 - › **Hostname:** Enter the IP address of your remote Liferay Portal instance's machine.
 - › **HTTP Port:** Enter the port it runs on (default: 8080).
 - › **Username and Password:** Enter your administrator credentials for the remote Liferay Portal instance.

Leave the *Liferay Portal Context Path* and *Server Manager Context Path* set to the defaults unless these values were changed for your remote Liferay Portal instance.

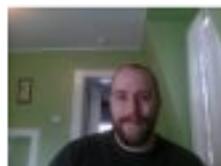
6. Your remote Liferay Portal instance needs the Remote IDE Connector application

4. The New Server wizard's next page directs you to define the Liferay Portal runtime stub. Doing so allows projects to create a stub for your remote server to use the runtime.

installed; otherwise, Liferay IDE can't connect to it. If you haven't installed Liferay IDE Connector yet, click the *Remote IDE Connector* link in the wizard. If you already downloaded the Remote IDE Connector application and installed it to your remote portal, skip to the next step and validate your connection.

Browse Liferay Marketplace for the Remote IDE Connector application. When you've found it, click *Free* to purchase it. Follow the on-screen prompts.

Once you've purchased the application, navigate to the *Purchased* page of the Control Panel's Marketplace interface.



Russell Bohl

[App Manager](#)

[Support](#)

[License](#)

App Manager

Purchased Products

We recommend that you log into your portal instance and install purchased apps to provide your portal instance with automatic update notices should they become available. You can also hot deploy by following instructions found in the FAQ.

Category

Title

Search

Product Information



[Remote IDE Conne...](#)

Liferay, Inc.



1 Rating



[Kaleo Workflow EE](#)

Liferay, Inc.



0 Ratings

Showing 2 results.

Find your application in the list of purchased products. Then click on the buttons to download

and install the application. Once it's been installed on your remote portal, return to the Remote Liferay Server configuration wizard in Liferay IDE.

7. Click the *Validate Connection* button; if no warnings or errors appear, your connection works! If you get any warning or error messages in the configuration wizard, check your connection settings.
8. Once Liferay IDE is connected to your remote Liferay Portal instance, click *Finish* in the Remote Liferay Server configuration wizard.

After you click *Finish*, the new remote server appears in Liferay IDE's *Servers* tab. This tab appears in the bottom left corner of the Eclipse window if you're using the Liferay perspective. If you entered your connection settings correctly, Eclipse connects to your remote server and displays the remote Liferay Portal instance's logs in the console. If your remote server is in debug mode, the Eclipse Java debugger is attached to the remote process automatically.

9. You can change the remote server settings at any time. Double-click on your remote server instance in the *Servers* tab to open the configuration editor, where you can modify the settings.

Now that your remote Liferay Portal server is configured, let's test the remote server adapter!

2.1.11.2. Using the Remote Server Adapter

Once your remote Liferay Portal server is correctly configured and your local Liferay IDE is connected to it, you can begin publishing projects to it and using it as you would a local Liferay Portal server.

Here's how to publish plugin projects to your remote server in Liferay IDE:

1. Right click on the server and choose *Add and Remove*....



Note: Make sure you have available projects configured in Liferay IDE. If not, you'll get an error message indicating there are no available resources to add or remove from the server.

2. Select the Liferay projects to publish to your remote server; click *Add* to add them to your remote server, then click *Finish*. Deployment begins immediately. If publication to the remote server was successful, your console displays a message that the plugin was successfully deployed!
3. As you make changes to your plugin project, republish them so they take effect on the remote server. To set your remote server's publication behavior, double click your remote server in the *Servers* tab. You can choose to automatically publish resources after changes are made, automatically publish after a build event, or never to publish automatically. To manually invoke the publishing operation after having modified project files, right click on the server in the Servers view and select *Publish*.

Next, let's examine the structure of Liferay's Plugins SDK. We'll also learn how to use it

independently of Liferay IDE.

2.2. Leveraging the Plugins SDK

Java developers use a wide variety of tools and development environments. Liferay makes every effort to remain tool agnostic, so you can choose the tools that work best for you. If you don't want to use Liferay IDE, you can use Liferay's Plugins Software Development Kit (SDK) all by itself. The Plugins SDK is based on Apache Ant and can be used along with any editor or Integrated Development Environment (IDE).

In this section, we'll explain how to set up a Plugins SDK. We'll also discuss its file structure and available Ant targets and share some best practices to help you get the most out of the Plugins SDK.

Setting up the Plugins SDK is easy. Let's get to it.

2.2.1. Installing the SDK

The first thing you should do is install Liferay Portal. If you haven't already installed a Liferay bundle, follow the instructions in the Installation and Setup chapter of *Using Liferay Portal 6.2*. Many people use the Tomcat bundle for development, as it's small, fast, and takes up fewer resources than most other servlet containers. Although you can use any application server supported by Liferay Portal for development, our examples use the Tomcat bundle.



Note: In Liferay Developer Studio, the SDK is already installed and ready to use. Liferay Portal Enterprise Edition (EE) comes with Liferay Developer Studio and much more (see CE vs. EE). Download a free trial of Liferay Portal EE today.

Installation steps:

1. Download The Plugins SDK from our web site at <http://www.liferay.com>.

Click the *Downloads* link at the top of the page.

From the *Liferay Portal 6.2 Community Edition* section, select the *Plugins SDK* option.

Click *Download*.

2. Unzip the archive to a folder of your choosing. Because some operating systems have trouble running Java applications from folders with names containing spaces, avoid using spaces when naming your folder.

On Windows, to build a plugin's services (see Generating Your Service Layer), the Plugins SDK and Liferay Portal instance must be on the same drive. E.g., if your Liferay Portal instance is on your C :\ drive, your Plugins SDK must also be on your C :\ drive in order for Service Builder to be able to run successfully.



Tip: By default, Liferay Portal Community Edition comes bundled with many plugins. It's common to remove them to speed up the server start-up. Just navigate to the

`liferay-portal-[version]/tomcat-[tomcat-version]/webapps` directory and delete all its subdirectories except for `ROOT`, `marketplace-portlet`, and `tunnel-web`.

Now that you've installed the Plugins SDK, let's configure Apache Ant for use in developing your plugins.

2.2.1.1. Ant Configuration

Building projects in the Plugins SDK requires that you install Ant (version 1.8 or higher) on your machine. Download the latest version of Ant from <http://ant.apache.org/>. Extract the archive's contents into a folder of your choosing.

Now that Ant is installed, create an `ANT_HOME` environment variable to capture your Ant installation location. Then put Ant's `bin` directory (e.g., `$ANT_HOME/bin`) in your path. We'll give you examples of doing this on Linux (Unix or Mac OS X) and Windows.

On Unix-like systems (Linux or Mac OS X), if your Ant installation directory is `/java/apache-ant-<version>` and your shell is Bash, set `ANT_HOME` and adjust your path by specifying the following in `.bash_profile` or from your terminal:

```
export ANT_HOME=/java/apache-ant-<version>
export PATH=$PATH:$ANT_HOME/bin
```

On Windows, if your Ant installation folder is `C:\Java\apache-ant-<version>`, set your `ANT_HOME` and path environment variables appropriately in your system properties:

1. Select *Start*, then right-select *Computer* → *Properties*.
2. In the *Advanced* tab, click *Environment Variables*....
3. In the *System variables* section, click *New*....
4. Set the `ANT_HOME` variable:
 - › **Variable name:** `ANT_HOME`
 - › **Variable value:** [Ant installation path] (e.g., `C:\Java\apache-ant-<version>`)

Click *OK*.

5. Also in the *System variables* section, select your path variable and click *Edit*....
6. Insert `%ANT_HOME%\bin`; after `%JAVA_HOME%\bin`; and click *OK*.
7. Click *OK* to close all system property windows.
8. Open a new command prompt for your new environment variables to take affect.

To verify Ant is in your path, execute `ant -version` from your terminal to make sure your output looks similar to this:

```
Apache Ant(TM) version <version> compiled on <date>
```

If the version information doesn't display, make sure your Ant installation is referenced in your path.

Now that Ant is configured, let's set up your Plugins SDK environment.

2.2.1.2. Plugins SDK Configuration

Now we have the proper tools set up. Next, we need to configure our Plugins SDK. It needs to know the location of our Liferay installation so it can compile plugins against Liferay's JAR files and can deploy plugins to your Liferay instance. The Plugins SDK contains a `build.properties` file that contains the default settings about the location of your Liferay installation and your deployment folder. You can use this file as a reference, but you shouldn't modify it directly (In fact, you will see the message "DO NOT EDIT THIS FILE" at the top if you open it). In order to override the default settings, create a new file named `build.[username].properties` in the same folder, where `[username]` is your user ID on your machine. For example, if your user name is `jbloggs`, your file name would be `build.jbloggs.properties`.

Edit this file and add the following lines:

```
app.server.type=[the name build.properties uses for your application server type]
app.server.parent.dir=[the directory containing your Liferay bundle]
app.server.tomcat.dir=[the directory containing your application server]
```

If you are using Liferay Portal bundled with Tomcat 7.0.42 and your bundle is in your

C:/liferay-portal-6.2 folder, you'd specify the following lines:

```
app.server.type=tomcat
app.server.parent.dir=C:/liferay-portal-6.2
app.server.tomcat.dir=${app.server.parent.dir}/tomcat-7.0.42
```

Since we're using the Tomcat application server, we specified `tomcat` as our app server type and we specified the `app.server.tomcat.dir` property. See the Plugins SDK's `build.properties` for the name of the app server property that matches your app server.

Save the file. Next, let's consider the structure of the Plugins SDK.

2.2.2. Structure of the SDK

Each folder in the Plugins SDK contains scripts for creating new plugins of that type. Here is the directory structure of the Plugins SDK:

- `liferay-plugins-<version>/` - Plugins SDK root directory.
 - `clients/` - client applications directory.
 - `dist/` - archived plugins for distribution and deployment.
 - `ext/` - Ext plugins directory. See Advanced Customization with Ext Plugins.
 - `hooks/` - hook plugins directory. See Customizing and Extending Functionality with Hooks.
 - `layouttpl/` - layout templates directory. See Creating Liferay Layout Templates.
 - `lib/` - commonly referenced libraries.
 - `misc/` - development configuration files. Example, a source code formatting specification file.

- › portlets/ - portlet plugins directory. See [Developing Portlet Applications](#).
- › themes/ - themes plugins directory. See [Creating Liferay Themes and Layout Templates](#).
- › tools/ - plugin templates and utilities.
- › webs/ - web plugins directory.
- › build.properties - default SDK properties.
- › build.<username>.properties - (optional) override SDK properties.
- › build.xml - contains targets to invoke in the SDK.
- › build-common.xml - contains common targets and properties referenced throughout the SDK.
- › build-common-plugin.xml - contains common targets and properties referenced by each plugin.
- › build-common-plugins.xml - contains common targets and properties referenced by each plugin type.

New plugins are placed in their own subdirectory of the appropriate plugin type. For instance, a new portlet called *greeting-portlet* would reside in `liferay-plugins-[version]/portlets/greeting-portlet`.

There's an Ant build file called `build.xml` in each of the plugins directories. Here are some Ant targets you'll commonly use in developing your plugins:

- `build-service` - builds the service layer for a plugin, using Liferay Service Builder.
- `clean` - cleans the residual files created by the invocations of the compilation, archiving, and deployment targets.
- `compile` - compiles the plugin source code.
- `deploy` - builds and deploys the plugin to your application server.
- `format-source` - formats the source code per Liferay's source code guidelines, informing you of violations that must be addressed. See the [Development Style](#) community wiki page for details.
- `format-javadoc` - formats the Javadoc per Liferay's Javadoc guidelines. See the [Javadoc Guidelines](#) community wiki page for details.

You're now familiar with the Plugins SDK's structure and Ant targets. Next, let's create a plugin using the Plugins SDK from a terminal environment.

2.2.3. Creating Plugins with Liferay SDK

You saw how easy it is to create and deploy Liferay plugin projects using Liferay IDE with an installed Liferay SDK. If you don't want to use Eclipse, you can still leverage the SDK to create your Liferay plugins.

Let's pretend that Harold Schnozz, Nose-ster's founder, despises Eclipse. We may not agree with his objections, but since he's paying us good money to create portlets for his organization, we'll make do without the benefit of Liferay IDE.

Navigate to the `portlets` folder of your Plugins SDK and follow these steps:

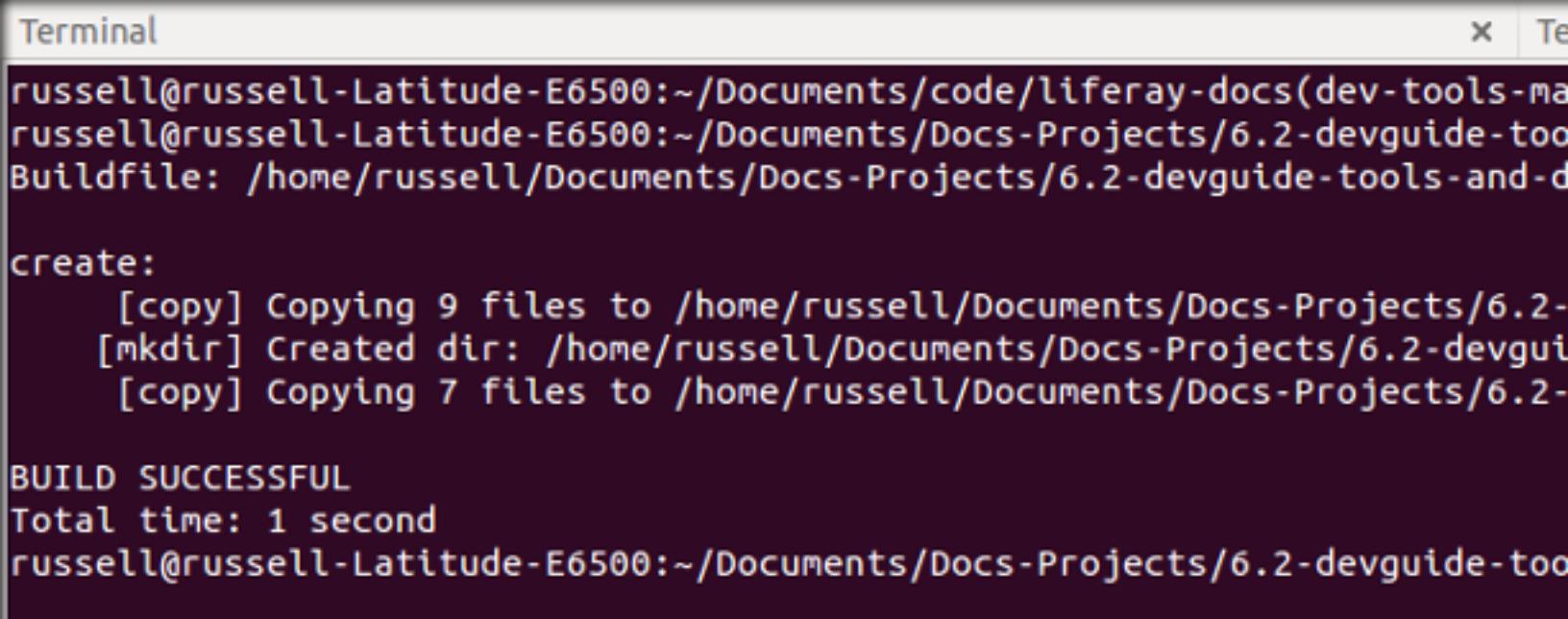
1. On Linux and Mac OS X, enter

```
./create.sh event-listing-portlet "Event Listing"
```

2. On Windows, enter

```
create.bat event-listing-portlet "Event Listing"
```

Your terminal will display a BUILD SUCCESSFUL message from Ant, and a new folder with your portlet plugin's directory structure will be created inside of the `portlets` folder in your Plugins SDK. This is where you'll work to implement your own functionality. Notice that the



The screenshot shows a terminal window titled "Terminal". The command `./create.sh event-listing-portlet "Event Listing"` is run, followed by a series of build logs. The logs show the creation of a directory structure under `/home/russell/Documents/Docs-Projects/6.2-devguide-tools-and-d`, with files being copied to `/home/russell/Documents/Docs-Projects/6.2-devguide-tools-and-d`. The output ends with "BUILD SUCCESSFUL" and a total time of 1 second.

```
russell@russell-Latitude-E6500:~/Documents/code/liferay-docs(dev-tools-manual)
russell@russell-Latitude-E6500:~/Documents/Docs-Projects/6.2-devguide-tools-and-d
Buildfile: /home/russell/Documents/Docs-Projects/6.2-devguide-tools-and-d

create:
[copy] Copying 9 files to /home/russell/Documents/Docs-Projects/6.2-devguide-tools-and-d
[mkdir] Created dir: /home/russell/Documents/Docs-Projects/6.2-devguide-tools-and-d
[copy] Copying 7 files to /home/russell/Documents/Docs-Projects/6.2-devguide-tools-and-d

BUILD SUCCESSFUL
Total time: 1 second
russell@russell-Latitude-E6500:~/Documents/Docs-Projects/6.2-devguide-tools-and-d
```

Plugins SDK automatically appends "-portlet" to the project name when creating its directory.



Tip: If you are using a source control system such as Subversion, CVS, Mercurial, Git, etc., this would be a good moment to do an initial check-in of your changes. After building the plugin for deployment, several additional files will be generated that should *not* be handled by the source control system.

Now you have a Liferay portlet project for our Event Listing portlet. We still need to deploy the project to our Liferay Server. With Liferay IDE you had multiple options: drag and drop your project onto the server, or right click the server and select *Add and Remove*.... It's almost as easy using an ant target. Simply open a terminal window in your `portlets/event-listing-portlet` directory and enter

```
ant deploy
```

A BUILD SUCCESSFUL message indicates your portlet is now being deployed. If you switch to

the terminal window running Liferay, within a few seconds you should see the message 1 portlet for my-greeting-portlet is available for use. If not, double-check your configuration.

In your web browser, log in to the portal as explained earlier. Hover over *Add* at the top of the page and click on *More*. Select the *Sample* category, and then click *Add* next to *Event Listing*. Your portlet appears in the page below.

Next, let's consider some best practices for developing plugins using the SDK.

2.2.4. Best Practices

The Plugins SDK can house all of your plugin projects enterprise-wide, or you can have separate Plugins SDK projects for each plugin. For example, if you have an internal Intranet using Liferay with some custom portlets, you can keep those portlets and themes in their own Plugins SDK project in your source code projects. Or, you can further separate your projects by having a different Plugins SDK project for each portlet or theme project.

It's also possible to use the Plugins SDK as a simple cross-platform project generator. Create a plugin project using the Plugins SDK and then copy the resulting project folder to your IDE of choice. You'll have to manually modify the Ant scripts, but this process makes it possible to create plugins with the Plugins SDK while conforming to the strict standards some organizations have for their Java projects.

Now you know of two Liferay-specific tools that streamline development in Liferay. But both use the Apache Ant build tool. If that's a deal breaker for you, consider using Liferay's Apache Maven archetypes to build your custom Liferay plugins. We'll look at Maven next, and we'll have some fun with classic poetry while doing it.

2.3. Developing Plugins Using Maven

"Once upon a midnight dreary, while I pondered weak and weary..."

Here's the scene--you're sitting in a luxurious armchair next to a dancing fire, hot beverage in hand. Shadows dance on the tapestry-covered wall, and your cat *Lenore II* is purring softly from her favorite perch atop the mantle.

"Ah, distinctly I remember it was in the bleak December..."

At least you're passing this cold December night in grand style (in front of your computer customizing Liferay Portal, of course).

"Eagerly I wished the morrow;--vainly I had sought to borrow From *Liferay* surcease of sorrow--sorrow for my last *Lenore*--"

We're sorry to hear your previous cat, the original *Lenore*, has passed away. Just take good care of *Lenore II*, would you?

"And the silken sad uncertain rustling of each purple curtain Thrilled me--filled me with *Antastic* terrors never felt before;"

Okay now you're being melodramatic; nobody can disdain *Apache Ant* that vehemently. What

about customizing Liferay Portal using the Ant-based Plugins SDK could make you feel sadness and terror?

"Deep into that darkness peering, long I stood there wondering, fearing..."

We get it! You don't want to use our Ant-based Plugins SDK. Give us surcease from the melodrama, okay?

"Open here I flung the shutter, when, with many a flirt and flutter, In there stepped a stately *Maven* of the saintly days of yore."

So, you'd rather use Apache Maven to develop your Liferay plugins?

"But *Apache Maven* still beguiling all my sad soul into smiling..."

Edgar Allen Poe liked Maven too, so you're in good company. Trust us; we know. But if your soul was made sad because you thought you had to use Liferay's Ant-based Plugins SDK to develop your plugins, Apache Maven will make your sad soul smile. And while you're at it, take care of Lenore II for all of us animal lovers, would you?

Quoth the Maven, "Let us proceed undaunted in exploration of these topics:"

- Installing Maven
- Using Maven Repositories
- Installing Required Liferay Artifacts
- Using a Parent Plugin Project
- Creating Liferay Plugins with Maven
- Deploying Liferay Plugins with Maven
- Liferay Plugin Types to Develop with Maven

As an alternative to developing plugins using the Plugins SDK, you can leverage the Apache Maven build management framework. Here's a list of some exciting Maven features:

- Offers a simple build process.
- Features a project object model.
- Has a defined project life cycle.
- Provides a dependency management system.

Maven's core installation is lightweight; there are core plugins for compiling source code and creating distributions, and there is an abundance of non-core plugins, letting you extend Maven easily for your customizations.

Many developers are switching from Ant to Maven because it offers a common interface for project builds. Maven's universal directory structure makes it easier for you to understand another developer's project more quickly. With Maven, there's a simple process to build, install, and deploy project artifacts.

Maven uses a *project object model (POM)* to describe a software project. The POM is specified as XML in a file named `pom.xml`. Think of `pom.xml` as a blueprint for your entire project; it describes your project's directories, required plugins, build sequence, and dependencies. The POM is your project's sole descriptive declaration. Once you create the `pom.xml` file and invoke the build process, Maven does the rest, downloading your project's inferred dependencies and building your project artifacts. If you're not already familiar with how Maven works, you can get familiar with Maven's project object model by reading Sonatype's documentation for it at

<http://www.sonatype.com/books/mvnref-book/reference/pom-relationships.html>.

Maven provides a clear definition of a project's structure and manages a project from a single piece of information--its POM. Understanding a Maven project can be much easier than understanding an Ant-based project's various build files. Maven forces projects to conform to a standard build process, whereas Ant projects can be built differently from project to project. Also, Maven provides an easy way to share artifacts (e.g., JARs, WARs, etc.) across projects via public repositories.

There are disadvantages to using Maven. You might find the Maven project structure too restrictive, or decide that reorganizing your projects to work with Maven is too cumbersome. Maven is intended primarily for Java-based projects, so it can be difficult to manage your project's non-Java source code. Consider Maven's advantages and disadvantages, then decide how you want to manage your projects. After you're finished reading about Maven here, you can read an in-depth book about Maven at *Maven: The Complete Reference* by Sonatype, Inc. at <http://www.sonatype.com/books/mvnref-book/reference/>.

Liferay provides Maven archetypes to help you build plugins of various types, including Liferay portlets, themes, hooks, layout templates, web plugins, and more. You can also install and deploy Liferay artifacts to your repositories. We'll dive into all these topics in this chapter.

"Straight I wheeled a cushioned seat in front of computer desk once more; Then, upon the velvet falling, I betook to Maven installing..."

2.3.1. Installing Maven

You can download Maven from <http://maven.apache.org/download.cgi>. We recommend putting your Maven installation's `bin` directory in your system's `$PATH`, so you can run the Maven executable (`mvn`) easily from your command prompt.

Let's learn about the types of repositories you can use with Maven projects.

2.3.2. Using Maven Repositories

Wouldn't it be nice to install and deploy your Liferay artifacts to a repository? Great news! Maven lets you install your artifacts to your machine's local repository and even deploy them to remote repositories; so you can share them privately with your team or with the public for general consumption. Your *local* repository holds your downloaded artifacts and those artifacts you install to it. *Remote* repositories are for sharing artifacts either privately (e.g., within your development team) or publicly. To learn more about using artifact repositories see <http://maven.apache.org/guides/introduction/introduction-to-repositories.html>.

Maven also lets you configure a proxy server; it mediates your requests to public Maven repositories and caches artifacts locally. Using a local proxy/repository helps you build projects faster and more reliably. You want this for two reasons: accessing remote repositories is slower, and remote repositories are sometimes unavailable. Most Maven proxy servers can also host private repositories that hold only your private artifacts. If you're interested in running your repository behind a proxy, see <http://www.sonatype.com/books/nexus-book/reference/install-sect-proxy.html>.

Now that you've been introduced to Maven repositories and proxy servers, let's consider using a repository management server to create and manage your Maven repositories.

2.3.2.1. Managing Maven Repositories

You'll frequently want to share Liferay artifacts and plugins with teammates, or manage your repositories using a GUI. For this, you'll want Nexus OSS. It's a Maven repository management server that facilitates creating and managing release servers, snapshot servers, and proxy servers. Release servers hold software that has met the software provider's criteria for planned features and quality. Snapshot servers hold software that is in a state of development. If you're not interested in using Nexus as a repository management server, feel free to skip this section.

Let's create a Maven repository using Nexus OSS. If you haven't already, download Nexus OSS from <http://www.sonatype.org/nexus/> and follow instructions at http://www.sonatype.com/books/nexus-book/reference/_installing_nexus.html to install and start it.

To create a 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 username is `admin` with password `admin123`.
2. Click on *Repositories* and navigate to *Add... → Hosted Repository*.

Repository	Type	Quality	Format	Policy
Public Repo	Proxy Repository	ANALYZE	maven2	
3rd party	Virtual Repository	ANALYZE	maven2	Release
Apache Snap	Repository Group	ANALYZE	maven2	Snapshot
Central	Repository Group	ANALYZE	maven2	Release
Central M1 shadow	virtual	ANALYZE	maven1	Release
Codehaus Snapshots	proxy	ANALYZE	maven2	Snapshot
Releases	hosted	ANALYZE	maven2	Release
Snapshots	hosted	ANALYZE	maven2	Snapshot



Note: To learn more about each type of Nexus repository, read Sonatype's *Managing Repositories* at <http://www.sonatype.com/books/nexus-book/reference/confignx-section-manage-repo.html>.

3. Enter repository properties appropriate to the access you'll provide its artifacts. We're installing release version artifacts into this repository, so specify *Release* as the repository policy. Below are examples of repository property values:
 - › **Repository ID:** *liferay-releases*
 - › **Repository Name:** *Liferay Release Repository*
 - › **Provider:** *Maven2*
 - › **Repository Policy:** *Release*
4. Click *Save*.

You just created a Maven repository accessible from your Nexus OSS repository server! Congratulations!

Let's create a Maven repository to hold snapshots of each Liferay plugin we create. Creating a *snapshot* repository is almost identical to creating a *release* repository. The only difference is that we'll specify *Snapshot* as its repository policy:

1. Go to your Nexus repository server in your web browser.
2. Click on *Repositories* and navigate to *Add... → Hosted Repository*.

3. Specify repository properties like the following:

- › **Repository ID:** *liferay-snapshots*
- › **Repository Name:** *Liferay Snapshot Repository*
- › **Provider:** *Maven2*
- › **Repository Policy:** *Snapshot*

4. Click *Save*.

Voila! You not only have a repository for your Liferay releases (i.e., `liferay-releases`), you also have a repository for your Liferay plugin snapshots (i.e., `liferay-snapshots`).

Let's configure your new repository servers in your Maven environment so you can install artifacts to them.

2.3.2.2. Configuring Local Maven Settings

Before using your repository servers and/or any repository mirrors, you must specify them in your Maven environment settings. Your repository settings enable Maven to find the repository and get access to it for retrieving and installing artifacts.



Note: You only need to configure a repository server if you're installing downloaded Liferay CE/EE artifacts from a zip file or if you want to share artifacts (e.g., Liferay artifacts and/or your plugins) with others. If you're automatically installing Liferay CE artifacts from the Central Repository and aren't interested in sharing artifacts, you don't need a repository server specified in your Maven settings.

However, configuring a mirror in your Maven settings is recommended as a best practice. Get more information on mirrors and their purpose in Maven's guide on mirrors at <http://maven.apache.org/guides/mini/guide-mirror-settings.html>.

To configure your Maven environment to access your `liferay-releases` repository server, do the following:

1. Navigate to your `[USER_HOME] / .m2 /` directory. Create that directory if it doesn't yet exist.
2. Open your `settings.xml` file. If it doesn't yet exist, create it.
3. Provide settings for your repository servers. Here are contents from a `settings.xml` file that has `liferay-releases` and `liferay-snapshots` repository servers configured:

```
<?xml version="1.0"?>
<settings>
  <servers>
    <server>
      <id>liferay-releases</id>
      <username>admin</username>
      <password>admin123</password>
```

```
</server>
<server>
    <id>liferay-snapshots</id>
    <username>admin</username>
    <password>admin123</password>
</server>
</servers>
</settings>
```



Note: The username `admin` and password `admin123` are the credentials of the default Nexus OSS 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 plugin artifacts you'll create!

Now, let's install the Liferay artifacts you'll need to create your plugins.

2.3.3. Installing Required Liferay Artifacts

To create Liferay plugins using Maven, you'll need the archives required by Liferay (e.g., required JAR and WAR files). This won't be problem--Liferay provides them as Maven artifacts.

So how do you get the Liferay artifacts? The exact process depends on whether you're building plugins for Liferay EE or Liferay CE. If you're building plugins for Liferay EE, you'll have to install the Liferay Artifacts manually from a zip file. You can do the same if you're building plugins for Liferay CE, but there's a simpler option available: you can automatically install CE artifacts from the Central Repository. Alternatively for Liferay CE, if you absolutely must the latest pre-release changes from our Liferay CE source repository, you can build the Liferay CE artifacts yourself. We'll demonstrate each of these options.



Note: The EE and CE zip files are a means to *install* the artifacts to a Maven repository of your choice. In the next few sections, we'll demonstrate the zip file and Central Repository installation options.

Let's look at the manual process first, by downloading and installing Liferay artifacts from a zip file.

2.3.3.1. Installing Artifacts from a Zip File

Whether you're building plugins for Liferay EE or CE, you can get the Liferay artifacts by manually installing them from a zip file.

Let's download the Liferay EE artifacts first.

Downloading a Liferay EE Artifact Zip File:

You can download the Liferay EE artifacts package from Liferay's Customer Portal. Just follow these steps:

1. Navigate to www.liferay.com and sign in.
2. Go to the Customer Portal by clicking your profile picture in the Dockbar and selecting *Customer Portal*.
3. Select *Liferay Portal* from the *Downloads* panel.
4. Inside *Filter by:*, select the appropriate Liferay version in the first field and select the *For Developers* value in the second field.

Liferay Downloads: Portal



For Developers

[Liferay Portal 6.2 EE SP1 Maven](#)

Type: For Developers Post Date: 2014-02-07
Affected Versions: 6.2 EE
Filesize: 310 MB
MD5: C20793E651A2E0323F79CC027F35690D

[Download](#) [Release Notes](#)

5. Click *Download* under the desired *Liferay Portal [Version] Maven*.

The Liferay Maven EE artifacts package downloads to your machine.

Downloading a Liferay CE Artifact Zip File:

You can download Liferay CE artifacts from SourceForge by following these steps:

1. Open your browser to *Liferay Portal* on SourceForge → <http://sourceforge.net/projects/liferay/files/Liferay%20Portal/>.
2. Select the Liferay version for which you need Maven artifacts. For example, if you need Maven artifacts for Liferay Portal 6.2.0 CE GA1, select version *6.2.0 GA1*.

Name	Modified	Size	Downloads	
Parent folder				
liferay-portal-tomcat-6.2.0-ce-ga1-2013...	2013-11-02	91 Bytes	19	 
liferay-portal-tomcat-6.2.0-ce-ga1-2013...	2013-11-02	293.2 MB	2,743	 
liferay-portal-src-6.2.0-ce-ga1-2013110...	2013-11-02	88 Bytes	9	 
liferay-portal-src-6.2.0-ce-ga1-2013110...	2013-11-02	297.2 MB	440	 
liferay-portal-sql-6.2.0-ce-ga1-2013110...	2013-11-02	88 Bytes	9	 
liferay-portal-sql-6.2.0-ce-ga1-2013110...	2013-11-02	1.9 MB	200	 
liferay-portal-resin-6.2.0-ce-ga1-201311...	2013-11-02	90 Bytes	4	 
liferay-portal-resin-6.2.0-ce-ga1-201311...	2013-11-02	311.0 MB	8	 
liferay-portal-maven-6.2.0-ce-ga1-2013...	2013-11-02	90 Bytes	9	 
liferay-portal-maven-6.2.0-ce-ga1-2013...	2013-11-02	325.6 MB	48	 
liferay-portal-jonas-6.2.0-ce-ga1-20131...	2013-11-02	90 Bytes	4	 

3. Select the appropriate zip file. The zip files use naming convention `liferay-portal-maven-[version]-[date].zip`.

The Liferay Maven CE artifacts package downloads to your machine.

You can extract the Liferay artifacts package zip file anywhere you like. The zip file not only includes the Liferay artifacts, but also includes a convenient script to install and deploy the artifacts to your repositories.

If you're using Liferay CE and you want the latest pre-release artifacts from the Liferay CE source repository, you can get them--but you'll have to build them yourself. Don't worry, it's easy. We'll show you how to build the artifacts from Liferay's source code next.

2.3.3.2. Building CE Maven Artifacts from Source

Downloading the Liferay Maven artifacts is useful if you're interested in using the artifacts for a particular release. However, if you'd like to use the very latest Liferay CE Maven artifacts, you can build them from source. To build the latest Liferay CE Maven artifacts from source, follow these steps:

1. Navigate to your local Liferay Portal CE source project. If you don't already have a local Liferay Portal CE source project on your machine, you can fork the Liferay Portal CE Github repository, found at <http://github.com/liferay/liferay-portal>, and clone it to your machine.
2. Create an `app.server.[user name].properties` file in your local Liferay Portal CE source project root directory. Specify the following properties in it:

```
app.server.type=[your application server's type. Look up your app  
server's type in the app.server.properties file in the same directory.]  
app.server.parent.dir=[your application server's parent directory]  
app.server.[type].dir=[your application server's directory]
```

For example, if you're using Liferay with Apache Tomcat 7.0.42 bundled in your `c:/bundles` folder, you'd specify the following properties:

```
app.server.type=tomcat  
app.server.parent.dir=c:/liferay-portal-6.2  
app.server.tomcat.dir=${app.server.parent.dir}/tomcat-7.0.42
```

Of course, you should specify the values appropriate to your application server and your bundle/parent directory. Note that your `app.server.[type].dir` directory doesn't need to exist yet; it is created by invoking an Ant target in the next step.

3. Run `ant -f build-dist.xml unzip-[app server name]` to unzip a copy of your preferred application server to the specified directory.

For example, to unzip Apache Tomcat to the directory specified by your `app.server.tomcat.dir` property, run:

```
ant -f build-dist.xml unzip-tomcat
```

4. Create a `releases.[user name].properties` in your local Liferay Portal CE source project root directory and specify the following properties:

```
gpg.keyname=[GPG key name]  
gpg.passphrase=[GPG passphrase]  
lp.maven.repository.url=http://localhost:8081/nexus/content/repositories/life  
ray-snapshots  
lp.maven.repository.id=liferay-snapshots
```

Of course, replace the values specified above with your own GPG and Maven repository credentials.

If you don't have GPG installed and don't have a public/private GPG key, you should visit <http://www.gnupg.org> and install GPG before continuing. Once you've installed GPG, generate a GPG key by running `gpg --gen-key` and following the instructions. Once you've generated a GPG key, you can find your GPG keyname by running `gpg --list-keys`.



Note: The `releases.[user name].properties` is not required if you only plan to install the Liferay artifacts locally and not deploy them.

5. Open a command prompt, navigate to your Liferay home directory, and build the Liferay Portal WAR file by running

```
ant -f build-dist.xml all zip-portal-war
```

6. Deploy the Liferay artifacts to your Maven repository by running

```
ant -f build-maven.xml deploy-artifacts
```

If you want the Liferay artifacts to be installed locally but don't have a remote Maven repository or don't want the artifacts to be remotely deployed, you can run the install target instead of the deploy target: `ant -f build-maven.xml install-artifacts`. The target installs the Liferay artifacts you built to your local .m2 repository (e.g., to your `[USER_HOME]/.m2/` directory).



Warning: During the process of packaging up the `javadoc.jar` files for your Liferay artifacts, your machine may experience sluggish performance or an insufficient amount of Java heap space. There are two solutions to this problem:

- *Increase the memory available for the Javadoc packaging process:* Navigate to `[Liferay home]/build.xml` and search for the `javadoc` target. Find the `maxmemory` property and increase it as desired.
- *Skip the Javadoc packaging process:* Navigate to `[Liferay home]/build-maven.xml` and find the `prepare-maven` target. Within this target, comment out the call to the `jar-javadoc` target, like below:

```
<!-- <antcall target="jar-javadoc" /> -->
```

Great! You now know how to build Liferay CE artifacts from your local portal source tree. As an alternative to building the artifacts, you may have downloaded Liferay release artifacts as a zip file. Once you've downloaded them, you'll need to install them to your Maven repository. We'll show you how to do that in the next section.

2.3.3.3. *Installing Artifacts to a Repository*

Let's install the Liferay release artifacts to your local Maven repository.

1. If you downloaded a Liferay artifacts zip file, navigate to the `liferay-portal-maven-[version]` directory. This is the root directory extracted from the Liferay artifacts zip file. If you built the artifacts from source, navigate to the time-stamped directory containing the artifacts in your Local Liferay Portal CE source project's root directory, (e.g., `liferay-portal/20121105174417071`).
2. To install the artifacts to your local repository, execute

```
ant install
```

Your console shows output from the artifacts being installed from the Liferay Maven package into your local repository, typically located in your `${USER_HOME}/.m2/repository` directory.

Your local repository now contains the Liferay artifacts required to build Liferay plugins. Wasn't that easy?

If you want to share your Liferay artifacts with teammates, you'll have to deploy them to a release repository server.

2.3.3.4. Deploying Artifacts to a Repository

You may find it worthwhile to share your Liferay artifacts with teammates.

Here's how you do it:

1. Make sure you've created a `liferay-releases` repository server to hold the Liferay Maven artifacts. If you haven't, see the *Managing Maven Repositories* section for instructions.
2. Make sure the repository that will hold your Liferay artifacts is specified as a server in Maven's `settings.xml` file. If it isn't, see the *Configuring Local Maven Settings* section for instructions on adding an entry for the server.

Here's an example setting for a repository server named `liferay-releases`:

```
<servers>
  ...
  <server>
    <id>liferay-releases</id>
    <username>admin</username>
    <password>admin123</password>
  </server>
  ...
</servers>
```

3. Navigate to the directory holding the Liferay artifacts you want to install to your Maven repository. If you unzipped the artifacts from a downloaded zip file, the artifacts are in a directory that follows the naming convention `liferay-portal-maven-[version]`. If you built the artifacts yourself, they'll be in a time-stamped directory in your `liferay-portal` repository directory.
4. Create a `build.[user name].properties` file in this directory. In the new properties file, specify values for the properties `lp.maven.repository.id` and `lp.maven.repository.url`. These refer to your repository's ID and URL, respectively.

Here are some example property values:

```
lp.maven.repository.id=liferay-releases
lp.maven.repository.url=http://localhost:8081/nexus/content/repositories/liferay-releases
```

Note, if you created a repository in Nexus OSS, as demonstrated in the section *Managing Maven Repositories*, you can specify that repository's ID and URL.

5. To deploy to your release repository server, execute
`ant deploy`

Your console shows output from the artifacts being deployed into your repository server.

To verify your artifacts are deployed, navigate to the *Repositories* page of your Nexus OSS server and select your repository. A window appears below displaying the Liferay artifacts now deployed to your repository.

The screenshot shows the Sonatype Nexus interface. On the left, there's a sidebar with tabs for 'Sonatype™ Servers' (selected), 'nexus', 'Artifact Search', 'Advanced Search', 'Views.Repositories', 'Repositories', 'Repository Targets', 'Routing', 'System Feeds', 'Security', 'Administration', and 'Help'. The main area has a 'Welcome' tab and a 'Repositories' tab. Under 'Repositories', there's a table with columns: Repository, Type, Quality, Format, and Policy. The table lists several repositories: Central (proxy, ANALYZE, maven2, Release), Central M1 shadow (virtual, ANALYZE, maven1, Release), Codehaus Snapshots (proxy, ANALYZE, maven2, Snapshot), Liferay CE Snapshots (hosted, ANALYZE, maven2, Snapshot), Liferay Release Repository (hosted, ANALYZE, maven2, Release), and Liferay Snapshot Repository (hosted, ANALYZE, maven2, Snapshot). Below the table, under 'Liferay Release Repository', there are tabs for 'Browse Index', 'Browse Storage', 'Configuration', 'Mirrors', 'Routing', and 'Summary'. The 'Browse Storage' tab is selected, showing a tree view of the repository structure: Liferay Release Repository > com > liferay > portal > portal-client, portal-impl, portal-pacl, portal-parent, portal-service, portal-web, support-tomcat, util-bridges, util-java, util-slf4j, and util-taglib.

Congratulations! You've downloaded the Liferay artifacts, installed them to your local repository, and deployed them to your release repository server for sharing with teammates.

Did you know that Liferay has its own Maven repository for artifacts? Let's learn how to install

artifacts from Liferay's repository next.

2.3.3.5. Installing Artifacts from the Liferay Repository

If you'd like to access Liferay's CE artifacts without downloading and installing a .zip file, you can configure Maven to automatically download and install them from Liferay's own repository: <https://repository.liferay.com>. The first time you use Maven to compile a Liferay plugin project, Maven automatically downloads the required artifacts from the Liferay Maven repository into your local repository, if they're not found in your local repository or any of your configured repository servers. You'll see it happen when you package your Liferay plugins.

In order to access artifacts from the Liferay Maven repository, you'll need to configure Maven to look for them there.

First, specify the Liferay Repository's credentials in your project's parent pom.xml file as follows:

```
<repositories>
    <repository>
        <id>liferay-ce</id>
        <name>Liferay CE</name>
        <url>https://repository.liferay.com/nexus/content/groups/liferay-ce</url>
        <releases><enabled>true</enabled></releases>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>
</repositories>

<pluginRepositories>
    <pluginRepository>
        <id>liferay-ce</id>
        <url>https://repository.liferay.com/nexus/content/groups/liferay-ce</url>
        <releases><enabled>true</enabled></releases>
        <snapshots><enabled>true</enabled></snapshots>
    </pluginRepository>
</pluginRepositories>
```



Important: Do not leave the Liferay repository configured when publishing artifacts to Maven Central. You must comment out the Liferay Repository credentials when publishing your artifacts.

Next, when interacting with the Liferay Repository, you'll need to use specialized commands to access it. Use the following command to access the CE repo:

```
mvn archetype:generate -DarchetypeCatalog=https://repository.liferay.com/nexus/content/groups/liferay-ce
```

Liferay makes its artifacts available on Maven's Central Repository as well. As with using Liferay's Maven repository, the Maven's Central Repository enables you to automatically

download and install Liferay Maven artifacts. Let's see how.

2.3.3.6. Installing Artifacts from the Central Repository



Important: Currently, the *GA1* Liferay Maven artifacts are not available in Maven's Central Repository. The Central Repository is only synced to Liferay's *6.2.0-RC5* release. As a current workaround to accessing Liferay's *6.2 GA1* artifacts, please reference the *Installing Artifacts from the Liferay Repository* section for setup.

Liferay offers an option for automatic download and installation of Liferay Maven artifacts publicly available on the *Central Repository*, located at <http://search.maven.org/#search|ga1|liferay maven>. They are updated with Liferay releases (e.g., 6.0.6, 6.1.1, 6.1.2, 6.1.20, 6.1.30, 6.2.0-RC5, etc.). The first time you use Maven to compile a Liferay plugin project, Maven automatically downloads the required artifacts from the Central Repository into your local repository if they're not found in your local repository or any of your configured repository servers. You'll see it happen when you package your Liferay plugins.

Now that we have our Maven artifacts set up, let's configure Liferay IDE with Maven.

2.3.4. Using Liferay IDE with Maven

Wouldn't it be nice if you could manage your Liferay Maven projects from Liferay IDE? You can! Liferay IDE 2.0 introduces the Maven project configurator (`m2e-liferay`), or the added support of configuring Maven projects as full Liferay IDE projects. Let's explore what the Maven project configurator does, how to install it, and how to install its dependencies.

2.3.4.1. Installing Maven Plugins for Liferay IDE

In order to properly support Maven projects in the IDE, you first need a mechanism to recognize Maven projects as Liferay IDE projects. IDE projects are recognized in Eclipse as faceted web projects that include the appropriate Liferay plugin facet. Therefore, all IDE projects are also Eclipse web projects (faceted projects with web facet installed). In order for the IDE to recognize the Maven project and for it to be able to leverage Java EE tooling features (e.g., the *Servers* view) with the project, the project must be a flexible web project. Liferay IDE requires that the following Eclipse plugins be installed in order for Maven projects to meet these requirements:

- `m2e-core` (Maven integration for Eclipse)
- `m2e-wtp` (Maven integration for WTP)

When you install the `m2e-liferay` plugin, these dependencies are installed by default. We'll flesh out the installation process soon, but first, let's get a deeper understanding of how these plugins work to give us our IDE/Maven compatibility.

The `m2e-core` plugin is the standard Maven tooling support for Eclipse. It provides dependency resolution classpath management and an abstract project configuration framework for adapters. Also, in order for a Liferay Maven project to be recognized as a flexible web

project, the Maven project must be mapped to a flexible web project counterpart. The `m2e-wtp` plugin provides project configuration mapping between the Maven models described in the Maven project's POMs and the corresponding flexible web project supported in Eclipse. With these integration features in place, the only remaining requirement is making sure that the `m2e-core` plugin can recognize the extra lifecycle metadata mappings necessary for supporting Liferay's custom goals. Let's break down the lifecycle mappings just a bit to get a better understanding of what this means.

Both Maven and Eclipse have their own standard build project lifecycles that are independent from each other. Therefore, for both to work together and run seamlessly within Liferay IDE, you need a lifecycle mapping to link both lifecycles into one combined lifecycle. Normally, this would have to be done manually by the user. However, with the `m2e-liferay` plugin, the lifecycle metadata mapping and Eclipse build lifecycles are automatically combined providing a seamless user experience. The lifecycle mappings for your project can be viewed by right-

Plugin execution	Mapping
<code>clean:clean</code>	ignore
<code>liferay:build-css</code>	configurator
<code>resources:resources</code>	execute
<code>compiler:compile</code>	configurator,configurator,configurator,configurator,configurator
<code>resources:testResources</code>	execute
<code>compiler:testCompile</code>	configurator,configurator,configurator,configurator
<code>surefire:test</code>	ignore
<code>war:war</code>	configurator,configurator,configurator,configurator,configurator,configurator
<code>install:install</code>	ignore
<code>deploy:deploy</code>	ignore
<code>site:site</code>	ignore

clicking your project and selecting *Properties* → *Maven* → *Lifecycle Mapping*.

When first installing Liferay IDE, the installation startup screen lets you select whether you'd like to install the Maven plugins automatically. Did you miss this during setup? No problem! To install the required Maven plugins, navigate to *Help* → *Install New Software*. In the *Work with*

field, insert the following: Liferay IDE repository - <http://releases.liferay.com/tools/ide/latest/milestone/>.

If the m2e-liferay plugin does not appear, this means you already have it installed. To verify, uncheck the *Hide items that are already installed* checkbox and look for m2e-liferay in the list of installed plugins. Also, if you'd like to view everything that is bundled with the m2e-liferay plugin, uncheck the *Group items by category* checkbox.

The screenshot shows the Eclipse Marketplace interface. At the top, there's a toolbar with icons for 'Install' and 'Search'. Below that is a header bar with tabs for 'Available Software' and 'Marketplace'. The main area is titled 'Available Software' with the sub-instruction 'Check the items that you wish to install.' A search bar at the top right contains the URL 'Work with: Liferay IDE repository - http://releases.liferay.com/tools/ide/latest/milestone/'. To the right of the search bar is an 'Add...' button. Below the search bar is a link 'Find more software by working with the ["Available Software Sites"](#) preferences'. There's also a 'type filter text' input field. The main list is a table with columns 'Name' and 'Version'. It shows three items:

Name	Version
Liferay IDE	2.0.0.201311271605-m3
Liferay IDE	2.0.0.201311271605-m3
m2e-liferay - Maven Integration for Liferay IDE	2.0.0.201311271605-m3

Awesome! The required Maven plugins are installed and your IDE instance is ready to be mavenized! Next, let's learn how to configure an existing Maven project.

2.3.4.2. Configuring your Liferay Maven Project

Now your Liferay IDE instance is Maven-ready and you have an existing Maven project. Let's investigate what is going on under the hood and configure your project. Note, if you'd like to learn how to create a new Maven project in the IDE, visit the *Creating Liferay Plugins with Maven* section. Furthermore, you can import an existing Maven project by navigating to *File → Import → Maven* and selecting the location of your Maven project source code.



Note: Due to the lifecycle mapping of Eclipse and Maven, it is unsafe to manually insert or overwrite the .classpath and .project files and .settings folder. IDE automatically generates these files when a project is imported and updates them appropriately.

The `m2e-core` plugin delegates your Liferay Maven plugin's project configuration to the `m2e-liferay` project configurator. The `m2e-wtp` project configurator then converts your Liferay WAR package into an Eclipse flexible web project. Next, the `m2e-liferay` configurator looks for the Liferay Maven plugin to be registered on the POM effective model for WAR type packages. If no Liferay Maven plugin is configured on the effective POM for the project, project configuration ceases. If the plugin is configured, the project configurator validates your project's configuration, checking its POM, parent POM, and the project's properties. The configurator detects invalid properties and reports them as errors in the IDE's POM editor. There are a list of key properties that your project must specify in order for it to become a valid Liferay IDE project. The next section titled *Using a Parent Plugin Project* identifies these properties and explains how they are used.

There are various ways to satisfy these properties--the Maven profile in the `Global settings.xml` file (recommended), in the `User settings.xml` file, in the parent `pom.xml`, or in the project `pom.xml` directly. You can think of these choices as a hierarchy for how your Maven plugins receive their properties. The project `pom.xml` overrides the parent `pom.xml`, the parent `pom.xml` overrides the `User settings.xml` file, and the `User settings.xml` file overrides the `Global settings.xml` file.

Global settings.xml: provides configuration for all plugins belonging to all users on a machine. This file resides in the `${MAVEN_HOME} /conf/settings.xml` directory.

User settings.xml: provides configuration for all plugins belonging to a single user on a machine. This file resides in the `${USER_HOME} /.m2/settings.xml` directory.

Parent pom.xml: provides configuration for all modules in the parent project.

Project pom.xml: provides configuration for the single plugin project.

Note that if a profile is active from your `settings.xml`, its values will override your properties in a POM. If you'd like to specify the properties in a POM, see the next section *Using a Parent Plugin Project* for more details.

Here's an example of what a Maven profile looks like inside the `settings.xml` file.

```
<profiles>
    <profile>
        <id>sample</id>
        <properties>
            <plugin.type>portlet</plugin.type>
            <liferay.version>6.2.0</liferay.version>

            <liferay.maven.plugin.version>6.2.0</liferay.maven.plugin.version>
                <liferay.auto.deploy.dir>E:\liferay-portal-tomcat-6.2.0-ce-ga1\deploy</liferay.auto.deploy.dir>
                    <liferay.app.server.deploy.dir>E:\liferay-portal-tomcat-6.2.0-ce-ga1\tomcat-7.0.42\webapps</liferay.app.server.deploy.dir>
                        <liferay.app.server.lib.global.dir>E:\liferay-portal-tomcat-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-tomcat-
6.2.0-ce-ga1\tomcat-7.0.42\webapps\ROOT</liferay.app.server.portal.dir>
    </properties>
</profile>
</profiles>

```

Once you have a Maven profile configured in the \${USER_HOME} / .m2/settings.xml file, you can activate the profile by right-clicking on your project → *Properties* → *Maven* and entering the profile IDs that supply the necessary settings into the *Active Maven Profiles* text field. For example, to reference the profile and properties we listed above, you'd enter *sample* for the Active Maven Profile. Once you've specified all the values, the configurator (m2e-liferay) validates the properties. If there are errors in the pom.xml file, the configurator marks them in Liferay IDE's POM editor. If you fix a project error, update the project to persist the fix by right-clicking the project → *Maven* → *Update Project*.

After your POM configuration meets the requirements, the configurator installs the Liferay plugin facet and your Maven project is officially a Liferay IDE project!

Once you have your Maven project configured, you may want to execute a specific Maven goal such as liferay:build-lang or liferay:build-db that is associated with your build phase. To access your project's Maven goals and execute them, right-click your project → *Liferay* → *Maven* and select the goal to execute. To learn more about Maven's build lifecycle and plugin goals, visit Apache's Build Lifecycle Basics guide.

When working with your pom.xml file in the IDE, you'll notice several different viewing modes to work with:

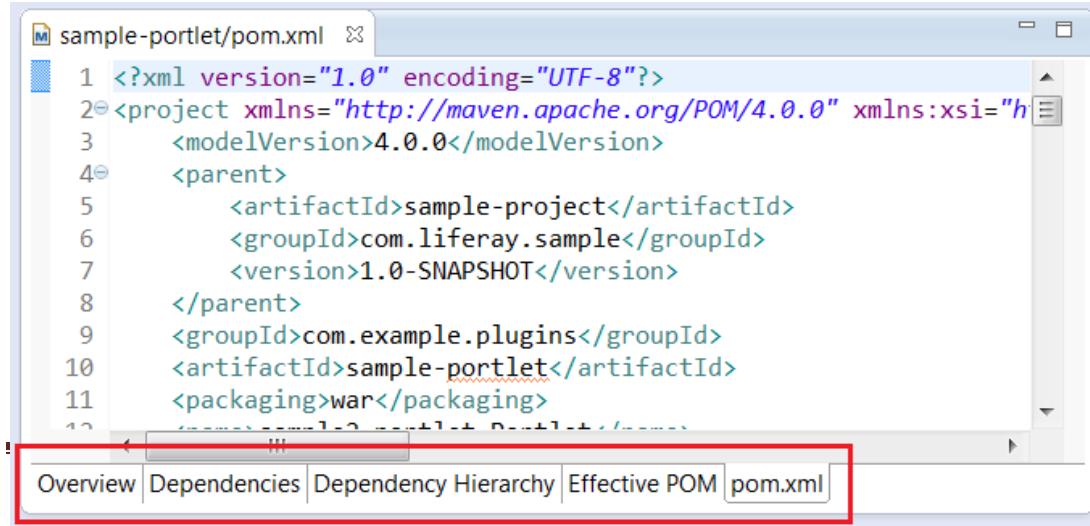
pom.xml: provides an editable POM as it appears on the file system.

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.

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.

Dependency Hierarchy: provides hierarchical view of project dependencies and interactive list of resolved dependencies.



By taking advantage of these interactive modes, modifying and

organizing your POM and its dependencies has never been easier!

Next, we'll consider the benefits of using a Maven parent project with your plugin projects.

2.3.5. Using a Parent Plugin Project

Maven supports project inheritance. You can create a *parent* project that contains properties child projects have in common, and *child* projects inherit those properties from the parent project. This saves time, since you don't need to specify those properties in each project. If you develop more than one project, it makes sense to leverage project inheritance so that all projects can share properties they have in common.

Our example demonstrates project inheritance; we'll build a project with a parent/child relationship. Even if you're not going to leverage Maven's project inheritance capabilities when you build your Liferay plugins with Maven, the process is the same for creating any Liferay plugin with Maven's Liferay artifacts. For more information on project inheritance, see Maven's documentation at <http://maven.apache.org/pom.html#Inheritance>.

We'll create our parent project and then specify the general settings needed to build your plugins for Liferay. The parent project is similar to the project root of the Liferay Plugins SDK. Its `pom.xml` file can specify information to be used by any plugin projects that refer to it. You can always specify information in each plugin's POM, but it's more convenient to use the parent project's POM for sharing common information.

Let's create a parent project named `sample-parent-project`. Start by creating a new directory for your parent project. For this example, we'll name the directory `sample-parent-project`. You can place the directory anywhere on your file system.

Next, create a POM file named `pom.xml` in your `sample-parent-project` directory. Insert the following XML code into the 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>com.liferay.sample</groupId>
  <artifactId>sample-parent-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <name>sample-parent-project</name>
  <url>http://www.liferay.com</url>

  <properties>
    <liferay.app.server.deploy.dir>
      ${liferay.app.server.deploy.dir}
    </liferay.app.server.deploy.dir>

    <liferay.app.server.lib.global.dir>
```

```

        ${liferay.app.server.lib.global.dir}
    </liferay.app.server.lib.global.dir>

    <liferay.app.server.portal.dir>
        ${liferay.app.server.portal.dir}
    </liferay.app.server.portal.dir>

    <liferay.auto.deploy.dir>
        ${liferay.auto.deploy.dir}
    </liferay.auto.deploy.dir>

    <liferay.version>
        ${liferay.version}
    </liferay.version>

    <liferay.maven.plugin.version>
        ${liferay.maven.plugin.version}
    </liferay.maven.plugin.version>
</properties>

<dependencies>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>portal-client</artifactId>
        <version>6.2.0-GA1</version>
    </dependency>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>portal-impl</artifactId>
        <version>6.2.0-GA1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>portal-pacl</artifactId>
        <version>6.2.0-GA1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>portal-service</artifactId>
        <version>6.2.0-GA1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>portal-web</artifactId>
        <version>6.2.0-GA1</version>
        <type>war</type>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>util-bridges</artifactId>

```

```

        <version>6.2.0-GA1</version>
    </dependency>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>util-java</artifactId>
        <version>6.2.0-GA1</version>
    </dependency>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>util-slf4j</artifactId>
        <version>6.2.0-GA1</version>
    </dependency>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>util-taglib</artifactId>
        <version>6.2.0-GA1</version>
    </dependency>
</dependencies>

</project>
```

The POM starts by specifying the model version that Maven supports, your project's Maven coordinates, your project's name, and your company's URL.

Next, the POM specifies some key Liferay property elements that your plugins require in order to be deployed to your Liferay portal. You can conveniently specify these values in a parent project for all of your plugin projects to leverage. A plugin project can override any of its parent's properties by specifying the desired property explicitly in the child plugin project's POM.

Replace each Liferay property value (e.g., replace

`${liferay.app.server.deploy.dir}` and other dereferenced `liferay.*` properties) with the appropriate value based on your Liferay environment. We've described these key properties here:

- `liferay.app.server.deploy.dir`: Your app server's deployment directory.
- `liferay.app.server.lib.global.dir`: Your app server's global library directory.
- `liferay.app.server.portal.dir`: The path to Liferay's deployment directory on the app server.
- `liferay.auto.deploy.dir`: The path of your Liferay bundle's hot-deploy directory `deploy/`. By specifying your Liferay instance's deploy directory in the POM, you're telling Maven exactly where to deploy your plugin artifacts.
- `liferay.maven.plugin.version`: The version of the Liferay Maven Plugin you are using.
- `liferay.version`: The version of Liferay you are using.

Here's an example where we've specified these *properties* for Liferay bundled with Apache Tomcat in a directory `C:\liferay-portal-6.2`:

```

<properties>
    <liferay.app.server.deploy.dir>
        C:\liferay-portal-6.2\tomcat-7.0.42\webapps
    </liferay.app.server.deploy.dir>
```

```

<liferay.app.server.lib.global.dir>
  C:\liferay-portal-6.2\tomcat-7.0.42\lib\ext
</liferay.app.server.lib.global.dir>

<liferay.app.server.portal.dir>
  C:\liferay-portal-6.2\tomcat-7.0.42\webapps\root
</liferay.app.server.portal.dir>

<liferay.auto.deploy.dir>
  C:\liferay-portal-6.2\deploy
</liferay.auto.deploy.dir>

<liferay.maven.plugin.version>
  6.2.0-GA1
</liferay.maven.plugin.version>

<liferay.version>
  6.2.0-GA1
</liferay.version>
</properties>

```

You can also specify these key properties in your `Global or User settings.xml` file. To learn more about this method, visit the *Configuring Your Liferay Maven Project* section.

The Liferay plugins you develop depend on several Liferay artifacts. We've included them in individual dependency elements within the POM's `dependencies` element. All of your parent project's modules (i.e., projects that refer to this parent) can leverage these dependencies.



Note: You could just as easily include such dependencies in the POM of each of your plugin projects, but specifying them in a parent project makes them accessible to child projects through inheritance.

Now that you specified your project's general information, your Liferay environment properties, and the Liferay artifacts on which Liferay plugin projects depend, let's create a plugin project using Liferay's archetypes.

2.3.6. Creating Liferay Plugins with Maven

Liferay offers many archetypes to help create Maven projects for multiple plugin types, including portlet, theme, hook, and layout template plugins. We provide archetypes for each of these plugin types for various versions of Liferay, so you almost certainly have the archetype you need.

Archetype is Maven's project templating toolkit. Let's use it to create a Liferay portlet project. With Archetype, you can use the same steps we detail below to generate Liferay plugin projects of any type.



Note: Make sure Maven is installed and that its executable is in your \$PATH environment variable.

We'll demonstrate two ways of creating Liferay plugins with Maven: using Liferay IDE and using the command line. First, let's learn how to use Maven archetypes to generate a Liferay plugin project using Liferay IDE:

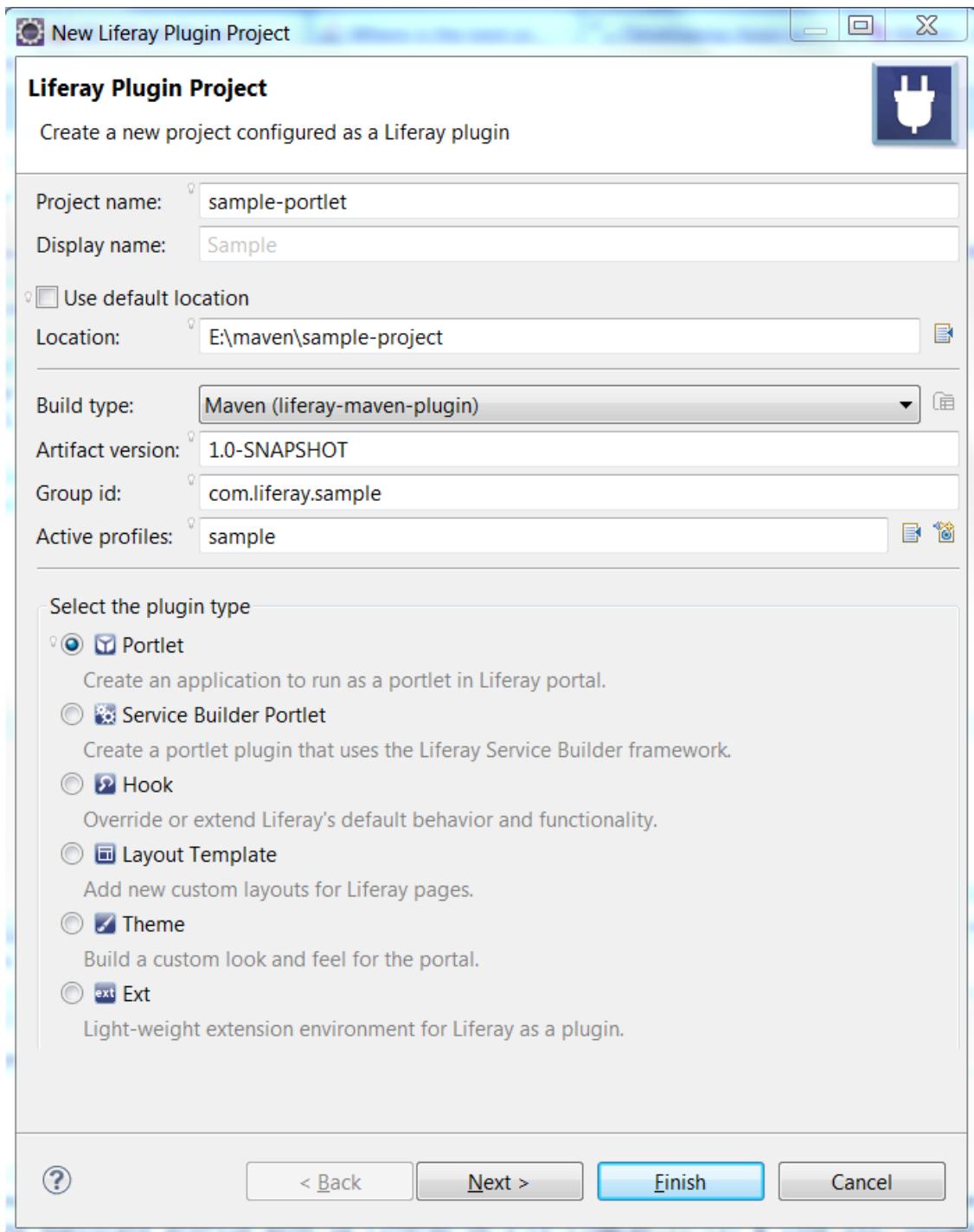
1. Navigate to *File* → *New* → *Liferay Plugin Project*.
2. Assign a project name and display name. For our example, we'll use *sample-portlet* and *Sample* for the project name and display name, respectively. Notice that upon entering *sample-portlet* as the project name, the wizard conveniently inserts *Sample* in grayed-out text as the portlet's default display name. The wizard derives the default display name from the project name, starts it in upper-case, and leaves off the plugin type suffix *Portlet* because the plugin type is automatically appended to the display name in Liferay Portal. The IDE saves the you from repetitively appending the plugin type to the display name; in fact, the IDE ignores any plugin type suffix if you happen to append it to the display name.
3. Select *Maven (liferay-maven-plugin)* for the build type. Notice that some of the options for your plugin project changed, including the *Location* field, which is set to the user's workspace by default. Choose the parent directory in which you want to create the plugin project. It is a best practice to create a parent project for your Maven plugins, so they can all share common project information. See section *Using a Parent Plugin Project* for details.
4. Specify the *Artifact version* and *Group id*. For our example, we'll use *1.0-SNAPSHOT* and *com.liferay.sample* for the artifact version and group id, respectively.
5. Specify the active profile that you'd like your Liferay plugin project to use. If you don't remember your active profile or haven't created one, click the *Select Active Profiles ...* icon immediately to the right of the text field. If you have any active profiles, they will be listed in the menu on the left. To select an existing profile, highlight its profile ID and select the illuminated *right arrow* button to transfer it to the menu on the right. Otherwise, if you don't have any existing profile, click the green *addition* button to create a profile and give it an ID.

If you're specifying a new profile, the wizard will still create your plugin, but it will need further attention before it is deployable. You'll need to specify the necessary properties within the new profile; we'll demonstrate specifying these properties in the *Configuring your Liferay Maven Project* section of this chapter.

You also have the option to create a profile based on a Liferay runtime. To do this, select the *Create New Maven Profile Based on Liferay Runtime* button to the far right of the *Active profiles* text field. Specify the *Liferay runtime*, *New profile id*, and *Liferay version*. For the new profile location you can choose to specify your profile in the *settings.xml* (recommended) or your project's *pom.xml*. When creating your Maven profile based on a Liferay runtime, the IDE automatically populates the new profile with the required properties, and no additional profile

configuration is needed for the plugin.

6. Select the *Portlet* plugin type and then click *Finish*.



Great!
You've
success-
fully
created
a
Liferay
portlet
project
using
Maven
in
Liferay
IDE!
Next,
let's
run
through
steps
for
creat-
ing your
Liferay
Maven
plugins
using
the
comma-
nd line.

1. O
pen the
comma-
nd
prompt
and
navigat
e to the
parent

directory in which you want to create the plugin project. Archetype will create a sub-directory for the plugin project you create.



Note: If you haven't already created a parent project, you may want to consider creating one to share common project information. See section *Using a Parent Plugin Project* for details.

2. Execute the command

```
mvn archetype:generate -  
DarchetypeCatalog=https://repository.liferay.com/nexus/content/groups/liferay  
-ce
```



Important: Currently, the new GA1 artifacts for CE and EE are only available from repository.liferay.com. Therefore, you must use the `-DarchetypeCatalog=...` portion to access the Liferay Repository. You'll also need to configure a couple other files to ensure the generation command completes successfully. Reference the *Installing CE Artifacts from the Central Repository* and *Installing EE Artifacts from the Liferay Repository* sections to configure Maven to access the Liferay Repository for CE and EE artifacts, respectively.

Archetype starts and lists the archetypes available to you. You're prompted to *choose* an archetype or *filter* archetypes by group / artifact ID. The output looks similar to the following text:

```
...  
4: https://repository.liferay.com/nexus/content/groups/liferay-ce/ ->  
com.liferay.  
maven.archetypes:liferay-portlet-jsf-archetype  
(Provides an archetype to create Liferay JSF portlets.)  
5: https://repository.liferay.com/nexus/content/groups/liferay-ce/ ->  
com.liferay.  
maven.archetypes:liferay-layouttpl-archetype  
(Provides an archetype to create Liferay layout templates.)  
6: https://repository.liferay.com/nexus/content/groups/liferay-ce/ ->  
com.liferay.  
maven.archetypes:liferay-portlet-archetype  
(Provides an archetype to create Liferay portlets.)  
7: https://repository.liferay.com/nexus/content/groups/liferay-ce/ ->  
com.liferay.  
maven.archetypes:liferay-portlet-liferay-faces-alloy-archetype  
(Provides an archetype to create Liferay Faces Alloy portlets.)  
8: https://repository.liferay.com/nexus/content/groups/liferay-ce/ ->  
com.liferay.  
maven.archetypes:liferay-portlet-primefaces-archetype  
(Provides an archetype to create Liferay PrimeFaces portlets.)  
...  
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains):
```

3. Choose a Liferay portlet archetype by entering its number. Since we're using the Liferay Repository, the newest archetype version is automatically selected. (6.2-GA1).
4. Enter values for the *groupId*, *artifactId*, *version*, and *package* coordinates (properties) of your project. Here are some examples:

```
groupId: com.liferay.sample
artifactId: sample-portlet
version: 1.0-SNAPSHOT
package: com.liferay.sample
```

```
Define value for property 'groupId': : com.liferay.sample
Define value for property 'artifactId': : sample-portlet
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': com.liferay.sample: :
Confirm properties configuration:
groupId: com.liferay.sample
artifactId: sample-portlet
version: 1.0-SNAPSHOT
package: com.liferay.sample
Y: :
```

This process is illustrated in the snapshot below:

For more information on defining

Maven coordinates, see http://maven.apache.org/pom.html#Maven_Coordinates.

5. Enter the letter *Y* to confirm your coordinates.

Maven's Archetype tool creates a Liferay plugin project directory with a new *pom.xml* file and source code.



Note: The archetype file is downloaded and installed automatically to your local repository (e.g., *.m2/repository/com/liferay/maven/archetypes/[archetype]*). If you configured a mirror pointing to your public repository on Nexus, the plugin is installed there.

Following these steps using Liferay IDE or the command line, you can use Archetype to generate all your Liferay plugin projects!

Plugin projects generated from a Liferay archetype are equipped with a POM that's ready to work with a parent project. It inherits the values for *liferay.version* and *liferay.auto.deploy.dir* properties from the parent.

When your plugin is created, you can package and deploy your project to a specified Liferay instance. You can even install and deploy the individual plugin to a remote repository.

Next we'll go through some brief examples to demonstrate deploying your plugins to Liferay Portal using Maven.

2.3.7. Deploying Liferay Plugins with Maven

With Maven it's easy to deploy plugins to a Liferay Portal instance. Just follow these steps:

1. Make sure you've specified the Liferay specific properties (e.g., those properties starting with `liferay.`) your plugin uses. See this chapter's section *Using a Parent Plugin Project* for descriptions of these Liferay properties.

Here's an example where we specified these *properties* for Liferay bundled with Apache Tomcat in a directory C:\liferay-portal-6.2:

```
<properties>
    <liferay.app.server.deploy.dir>
        C:\liferay-portal-6.2\tomcat-7.0.42\webapps
    </liferay.app.server.deploy.dir>

    <liferay.app.server.lib.global.dir>
        C:\liferay-portal-6.2\tomcat-7.0.42\lib\ext
    </liferay.app.server.lib.global.dir>

    <liferay.app.server.portal.dir>
        C:\liferay-portal-6.2\tomcat-7.0.42\webapps\root
    </liferay.app.server.portal.dir>

    <liferay.auto.deploy.dir>
        C:\liferay-portal-6.2\deploy
    </liferay.auto.deploy.dir>

    <liferay.maven.plugin.version>
        6.2.0-GA1
    </liferay.maven.plugin.version>

    <liferay.version>
        6.2.0-GA1
    </liferay.version>
</properties>
```

2. In your command prompt, navigate to your Liferay plugin project's directory.

3. Package your plugin by entering

```
mvn package
```

Your command output should be similar to the following output:

```
[INFO] Building war:
E:\liferay-plugins-maven\sample-parent-project\sample-portlet\target\sample-
portlet-1.0-SNAPSHOT.war
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

4. Deploy the plugin into your Liferay bundle by entering

```
mvn liferay:deploy
```

The command output should look similar to the following output:

```
[INFO] Deploying sample-portlet-1.0-SNAPSHOT.war to [liferay version]\deploy
[INFO] -----
[INFO] BUILD SUCCESS
```

[INFO] -----

Your Liferay console output shows your plugin deploying. It looks like this:

```
INFO: Deploying web application directory [liferay version]\[tomcat version]\webapps\sample-portlet
INFO  [pool-2-thread-2] [HotDeployImpl:178] Deploying sample-portlet from queue
INFO  [pool-2-thread-2] [PluginPackageUtil:1033] Reading plugin package for sample-portlet
```



Note: If you get the following error after executing `mvn liferay:deploy`, make sure you're executing the command from your plugin's directory (e.g., `sample-portlet`)--not your parent project's directory.

```
[ERROR] No plugin found for prefix 'liferay' in the current project and in the plugin groups [org.apache.maven.plugins, org.codehaus.mojo] available from the repositories [local (C:\Users\cdhoag\.m2\repository), central (http://repo.maven.apache.org/maven2)] -> [Help 1]
```

5. If you're deploying the plugin to a release or snapshot repository, specify the repository by adding a distribution management section to your plugin's `pom.xml` or your parent project's `pom.xml`.

Here's an example distribution management section for a snapshot repository:

```
<distributionManagement>
    <repository>
        <id>liferay-releases</id>
        <url>http://localhost:8081/nexus/content/repositories/liferay-releases</url>
    </repository>
    <snapshotRepository>
        <id>liferay-snapshots</id>
        <name>Liferay Snapshots Repository</name>
        <url>http://localhost:8081/nexus/content/repositories/liferay-snapshots</url>
    </snapshotRepository>
</distributionManagement>
```

The screenshot shows the Liferay Release Repository interface. At the top, there are tabs: Browse Storage, Browse Index, Configuration, Mirrors, Summary (which is selected and highlighted in blue), and Artifact Upload. Below the tabs, there's a section titled "Repository Information" containing details about the repository ID, name, type, policy, format, and groups. At the bottom, there's a "Distribution Management" section containing the XML code for the distribution management configuration.

The proper contents for your `<distributionManagement>` element can be found in the *Summary* tab for each of your repositories.

Since you created the plugin as a snapshot, you'll have to

deploy it to a snapshot repository. You can deploy a plugin as a release, but the plugin's POM must specify a valid release version (e.g., <version>1.0</version>), not a snapshot version (e.g., <version>1.0-SNAPSHOT</version>).

6. Deploy your plugin into your specified Nexus repository:

```
mvn deploy
```

Your plugin is now available in your Nexus repository!



Note: There are three build phases you'll use when developing plugins with Maven:

- In Maven's *compile* phase, explicit dependencies are downloaded to your local repository (i.e., .m2/repository/com/liferay/portal).
- In Maven's *package* phase, the plugin's inferred dependencies are downloaded to your local repository (i.e., .m2/repository).
- In Maven's *install* phase, your plugin is installed to your local repository.

Now that you've deployed a plugin using Maven, let's consider the types of Liferay plugins you can develop with Liferay Maven archetypes.

2.3.8. Liferay Plugin Types to Develop with Maven

You can develop all Liferay plugin types with Maven: portlets, themes, layout templates, hooks, and Ext. Next, you'll learn how to create each plugin type using Maven, and we'll point out where each plugin's directory structure is different than if you created it using the Plugins SDK. We'll often refer to the previous sections for creating and deploying these plugin types in Maven using Liferay artifacts. We'll also reference sections of some other chapters in this guide, since they're still relevant to Maven developers: they explain how you develop each type of plugin regardless of development environment.

Let's start with portlet plugins.

2.3.8.1. Creating a Portlet Plugin

To create a Liferay portlet plugin project, follow the *Creating Liferay Plugins with Maven* section.



Tip: As you use Maven's Archetype tool to generate your portlet project, you can filter on group ID `liferay`, or even the group ID/artifact ID combination `liferay:portlet`, to find the Liferay portlet archetypes more easily.

2.3.8.1.1. Anatomy

A portlet project created from the `com.liferay.maven.archetypes:liferay-portlet-archetype` has the following directory structure:

- `portlet-plugin/`

```

    > src/
      - main/
        • java/
        • resources/
        • webapp/
          > css/
            - main.css
          > js/
            - main.js
          > WEB-INF/
            - liferay-display.xml
            - liferay-plugin-package.properties
            - liferay-portlet.xml
            - portlet.xml
            - web.xml
          > icon.png
          > view.jsp
      > pom.xml

```

Maven creates the `src/main/java/` directory automatically. It holds the portlet's Java source code (e.g., `com.liferay.sample.SamplePortlet.java`), and `portlet-plugin/src/main/webapp` holds its web source code. If you've created any portlet plugins using the Plugins SDK, you might have noted it uses a different directory structure.

The following table illustrates the differences in location of the Java source and web source code for a Maven project and a Plugins SDK project:

Location	Maven project	Plugins SDK project
Java source	<code>src/main/java</code>	<code>docroot/WEB-INF/src</code>
Web source	<code>src/main/webapp</code>	<code>docroot</code>

To view the full directory structure of a portlet developed by Ant, visit our Anatomy of a Portlet section in this guide.

2.3.8.1.2. Deployment

To deploy your portlet plugin, follow the instructions detailed above in *Deploying Liferay Plugins with Maven*.

Congratulations! You successfully created a Liferay portlet plugin using Maven.

2.3.8.1.3. More Information

For detailed information on creating portlet plugins, see Developing Portlet Applications.

Next, let's run through a brief example for developing a theme plugin using Maven.

2.3.8.2. Developing Liferay Theme Plugins with Maven

So you're sitting in your armchair next to the fire, just as we described in our chapter

introduction; shadows dance on the tapestry-covered wall, and Lenore II (your cat) is purring atop the mantle. Yes, you're passing this cold winter's night in grand style (in front of your computer, of course). Now imagine yourself sitting on a cold hard wooden chair inside an off-white cubicle with empty walls (you're still in front of your computer, of course). These two descriptions paint two very different pictures, but both describe what you're doing (sitting and computing). Changing the "scenery" of your portal sets the mood for your users. We'll show you how to develop your own theme plugin (i.e., your "scenery") using Maven so your portal has a lasting impression on anyone who visits.

2.3.8.2.1. Creating a Theme Plugin

Theme plugin creation is similar to portlet plugin creation. We'll start by assuming you already created the `sample-parent-project` and its `pom.xml`.

To create your Liferay theme plugin project follow the *Creating Liferay Plugins with Maven* section, making sure to select *Theme* as the plugin type.



Tip: As you use Maven's Archetype tool to generate your theme project, you can filter on group ID `liferay`, or even the group ID/artifact ID combination `liferay:theme`, to more easily find the Liferay portlet archetypes.

2.3.8.2.2. Anatomy

A theme project created from the `com.liferay.maven.archetypes:liferay-theme-archetype` has the following directory structure:

- `sample-theme/`
 - › `pom.xml`
 - › `src/`
 - `main/`
 - `resources/`
 - › `resources-importer/`
 - `document_library/`
 - `journal/`
 - `articles/`
 - `structures/`
 - `templates/`
 - `readme.txt`
 - `sitemap.json`
 - `webapp/`
 - › `WEB-INF/`
 - `liferay-plugin-package.properties`
 - `web.xml`
 - › `css/` * Optionally add to hold CSS customizations
 - › `images/` * Optionally add to hold custom images

- › js/* Optionally add to hold JavaScript customizations
- › templates/* Optionally add to hold template customizations

The `src/main/webapp/` folder holds your theme's customizations. If you've ever created a theme plugin using Liferay IDE or the Plugins SDK, this folder is used the same way as the `docroot/_diffs/` folder. For example, `custom.css` should go in `src/main/webapp/css/custom.css`.

Here's a table describing the directory structure differences between themes created using Maven and themes created using the Plugins SDK:

Location	Maven project	Plugins SDK project
	<code>customizations</code>	<code>src/main/webapp/</code>

To view the directory structure of a theme developed by Ant, visit the Anatomy of a Theme Project section in this guide.

2.3.8.2.3. Theme POM

The theme plugin project POM has two additional properties:

- `liferay.theme.parent`: Sets the parent theme. You can define almost any WAR artifact as the parent using the syntax `groupId:artifactId:version`, or use the core themes by specifying `_unstyled`, `_styled`, `classic`, or `control_panel`.
- `liferay.theme.type`: Sets the template theme language.

The default settings for the two theme properties look like this:

```
<properties>
    <liferay.theme.parent>_styled</liferay.theme.parent>
    <liferay.theme.type>vm</liferay.theme.type>
</properties>
```

2.3.8.2.4. Deployment

To deploy your theme plugin, follow the instructions in the *Deploying Liferay Plugins with Maven* section.



Note: When you execute the `package` goal, a WAR file is created; it's just like the Maven WAR type project. Simultaneously, the parent theme is downloaded and copied, and your theme's customizations are overlaid last. A thumbnail image of the theme is created and placed in the `target` directory. Its path is `target/[theme]/images/screenshot.png` in your theme project.

2.3.8.2.5. More Information

For more information on Liferay themes and its settings, see Creating Liferay Themes and Layout Templates.

You successfully developed a Liferay theme using Maven. Find out about developing hook plugins next.

2.3.8.3. Developing Liferay Hook Plugins with Maven

Hooks are the optimal plugin type for customizing Liferay's core features. Creating a hook is almost identical to portlet plugin creation in Maven. Let's take a look.

2.3.8.3.1. Creating a Hook Plugin

To create a Liferay hook plugin project, follow the steps outlined in the *Creating Liferay Plugins with Maven* section, making sure to select *Hook* as the plugin type.



Tip: As you use Maven's Archetype tool to generate your hook you can filter on group ID `liferay`, or even the group ID/artifact ID combination `liferay:hook`, to more easily find the Liferay portlet archetypes.

2.3.8.3.2. Anatomy

A hook project created from the `com.liferay.maven.archetypes:liferay-hook-archetype` has the following directory structure:

- hook-plugin/
 - src/
 - main/
 - java/
 - resources/
 - webapp/
 - WEB-INF/
 - lib/ * Optionally add to hold required libraries
 - liferay-hook.xml
 - liferay-plugin-package.properties
 - web.xml
 - pom.xml

The `hook-plugin/src/main/java/` directory holds the hook's Java source code (e.g., `com.liferay.sample.SampleHook.java`) and `hook-plugin/src/main/webapp` holds the hook's web source code. If you're familiar with creating hook plugins using the Plugins SDK, you probably noticed that Maven uses a different plugin directory structure.

The following table illustrates the differences in location of the Java source and web source code for a Maven project and a Plugins SDK project:

Location	Maven project	Plugins SDK project
Java source	<code>src/main/java</code>	<code>docroot/WEB-INF/src</code>
Web source	<code>src/main/webapp</code>	<code>docroot</code>

To view the directory structure of a hook developed by Ant, visit the *Anatomy of the Hook* section of the Creating a Hook section in this guide.

2.3.8.3.3. Deployment

To deploy your hook plugin, follow the instructions from the *Deploying Liferay Plugins with Maven* section.

2.3.8.3.4. More Information

For detailed information on creating hooks, see Customizing and Extending Functionality with Hooks.

You're nearly a Maven expert now; you're able to create portlets, themes, and hooks. Let's round things out by learning to develop layout templates.

2.3.8.4. Developing Liferay Layout Template Plugins with Maven

You can create layout templates to customize the display of portlets on your page and to embed commonly used portlets. In our introduction to themes, we described a nice scene where you're relaxing in a luxurious chair, computer in your lap, Lenore II (your cat) purring on the mantle above a dancing fire. Sounds nice, doesn't it? It would be, but the chair's too small, so your knees are up in the air when your feet are flat on the ground, and your laptop is balanced precariously on top of them. The fire is also surprisingly large for that fireplace. In fact, its flames are already licking at the bottom of the mantle--which is made of wood! Remember Lenore II, softly purring on the mantle? She's going to cook just like the original Lenore if we don't do something! But it's so hard to get out of this tiny chair. Someone save Lenore II!

"Tell this soul with sorrow laden if, within the distant Aidenn, It shall clasp a sainted kitten whom the angels named Lenore II--"

In memory of the late, now crispy Lenore II, let's create a layout template plugin with Maven.

2.3.8.4.1. Creating a Layout Template Plugin

To create a Liferay layout template plugin project follow the *Creating Liferay Plugins with Maven* section, making sure to select *Layout Template* as the plugin type.



Tip: As you use Maven's Archetype tool to generate your layout template project, you can filter on group ID `liferay`, or even group ID / artifact ID combination `liferay:layout`, to find the Liferay layout template archetypes.

2.3.8.4.2. Anatomy

A layout template project created from the `com.liferay.maven.archetypes:liferay-layouttpl-archetype` has the following directory structure:

- `layouttpl-plugin/`
 - `src/`

- main/
 - resources/
 - webapp/
 - › WEB-INF/
 - liferay-layout-templates.xml
 - liferay-plugin-package.properties
 - web.xml
 - › sample-layout.png
 - › sample-layout.tpl
 - › sample-layout.wap.tpl
 - › pom.xml

There's a directory structure difference between plugin projects created using Liferay Maven archetypes and those created using the Liferay Plugins SDK. The following table illustrates this difference:

Location	Maven project	Plugins SDK project
Web source	src/main/webapp	docroot

To view the directory structure of a layout template developed by Ant, visit the Anatomy of a Layout Template Project section in this guide.

2.3.8.4.3. Deployment

To deploy your layout template plugin, follow the instructions detailed above in the *Deploying Liferay Plugins with Maven* section.

2.3.8.4.4. More Information

For detailed information on creating layout templates, see Creating Liferay Themes and Layout Templates.

You've passed your trial by fire (the cat thanks you), developing yet another plugin type with Maven. Way to go! In the next section we'll cover other Liferay-provided Maven archetypes.

2.3.8.5. Developing More Liferay Plugins with Maven

Did you think we covered all the available archetypes for developing Liferay plugins? The Liferay team has been busy expanding our archetype list, and we're proud to show you some additional plugins that you can create using Maven archetypes.

Check out these exciting archetypes that are available:

- Liferay ServiceBuilder portlets
- Liferay webs
- Liferay Ext
- JSF Portlet Archetype
- ICEFaces Portlet Archetype
- PrimeFaces Portlet Archetype
- Liferay Faces Alloy Portlet Archetype
- Liferay Rich Faces Portlet Archetype

In addition, there are some Maven *goals* Liferay has provided:

- DBBuilder - The `build-db` goal lets you execute the DBBuilder to generate SQL files.
- SassToCSSBuilder - The `build-css` goal precompiles SASS in your css; this goal has been added to theme archetype.

You now have plenty of archetypes at your disposal!

"But the chair whose violet lining with the lamp-light gloating o'er, Lenore II shall press, ah, nevermore!"

Lenore II didn't make it through the Maven section, but you did. You can develop all your Liferay plugins using Maven; there's a standard process for generating the archetypes and selecting your plugin options for each plugin type. You can then customize the archetype to your liking. Using Maven to develop plugins offers an easy and effective way to customize your Liferay Portal.

Are you wondering if we're going to make more terrible jokes that steal from classic poetry? Quoth the Maven, "Probably." Let's move on to exploring the differences between hot deploy and auto deploy, with respect to your plugins.

2.4. Deploying Your Plugins: Hot Deploy vs. Auto Deploy

As you develop plugins you'll want to deploy them to your test servers and as you finish developing plugins you'll want to deploy them to your production servers. There are *hot* deploy and *auto* deploy options to use in deploying your plugins. Most people confuse the two concepts, believing them to be one and the same. In reality, Liferay has TWO completely separate and different concepts for them.

How, you say? We'll give a brief synopsis of each deployment method in this section. Let's get started by explaining the hot deployment method.

2.4.1. Using Hot Deployment

The first deployment method we'll explore is *hot* deployment. You may be familiar with hot deployment in the context of JEE application servers. In summary, you place an application artifact (WAR or EAR) file into a specifically configured directory, your application server (Tomcat, WebSphere, WebLogic, etc.) picks up that artifact, deploys it within the application server, and starts the application.

This model works really well for development purposes, since a server restart is not required to view updates from your code changes. This model also works for single node production deployments.

This model completely breaks down when you deploy to a multi-node production deployment. In a multi-node environment, you have many more constraints to deal with, which require you to:

- Ensure the application archive is available to all nodes
- Ensure the application deploys successfully across all nodes, simultaneously

Most application servers solve these constraints by using a master/slave type of design: an admin server with multiple managed servers. When you hot deploy a plugin, you use the admin server's user interface, or vendor console tool like Wsadmin, to add the archive, select which managed

servers should deploy it, and start the application. Application server vendors often have different names and tools for these modes and tools:

- JBoss "domain" mode
- WebLogic "production" mode
- WebSphere deployment manager
- Tomcat FarmWarDeployer

These modes and tools reside completely outside of Liferay Portal and are strictly in the application server's realm. However, Liferay piggybacks off the application server's hot deploy capability and performs additional initialization after a given application starts (e.g., via `javax.servlet.ServletContextListener` mechanisms).

There are some specific Liferay capabilities that won't work unless your application server has hot deployment capabilities. Specifically, hot deploying custom JSPs in hooks won't work, because Liferay's JSP hook overriding capabilities depend on the application server's ability to:

- Deploy based on an exploded portal WAR
- Load changes to JSP files at runtime

Application servers running in "production" and "domain" modes cannot support these abilities, because in these deployment models, most servers don't use exploded WARs. As such, these application servers don't support JSP reloading/recompilation in these modes. Even for Tomcat, it's generally advisable to deactivate JSP reloading for production deployments.

So what do you do if you use hooks to override Liferay JSPs AND you must use non-expoded WAR deployments? The answer is simple: inject a pre-processing stage as part of your build process. You deploy the hooks, allowing them to make changes to the portal WAR file. Then you rebundle the portal WAR file and deploy it using the application server's deployment tools. Of course, you still need to deploy your hook as well, but you no longer need to worry about the JSP overrides not being loaded by your application server.

Hopefully this whets your appetite for doing hot deployments. Stay hungry, as it's time to explore auto deployment next.

2.4.2. Using Auto Deployment

The Liferay *auto* deployment feature is a mostly optional feature that works in conjunction with the hot deployment capabilities of your application server. Where Liferay's hot deploy leverages the hot deploy capabilities of your app server and performs additional initializations, auto deploy injects required JAR files and descriptors into your application's archive file. Executing `ant deploy` invokes both hot deployment and auto deployment tasks for your plugin.

So how does auto deployment work with Liferay plugins? Auto deployment completes the following tasks:

1. Picks up a Liferay recognized archive (e.g., `*-portlet.*`, `*-theme.*`, `*-web.*`, `*.lpkg`)
2. Injects required libraries (e.g., `util-java.jar`, `util-taglib.jar`)
3. Injects dependent JAR files (specified in `liferay-plugins.properties`)
4. Injects required taglib descriptors (e.g., `liferay-theme.tld`)

5. Injects required deployment descriptors (e.g., app server specific descriptors)
6. Injects any missing Liferay specific deployment descriptors (e.g., `liferay-portlet.xml`)

By relying on auto deployment to complete these tasks automatically, you save time and you don't even have to learn all of Liferay's deployment descriptors. However, this feature is incompatible with application server farms and multi-node modes.

So now you're probably wondering how to configure your application server in these situations? The answer is simple: Do not use the auto deployment method at runtime; use it at build time.

The Liferay Plugins SDK allows you to preprocess your archives and inject all the required elements, thus, bypassing auto deployer at runtime. You simply need to call the following Ant task:

```
ant direct-deploy
```

The `direct-deploy` Ant task creates an exploded WAR from which you can easily create a WAR file. The location of the exploded WAR depends on the deployment directory of the application server you've configured in your Plugins SDK environment. See the *Plugins SDK Configuration* section of Leveraging the Plugins SDK for instructions on configuring the Plugins SDK for your app server. The Plugins SDK's `build.properties` provides a default deployment directory value for each supported app server. But you can override the default value by specifying your desired value for the `app.server.[type].deploy.dir` (replace `[type]` with your app server type) in your `build.[username].properties` file.

If you choose not to use the Liferay Plugins SDK to do direct deployment, you can examine the `build-common.xml` file in the Plugins SDK to see how Liferay invokes the deployer tools.

Terrific! You now know the differences between hot deploy and auto deploy. Understanding what's going on during the deployment of your plugins is crucial for troubleshooting anything that goes wrong, and can help you simplify your deployment process and make it more efficient. Let's summarize what we learned in this chapter.

2.5. Summary

In this chapter, we covered many of Liferay's most popular development strategies. You covered Liferay IDE, which serves as a workspace used for customizing your Liferay instance. You also learned that when Liferay IDE is paired with the Plugins SDK, you have a one stop development environment where you can develop Liferay plugins, build them, and deploy them to your Liferay instance.

You also learned all about developing Liferay plugins with the Maven build framework. You configured Maven locally, downloaded and installed the required Liferay Maven artifacts, and learned to create Liferay plugins with Maven. You're ready to create all kinds of Liferay plugins based on Liferay's plugin archetypes. Just don't let Lenore III sleep near the fire this time.

Lastly, you learned about Liferay's deployment process, and the difference between hot deploy and auto deploy.

Feeling confused by the number of features provided by Eclipse and Liferay IDE? You can easily come across difficult questions and run into very specific problems, but someone else might have

already solved your issue or answered your question. So where would you go to find out? Don't reinvent the wheel, visit the Liferay IDE Community page! On the *Forums* page, you can look up solutions to specific issues and ask questions. Be sure to fully describe any problems you have to ensure you get a working answer. You can even track known issues from the *Issue Tracker* page.

Now that you have a firm grasp on Liferay's development tools, let's learn about developing one of Liferay's most important components: portlets.

3. Developing Portlet Applications

Think of your Liferay portal as a pizza crust (sit down, you can go order a real pizza when we're done here). In Chapter 2 we equipped you with Liferay's tools for developing your pizza, and Liferay comes with some basic toppings that make for a pretty good pizza out of the box (i.e., our core portlets and built-in functionality). Of course, your boss might demand anchovies, and Liferay definitely doesn't come with anchovies. So what do you do? You take our tools, get some anchovies (your app's source code), and integrate them with the pizza (Liferay). In this chapter we're going to show you how to develop portlet projects to top your Liferay pizza in such a way that the end-user won't be able to tell the difference between your custom portlet and our core portlets.

In the last chapter we showed you how to create Liferay plugin projects, and if you followed along with our exercises, you now have a project to hold Liferay portlets. Unfortunately we don't really have any portlets in there yet. So we're going to get to business on actually creating an application with the Liferay development tools we've already introduced to you. It's fitting to start with portlet development, because portlets are the most basic, most commonly used type of Liferay plugin you'll develop. We'll create and deploy a simple portlet using the Plugins SDK. It will allow a customized greeting to be saved in the portlet's preferences and then display it whenever the portlet is viewed. Then we'll clean up the portlet's URLs by adding a friendly URL mapping. Lastly we'll localize the portlet.

You're free to use any framework you prefer to develop your portlets, including Struts, Spring MVC, JSF, and Vaadin. Here we'll use the Liferay MVCPortlet framework, because it's simple, lightweight, and easy to understand.

You don't have to be a Java developer to take advantage of Liferay's built-in features (such as user and organization management, page building, and content management). An application developed using Ruby or PHP can be deployed as a portlet using the Plugins SDK, and it will run seamlessly inside of Liferay. For examples, check out the `liferay-plugins` repository from GitHub at <http://github.com/liferay>.

We'll discuss the following topics as we learn about developing portlets for Liferay:

- Creating a Portlet Project
- Anatomy of a Portlet Project
- Writing the My Greeting Portlet
- Understanding the Two Phases of Portlet Execution
- Passing Information from the Action Phase to the Render Phase
- Developing a Portlet with Multiple Actions

- Adding Friendly URL Mapping to the Portlet
- Localizing Your Portlet
- Implementing Configurable Portlet Preferences
- Creating Plugins to Share Templates, Structures, and More

First, let's create the portlet that we'll use throughout this chapter.

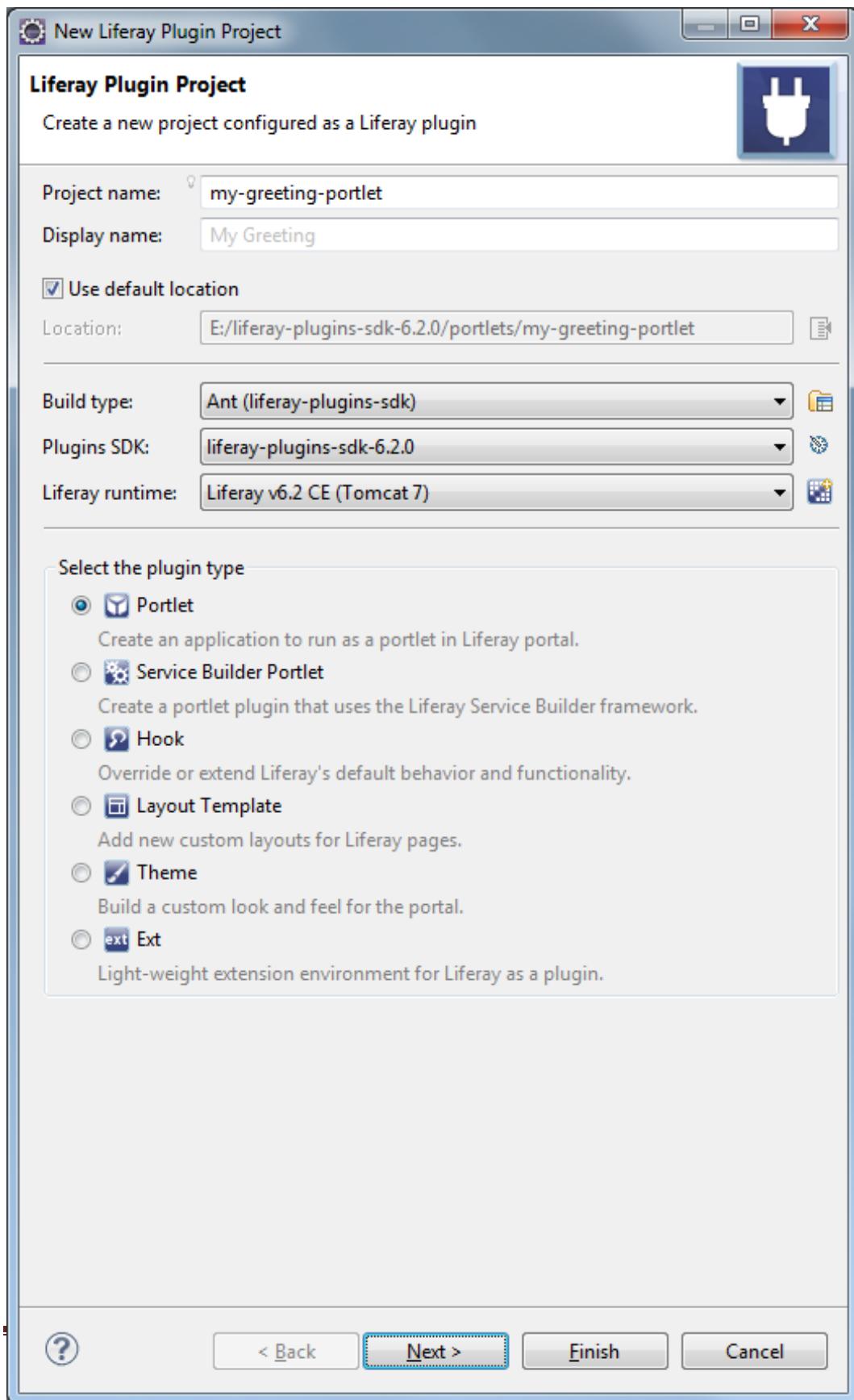
3.1. *Creating a Portlet Project*

Portlet creation using the Plugins SDK is simple. There's a `portlets` folder inside the Plugins SDK folder, where your portlet projects reside. The first thing to do is give your portlet a project name (without spaces) and a display name (which can have spaces). For the greeting portlet, the project name is `my-greeting`, and the portlet title is `My Greeting`.

Once you've named your portlet, you're ready to begin creating the project. There are several different ways to create this portlet. Let's try it using Liferay Developer Studio first, then by using the terminal.

In Developer Studio:

1. Go to File → New → Liferay Project.
2. Fill in the *Project name* and *Display name* with `my-greeting-portlet` and `My Greeting`, respectively.
3. Leave the *Use default location* checkbox checked. By default, the default location is set to your current workspace. If you'd like to change where your plugin project is saved in your file system, uncheck the box and specify your alternate location.



4. Select the *Ant (liferay-plugins-sdk)* option for your build type. If you'd like to use *Maven* for your build type, navigate to the [Developing Plugins Using Maven](#) section for details.

5. Your configured SDK and Liferay Runtime should already be selected. If you haven't yet pointed Liferay IDE to a Plugins SDK, click *Configure SDKs* to open the *Installed Plugin SDKs* management wizard. You can also access the

New Server Runtime Environment wizard if you need to set up your runtime server; just click the *New Liferay Runtime* button next to the *Liferay Portal Runtime* dropdown menu.

6. Select *Portlet* as your Plugin type.
7. Click *Next*.
8. In the next window, make sure that the *Liferay MVC* framework is selected and click *Finish*.

With Developer Studio, you can create a new plugin project or if you already have a project, create a new plugin in an existing project. A single Liferay project can contain multiple plugins.

Using the Terminal: Navigate to the `portlets` directory in the terminal and enter the appropriate command for your operating system:

1. In Linux and Mac OS X, enter

```
./create.sh my-greeting "My Greeting"
```

2. In Windows, enter

```
create.bat my-greeting "My Greeting"
```

You should get a **BUILD SUCCESSFUL** message from Ant, and there will now be a new folder inside of the `portlets` folder in your Plugins SDK. This folder is your new portlet project. This is where you will be implementing your own functionality. Notice that the Plugins SDK automatically appends `"-portlet"` to the project name when creating this folder.

Alternatively, if you will not be using the Plugins SDK to house your portlet projects, you can copy your newly created portlet project into your IDE of choice and work with it there. If you do this, you will need to make sure the project references some `.jar` files from your Liferay installation, or you may get compile errors. Since the Ant scripts in the Plugins SDK do this for you automatically, you don't get these errors when working with the Plugins SDK.

To resolve the dependencies for portlet projects, see the classpath entries in the `build-common.xml` file in the Plugins SDK project. You can determine from the `plugin.classpath` and `portal.classpath` entries, which `.jar` files are necessary to build your newly created portlet project. This is not a recommended configuration, and we encourage you to keep your projects in the Plugins SDK.



Tip: If you are using a source control system such as Subversion, CVS, Mercurial, Git, etc., this might be a good moment to do an initial check-in of your changes. After building the plugin for deployment, several additional files will be generated that should *not* be handled by the source control system.

3.1.1. Deploying the Portlet

Liferay provides a mechanism called auto-deploy that makes deploying portlets (and any other plugin types) a breeze. All you need to do is drop the plugin's WAR file into the deploy directory, and the portal makes the necessary changes specific to Liferay and then deploys the plugin to the

application server. This is a method of deployment used throughout this guide.



Note: Liferay supports a wide variety of application servers. Many, such as Tomcat and Jboss, provide a simple way to deploy web applications by just copying a file into a folder and Liferay's auto-deploy mechanism takes advantage of that ability. You should be aware though, that some application servers, such as Websphere or Weblogic, require the use of specific tools to deploy web applications; Liferay's auto-deploy process won't work for them.

Deploying in Developer Studio: Drag your portlet project onto your server. When deploying your plugin, your server displays messages indicating that your plugin was read, registered and is now available for use.

```
Reading plugin package for my-greeting-portlet
```

```
Registering portlets for my-greeting-portlet
```

```
1 portlet for my-greeting-portlet is available for use
```

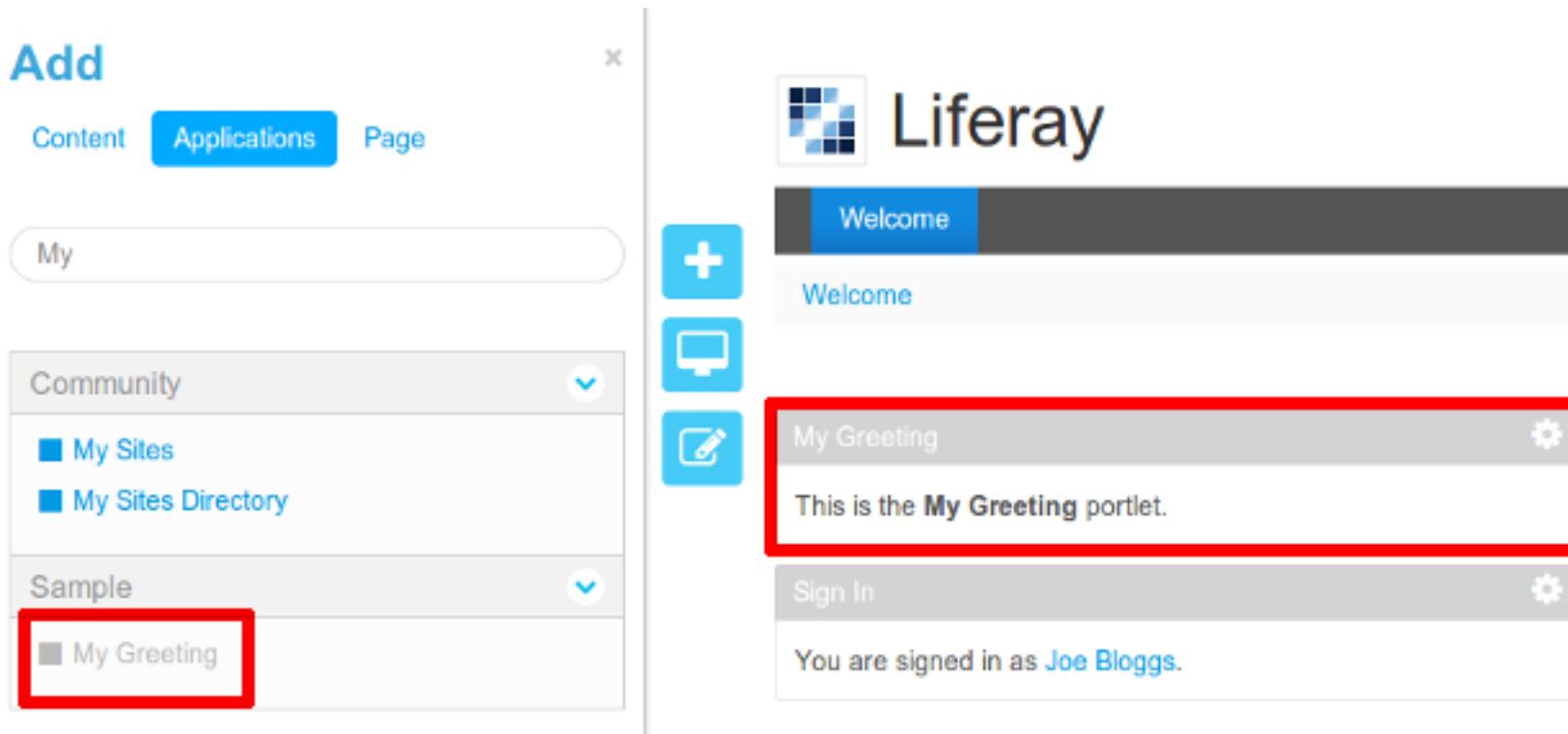
If at any time you need to redeploy your portlet while in Developer Studio, right-click your portlet located underneath your server and select *Redeploy*.

Deploying in the terminal: Open a terminal window in your `portlets/my-greeting-portlet` directory and enter

```
ant deploy
```

A BUILD SUCCESSFUL message indicates your portlet is now being deployed. If you switch to the terminal window running Liferay, within a few seconds you should see the message 1 portlet for my-greeting-portlet is available for use. If not, double-check your configuration.

In your web browser, log in to the portal as explained earlier. Click the Add button, which appears as a *Plus* symbol in the top right hand section of your browser. Then click *Applications*, find the My Greeting portlet in the *Sample* category, and click *Add*. Your portlet appears in the page.



Congratulations, you've just created your first portlet!

3.2. *Anatomy of a Portlet Project*

A portlet project is made up of at least three components:

1. Java Source.
2. Configuration files.
3. Client-side files (.jsp, .css, .js, graphics files, etc.).

When using Liferay's Plugins SDK, these files are stored in a standard directory structure:

- PORTLET-NAME/
 - build.xml
 - docroot/
 - css/
 - js/
 - META-INF/
 - WEB-INF/
 - lib/
 - src/ - this folder is not created by default.
 - tld/
 - liferay-display.xml
 - liferay-plugin-package.properties

- liferay-portlet.xml
- portlet.xml
- web.xml
- icon.png
- view.jsp

The portlet we just created is fully functional and deployable to your Liferay instance.

By default, new portlets use the MVCPortlet framework, a light framework that hides part of the complexity of portlets and makes the most common operations easier. The default MVCPortlet project uses separate JSPs for each portlet mode: each of the registered portlet modes has a corresponding JSP with the same name as the mode. For example, 'edit.jsp' is for edit mode and 'help.jsp' is for help mode.

The *Java Source* is stored in the `docroot/WEB-INF/src` folder.

The *Configuration Files* are stored in the `docroot/WEB-INF` folder. Files stored here include the standard JSR-286 portlet configuration file `portlet.xml`, as well as three optional Liferay-specific configuration files. The Liferay-specific configuration files, while optional, are important if your portlets will be deployed on a Liferay Portal server. Here's a description of the Liferay-specific files:

- `liferay-display.xml`- Describes the category the portlet appears under in the *Add* menu of the Dockbar (the horizontal bar that appears at the top of the page to all logged-in users).
- `liferay-plugin-package.properties`- Describes the plugin to Liferay's hot deployer. You can configure Portal Access Control List (PACL) properties, `.jar` dependencies, and more.
- `liferay-portlet.xml`- Describes Liferay-specific enhancements for JSR-286 portlets installed on a Liferay Portal server. For example, you can set an image icon to represent the app, trigger a job for the scheduler, and much more. A complete listing of this file's settings is in its DTD in the `definitions` folder in the Liferay Portal source code.

Client Side Files are the `.jsp`, `.css`, and `.js` files that you write to implement your portlet's user interface. These files should go in the `docroot` folder; `.jsp` files can be placed in the root of the folder, while `.css` and `.js` files are given their own subfolders in `docroot`. Remember, with portlets you're only dealing with a portion of the HTML document that is getting returned to the browser. Any HTML code in your client side files must be free of global tags like `<html>` or `<head>`. Additionally, namespace all CSS classes and element IDs to prevent conflicts with other portlets. Liferay provides two tools, a taglib and API methods, to generate a namespace for you. See the *Using Portlet Namespacing* section of this chapter to learn more about namespacing.

Let's continue exploring portlet anatomy by studying your new My Greeting portlet.

3.2.1. A Closer Look at the My Greeting Portlet

If you're new to portlet development, this section will enhance your understanding of portlet

configuration options.

In the Plugins SDK, the portlet descriptor's default content in `docroot/WEB-INF/portlet.xml` looks like this (shown using Developer Studio's Portlet Application Configuration Editor):

portlet.xml

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

<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.
    http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd" version="2.0">
  <portlet>
    <portlet-name>my-greeting</portlet-name>
    <display-name>My Greeting</display-name>
    <portlet-class>com.liferay.util.bridges.mvc.MVCPortlet</portlet-class>
    <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>
    <portlet-info>
      <title>My Greeting</title>
      <short-title>My Greeting</short-title>
      <keywords>My Greeting</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>
```

Here's a basic summary of what each element represents:

- **portlet-name**: Contains the portlet's canonical name. Each portlet name is unique within the portlet application (that is, within the portlet plugin). In Liferay Portal, this is also referred to as the portlet ID.
- **display-name**: Contains a short name that's shown by the portal whenever this application needs to be identified. It's used by **display-name** elements. The display name need not be unique.
- **portlet-class**: Contains the fully qualified name of the class that handles invocations to the portlet.
- **init-param**: Contains a name/value pair as an initialization parameter of the portlet.
- **expiration-cache**: Indicates the time, in seconds, after which the portlet output expires. A value of **-1** indicates that the output never expires.
- **supports**: Contains the supported mime-type, and indicates the portlet modes supported for a specific content type. The concept of "portlet modes" is defined by the portlet specification. Modes are used to separate certain views of the portlet from others. The portal is aware of the portlet modes and provides generic ways to navigate between them (for example, using links in the box surrounding the portlet when it's added to a page), so they're useful for operations that are common to all or most portlets. The most common usage is to create an edit screen where each user can specify personal preferences for the portlet. All portlets must support the view mode.
- **portlet-info**: Defines information that can be used for the portlet title-bar and for the portal's categorization of the portlet. The JSR-286 specification defines a few resource elements that can be used for these purposes: **title**, **short-title**, and **keywords**. You can either include resource elements directly in a **portlet-info** element or you can place them in resource bundles.

Specifying the information directly into the **portlet-info** element in your **portlet.xml** file is straightforward. For example, you could specify a weather portlet's information, like this:

```
<portlet>
  ...
  <portlet-info>
    <title>Weather Portlet</title>
    <short-title>Weather</short-title>
    <keywords>weather, forecast</keywords>
  </portlet-info>
  ...
</portlet>
```

Alternatively, you can specify this same information as resources in a resource bundle file for your portlet. For example, you could create the file **docroot/WEB-INF/src/content/Language.properties**, in your portlet project, to specify your portlet's title, short title, and keywords:

```
# Default Resource Bundle
#
# filename: Language.properties
# Portlet Info resource bundle example
javax.portlet.title=Weather Portlet
```

```
javax.portlet.short-title=Weather  
javax.portlet.keywords=weather,forecast  
To use the resource bundle, you'd reference it in your portlet.xml file:  
<portlet>
```

```
...  
<resource-bundle>content.Language</resource-bundle>  
<portlet-info>...</portlet-info>  
...  
</portlet>
```

As a best practice, if you're not planning on supporting localized title, short title, and keywords values for your portlet, simply specify them within the `<portlet-info>` element in your `portlet.xml` file. Otherwise, if you're ready to provide localized values, use a resource bundle for specifying your default values and specify the localized values in separate resource bundles.



Note: You should not specify values for a portlet's title, short title, and keywords in both a portlet's `<portlet-info>` element in `portlet.xml` and in a resource bundle. If you do so unintentionally, the values in the resource bundle take precedence over the values in the `<portlet-info>` element.

Specifying *localized* values for your portlet's title, short title, and keywords in resource bundles is easy. For example, if you're supporting German and English locales, you'd create `Language_de.properties` and `Language_en.properties` files, respectively, in your portlet's `docroot/WEB-INF/src/content/` directory. This is the same directory as your default resource bundle file `Language.properties`. The contents of the German and English resource bundles may look like the following:

```
# English Resource Bundle  
#  
# filename: Language_en.properties  
# Portlet Info resource bundle example  
javax.portlet.title=Weather Portlet  
javax.portlet.short-title=Weather  
javax.portlet.keywords=weather,forecast  
  
# German Resource Bundle  
#  
# filename: Language_de.properties  
# Portlet Info resource bundle example  
javax.portlet.title=Wetter Portlet  
javax.portlet.short-title=Wetter  
javax.portlet.keywords=wetter,vorhersage
```

You'd reference your default bundle and these localized bundles in your `portlet.xml` file, like this:

```
<portlet>  
...  
<resource-bundle>content.Language</resource-bundle>  
<resource-bundle>content.Language_de</resource-bundle>
```

```
<resource-bundle>content.Language_en</resource-bundle>
<portlet-info>...</portlet-info>
...
</portlet>
```

If you're mavenizing your portlet, make sure to copy your `content` folder into your portlet's `src/main/webapp/WEB-INF/classes` folder.

For more information, see the JSR-286 portlet specification, at
<http://www.jcp.org/en/jsr/detail?id=286>.

- `security-role-ref`: Contains the declaration of a security role reference in the code of the web application. Specifically in Liferay, the `role-name` references which roles can access the portlet.

docroot/WEB-INF/liferay-portlet.xml: In addition to the standard `portlet.xml` options, there are optional Liferay-specific enhancements for Java Standard portlets that are installed on a Liferay Portal server. By default, the Plugins SDK sets the contents of this descriptor, as shown in Developer Studio:

liferay-portlet.xml

```
<?xml version="1.0"?>
<!DOCTYPE liferay-portlet-app PUBLIC "-//Liferay//DTD Portlet Application 6.2.0//EN" "http://www.liferay.com/dtd/liferay-portlet-app_6_2_0.dtd">

<liferay-portlet-app>
    <portlet>
        <portlet-name>my-greeting</portlet-name>
        <icon>/icon.png</icon>
        <header-portlet-css>/css/main.css</header-portlet-css>
        <footer-portlet-javascript>/js/main.js</footer-portlet-javascript>
        <css-class-wrapper>my-greeting-portlet</css-class-wrapper>
    </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>
```

Here's a basic summary of what some of the elements represent.

- **portlet-name:** Contains the canonical name of the portlet. This needs to be the same as the `portlet-name` specified in the `portlet.xml` file.

- `icon`: Path to icon image for this portlet.
- `instanceable`: Indicates whether multiple instances of this portlet can appear on the same page.
- `header-portlet-css`: The path to the `.css` file for this portlet to include in the `<head>` tag of the page.
- `footer-portlet-javascript`: The path to the `.js` file for this portlet, to be included at the end of the page before the `</body>` tag.

There are many more elements that you should be aware of for more advanced development. They're all listed in the DTD for this file, which is found in the `definitions` folder in the Liferay Portal source code.

3.3. Writing the My Greeting Portlet

Let's make our portlet do something useful. First, we'll give it two pages:

- `view.jsp`: displays the greeting and provides a link to the `edit` page.
- `edit.jsp`: shows a form with a text field, allowing the greeting to be changed, and including a link back to the `view` page.

The `MVCPortlet` class handles the rendering of our JSPs, so for this example, we won't write a single Java class.

First, since we don't want multiple greetings on the same page, let's make the My Greeting portlet non-instanceable. Just edit `liferay-portlet.xml`. If your `portlet` element already has an `instanceable` element, change its value from `true` to `false`. If you don't already have an `instanceable` element for your portlet, add it. Here's what it looks like in the context of the `portlet` element:

```
<portlet>
    <portlet-name>my-greeting</portlet-name>
    <icon>/icon.png</icon>
    <instanceable>false</instanceable>
    <header-portlet-css>/css/main.css</header-portlet-css>
    <footer-portlet-javascript>/js/main.js</footer-portlet-javascript>
    <css-class-wrapper>my-greeting-portlet</css-class-wrapper>
</portlet>
```

Now we'll create our JSP templates. Start by editing `view.jsp`, found in your portlet's `docroot` directory. Replace its current contents with the following:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@ page import="javax.portlet.PortletPreferences" %>

<portlet:defineObjects />

<%
PortletPreferences prefs = renderRequest.getPreferences();
String greeting = (String)prefs.getValue(
"greeting", "Hello! Welcome to our portal.");
%>

<p><%= greeting %></p>
```

```

<portlet:renderURL var="editGreetingURL">
    <portlet:param name="mvcPath" value="/edit.jsp" />
</portlet:renderURL>

<p><a href="<% editGreetingURL %>">Edit greeting</a></p>
Next, create edit.jsp in the same directory as view.jsp, with the following content:
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>

<%@ page import="javax.portlet.PortletPreferences" %>

<portlet:defineObjects />

<%
PortletPreferences prefs = renderRequest.getPreferences();
String greeting = renderRequest.getParameter("greeting");
if (greeting != null) {
    prefs.setValue("greeting", greeting);
    prefs.store();
}
%>

<p>Greeting saved successfully!</p>

<%
}
%>

<%
greeting = (String)prefs.getValue(
    "greeting", "Hello! Welcome to our portal.");
%>

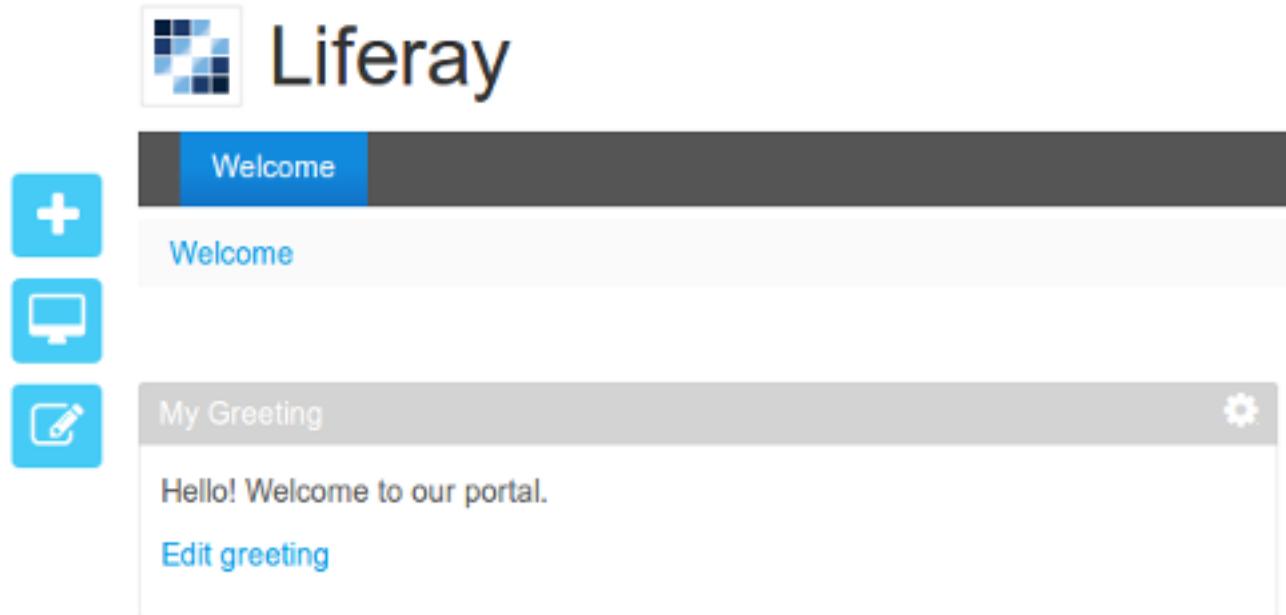
<portlet:renderURL var="editGreetingURL">
    <portlet:param name="mvcPath" value="/edit.jsp" />
</portlet:renderURL>

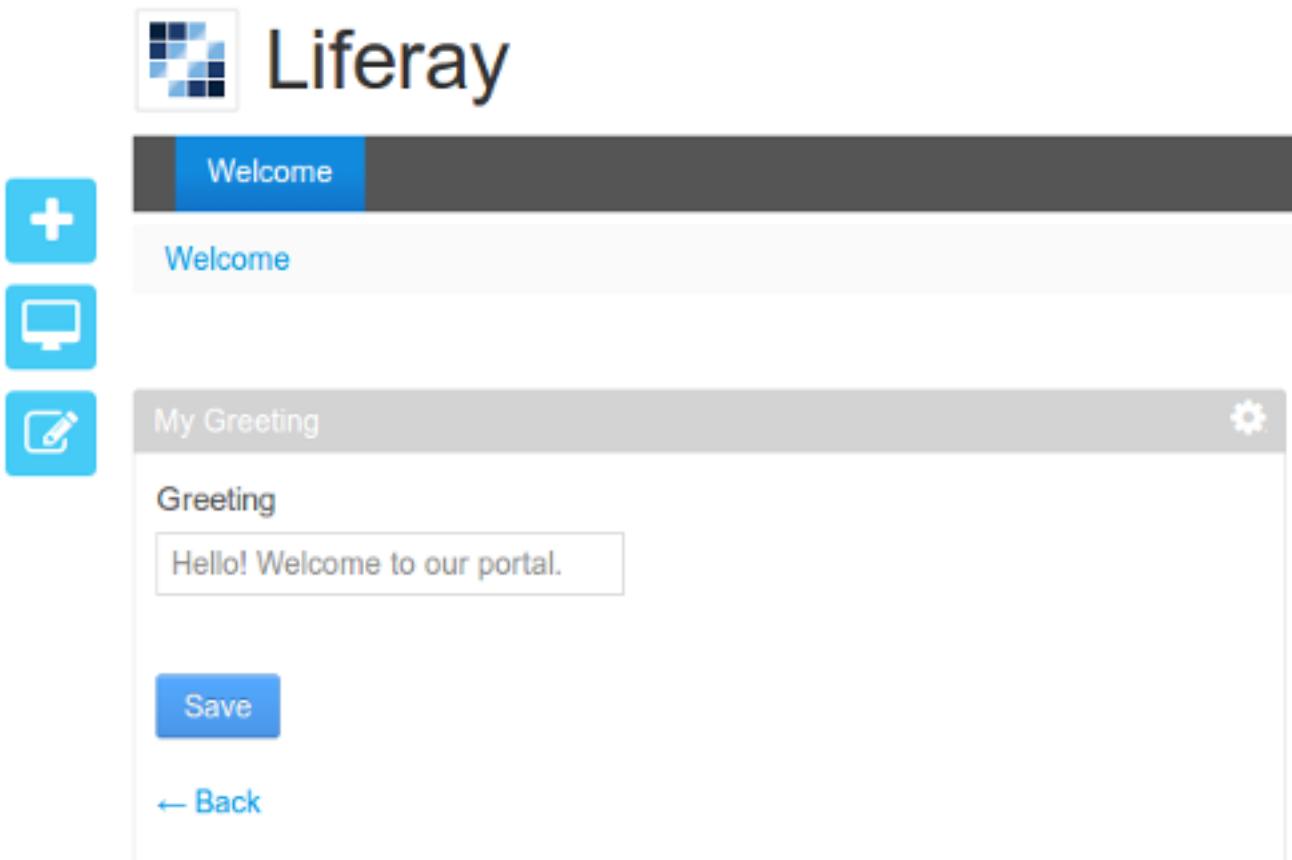
<aui:form action="<% editGreetingURL %>" method="post">
    <aui:input label="greeting" name="greeting" type="text" value="<%=
greeting %>" />
    <aui:button type="submit" />
</aui:form>

<portlet:renderURL var="viewGreetingURL">
    <portlet:param name="mvcPath" value="/view.jsp" />
</portlet:renderURL>

<p><a href="<% viewGreetingURL %>">&larr; Back</a></p>
Redeploy the portlet in Developer Studio or redeploy it in a terminal by executing the command
ant deploy from your my-greeting-portlet folder. Go back to your web browser and
refresh the page; you should now be able to use the portlet to save and display a custom greeting.

```





Tip: If your portlet deployed successfully, but you don't see any changes in your browser after refreshing the page, Tomcat may have failed to rebuild your JSPs. To fix this, delete the work folder in `liferay-portal-[version]/tomcat-[tomcat-version]` and refresh the page again to force them to be rebuilt.

There are a few important details to note concerning this implementation. First, the links between pages are created using the `<portlet:renderURL>` tag, which is defined by the `http://java.sun.com/portlet_2_0` tag library. These URLs have only one parameter named `mvcPath`. This is used by `MVCPortlet` to determine which JSP to render for each request. Always use taglibs to generate URLs to your portlet, because the portlet doesn't own the whole page, only a fragment of it. The URL must always go to the portal responsible for rendering, and this applies to your portlet and any others that the user might put in the page. The portal will be able to interpret the taglib and create a URL with enough information to render the whole page.

Second, notice that the form in `edit.jsp` has the prefix `aui`, signifying that it's part of the AlloyUI tag library. AlloyUI greatly simplifies the code required to create attractive and

accessible forms by providing tags that render both the label and the field at once. You can also use regular HTML or any other taglibs to create forms based on your own preferences.

Another JSP tag you may have noticed is `<portlet:defineObjects/>`. The portlet specification defined this tag in order to be able to insert a set of implicit variables into the JSP that are useful for portlet developers, including `renderRequest`, `portletConfig`, `portletPreferences`, etc. 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. The `<portlet:defineObjects>` tag makes the following portlet objects available to a JSP:

- `RenderRequest renderRequest`: represents the request sent to the portlet to handle a render. `renderRequest` 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.
- `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.
- `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.
- `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.
- `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.
- `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.
- `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.
- `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.
- `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.getAttributeMap()` call or an empty Map if no session attributes exist.
- `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.

The variables made available by the `<portlet:defineObjects/>` tag reference are the same portlet API objects that are stored in the request object of the JSP. For more information about these objects, please refer to the Liferay's Portlet 2.0 Javadocs at <http://docs.liferay.com/portlet-api/2.0/javadocs/>.



Note: For the purpose of making our example easy to follow, we cheated a little bit. The portlet specification doesn't allow setting preferences from a JSP, because they are executed in what is known as the render state. There are good reasons for this restriction, and they're explained in the next section.

Let's talk about why we need two phases of execution for our portlets.

3.4. *Understanding the Two Phases of Portlet Execution*

Our portlet needs two execution phases, the action phase and the render phase. Multiple execution phases can be confusing to developers used to regular servlet development or used to other environments such as PHP, Python or Ruby. However, once you're acquainted with them, you'll find the action and render phase to be simple and useful. Let's talk about why they're necessary before defining each phase.

Our portlet doesn't own the entire HTML page, but shares the page with other portlets and the portal itself. The portal generates the page by invoking one or more portlets and adding some additional HTML around them. When a user invokes an action within a portlet, each of the page's portlets are rendered anew. The portal can't just allow each portlet to repeat its last invocation, and the scenario described below illustrates why.

Pretend we have a page with two portlets: a navigation portlet and a shopping portlet. Here's what would happen to a user if portals didn't have two execution phases:

1. First, the user would navigate to an item she wants to buy, and eventually submit the order, charging an amount on her credit card. After this operation, the portal would also invoke the navigation portlet with its default view.
2. Next, say the user clicks a link in the navigation portlet. This initiates an HTTP request/response cycle, and causes the content of the portlet to change. But all the

parameters are preserved during that cycle, including the ones from the shopping cart! Since the portal must also show the content of the shopping portlet, it repeats the last action (the one in which the user clicked a button), which causes a new charge on the credit card and the start of a new shipping process!

Why does this happen? Because the portal cannot know at runtime which portlets a user has added to a page. Obviously, when writing a standard web application, developers can design it so that certain URLs perform actions, and certain URLs navigate to other pages. Since an end user of a portal can add any portlet to a page, the portal must separate "actions" from a simple re-draw (or re-render) of the portlet.

Obviously, we'd like to avoid the situation described in step 2 above, but without the two phases, the portal wouldn't know whether the last operation on a portlet was an action. It would have no option but to repeat the last action over and over to obtain the content of the portlet (at least until the Credit Card reached its limit).

Fortunately, that's not how portals work. To prevent situations like the one described above, the portlet specification defines two phases for every request of a portlet, allowing the portal to differentiate *when an action is being performed* (and should not be repeated) and *when the content is being produced* (rendered):

- *Action phase*: The action phase can only be invoked for one portlet at a time. It is the result of a user interaction with the portlet. In this phase the portlet can change its status, for instance changing the user preferences of the portlet. Any inserts and modifications in the database or operations that should not be repeated must be performed in this phase.
- *Render phase*: The render phase is always invoked for all portlets on the page after the action phase (which may or not exist). This includes the portlet that also had executed its action phase. It's important to note that the order in which the render phase of the portlets in a page gets executed is not guaranteed by the portlet specification. Liferay has an extension to the specification through the element `render-weight` in `liferay-portlet.xml`. Portlets with a higher render weight will be rendered before those with a lower weight.

In our example so far, we've used a portlet class called `MVCPortlet`. That's all the portlet needs if it only has a render phase. In order to be able to add custom code that's executed in the action phase (and thus is *not* executed when the portlet is shown again) you must create a subclass of `MVCPortlet` or create a subclass of `GenericPortlet` directly (if you don't want to use Liferay's lightweight framework).

Our example above could be enhanced by creating the following class:

```
package com.liferay.samples;

import java.io.IOException;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.PortletException;
import javax.portlet.PortletPreferences;
import com.liferay.util.bridges.mvc.MVCPortlet;

public class MyGreetingPortlet extends MVCPortlet {
```

```

@Override
public void processAction(
    ActionRequest actionRequest, ActionResponse actionResponse)
throws IOException, PortletException {
    PortletPreferences prefs = actionRequest.getPreferences();
    String greeting = actionRequest.getParameter("greeting");

    if (greeting != null) {
        prefs.setValue("greeting", greeting);
        prefs.store();
    }

    super.processAction(actionRequest, actionResponse);
}
}

```

Create the above class, and its package, in the directory docroot/WEB-INF/src in your portal project.

The file portlet.xml must also be changed so that it points to your new portlet class com.liferay.samples.MyGreetingPortlet, instead of com.liferay.util.bridges.mvc.MVCPortlet:

```

<portlet>
<portlet-name>my-greeting</portlet-name>
<display-name>My Greeting</display-name>
<portlet-class>com.liferay.samples.MyGreetingPortlet</portlet-class>
<init-param>
    <name>view-template</name>
    <value>/view.jsp</value>
</init-param>
...

```

Finally, make a minor change in the edit.jsp file, changing the URL to which the form is sent in order to let the portal know to execute the action phase. There are three types of URLs that can be generated by a portlet:

- *renderURL*: Invokes a portlet using only its render phase.
- *actionURL*: Executes an action phase before rendering all the portlets in the page.
- *resourceURL*: Is used to retrieve images, XML, JSON or any other type of resource. It's often used to dynamically generate images or other media types, as well as making AJAX requests to the server. Most importantly, it differs from the other two in that the portlet has full control of the data that is sent in response.

Let's change the edit.jsp file to use an *actionURL*, using the JSP tag of the same name. We'll also remove the previous code that was saving the preference. Overwrite the edit.jsp file contents with the following:

```

<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>

<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>
<%@ page import="com.liferay.portal.kernel.util.Validator" %>
<%@ page import="javax.portlet.PortletPreferences" %>

<portlet:defineObjects />

```

```

<%
    PortletPreferences prefs = renderRequest.getPreferences();
    String greeting = (String)prefs.getValue(
        "greeting", "Hello! Welcome to our portal.");
%>

<portlet:actionURL var="editGreetingURL">
    <portlet:param name="mvcPath" value="/edit.jsp" />
</portlet:actionURL>

<aui:form action="<% editGreetingURL %>" method="post">
    <aui:input label="greeting" name="greeting" type="text" value="<%=
        greeting %>" />
    <aui:button type="submit" />
</aui:form>

<portlet:renderURL var="viewGreetingURL">
    <portlet:param name="mvcPath" value="/view.jsp" />
</portlet:renderURL>

<p><a href="<% viewGreetingURL %>">&larr; Back</a></p>

```

Redeploy the portlet after making these changes; everything should work exactly like before. Well, almost. Unless you paid close attention, you may have missed something: the portlet no longer shows a message to the user that the preference has been saved after she clicks the save button. To implement that, information must pass from the action phase to the render phase, so that the JSP knows that the preference has just been saved and can show a message to the user.

3.5. *Passing Information from the Action Phase to the Render Phase*

There are two ways to pass information from the action phase to the render phase. The first way is through render parameters. In the `processAction` method you can invoke the `setRenderParameter` method to add a new parameter to the request. The render phase can read this:

```
actionResponse.setRenderParameter("parameter-name", "value");
```

From the render phase (in our case, the JSP), this value is read like this:

```
renderRequest.getParameter("parameter-name");
```

It's important to be aware that when invoking an action URL, the parameters specified in the URL are only readable from the action phase (that is within the `processAction` method). In order to pass parameter values to the render phase you must read them from the `actionRequest` and then invoke the `setRenderParameter` method for each parameter needed.



Tip: Liferay offers a convenient extension to the portlet specification through the `MVCPortlet` class to copy all action parameters directly as render parameters. You can achieve this by setting the following `init-param` in your `portlet.xml`:

```
<init-param>
```

```

<name>copy-request-parameters</name>
<value>true</value>
</init-param>

```

One final note about render parameters: the portal remembers them for all later executions of the portlet until the portlet is invoked with *different* parameters. That is, if a user clicks a link in our portlet and a render parameter is set, and then the user continues browsing through other portlets on the page, each time the page is reloaded, the portal renders our portlet using the render parameters that we initially set. If we used render parameters in our example, then the success message will be shown not only right after saving, but also every time the portlet is rendered until the portlet is invoked again *without* that render parameter.

The second way of passing information from the action phase to the render phase is not unique to portlets, so it might be familiar to you--using the session. Your code can set an attribute in the `actionRequest` that is then read from the JSP. In our case, the JSP would also immediately remove the attribute from the session so the message is only shown once. Liferay provides a helper class and taglib to do this operation easily. In the `processAction` method, you need to use the `SessionMessages` class:

```

package com.liferay.samples;

import java.io.IOException;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.PortletException;
import javax.portlet.PortletPreferences;
import com.liferay.portal.kernel.servlet.SessionMessages;
import com.liferay.util.bridges.mvc.MVCPortlet;

public class MyGreetingPortlet extends MVCPortlet {
    @Override
    public void processAction(
        ActionRequest actionRequest, ActionResponse actionResponse)
        throws IOException, PortletException {
        PortletPreferences prefs = actionRequest.getPreferences();
        String greeting = actionRequest.getParameter("greeting");

        if (greeting != null) {
            prefs.setValue("greeting", greeting);
            prefs.store();
        }

        SessionMessages.add(actionRequest, "success");
        super.processAction(actionRequest, actionResponse);
    }
}

```

Next, in `view.jsp`, add the `liferay-ui:success` JSP tag and add the taglib declarations below:

```

<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>

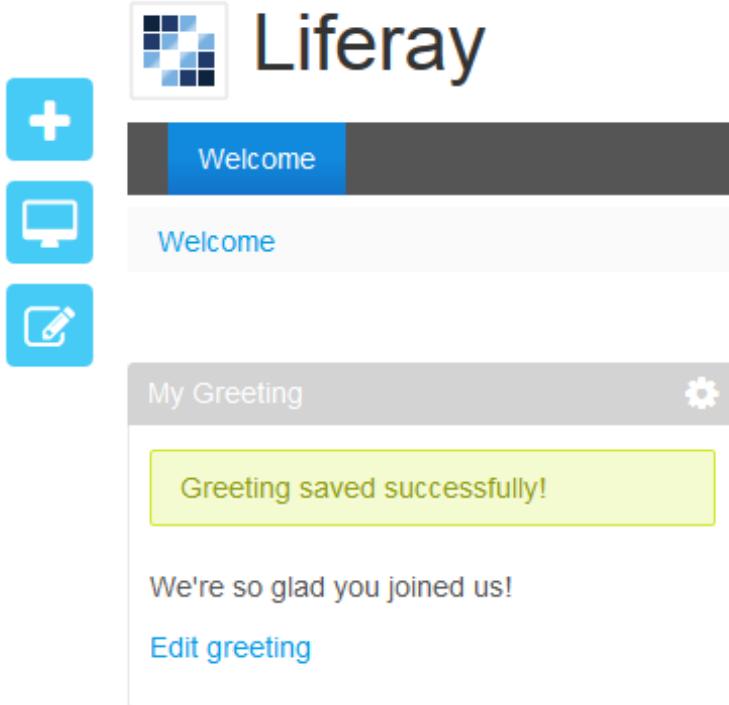
```

```

<%@ page import="javax.portlet.PortletPreferences" %>
<portlet:defineObjects />
<liferay-ui:success key="success" message="Greeting saved successfully!" />
<% PortletPreferences prefs = renderRequest.getPreferences(); String greeting = (String)prefs.getValue("greeting", "Hello! Welcome to our portal."); %>
<p><%= greeting %></p>
<portlet:renderURL var="editGreetingURL">
    <portlet:param name="mvcPath" value="/edit.jsp" />
</portlet:renderURL>
<p><a href="<% editGreetingURL %>">Edit greeting</a></p>

```

After this change, redeploy the portlet, go to the edit screen, edit the greeting, and save it. You should see a nice message that looks like this:



```

quest, "success");
}
catch(Exception e) {
    SessionErrors.add(actionRequest, "error");
}

```

Your view.jsp shows the error message in your portlet, if an error occurs while processing the action request.

There's also an equivalent utility class for error notification. You can add the liferay-ui:error tag to your view.jsp after the liferay-ui:success tag:

```
<liferay-ui:error key="error" message="Sorry, an error prevented saving your greeting" />
```

This error utility is commonly used after catching an exception in the processAction method. For example:

```
try {
    prefs.setValue("greeting", greeting);
    prefs.store();
}
```

```
SessionMessages.add(actionRe
```

Your request failed to complete.

Sorry, an error prevented saving your greeting

Hello! Welcome to our portal.

Edit greeting

The first message is automatically added by Liferay. The second one is the one you added in your JSP. You've successfully created and rendered your portlet's error message. Terrific!

Have you ever wondered how Liferay Portal determines which portlet to associate with a request parameter--especially when the portal receives

multiple parameters, with the same name, coming from different portlets? Each of Liferay's core portlets namespaces its request parameters, so that Liferay can distinguish them from other request parameters. And you can leverage namespacing in your portlets, too. Let's discuss portlet namespacing and how to turn on/off the portal's namespacing logic for a portlet.

3.5.1. Using Portlet Namespacing

Namespacing ensures that a given portlet's name is uniquely associated with elements in request parameters it sends to the portal. This prevents name conflicts with other elements on the portal page and with elements from other portlets on the page. Namespacing your portlet elements is easy. Simply use the `<portlet:namespace />` tag to produce a unique value for your portlet's elements. The following example code uses the `<portlet:namespace />` tag to reference the portlet's *fm* form during submission:

```
submitForm(document.<portlet:namespace />fm);
```

To illustrate the benefits of namespacing an element, such as the *fm* form from the example code above, suppose you have portlets named A and B in your portal and they both have a form named *fm*. Without portlet namespacing, the portal would be unable to differentiate between the two forms and, likewise, would be unable to determine their associated portlets. But, submitting both portlet A's form and portlet B's form as `<portlet:namespace />fm` would distinguish the forms as `*_Afm*` and `*_Bfm*`, respectively. Liferay associates each namespaced element, such as these namespaced forms, with the portlet that produced it.

By default, Liferay only allows *namespaced* parameters to access portlets. However, many third-party portlets send *unnamespaced* parameters. Therefore, Liferay gives you the option to turn off the unnamespaced parameters filter for portlets, to avoid third-party portlets from breaking. To turn the filter off for a portlet, navigate to the portlet's `liferay-portlet.xml` file and enter the following tag:

```
<requires-namespaced-parameters>false</requires-namespaced-parameters>  
Turning this filter off is on a per portlet basis, so you'll need to set the <requires-  
namespaced-parameters/> tag to false for every third-party portlet that sends  
unnamespaced parameters.
```

Interested in developing your custom portlet with multiple actions? Then you'll definitely want to check out the next section!

3.6. ***Developing a Portlet with Multiple Actions***

Right now our portlet only has two views: the default view and edit view. Adding more views is easy, and you can link to them using the `mvcPath` parameter in your `renderURL`. But we only have one action. What if we want to add another action, like sending an email to the user?

You can have as many actions as you want in a portlet. Implement each one as a method that receives two parameters: an `ActionRequest` and an `ActionResponse`. Name the method whatever you want, but note that the method name must match the URL name that points to it.

Let's rewrite the example from the previous section using a custom name for the action method that sets the greeting, and add a second action method for sending emails.

```
public class MyGreetingPortlet extends MVCPortlet {
    public void setGreeting(
        ActionRequest actionRequest, ActionResponse actionResponse)
    throws IOException, PortletException {
        PortletPreferences prefs = actionRequest.getPreferences();
        String greeting = actionRequest.getParameter("greeting");

        if (greeting != null) {
            try {
                prefs.setValue("greeting", greeting);
                prefs.store();
                SessionMessages.add(actionRequest, "success");
            }
            catch(Exception e) {
                SessionErrors.add(actionRequest, "error");
            }
        }
    }

    public void sendEmail(
        ActionRequest actionRequest, ActionResponse actionResponse)
    throws IOException, PortletException {
        // Add code here to send an email
    }
}
```

We no longer need to invoke the `processAction` method of the super class, since we're not overriding it.

This name change also requires a simple change in the URL so its name matches the method that is invoked to execute the action. In the `edit.jsp`, modify the `actionURL` so it looks like this:

```
<portlet:actionURL var="editGreetingURL" name="setGreeting">
    <portlet:param name="mvcPath" value="/edit.jsp" />
</portlet:actionURL>
```

Now you know all the basics of portlet development, and can use your Java knowledge to build portlets that get integrated in Liferay. Let's put the finishing touches on your portlet by first

learning about an extension to Liferay's portlet specification that generates more elegant URLs for your portlets.

3.7. Adding Friendly URL Mapping to the Portlet

When you click the *Edit greeting* link, you're taken to a page with a URL that looks like this:

```
http://localhost:8080/web/guest/home?p_p_id=mygreeting_WAR_mygreetingportlet
&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&p_p_col_id=column-1
&p_p_col_count=2&_mygreeting_WAR_mygreetingportlet_mvcPath=%2Fedit.jsp
```

Since Liferay 6, there's a built-in feature that can easily change the ugly URL above to this:

```
http://localhost:8080/web/guest/home/-/my-greeting/edit
```

The feature is called friendly URL mapping. It takes unnecessary parameters out of the URL and allows you to place the important parameters in the URL path, rather than in the query string. To add this functionality, first edit `liferay-portlet.xml` and add the following lines directly after `</icon>` and before `<instanceable>` (remove the line breaks):

```
<friendly-url-mapper-class>com.liferay.portal.kernel.portlet.DefaultFriendlyURLMapper</friendly-url-mapper-class>
<friendly-url-mapping>my-greeting</friendly-url-mapping>
<friendly-url-routes>com/liferay/samples/my-greeting-friendly-url-routes.xml\
</friendly-url-routes>
```

Next, create the file (remove the line break):

```
my-greeting-portlet/docroot/WEB-INF/src/com/liferay/samples/my\
-greeting-friendly-url-routes.xml
```

Place the following content into the new file (remove the line break after

```
{mvcPathName}.jsp):
<?xml version="1.0"?>
<!DOCTYPE routes PUBLIC "-//Liferay//DTD Friendly URL Routes 6.2.0//EN"
"http://www.liferay.com/dtd/liferay-friendly-url-routes_6_2_0.dtd">
```

```
<routes>
  <route>
    <pattern>/ {mvcPathName} </pattern>
    <generated-parameter name="mvcPath">/ {mvcPathName}.jsp \
    </generated-parameter>
  </route>
</routes>
```

Redeploy your portlet, refresh the page, and look at the URL after clicking the *Edit greeting* link. Notice how much shorter and more user-friendly the URL is, without even having to modify the JSPs.

The screenshot shows a Liferay web application interface. At the top, there's a header bar with a back arrow icon and the URL "localhost:8080/web/guest/home/-/my-greeting/edit". Below the header is the Liferay logo. A navigation bar contains a "Welcome" tab, which is highlighted in blue, and a "Welcome" link below it. The main content area has a title "My Greeting". Underneath, there's a "Greeting" section containing a text input field with the placeholder "Hello You! Welcome to our porta". Below the input field is a blue "Save" button. At the bottom left of the content area is a "← Back" link.

For more information on friendly URL mapping, there's a detailed discussion in *Liferay in Action*. Our next step here is to explore localization of the portlet's user interface.

3.8. Localizing Your Portlets

If your portlets

target an international audience, you can localize your portlets' user interfaces. To localize a portlet, you need to create language properties files, also called resource bundles, for each language you wish to support. You can translate language properties manually or use a web service to translate them for you. Conveniently, all of the translated messages used by Liferay Portal are also accessible to plugin projects. To localize messages in addition to portal's localized messages, you must create language keys in one or more resource bundles within your plugin project. When planning your portlet's localization, you should consider the following questions.

Are there messages that Portal uses that you'd like to use in your portlets? Does your plugin contain multiple portlets? If so, do any of its portlets need to be available for administrative purposes in the Control Panel? If any of its portlets need to be in the Control Panel, you should create separate resource bundles for each of these portlets. Otherwise, your portlets should share the same resource bundle so that you can leverage Liferay's language building capabilities from Liferay IDE and the Plugins SDK. We'll show you how to localize your portlets in all of these scenarios. Let's start by leveraging the messages that Liferay Portal has already localized in its core set of language keys.

3.8.1. Using Liferay's Language Keys

Liferay specifies a host of language keys in its core `Language.properties` file found in the content folder of your `portal-impl.jar`, or `portal-impl/src/content` of your Liferay Portal source tree. Leveraging Portal's core language keys saves you time, since these keys always have up to date translations for multiple languages. Additionally, your portlet blends better into Liferay's UI conventions.

You can use a language key in your JSP via a `<liferay-ui:message />` tag.

```
<liferay-ui:message key="message-key" />
```

You specify the message key corresponding to the language key in the `Language.properties` file you want to display. For example, to welcome a user in their language, specify the message key named `welcome`.

```
<liferay-ui:message key="welcome" />
```

This key maps to the word "Welcome", in your translation of it to the user's locale. Here is the `welcome` language key from Liferay's `Language.properties` file.

```
welcome=Welcome
```

Let's add the `welcome` language key in front of our greeting in the `view.jsp` file of the `my-greeting-portlet` we created earlier. Replace its current greeting paragraph with this:

```
<p><liferay-ui:message key="welcome" />! <%= greeting %></p>
```

Revisit the page to confirm that the word "Welcome", from `Language.properties`, now precedes your greeting!

Note, in order to use the `<liferay-ui:message />` tag, or any of the `liferay-ui` tags, you must include the following line in your JSP. It imports the `liferay-ui` tag library.

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

The `<liferay-ui:message />` tag also supports passing strings as arguments to a language key. For example, the `welcome-x` key expects one argument. Here is the `welcome-x` key from the `Language.properties` file:

```
welcome-x=Welcome{0}!
```

It references `{0}`, which denotes the first argument of the argument list. An arbitrary number of arguments can be passed in via a message tag, but only those arguments expected by the language key are used. The arguments are referenced in order as `{0}`, `{1}`, etc. Let's pass in the user's screen name as an argument to the `welcome-x` language key in the "My Greeting" portlet.

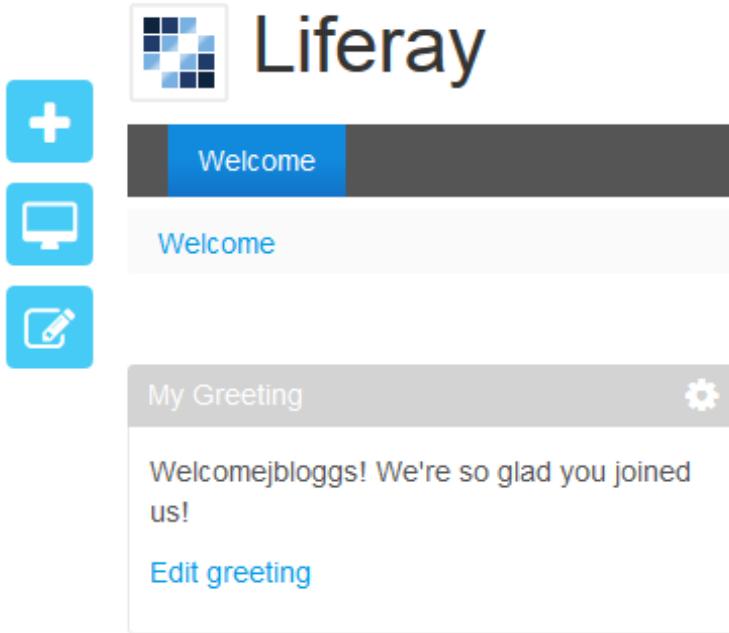
1. Open the `view.jsp` file.
2. Add the following lines near the top of the JSP, just above the `<portlet:defineObjects />` tag. The first line imports the `liferay-theme` tag library. The second line defines the library's objects, providing access to the `user` object holding the user's screen name.

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

```
<liferay-theme:defineObjects />
```

3. Replace the current welcome message tag and exclamation point, `<liferay-`

```
ui:message key="welcome" />!, in the JSP with the following:  
<liferay-ui:message key="welcome-x" arguments="<% user.getScreenName() %>" />
```



When you refresh your page, your "My Greeting" portlet greets you by your screen name!

Other message tags you'll want to use are the `<liferay-ui:success />` and `<liferay-ui:error />` tags. The `<liferay-ui:success />` helps you give positive feedback, marked up in a pleasant green background. The `<liferay-ui:error />` tag helps you warn your users of invalid input or exceptional conditions. Error messages are marked up in an appropriately alarming red background.

The `<liferay-ui:success />` tag is triggered when its key value is found in the `SessionMessages` object. Earlier in our `MyGreetingPortlet` class, we triggered the success message `<liferay-ui:success key="success" ... />` by adding its key to the `SessionMessages` object with the following call:

```
SessionMessages.add(actionRequest, "success");
```

Similarly, the `<liferay-ui:error />` tag is triggered when its key value is found in the `SessionErrors` object. Likewise, in our `MyGreetingPortlet` class, we triggered the error message `<liferay-ui:error key="error" ... />` by adding its key to the `SessionErrors` object with the following call:

```
SessionErrors.add(actionRequest, "error");
```

That's all you need to do to leverage Liferay's core localization keys. If you need to add localization keys, follow the instructions below to deliver locally tailored portlets to your customers.

3.8.2. Sharing Language Keys Between Your Portlets

It's likely that you'll have messages that you want to localize that aren't one of Liferay's core language keys. So you'll need to specify these language keys in one or more resource bundles in your plugin. If one of your portlets is going to be used in the Control Panel and you want to localize its title and description used in the Control Panel, then it's best to use a separate resource bundle for that portlet. If none of your portlets are going to be used in the Control Panel, then the portlets can share the same resource bundle. We'll show you how to share a resource bundle

between portlets first.

Let's add a resource bundle to the `event-listing-portlet` plugin project we created earlier in Chapter 2:

1. Create a content package in your project's source folder `doocroot/WEB-INF/src`.
2. Create a file `Language.properties` in the content folder you just created and add the following language key:

```
your-nose-knows-best=Your nose knows best
```

3. Create another language key file `Language_es.properties` in the content folder and add the equivalent `your-nose-knows-best` key translated to Spanish:

```
your-nose-knows-best=La nariz sabe mejor
```

4. Add the following line below your last included JSP (e.g., after `<%@include file="/html/init.jsp" %>`) in the `view.jsp` files for each of your portlets. This line brings your translated language key value into your JSP:

```
Nose-ster - <liferay-ui:message key="your-nose-knows-best" />!
```

5. For both portlets, update their `<portlet>` node in the `portlet.xml` file to refer to the same resource bundle:

```
<portlet>
    <portlet-name>eventlisting</portlet-name>
    ...
    <resource-bundle>content.Language</resource-bundle>
    <portlet-info>...</portlet-info>
    ...
</portlet>
<portlet>
    <portlet-name>locationlisting</portlet-name>
    ...
    <resource-bundle>content.Language</resource-bundle>
    <portlet-info>...</portlet-info>
    ...
</portlet>
```

Make sure to put each `resource-bundle` element in its proper place in the `portlet` element. See the `portlet.xml` file's schema http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd for details.

6. Redeploy the plugin and go to the page where you added the Event Listing and Location Listing portlets to verify that they display the same message "Nose-ster - Your nose knows best!".
7. Switch your portal's locale to Spanish by adding `/es` after `localhost:8080` and refresh the page. Notice how both portlets display your translated language key.

At this point any language keys you specify in the `Language.properties` file are accessible from either of the portlets.



Note: It's best to use the Liferay naming convention for language bundle file and folder so your portlets can access the bundle and you can use the automatic language building capabilities of Liferay IDE and the Plugins SDK with the bundle.

Before we cover localizing Control Panel portlets, let's learn how Liferay facilitates generating language key files and translating the keys to languages you want to support.

3.8.3. Generating Language Properties File and Automated Translations

In order for a user to see a message in his own locale, the message value must be specified in a resource bundle file with a name ending in his locale's two character code. For example, a resource bundle file named `Language_es.properties` containing a message property with key `welcome` must be present with a Spanish translation of the word "Welcome". Don't worry, the Plugins SDK provides a means for you to get translations for your default resource bundle.

The Plugins SDK uses the Bing Translator service <http://www.microsofttranslator.com/> to translate all of the resources in your `Language.properties` file to multiple languages. It provides a base translation for you to start with. To create base translations using the Bing Translator service, you'll need to do the following:

1. Sign up for an Azure Marketplace account and register your application. Be sure to write down your client ID and client secret given to you for your application.
2. Make sure that you have a `build.[username].properties` file in your Plugins SDK root directory. This `build.[username].properties` file should contain a reference to a Liferay bundle. If you have a Liferay Tomcat bundle, for example, your reference should look like this:

```
app.server.dir=[Liferay Home]/tomcat-7.0.42
auto.deploy.dir=[Liferay Home]/deploy
```

[Liferay Home] refers to your bundle's root directory.

3. Edit the `portal-ext.properties` file in your Liferay Home directory by adding the following two lines replaced with your values:

```
microsoft.translator.client.id=your-[client-id]
microsoft.translator.client.secret=your-[client-secret]
```

Liferay copies the `portal-ext.properties` file from your Liferay Home directory to the `tomcat-[version]/webapps/ROOT/WEB-INF/classes` directory upon startup. So either start Liferay or manually copy your `portal-ext.properties` file from Liferay Home to this location.

4. Edit the `Language.properties` file of the plugin for which you'd like to add properties to be translated. For example, if you have a `hello-world` portlet in your Plugins SDK, you'd edit the following file:

`[Liferay Plugins SDK]/portlets/hello-world-portlet/docroot/WEB-INF/src/content/Language.properties`

You can add properties, remove properties, or edit properties. However, translations will *not* be generated for existing properties.

5. Run `ant build-lang` from the plugin directory of the plugin for which you'd like to generate translations. For example, in the case of the `hello-world` portlet example, you'd run `ant build-lang` from the `[Liferay Plugins SDK]/portlets/hello-world-portlet` directory.

When the build completes, you'll find the generated files with all of the translations in the same folder as your `Language.properties` file.



Note: Since translations aren't generated for existing properties, use two steps if you need to edit existing properties. First, remove the properties from `Language.properties` and run `ant build-lang` to remove the properties from all the other resource bundles. Then re-add the properties with new values and run `ant build-lang` again. Now the Microsoft Translator should generate new translations for your properties.



Note: If you're Mavenizing your portlet, make sure to copy your `content` folder into your portlet's `src/main/webapp/WEB-INF/classes` folder.

By using the Plugins SDK's language building capability, you can keep all created translations synchronized with your default `Language.properties`. You can run it any time during development. It significantly reduces the time spent on the maintenance of translations. However, remember that a *machine* translation is generated by the Microsoft Translator. Machine translations can often come across as rude or (unintentionally) humorous. Sometimes they are simply inaccurate. Someone fluent in each language should review the translations before the translations are deployed to a production environment.

Now that you know how to create a shared resource bundle and how to generate translations, let's consider why you may need to use separate resource bundles for each portlet. For example, to localize the title and description of each of your plugin's Control Panel-enabled portlets, you

must use separate resource bundles. We'll show you how to implement them.

3.8.4. Localizing Control Panel Portlets

You may have noticed that your Control Panel-enabled portlets are missing that super-fancy must-have portlet title and description in the Control Panel. To make your portlet look cool within the Control Panel, create specially tailored description and title keys in separate `Language.properties` files for each portlet in your project. You'll use the `javax.portlet.title` and `javax.portlet.description` language keys.

For demonstration purposes, let's consider a project that has one portlet named `eventlisting` and another portlet named `locationlisting`. We'll need to create a resource bundle for each of them to specify their localized title and description values.



Note: If your project only has one portlet, it's best to put your resource bundle directly in the `content` folder. Specifying your bundle in file `content/Language.properties` lets you leverage the Plugins SDK's language building capabilities, via right-clicking on the `Language.properties` file → Liferay → Build Languages in Developer Studio or executing `ant build-lang` from the terminal.

Here's what you'd do to localize the title and description for each portlet in the project:

1. If you haven't done so already, configure each portlet to display in the Control Panel. For our example, we would display them in the *Content* portion and give them an arbitrary *weight* value for determining where they're to be placed in the column with respect to the other portlets. Here's a sample of how to specify this in our project's `liferay-portlet.xml` file (Replace line breaks):

```
<portlet>
  <portlet-name>eventlisting</portlet-name>
  <icon>/icon.png</icon>
  <control-panel-entry-category>site_administration.content</control-
  panel-entry-category>
  <control-panel-entry-weight>1.5</control-panel-entry-weight>
  ...
</portlet>
<portlet>
  <portlet-name>locationlisting</portlet-name>
  <icon>/icon.png</icon>
  <control-panel-entry-category>site_administration.content</control-
  panel-entry-category>
  <control-panel-entry-weight>1.6</control-panel-entry-weight>
  ...
</portlet>
```

2. Create a namespaced folder to hold each portlet's resource bundle. It's a best practice to name each resource bundle folder based on the name of its portlet.

For example, you could create a resource bundler folder `content/eventlisting` for the `eventlisting` portlet and a folder `content/locationlisting` for the `locationlisting` portlet.

3. Create a `Language.properties` file in the resource bundle folders you just created. Specify the `javax.portlet.title` and `javax.portlet.description` language key/values in each of these `Language.properties` files.

The `eventlisting` portlet could have the following key/value pairs in its `content/eventlisting/Language.properties` file:

```
javax.portlet.title=Event Listing Portlet  
javax.portlet.description=Lists important upcoming events.
```

And the `locationlisting` portlet could have these key/value pairs in its `content/locationlisting/Language.properties` file:

```
javax.portlet.title=Location Listing Portlet  
javax.portlet.description=Lists event locations.
```

4. Specify the resource bundles for the portlets in the project's `portlet.xml` file. The example `portlet.xml` file code snippet below demonstrates specifying the resource bundles for the `eventlisting` and `locationlisting` example portlets:

```
<portlet>  
    <portlet-name>eventlisting</portlet-name>  
    ...  
    <resource-bundle>content.eventlisting.Language</resource-bundle>  
    <portlet-info>...</portlet-info>  
    ...  
</portlet>  
<portlet>  
    <portlet-name>locationlisting</portlet-name>  
    ...  
    <resource-bundle>content.locationlisting.Language</resource-bundle>  
    <portlet-info>...</portlet-info>  
    ...  
</portlet>
```

5. Redeploy your plugin project.
6. Go to the Control Panel and select the Event Locations portlet.
7. Add `en` to your portal context in your URL to interface with the portal in Spanish. For example, your URL would start like this:

`http://localhost:8080/es/group/control_panel/...`

Portal's Control Panel displays your portlet's localized title and description.

The screenshot shows the Liferay Control Panel interface. The URL in the address bar is `localhost:8080/es/group/control_panel/manage?p_p_id=locationlisting_WAR_eventlistingportlet&p_p_lifecycle=0&p_p_state=maximized`. The title bar says "Liferay ▾ / Administración de sitio web". On the left, there's a sidebar with links: "Páginas" (with a right arrow), "Contenido" (with a dropdown arrow), "Eventos", "Lugares de Eventos" (which is highlighted in blue), and "Contenido Web". The main content area is titled "Lugares de Eventos" with a question mark icon. A tooltip says "Lista contiene los lugares de actividades." Below it, the text "This is the Location L" is partially visible. There's a button labeled "Añadir localización". A message box says "There are currently no locations."

You're becoming an expert localizer!



Tip: Do you know how your portlet title is processed? If your portlet doesn't define a resource bundle or `javax.portlet.title`, the portal container next checks the `<portlet-info>` and inner `<portlet-title>` node in the `portlet.xml` descriptor. If they're missing too, the `<portlet-name>` node value is rendered as portlet title.



Note: Be aware that using Struts portlet and referring to a `StrutsResource` bundle in your `portlet.xml` engages a different title and description algorithm. Titles and long titles are pulled using two different keys:

- `javax.portlet.long-title`
- `javax.portlet.title`

Now that you're comfortable localizing portlet content, you may want to learn how to make translations available throughout the portal or how to override an existing translation. For instructions on doing that, refer to Chapter 10 of this guide, specifically the *Overriding a Language.properties File* section. It describes how to use a hook to override existing Liferay translations. You can share your keys with other portlets, as well as override existing Liferay translations.

Next, let's learn how to configure your portlets' preferences using configuration actions.

3.9. Implementing Configurable Portlet Preferences

Portlet Preferences are properties for storing basic portlet configuration data. Preferences are often used by administrators to provide customized views of a portlet to subsets of users or even all of a portlet's users. Portlet preferences are sometimes made accessible to users themselves for configuring a portlet just the way they like it. Liferay simplifies making portlet preferences configurable in your portlet's JSPs. In this section, we'll show you how to create a default configuration JSP page and add a portlet preferences control to it.

We'll use the Location Listing Portlet we developed in the Generating Your Service Layer chapter. We'll create a configuration page and add a custom option to it, allowing administrators to hide the address portion of the locations. Let's dive into portlet preferences by running through an example of creating a configuration page for the Location Listing Portlet and setting up a new portlet preference for it.

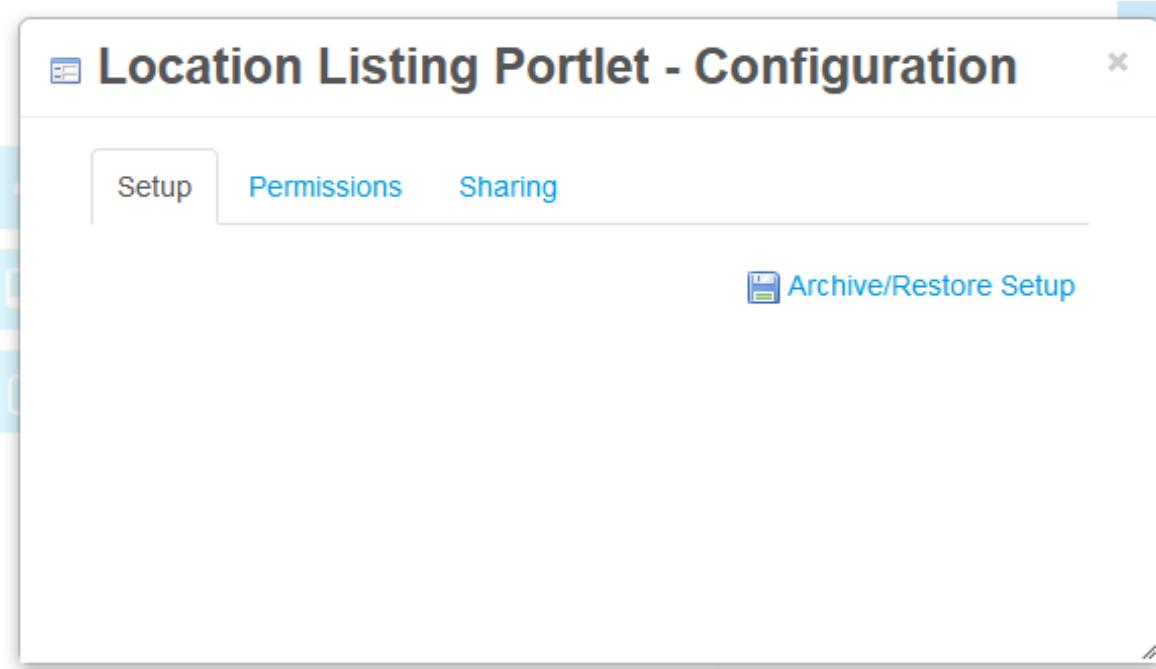
First, if your Location Listing Portlet doesn't already have a setup tab in its configuration page, you'll need to follow the steps below to create one. To check to see whether your portlet has a setup tab, click the portlet's options icon in the upper right corner and select *Configuration*. If you already have a *Setup* tab, you can skip the next section. Otherwise we'll show you how to create the default setup tab for your portlet's configuration page.

3.9.1. Creating a Default Setup Tab in the Portlet's Configuration Page

Open the `liferay-portlet.xml` file and add the element `<configuration-action-class>com.liferay.portal.kernel.portlet.DefaultConfigurationAction</configuration-action-class>` below your Location Listing Portlet's `<icon>...</icon>` tag. Here's a snippet to show you where it goes in the context of your `liferay-portlet.xml` file:

```
....  
<portlet>  
    <portlet-name>locationlisting</portlet-name>  
    <icon>/icon.png</icon>  
    <configuration-action-  
class>com.liferay.portal.kernel.portlet.DefaultConfigurationAction</configura-  
tion-action-class>  
    <header-portlet-css>/css/main.css</header-portlet-css>  
    <footer-portlet-javascript>  
        /js/main.js  
    </footer-portlet-javascript>  
    <css-class-wrapper>locationlisting-portlet</css-class-wrapper>  
</portlet>  
....
```

Notice that we've listed the *default* configuration action class. We'll update this tag with a custom configuration class later in the exercise. If you redeploy your portlet and open your portlet's *Configuration* page, you'll find the new *Setup* tab. It's empty for now. But we'll add a portlet preference control to it shortly.



In order to add a configurable portlet preference to the portlet, we must do the following things:

1. Specify a Configuration JSP in the `portlet.xml`
 2. Create the Configuration JSP for Displaying the Portlet Preference Options
 3. Create a Configuration Action Implementation Class for Processing the Portlet Preference Value
 4. Modify the View JSP to Respond to the Current Portlet Preference Value
- Let's specify a configuration JSP file, first.

3.9.2. Step 1: Specify a Configuration JSP in the `portlet.xml`

Your portlet will need a way to display configuration options to the user. Liferay checks to see if your portlet specifies a configuration JSP via a `config-template` initialization parameter in your `portlet.xml` file. Let's specify one for the Location Listing Portlet.

Open the `portlet.xml` file and insert the following lines after the Location Listing Portlet's `<portlet-class>...</portlet-class>` tag:

```
<init-param>
  <name>config-template</name>
  <value>/html/locationlisting/configuration.jsp</value>
</init-param>
```

3.9.3. Step 2: Create the Configuration JSP for Displaying the Portlet Preference Options

We'll create a configuration JSP file and add JavaScript to let the user select a portlet preference

value. For our example, we'll provide a custom option in the configuration page's setup tab, allowing administrators to show or hide location address information.

In the docroot/html/locationlisting directory, create a file named configuration.jsp, if you don't have one already.

Now let's begin creating our portlet preference for the configuration page's setup content. Assuming that you began with a blank configuration.jsp file, add the following code to it:

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

<liferay-portlet:actionURL portletConfiguration="true" var="configurationURL"
/>

<%
boolean showLocationAddress_cfg =
GetterUtil.getBoolean(portletPreferences.getValue("showLocationAddress",
StringPool.TRUE));
%>

<aui:form action="<%= configurationURL %>" method="post" name="fm">
    <aui:input name="<%= Constants.CMD %>" type="hidden" value="<%= Constants.UPDATE %>" />

    <aui:input name="preferences--showLocationAddress--" type="checkbox"
value="<%= showLocationAddress_cfg %>" />

    <aui:button-row>
        <aui:button type="submit" />
    </aui:button-row>
</aui:form>
```

The showLocationAddress_cfg variable holds the current value of whether to show the location addresses or not. The input checkbox lets the user set the value of the showLocationAddress portlet preference *key*. In order for the value to be persisted, the input must follow the naming convention preferences--somePreferenceKey--. For our example, the input name preferences--showLocationAddress-- maps the input value to a portlet preference key named showLocationAddress.

You've probably noticed some JSP compile errors and warnings with respect to the code we've added. We'll address them by adding directives to the init.jsp that our configuration JSP includes. Adding the directives will allow the JSP to access the classes and taglibs we're using.

In your init.jsp file, add the following directives:

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

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

<%@ page import="com.liferay.portal.kernel.util.Constants" %>
<%@ page import="com.liferay.portal.kernel.util.GetterUtil" %>
<%@ page import="com.liferay.portal.kernel.util.StringPool" %>
```

```
<liferay-theme:defineObjects />
```

The `tablib` directives access the JSP Standard Tag Library (JSTL), Liferay's theme taglib, and Liferay's portlet taglib. Then, we added directives for importing the classes we're using. Lastly, we inserted the `<portlet:defineObjects />` tag to access implicit variables that we'll need. It provides useful portlet variables such as `renderRequest`, `portletConfig`, and `portletPreferences`.

Your `configuration.jsp` is all set to display your portlet preference options. Let's implement a custom class to handle the configuration action.

3.9.4. Step 3: Create a Configuration Action Implementation Class for Processing the Portlet Preference Value

Now let's create a custom configuration action class for accessing the portlet preference. We'll have it extend the `DefaultConfigurationAction` class.

Create a package named `com.nosester.portlet.eventlisting.action` in the portlet's docroot/WEB-INF/src directory. In the new package, create a class named `ConfigurationActionImpl`, and specify `DefaultConfigurationAction` as its superclass.

Replace the contents of `ConfigurationActionImpl.java` with the following code:

```
package com.nosester.portlet.eventlisting.action;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.PortletConfig;
import javax.portlet.PortletPreferences;

import com.liferay.portal.kernel.portlet.DefaultConfigurationAction;

public class ConfigurationActionImpl extends DefaultConfigurationAction {

    @Override
    public void processAction(
        PortletConfig portletConfig, ActionRequest actionRequest,
        ActionResponse actionResponse) throws Exception {

        super.processAction(portletConfig, actionRequest, actionResponse);

        PortletPreferences prefs = actionRequest.getPreferences();

        String showLocationAddress = prefs.getValue(
            "showLocationAddress", "true");

        System.out.println("showLocationAddress=" + showLocationAddress +
            " in ConfigurationActionImpl.processAction() .");
    }
}
```

Notice we've extended the `DefaultConfigurationAction` class and added a new

`processAction()` method. The super-class's `processAction()` method is responsible for reading the portlet preferences from the configuration form and storing them in the database. Usually, you'd add appropriate validation logic for the parameters received from the form. Our bare-bones example simply demonstrates accessing the preferences from the action request.

Another common method to include in a custom configuration action class is a `render()` method. The render method is invoked when the user clicks the configuration icon. For this example, we'll stick with the original render method from the `DefaultConfigurationAction` class we're extending.



Note: You won't need to store portlet preferences by calling `preferences.store()` since they're automatically stored in the `DefaultConfigurationAction` class, which your configuration class extends.

Lastly, let's specify our new custom configuration class in the `liferay-portlet.xml`. Replace the existing `<configuration-action-class>...</configuration-action-class>` with `<configuration-action-class>com.nosester.portlet.eventlisting.action.ConfigurationActionImpl</configuration-action-class>`. Here's a snippet to show you where it goes in the context of the `liferay-portlet.xml` file:

```
....  
<portlet>  
    <portlet-name>locationlisting</portlet-name>  
    <icon>/icon.png</icon>  
    <configuration-action-  
class>com.nosester.portlet.eventlisting.action.ConfigurationActionImpl</confi  
guration-action-class>  
    <header-portlet-css>/css/main.css</header-portlet-css>  
    <footer-portlet-javascript>  
        /js/main.js  
    </footer-portlet-javascript>  
    <css-class-wrapper>locationlisting-portlet</css-class-wrapper>  
</portlet>  
....
```

Since your configuration action implementation is ready to process your portlet preference, let's update the view JSP to respond to the portlet preference.

3.9.5. Step 4: Modify the View JSP to Respond to the Current Portlet Preference Value

Let's add logic in our `view.jsp` to show/hide the location addresses based on the value of our portlet preference key `showLocationAddress`.

In the `view.jsp` file, we'll get the value of the `showLocationAddress` portlet preference key. If its value is `true`, we'll display all of the location fields, including the address fields; otherwise, we'll hide the address fields.

Below are the contents of the entire `view.jsp` file. We'll point out the code that handles the portlet preference, after this code listing:

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

This is the <b>Location Listing Portlet</b> in View mode.

<%
    String redirect = PortalUtil.getCurrentURL(renderRequest);
%>

<aui:button-row>
    <portlet:renderURL var="addLocationURL">
        <portlet:param name="mvcPath"
value="/html/locationlisting/edit_location.jsp" />
        <portlet:param name="redirect" value="<%= redirect %>" />
    </portlet:renderURL>

    <aui:button onClick="<%= addLocationURL.toString() %>" value="add-
location" />
</aui:button-row>

<%
boolean showLocationAddress_view =
GetterUtil.getBoolean(portletPreferences.getValue("showLocationAddress",
StringPool.TRUE));
%>

<liferay-ui:search-container emptyResultsMessage="location-empty-results-
message">
    <liferay-ui:search-container-results
        results="<%=
LocationLocalServiceUtil.getLocationsByGroupId(scopeGroupId,
searchContainer.getStart(), searchContainer.getEnd()) %>"
        total="<%=
LocationLocalServiceUtil.getLocationsCountByGroupId(scopeGroupId) %>"
    />

    <liferay-ui:search-container-row
        className="com.nosester.portlet.eventlisting.model.Location"
        keyProperty="locationId"
        modelVar="location" escapedModel="<%= true %>">
        <liferay-ui:search-container-column-text
            name="name"
            value="<%= location.getName() %>">
        />

        <liferay-ui:search-container-column-text
            name="description"
            property="description">
        />

    <c:choose>
```

```

<c:when test="<% showLocationAddress_view == true %>">
    <liferay-ui:search-container-column-text
        name="street-address"
        property="streetAddress"
    />

    <liferay-ui:search-container-column-text
        name="city"
        property="city"
    />

    <liferay-ui:search-container-column-text
        name="state-province"
        property="stateOrProvince"
    />

    <liferay-ui:search-container-column-text
        name="country"
        property="country"
    />
</c:when>
</c:choose>

<liferay-ui:search-container-column-jsp
    align="right"
    path="/html/locationlisting/location_actions.jsp"
/>
</liferay-ui:search-container-row>

<liferay-ui:search-iterator />

</liferay-ui:search-container>

```

Let's breakdown the above code. We start by getting the value of the `showLocationAddress` portlet preference key and assigning it to a boolean variable `showLocationAddress_view`:

```

<%
boolean showLocationAddress_view =
GetterUtil.getBoolean(portletPreferences.getValue("showLocationAddress",
StringPool.TRUE));
%>

```

If no `showLocationAddress` key is found, the value will default to `true` based on the `StringPool.TRUE` default value we pass to the method `portletPreferences.getValue(key, default)`. Then, we wrap the street address, city, state, and country column text elements in a conditional code block using `<c:choose><c:when test="..."> ... <c:when></c:choose>` tags:

```

<liferay-ui:search-container-row ... />
...
<c:choose>
    <c:when test="<% showLocationAddress_view == true %>">
        <liferay-ui:search-container-column-text
            name="street-address"
            property="streetAddress"

```

```

    />

    <liferay-ui:search-container-column-text
        name="city"
        property="city"
    />

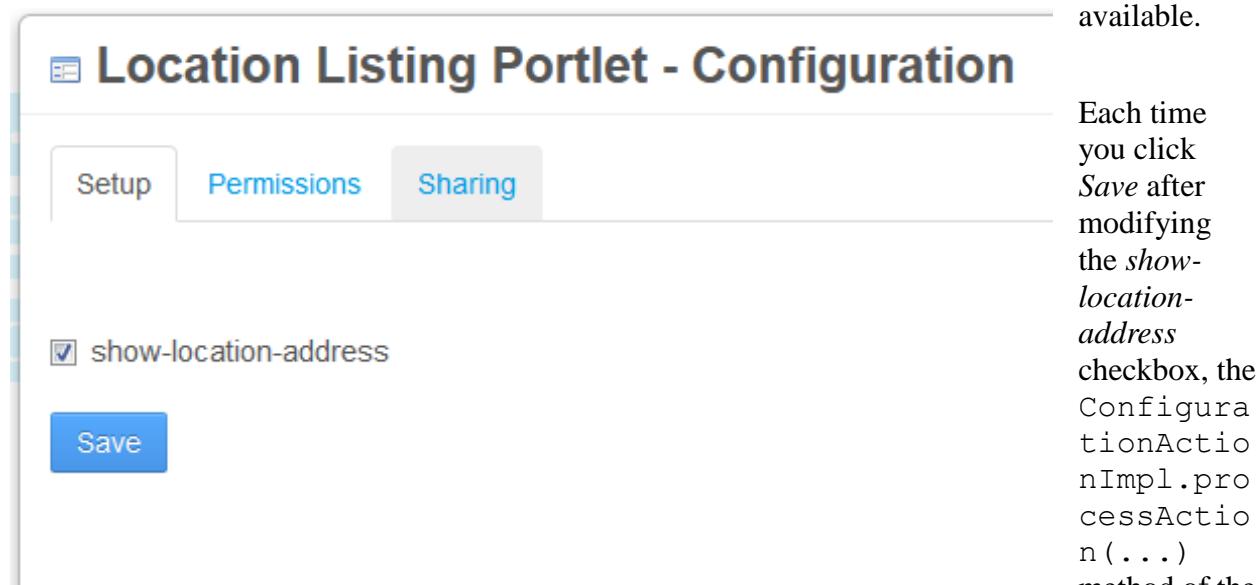
    <liferay-ui:search-container-column-text
        name="state-province"
        property="stateOrProvince"
    />

    <liferay-ui:search-container-column-text
        name="country"
        property="country"
    />
</c:when>
</c:choose>
...
</liferay-ui:search-container-row>
```

If `showLocationAddress_view` is true, *all* of the location's fields are displayed. If it is false, then the address fields are omitted.

That's it! You've created a custom configuration page and added a portlet preference to your portlet. Let's see the configuration page and portlet preference in action! Navigate to your Location Listing Portlet's *Configuration* page. You now have the `show-location-address`

checkbox available.



The screenshot shows the 'Location Listing Portlet - Configuration' page. At the top, there are three tabs: 'Setup', 'Permissions' (which is selected and highlighted in blue), and 'Sharing'. Below the tabs, there is a checkbox labeled 'show-location-address' with a checked mark. At the bottom left, there is a blue 'Save' button. To the right of the screenshot, there is a vertical text block that reads: 'Each time you click Save after modifying the show-location-address checkbox, the ConfigurationActionImpl.processAction(...) method of the'.

`ConfigurationActionImpl` class prints out the value of the `showLocationAddress` portlet preference key:

`showLocationAddress=true` in `ConfigurationActionImpl.processAction()`.

By unchecking the checkbox, the location addresses are hidden from view in the Location

Listing Portlet.

Name	Description	
Eiffel Tower	Beautiful steel structure right in the heart of Paris.	Actions
Golden Gate Bridge	Visitors are greeted by its international orange color.	Actions
Grand Canyon	That's some big whole in the ground!	Actions

Great job! Now you know how to use Liferay's portlet preferences in the portlets you develop. Next, let's use the Plugins SDK to create a plugin that extends another plugin.

3.10. *Creating Plugins to Extend Plugins*

For Liferay plugins, you can create a new plugin that extends an existing one. By extending a plugin, you can use all its features in your new plugin while keeping your changes/extensions separate from the existing plugin's source code.

To create a plugin which extends another, follow these steps:

1. Create a new empty plugin in the Plugins SDK.
2. Remove all the auto-generated files except `build.xml` and the `docroot` folder, which should be empty.
3. Copy the original WAR file of the plugin you'd like to extend (for example, `social-networking-portlet-6.2.0.1.war`) to the root folder of your new plugin.
4. Add the following line to your `build.xml` inside of the `<project>` tag to reference the original WAR file you are going to extend:

```
<property  
    name="original.war.file"  
    value="social-networking-portlet-6.2.0.1.war"
```

/>

5. Copy any files from the original plugin that you're overwriting to your new plugin (using the same folder structure) and run the Ant target merge. Please note that the merge target is called whenever the plugin is compiled. All you have to do is to check the Ant output:

```
dsanz@host:~/sdk/portlets/my-social-networking-portlet$ ant war  
Buildfile:  
/home/dsanz/sdk/portlets/my-social-networking-portlet/build.xml
```

compile:

```
merge:  
[mkdir] Created dir:  
/home/dsanz/sdk/portlets/my-social-networking-portlet/tmp  
[mkdir] Created dir:  
/home/dsanz/sdk/portlets/my-social-networking-portlet/tmp/WEB-INF/  
classes  
[mkdir] Created dir:  
/home/dsanz/sdk/portlets/my-social-networking-portlet/tmp/WEB-INF/  
lib  
  
merge-unzip:  
[unzip] Expanding:  
/home/dsanz/sdk/portlets/my-social-networking-portlet/social-  
networking-portlet-6.2.0.1.war into /home/dsanz/sdk/  
portlets/my-social-networking-portlet/tmp  
[copy] Copying 2 files to  
/home/dsanz/sdk/portlets/my-social-networking-portlet/tmp  
[mkdir] Created dir:  
/home/dsanz/sdk/portlets/my-social-networking-portlet/docroot/  
WEB-INF/classes
```

...

6. If the plugin that you're extending contains a service, you need to overwrite the ClpSerializer.java file. The Service Builder-generated ClpSerializer.java file contains a hard-coded project for _servletContextName. You need to change this to the name of your plugin.

This generates a plugin (you can find the WAR file in the /dist folder of your plugins SDK) which combines the original one with your changes.

3.11. Creating Plugins to Share Templates, Structures, and More

Have you ever wanted to share page templates with other users? Are colleagues and clients banging at your door to get a hold of the templates and structures you use for your web content articles and wikis? If so, you can bundle these up in a Liferay plugin to distribute to them. You can even put them in a Marketplace app for them to purchase. When they install your plugin, its templates and structures are automatically imported into their portal's global site. How is this possible? The Templates Importer feature of the Resources Importer app, makes it happen!

The Templates Importer is a part of the Resources Importer app. It lets you import the following things:

- Page templates
- Web content templates and structures
- Application Display Templates (ADTs) for any portlet that supports ADTs, such as the Asset Publisher, Blogs, Categories Navigation, Documents and Media, Site Map, Tags Navigation, and Wiki portlets.
- DDL structures and templates, including display templates and form templates.

You can include the template importing capability in any Liferay plugin you develop; but a portlet plugin is the most common type of plugin used for importing templates. Let's build a portlet plugin that imports some templates and structures.

1. Download, install, and activate the Resources Importer app from Liferay Marketplace.
2. Create a portlet plugin project named `sample-templates-importer-portlet`.
3. Edit your `liferay-plugin-package.properties` file to include the following property settings:

```
name=

required-deployment-contexts=\
resources-importer-web

resources-importer-developer-mode-enabled=true

module-incremental-version=1
```

Here's a summary of what we're accomplishing with these settings: - We remove the plugin's name value to prevent the portal from displaying the plugin as an available app. - Since the Templates Importer feature resides in the Resources Importer web plugin, we include it as a required context. - By enabling developer mode, if the templates we're importing to the Global site already exist on it, the Templates Importer conveniently overwrites them. - Setting the module increment version to 1 is an appropriate version starting point for the plugin.

4. Edit the portlet's `portlet.xml` file and delete the value of its `display-name` element to keep the portal from displaying the portlet as an available app.
5. Create a folder named `templates-importer` in the plugin's `WEB-INF\src` folder. This folder will hold all of the templates and structures to import into the portal's Global site.

Let's stop here for a moment and consider how to specify templates and structures. The Templates Importer expects them to be specified in a directory structure under the plugin project's `templates-importer` folder. You must create folders to contain the template and structure files to apply to the portal.

Here's the directory structure to follow for specifying folders to contain your templates and structures:

- `templates-importer/`

- › journal/
 - structures / - contains structures (XML) and folders of child structures. Each folder name must match the file name of the corresponding parent structure. For example, to include a child structure of parent structure Parent 1.xml, create a folder named Parent 1/ at the same level as the Parent 1.xml file, for holding a child structures.
 - templates / - groups templates (FTL or VM) 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.xml.
- › templates/
 - application_display / - contains application display template (ADT) script files written in either the FreeMarker Template Language (.ftl) or Velocity (.vm). The extension of the files, .ftl for FreeMarker or .vm for Velocity must reflect the language that the templates are written in.
 - asset_category / - contains categories navigation templates
 - asset_entry / - contains asset publisher templates
 - asset_tag / - contains tags navigation templates
 - blogs_entry / - contains blogs templates
 - document_library / - contains documents and media templates
 - site_map / - contains site map templates
 - wiki_page / - contains wiki templates
 - dynamic_data_list / - contains dynamic data list templates and structures
 - display_template / - groups templates (FTL or VM) 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.xml.
 - form_template / - groups templates (FTL or VM) 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.xml.
 - structure / - contains structures (XML)
 - page / - contains page templates (JSON)

For templates and structures in your custom plugins, you only need to create folders to support the templates and/or structures you're adding.

Conveniently we've provided a ZIP file of the folders, templates, and structures for the sample-templates-importer-portlet plugin:

1. Download the file sample-templates-importer-contents.zip.
2. Extract its contents into the templates-importer folder of the sample-templates-importer-portlet plugin.
3. Deploy the sample-templates-importer-portlet plugin into a Liferay

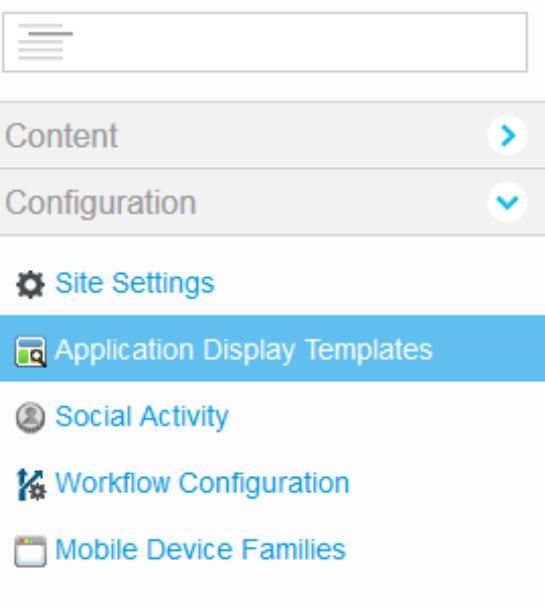
instance.

The console output should be similar to this:

```
INFO [localhost-startStop-8][PortletHotDeployListener:343] Registering
portlets for sample-templates-importer-portlet
INFO [localhost-startStop-8][PortletHotDeployListener:490] 1 portlet for
sample-templates-importer-portlet is available
for use
INFO [liferay/hot_deploy-1][ResourcesImporterHotDeployMessageListener:256]
Importing resources from sample-templates-
importer-portlet to group 10194 takes 1294 ms
...
```

4. View your resources from within Liferay. Log in to portal as an administrator and check the Global site to make sure that your resources were deployed correctly. Here's how you can use the Control Panel to view your templates and structures:
 1. Go to Sites in the Control Panel
 2. Select the *Global* site
 3. You can view the imported structures and templates here:
 - The Journal Article structures and templates can be viewed in the Web Content control panel portlet → *Manage* → *Structures* or *Manage* → *Templates*
 - The Dynamic Data List templates can be viewed in the Dynamic Data Lists control panel portlet → *Manage Data Definitions*. The templates can be viewed by going to the Actions menu of the Dynamic Data List structure and then clicking on *Manage Templates*.
 - The Application Display templates can be viewed under the *Configuration* category → *Application Display Templates*.
 - The page templates can be viewed in the *Page Templates* category from the Control Panel menu

The figure below shows some of the ADTs that were imported.



-  Content ▶
-  Configuration ▼
-  Site Settings
-  Application Display Templates
-  Social Activity
-  Workflow Configuration
-  Mobile Device Families

Application Display Templates

[+ Add ▾](#)[Delete](#)

20 Items per Page ▾

Page 1 of 1 ▾

Showing 14 results.

<input type="checkbox"/>	ID	Name	Description
<input type="checkbox"/>	13001	Asset Category - 6.2.0.1	
<input type="checkbox"/>	13003	Asset Entry 2 - 6.2.0.1	
<input type="checkbox"/>	13005	Asset Entry 1 - 6.2.0.1	
<input type="checkbox"/>	13007	test - 6.2.0.1	
<input type="checkbox"/>	13009	Asset Tag - 6.2.0.1	
<input type="checkbox"/>	13011	Blogs Entry -	

As you take a look around the folders and files within the plugin's `templates-importer` folder, notice the different kinds of templates and structures.

Page templates are specified in `.json` files in the `templates-`

importer/templates/page folder. Each one specifies the layout template, web content, assets, and portlet configurations to be imported with that page template.

Here's the contents of the page_3.json file:

```
{  
    "layoutTemplate": {  
        "columns": [  
            [  
                {  
                    "portletId": "58"  
                }  
            ],  
            [  
                {  
                    "portletId": "47"  
                },  
                {  
                    "portletId": "118",  
                    "portletPreferences": {  
                        "columns": [  
                            [  
                                {  
                                    "portletId": "3"  
                                }  
                            ],  
                            [  
                                {  
                                    "portletId": "16"  
                                }  
                            ]  
                        ],  
                        "layoutTemplateId": "2_columns_i"  
                    }  
                }  
            ]  
        ],  
        "friendlyURL": "/page-3",  
        "name": "Page 3",  
        "title": "Page 3"  
    },  
    "layoutTemplateId": "2_columns_ii"  
}
```

At the bottom of the JSON file, there are several important things specified for the page template. The layoutTemplateId references the layout template to use for the page. You can specify different layout templates to use for individual pages. You can find layout templates in your Liferay installation's /layouttpl folder. You can specify a name, title, and friendly URL for the page using the respective name, title, and friendlyURL fields. And, although it's not demonstrated in this page template, you can even set a page to be hidden.

Turning your attention to the columns of the JSON file, notice that you can declare portlets by specifying their portlet IDs. To lookup the IDs of Liferay's core portlets see the WEB-

`INF/portlet-custom.xml` file deployed in Liferay on your app server. If you're using the Web Content portlet, you can declare articles to be displayed on a page, by specifying HTML files. Interestingly, the `page_3.json` file demonstrates using the Nested Portlets portlet to display other portlets: the Search and Currency Converter portlets. Lastly, you can also specify portlet preferences for each portlet using the `portletPreferences` field.



Tip: You can also import resources, such as web content articles, using the Resources Importer. For example, it's very useful to import web content articles along with a page template that references the articles, in a nested Web Content Display portlet. For more information on importing resources, see [Importing Resources with Your Themes](#).

The figure below, shows a page created using the Page 3 template.

The screenshot shows a Liferay portal page titled "Page 3". At the top, there's a navigation bar with "Welcome" and "Page 3". Below the navigation bar, there are three portlets:

- Sign In**: A portlet where the user is signed in as "Test Test".
- Hello World**: A portlet displaying the message "Welcome to Liferay Portal Enterprise Edition 6.2.10 EE G".
- Nested Portlets**: A portlet containing a search bar with a placeholder "Search" and a dropdown menu set to "Everything".

Now that you've learned about the directory structure for your templates and the JSON file for the page templates, it's time to learn how to put template and structure files into your plugin. You can create templates and structures from scratch and/or leverage ones you've already created in Liferay. Let's go over how to leverage bringing in XML (structures) and FTL or VM template files from Liferay.

The sections below explain how to create structure and template files to put within the defined directory structure of the portlet's `templates-importer`/ folder.

Structure:

- **Dynamic Data Lists:** Edit the structure by clicking on *Manage Data Definitions*. Click on a structure that you want to export and then click on the *Source* tab. Copy and paste its contents into a new XML file for the structure in the `templates-importer/journal/dynamic_data_list/structure` folder. The structure XML sets a wireframe, or blueprint, for a dynamic data list's data.
- **Web Content:** Edit the structure by clicking on *Manage* and then *Structures*. Click on a structure that you want to export and then click on the *Source* tab. Copy and paste its contents into a new XML file for the structure in the `templates-importer/journal/structures/` folder. The structure XML sets a wireframe, or blueprint, for an article's data.

Template:

- **Application Display:** Edit the template by clicking on the template you want to export. Copy and paste its contents into a new FTL or VM file and place it in `templates-importer/templates/application_display/[your application display template type]/`.
- **Dynamic Data List:** Edit the template by clicking on *Manage Data Definition*. Click on *Manage Templates* from the Actions menu of the structure that your template is linked to. Choose the template that you want to export. Copy and paste its contents into a new FTL or VM file and place it in `templates-importer/templates/display_template/[structure name]/` or `templates-importer/templates/form_template/[structure name]/`
- **Page:** You will have to create the page template from scratch based on the `.json` file example for the page template above.

Importantly, you must name the files of all structures and templates, except page templates, after their source structures and templates. You can go back to any of the beginning steps in this section to make refinements to the sample plugin to try importing different structures and templates. The final `sample-templates-importer-portlet` project is available [here](#).

As you've seen for yourself, importing templates and structures with your plugin isn't difficult at all. The Resource Importer app's Templates Importer feature makes it easy. Have fun distributing your templates and structures!

3.12. Summary

You've covered a lot of ground learning Liferay Portlet development. You created a portlet project, studied its anatomy, and created the "My Greeting Portlet". You understood the Action phase and Render phase, and have passed information between them in a portlet. You've enhanced a portlet with multiple actions and have mapped a friendly URL to it. Lastly, you've found how easy it is to start localizing your portlets. You're really on a roll!

Now that you know how to create portlets, you'll need to consider a few things, such as persisting your objects to a database, maintaining separation between your persistence layer, business logic, and presentation layer, and allowing for flexible implementations. Lastly, you'll want the ability to publish your portlet's operations as services. So how do you address all of this? Hibernate probably comes to mind for persisting your data model, and Spring probably comes to mind with regards to supporting implementation flexibility. Sounds complicated, right? No need to worry! Liferay's Service Builder helps you build portlet services while hiding the complexities of using Spring and Hibernate under the hood. We'll cover Service Builder next.

4. Developing JSF Portlets with Liferay Faces



Liferay Faces

Do you want to develop MVC-based portlets using the Java EE standard? Do you want to use a portlet development framework with a UI component model that makes it easy to

develop sophisticated, rich UIs? Or have you been writing web apps using JSF that you'd like to use in Liferay Portal? If you answered yes to any of these questions, you're in luck! *Liferay Faces* provides all of these capabilities and more.

Liferay Faces is an umbrella project that provides support for the JavaServer™ Faces (JSF) standard within Liferay Portal. It encompasses the following projects:

- **Liferay Faces Bridge** enables you to deploy JSF web apps as portlets without writing portlet-specific Java code. It also contains innovative features that make it possible to leverage the power of JSF 2.x inside a portlet application. Liferay Faces Bridge implements the JSR 329 Portlet Bridge Standard.
- **Liferay Faces Alloy** enables you to use AlloyUI components in way that is consistent with JSF development.
- **Liferay Faces Portal** enables you to leverage Liferay-specific utilities and UI components in JSF portlets.

If you're new to JSF, you probably want to know its strengths, its weaknesses, and how it stacks up to developing portlets with CSS/JavaScript. We'll give you information on JSF and Liferay Faces to help you decide what framework is best for your needs.

Here some reasons why to use JSF and Liferay Faces:

- JSF is the Java EE standard for developing web applications that utilize the Model/View/Controller (MVC) design pattern. As a standard, the specification is actively

maintained by the 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 therefore a mature technology for developing web applications that are (arguably) simpler and easier 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 EE 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, ICEfaces, Primefaces, and RichFaces. Each of these component suites fortify JSF with a variety of UI components and complimentary technologies such as Ajax Push.
- JSF is a good choice for server-side developers that need to build web user interfaces. This enables server-side developers to focus on their core competencies rather than being experts in HTML/CSS/JavaScript.
- JSF provides the Facelets templating engine which makes it possible to create reusable UI components that are encapsulated as markup.
- JSF provides good integration with HTML5 markup
- JSF provides the Faces Flows feature which makes it easy for developers to create wizard-like applications that flow from view-to-view.
- JSF has good integration with dependency injection frameworks such as CDI and Spring that make it easy for developers to create beans that are placed within a scope managed by a container: @RequestScoped, @ViewScoped, @SessionScoped, @FlowScoped
- Since JSF is a stateful technology, the framework encapsulates the complexities of managing application state so that 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. So, sticking with JavaScript and leveraging AlloyUI 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. That's where this chapter is headed. You'll start with the basics: how to use the bridge to develop your own JSF portlet applications. From there, you'll learn how

the Liferay Faces Bridge works. After this, you'll examine using Liferay UI Components and Utilities in JSF applications. From this, you'll be ready to graduate on to leveraging AlloyUI components using Liferay Faces Alloy. Finally, you'll see how to migrate an existing project to Liferay Faces, and for those who want to contribute to the Liferay Faces project, you'll learn how to build Liferay Faces from source.

Your first task, however, is to learn how to develop portlets with Liferay Faces. This will introduce you to each of the Liferay Faces projects: Liferay Faces Bridge, Liferay Faces Alloy, and the Liferay Faces Portal Projects. We'll explain everything from choosing the correct Liferay Faces Version for your project, to updating your project from PortletFaces to Liferay Faces.

Let's start developing JSF portlets using Liferay Faces.

4.1. *Developing JSF Portlets*

Liferay supports developing and deploying JSF portlets on Liferay Portal with the help of Liferay Faces Bridge. Liferay Faces Bridge provides the means for deploying JSF portlets on Liferay Portal. In fact, the bridge supports deploying JSF web applications as portlets on any JSR 286 (Portlet 2.0) compliant portlet container, like Liferay Portal 5.2, 6.0, 6.1, and 6.2. Liferay Faces Bridge makes developing JSF portlets as similar as possible to JSF web app development. We'll take you through the portlet development process and show you how to leverage Liferay Faces Bridge's full potential with your JSF portlets.

In this section, you'll see how to develop JSF portlets with the standard features you expect, as well as additional features that'll help you build JSF applications that are powerful and easy to maintain.

Here are the topics we'll cover:

- Creating a JSF Portlet Project
- Specifying Your JSF Portlet's portlet.xml Descriptor
- Utilizing Portlet Preferences
- Accessing the Portlet API with ExternalContext
- Internationalizing JSF Portlets
- Utilizing IPC with JSF Portlets
- Leveraging CDI in JSF portlets
- Using Liferay Faces Bridge JSF Component Tags
- Dynamically Adding JSF Portlets to Liferay Portal
- Extending Liferay Faces Bridge Using Factory Wrappers

Let's get started with simple tutorial on creating and deploying a JSF portlet.

4.1.1. *Creating a JSF Portlet Project*

We want to make it easy for you to implement portlets using JSF. And Liferay IDE, with its powerful portlet plugin wizard, provides you with a great environment to do just that. The wizard lets you select a component suite that's right for your project, including JSF's standard UI component suite, ICEfaces, Liferay Faces Alloy, PrimeFaces, and RichFaces. Of course, you can use any development environment you like for building JSF portlets, but Liferay IDE is hard to

beat.

You'll create a JSF portlet project using Liferay IDE/Developer Studio, so you can see just how easy it is. If you don't have it installed yet, see Chapter 2 of this guide. If you do have it installed, launch it.

1. Go to *File → New → Liferay Plugin Project*.
2. In the project creation wizard's first window, you'll name your project and specify its development and runtime environments.

2.1 Fill in the *Project name* and *Display name* with *my-jsf-portlet* and *My JSF*, respectively.

2.2. Leave the *Use default location* checkbox checked. By default, the default location is set to your current Plugins SDK. If you'd like to change where your plugin project is saved in your file system, uncheck the box and specify your alternate location.

2.3. Select the *Ant (liferay-plugins-sdk)* option for your build type. If you'd like to use *Maven* for your build type, navigate to the Developing Plugins Using Maven section for details.

2.3. Your configured SDK and Liferay Runtime should already be selected. If you haven't yet pointed Liferay IDE to a Plugins SDK, click *Configure SDKs* to open the *Installed Plugin SDKs* management wizard. You can also access the *New Server Runtime Environment* wizard if you need to set up your runtime server; just click the *New Liferay Runtime* button next to the *Liferay Portal Runtime* dropdown menu.

2.4. Select *Portlet* as your Plugin type and click *Next*.

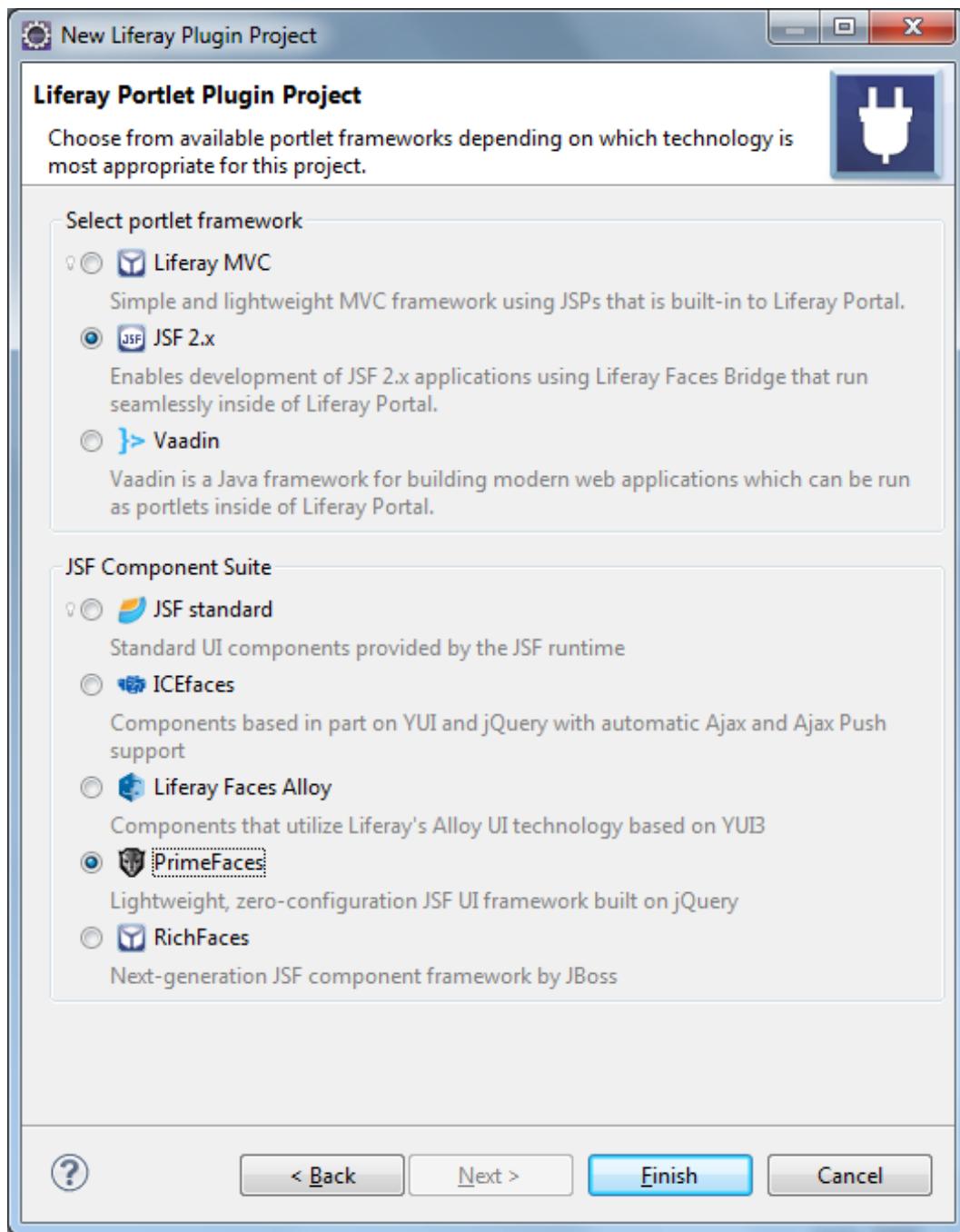
3. In this window, you'll select the portlet framework for your portlet and a UI component suite.

3.1 Select the *JSF 2.x* portlet framework.

Immediately, the wizard lists the available JSF component suites in the bottom section of the window. The list of component suites includes the JSF Standard suite, ICEfaces, Liferay Faces Alloy, PrimeFaces, and RichFaces.

3.2. Select the *PrimeFaces* UI component suite and click *Finish*.

Great! Your new JSF portlet plugin project is ready for you to develop JSF portlets.



Let's make the portlet display a calendar. We'll replace the portlet's default "hello world" text output with a PrimeFaces calendar component.

Open the view.xhtml file from the portlet project's docroot/views folder and replace the element <h:outputText value="#{i18n['my-jsf-hello-world']}> /> with the following

lines of code:

```
<h:form>
    <p:calendar></p:calendar>
</h:form>
```

Your view.xhtml facelet should look like this:

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

```

<f:view
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p="http://primefaces.org/ui"
    xmlns:ui="http://java.sun.com/jsf/facelets"
>
    <h:head />
    <h:body>
        <h:form>
            <p:calendar mode="inline" />
        </h:form>
    </h:body>
</f:view>

```

It's time to deploy your JSF portlet to the portal and see what it looks like.

4.1.2. Deploying JSF Portlets

Liferay provides a mechanism called auto-deploy that makes deploying portlets (and any other plugin types) a breeze. All you need to do is drop the plugin's .war file into the deploy directory, and the portal makes the necessary changes specific to Liferay and then deploys the plugin to the application server. This is a method of deployment used throughout this guide.



Note: Liferay supports a wide variety of application servers. Many, such as Tomcat and JBoss, provide a simple way to deploy web applications by just copying a file into a folder and Liferay's auto-deploy mechanism takes advantage of that ability. You should be aware though, that some application servers, such as WebSphere or Weblogic, require the use of specific tools to deploy web applications; Liferay's auto-deploy process won't work for them.

Deploying in Developer Studio: Drag your portlet project onto your server. When deploying your plugin, your server displays messages indicating that your plugin was read, registered and is now available for use.

```

Reading plugin package for my-jsf-portlet
Registering portlets for my-jsf-portlet
1 portlet for my-jsf-portlet is available for use

```

If at any time you need to redeploy your portlet while in Developer Studio, right-click your portlet located underneath your server and select *Redeploy*.

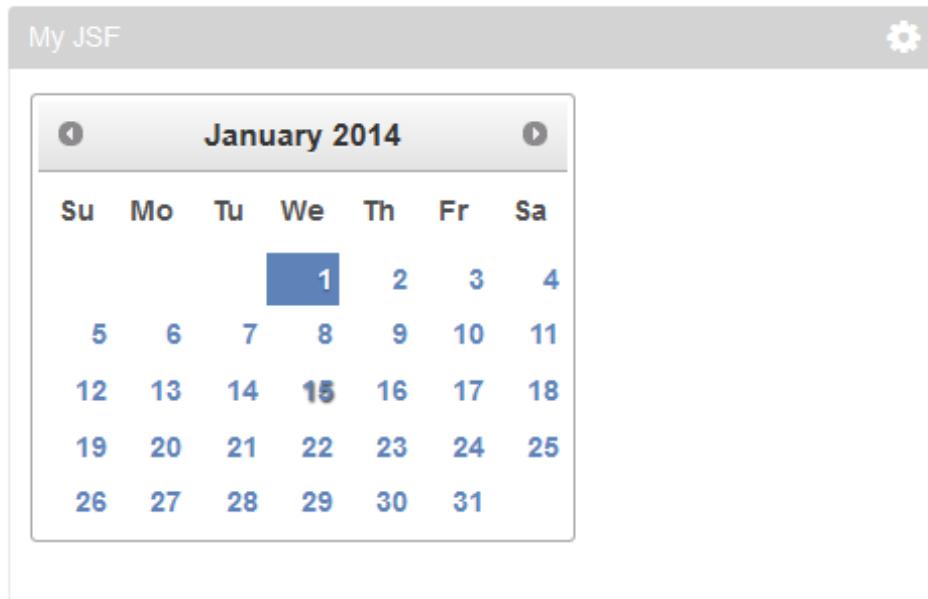
Deploying in the terminal: Open a terminal window in your `portlets/my-jsf-portlet` directory and enter

```
ant deploy
```

A `BUILD SUCCESSFUL` message indicates your portlet is now being deployed. If you switch to the terminal window running Liferay, within a few seconds you should see the message `1 portlet for my-jsf-portlet is available for use`. If not, double-check your configuration.

In your web browser, log in to the portal. Click the Add button, which appears as a *Plus* symbol in the top right hand section of your browser. Then click *Applications*, find the My JSF portlet in the *Sample* category, and click *Add*. Your portlet appears on the page, but Liferay Faces lets you know when a UI component requires a page refresh to render the first time.

Refresh the page and the portal renders your portlet's calendar component.



It's just that easy to create and deploy JSF portlet plugins!

Next, let's get familiar with the portlet deployment descriptor file (*portlet.xml*) and consider the descriptor requirements for JSF portlets.

4.1.3. Specifying the *portlet.xml* for Your JSF Portlet

Each portlet project must have a *WEB-INF/portlet.xml* deployment descriptor file. As we demonstrated in the previous section, Liferay IDE and the Plugins SDK create this file for you. But there are a couple unique requirements for JSF portlets with respect to their deployment descriptors.

First, using JSF 2.x in a portlet requires specifying

javax.portlet.faces.GenericFacesPortlet as the portlet's class. You specify this class name in the portlet's *<portlet-class>* entity. Notice that the portlet in the following *portlet.xml* snippet meets this requirement:

```
<portlet-app>
  <portlet>
    <portlet-name>jsf_portlet</portlet-name>
    <display-name>JSF Portlet</display-name>
    <portlet-class>
      javax.portlet.faces.GenericFacesPortlet
    </portlet-class>
    <init-param>
      <name>javax.portlet.faces.defaultViewId.view</name>
      <value>/view.xhtml</value>
    </init-param>
    <init-param>
```

```

        <name>javax.portlet.faces.defaultViewId.edit</name>
        <value>/preferences.xhtml</value>
    </init-param>
    <init-param>
        <name>javax.portlet.faces.defaultViewId.help</name>
        <value>/help.xhtml</value>
    </init-param>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>view</portlet-mode>
        <portlet-mode>edit</portlet-mode>
        <portlet-mode>help</portlet-mode>
    </supports>
    ...
</portlet>
</portlet-app>
```

Second, each portlet must map a facelet to each portlet mode that it supports. The `portlet.xml` file content above supports the VIEW, EDIT, and HELP portlet modes, and maps each of those modes to a specific facelet.

In the example code above, VIEW mode support is specified by the `<portlet-mode>view</portlet-mode>` element. The VIEW mode is mapped to the `/view.xhtml` facelet by the `<init-param>` element:

```

<init-param>
    <name>javax.portlet.faces.defaultViewId.view</name>
    <value>/view.xhtml</value>
</init-param>
```

Now that we've got `WEB-INF/portlet.xml` set up, let's move on to portlet preferences.

4.1.4. Using Portlet Preferences with JSF

JSF portlet developers often must enable the end-user to personalize portlets in some way. To meet this requirement, the Portlet 2.0 specification lets you define portlet preferences for each portlet. Preference names and default values are defined in the `WEB-INF/portlet.xml` descriptor. Portal end-users start out interacting with the portlet user interface in portlet VIEW mode but switch to portlet EDIT mode in order to select custom preference values.

```

<portlet-preferences>
    <preference>
        <name>datePattern</name>
        <value>MM/dd/yyyy</value>
    </preference>
</portlet-preferences>
```

Additionally, Portlet 2.0 lets you specify support for EDIT mode in the `WEB-INF/portlet.xml` descriptor.

```

<supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>view</portlet-mode>
    <portlet-mode>edit</portlet-mode>
```

```
</supports>
```

Although support for portlet EDIT mode has been specified, the portlet container does not necessarily know which JSF view should be rendered when the user enters portlet EDIT mode. JSF portlet developers must specify the Facelet view in the WEB-INF/portlet.xml descriptor that should be displayed for each supported portlet mode.

```
<init-param>
    <name>javax.portlet.faces.defaultViewId.edit</name>
    <value>/edit.xhtml</value>
</init-param>
```

Now that we've considered how to implement portal preferences, let's learn how to access the Portlet API.

4.1.5. Accessing the Portlet API with ExternalContext

Just as JSF *web app* developers rely on ExternalContext to access to the Servlet API, JSF *portlet* developers rely on it to access to the Portlet API.

As you develop JSF portlets, you'll often need to access instances of the javax.portlet.PortletRequest and javax.portlet.PortletResponse classes. You access these instances similarly to how you'd access the javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse classes in a servlet environment, except that you cast them to the portlet versions of the classes.

In the example code snippet below, the request object from externalContext.getRequest() is cast to the PortletRequest class and the response object from externalContext.getResponse() is cast to the PortletResponse class:

```
import javax.portlet.PortletRequest;
import javax.portlet.PortletResponse;
....
```



```
public class PortletBackingBean {

    public void submit() {
        FacesContext facesContext =
            FacesContext.getCurrentInstance();

        ExternalContext externalContext =
            facesContext.getExternalContext();

        PortletRequest portletRequest =
            (PortletRequest) externalContext.getRequest();

        PortletResponse portletResponse =
            (PortletResponse) externalContext.getResponse();
    }
}
```

The code listing above uses the singleton class LiferayFacesContext, which has methods

`getPortletRequest()` and `getPortletResponse()`. You can leverage the `LiferayFacesContext` class in your JSF portlets on Liferay to get easy access to the portlet requests and responses. This class comes with Liferay Faces Portal, which we'll cover in detail in later sections.

In the next section, we'll explain how to internationalize your JSF portlets.

4.1.6. Internationalizing JSF Portlets

There are at least two ways to handle internationalization with JSF portlets in Liferay Portal:

1. Using the standard JSF mechanism to create your own `i18n` keyword, as shown in the jsf2-portlet demo.
 - › Create a properties file in the classpath like `i18n.properties`
 - › Create a `<resource-bundle>` entry in the `<faces-config>` element, as demonstrated in `faces-config.xml`
 - › Use your custom `i18n` keyword Expression Language (EL) in your Facelet view, like `applicant.xhtml`
2. Using the built-in `i18n` keyword provided by Liferay Faces Portal, as shown in the jsf2-registration-portlet demo. This method integrates JSF and Liferay very well, because it allows you to "hook" into thousands of existing internationalized keys that Liferay Portal includes and allows you to add your own keys.
 - › Create a hook, like `liferay-hook.xml`, inside your portlet plugin
 - › Create internationalized Langauge properties files, like `Language_en_US.properties`
 - › Use the built-in `i18n` keyword Expression Language (EL) in your Facelet view, like `registrant.xhtml`

Internationalizing your portlets is especially easy to do using the options that Liferay Faces provides.

Next, we'll learn how to communicate between JSF portlets using IPC.

4.1.7. Using IPC with JSF Portlets

Liferay Faces Bridge supports Portlet 2.0 Inter Portlet Communication (IPC), using the JSR 329 approach for supporting Portlet 2.0 Events and Portlet 2.0 Public Render Parameters.



Note: Visit <http://www.liferay.com/community/liferay-projects/liferay-faces/demos> to see portlets that demonstrate the IPC techniques described in this section. At that location, you'll also find portlets that implement Ajax Push for IPC, using ICEfaces+ICEPush and PrimeFaces+PrimePush.

4.1.7.1. Using Portlet 2.0 Public Render Parameters

The Public Render Parameters technique provides a way for portlets to share data by setting public/shared parameter names in a URL controlled by the portal. While the benefit of this

approach is that it is relatively easy to implement, the drawback is that only small amounts of data can be shared. Typically the kind of data that is shared is simply the value of a database primary key. As required by the Portlet 2.0 standard, Public Render Parameters must be declared in the WEB-INF/portlet.xml descriptor.

This example excerpt from a WEB-INF/portlet.xml descriptor demonstrates setting a public render parameter for a customer ID, shared between a Customers portlet and a Bookings portlet:

```
<portlet>
    <portlet-name>customersPortlet</portlet-name>
    <supported-public-render-parameter>selectedCustomerId</supported-public-
render-parameter>
</portlet>
<portlet>
    <portlet-name>bookingsPortlet</portlet-name>
    <supported-public-render-parameter>selectedCustomerId</supported-public-
render-parameter>
</portlet>
<public-render-parameter>
    <identifier>selectedCustomerId</identifier>
    <qname xmlns:x="http://liferay.com/pub-render-
params">x:selectedCustomerId</qname>
</public-render-parameter>
```

Fortunately, the JSR 329 standard defines a mechanism for you to use Portlet 2.0 Public Render Parameters for IPC in a way that is more natural to JSF development. Section 5.3.2 of this standard requires the bridge to inject the public render parameters into the Model concern of the MVC design pattern (as in JSF model managed-beans) after the RESTORE_VIEW phase completes. This is accomplished by evaluating the EL expressions found in the <model-el>...</model-el> section of the WEB-INF/faces-config.xml descriptor. The WEB-INF/faces-config.xml descriptor excerpt below demonstrates using this mechanism in the example Customers portlet and Bookings portlet:

```
<faces-config>
    <application>
        <application-extension>
            <bridge:public-parameter-mappings>
                <bridge:public-parameter-mapping>

<parameter>customersPortlet:selectedCustomerId</parameter>
                <model-
el>#{customersModelBean.selectedCustomerId}</model-el>
                </bridge:public-parameter-mapping>
                <bridge:public-parameter-mapping>
                    <parameter>bookingsPortlet:selectedCustomerId</parameter>
                    <model-el>#{bookingsModelBean.selectedCustomerId}</model-
el>
                    </bridge:public-parameter-mapping>
                </bridge:public-parameter-mappings>
            </application-extension>
        </application>
    </faces-config>
```

Section 5.3.2 of the JSR 329 standard also requires that if a `bridgePublicRenderParameterHandler` has been registered in the `WEB-INF/portlet.xml` descriptor, then the handler must be invoked so that it can perform any processing that might be necessary. Optionally, you can implement and register a `bridgePublicRenderParameterHandler` for processing public render parameters.

For example, a `BridgePublicRenderParameterHandler` for processing public render params for the Bookings portlet's currently selected Customer could be stubbed out like the following class code:

```
package com.liferay.faces.example.handler;

import javax.faces.context.FacesContext;

import com.liferay.faces.bridge.BridgePublicRenderParameterHandler;

public class CustomerSelectedHandler
implements BridgePublicRenderParameterHandler {

    public void processUpdates(FacesContext facesContext) {
        // Here is where you would perform any necessary processing of public
        // render parameters
    }

}
```

For the `BridgePublicRenderParameterHandler` to be invoked, it must be registered in an `<init-param>` element within the portlet's `<portlet>` element in the `WEB-INF/portlet.xml` descriptor:

```
<init-param>
    <name>javax.portlet.faces.bridgePublicRenderParameterHandler</name>
    <value>com.liferay.faces.example.handler.CustomerSelectedHandler</value>
</init-param>
```



Note: For a complete example demonstrating public render parameters and a `bridgePublicRenderParameterHandler`, see the JSF2 IPC Public Render Parameters Portlet demo on GitHub.

Now that we've discussed Public Render Parameters for JSF in IPC, let's look at Events in IPC.

4.1.7.2. Handling Portlet 2.0 Events

In Portlet 2.0, you can leverage a server-side events technique that uses an event-listener design to share data between portlets. When using this form of IPC, the portlet container acts as broker and distributes events and payload (data) to portlets. One requirement of this approach is that the payload must implement the `java.io.Serializable` interface since it might be sent to a portlet in another WAR running in a different classloader. In addition, the Portlet 2.0 standard requires the events to be declared in the `WEB-INF/portlet.xml` descriptors of the involved

portlets.

The following example WEB-INF/portlet.xml descriptor snippet defines an IPC event for when a Customer is edited in the example Bookings portlet. The bookingsPortlet portlet is registered as the event's publisher (or sender). The customersPortlet portlet, on the other hand, is registered as a processor (or listener) for that event type. Consequently, when a Customer is edited in the bookingsPortlet portlet, that portlet publishes the event and the customersPortlet portlet is notified for processing the event.

Here's a snippet from the example's WEB-INF/portlet.xml descriptor:

```
<portlet>
    <portlet-name>customersPortlet</portlet-name>
    <supported-processing-event>
        <qname
xmlns:x="http://liferay.com/events">x:ipc.customerEdited</qname>
        </supported-processing-event>
    </portlet>
    <portlet>
        <portlet-name>bookingsPortlet</portlet-name>
        <supported-publishing-event>
            <qname
xmlns:x="http://liferay.com/events">x:ipc.customerEdited</qname>
            </supported-publishing-event>
        </portlet>
        ....
        <event-definition>
            <qname xmlns:x="http://liferay.com/events">x:ipc.customerEdited</qname>
            <value-type>com.liferay.faces.example.dto.Customer</value-type>
        </event-definition>
```

Optionally, you can implement a BridgeEventHandler for an event type and register the handler in the WEB-INF/portlet.xml descriptor. If a BridgeEventHandler has been registered in the WEB-INF/portlet.xml descriptor, Section 5.2.5 of the JSR 329 standard requires that the handler must be invoked so that it can perform any event processing that might be necessary.

When the customer's details (such as first name/last name) are edited in the Bookings portlet, the event named ipc.customerEdited is sent back to the Customers portlet and is processed by the following CustomerEditedEventHandler class:

```
...
import javax.faces.context.FacesContext;
import javax.portlet.Event;
import javax.portlet.faces.BridgeEventHandler;
import javax.portlet.faces.event.EventNavigationResult;
...
public class CustomerEditedEventHandler implements BridgeEventHandler {
```

```

    ...
    public EventNavigationResult handleEvent(FacesContext facesContext,
Event event) {
    EventNavigationResult eventNavigationResult = null;
    String eventQName = event.getQName().toString();

    if
(eventQName.equals("{http://liferay.com/events}ipc.customerEdited")) {
        ...
    }

    return eventNavigationResult;
}

...
}

```

And here's the descriptor for registering the `CustomerEditedEventHandler` class as a bridge event handler for the Customers portlet. The following `<init-param>` belongs in the Customers portlet's `<portlet>` element, in the `WEB-INF/portlet.xml` descriptor.

```

<init-param>
    <name>javax.portlet.faces.bridgeEventHandler</name>
    <value>com.liferay.faces.example.event.CustomerEditedEventHandler</value>
</init-param>

```



Note: For a complete example demonstrating JSF 2 IPC events, see the [JSF2 IPC Events - Customers and JSF2 IPC Events - Bookings demo portlets](#) on GitHub.

Now that we've discussed some common basic JSF portlet development topics, let's consider how to use dependency injection in JSF portlets.

4.1.8. Developing JSF Portlets with CDI

In December 2009, JSR 299 introduced the Contexts and Dependency Injection (CDI) 1.0 standard into the Java EE 6 platform. In April 2013, JSR 346 updated CDI to version 1.1 for Java EE 7. In addition, JSR 344, the JSF 2.2 specification which is another component of Java EE 7, introduced a dependency on the CDI API for the `javax.faces.view.ViewScoped` annotation and for the Faces Flows feature. JBoss Weld is the Reference Implementation (RI) for CDI and Apache OpenWebBeans is another open source implementation.

In this section we'll cover the following topics:

- Configuring CDI on Liferay Portal
- Configuring the Liferay CDI Portlet Bridge
- Understanding CDI in JSF Annotations

Let's look at configuring Weld on Liferay Portal for leveraging CDI with JSF portlets.

4.1.8.1. Configuring CDI on Liferay Portal

You must use one of the following portal/app-server combinations to use Weld with Liferay Portal:

- Liferay Portal 6.1/6.2 (Tomcat)
- Liferay Portal 6.1/6.2 (GlassFish)
 - Apply patch attached to LPS-35558.
 - Upgrade Mojarra in GlassFish to version 2.1.21 (or higher).
 - Upgrade Weld in GlassFish to version 1.1.10.Final (or higher).
- Liferay Portal 6.1/6.2 (JBoss AS)
 - Apply patch attached to LPS-35558
 - Upgrade Mojarra in JBoss AS to version 2.1.21 (or higher).
 - Upgrade Weld in JBoss AS to version 1.1.10.Final (or higher).
- Liferay Portal 6.1/6.2 (Resin)

When developing portlets with CDI 1.0, you must include a `WEB-INF/beans.xml` descriptor in your JSF portlet plugin's WAR deployment, so that the CDI implementation can detect the CDI-related annotations of your classes when it scans the classpath.

Here's an example `WEB-INF/beans.xml` descriptor:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/
                           http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

For JBoss AS 7, you must also include a `WEB-INF/jboss-deployment-structure.xml` descriptor in your portlet plugin's WAR deployment to include the CDI-related modules. Here's an example of a `WEB-INF/jboss-deployment-structure.xml` descriptor for JBoss:

```
<?xml version="1.0"?>
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
    <deployment>
        <exclusions>
            <module name="javaee.api" />
            <module name="org.apache.log4j" />
        </exclusions>
        <dependencies>
            <module name="com.liferay.portal" />
            <module name="javax.annotation.api" />
            <module name="javax.enterprise.api" />
            <module name="javax.inject.api" />
            <module name="javax.interceptor.api" />
            <module name="javax.validation.api" />
            <module name="javax.mail.api" />
            <module name="org.jboss.modules" />
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

Next, we'll cover Weld configuration on the app server. There are some different configuration steps for different app servers. We'll look at the most common configuration steps first.

For most app servers (excluding Resin), the portlet's WEB-INF/web.xml descriptor must include the following filter and filter mapping:

```
<filter>
    <filter-name>WeldCrossContextFilter</filter-name>
    <filter-class>org.jboss.weld.servlet.WeldCrossContextFilter</filter-
class>
</filter>
<filter-mapping>
    <filter-name>WeldCrossContextFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

If you are using Resin as your app server, you don't need JBoss Weld, as Resin includes the CanDI implementation of CDI by default.

The next section contains information about specifically configuring Tomcat, so developers running other application servers can skip it.

Additional Weld Configuration for Tomcat

If Weld is running in a Java EE application server like Oracle GlassFish or JBoss AS, then Weld is automatically included in the global classpath. But on Tomcat, it is necessary to include the weld-servlet.jar dependency in either the tomcat/lib global classpath, or directly in the WEB-INF/lib folder of a portlet:

```
<!-- Required only for Tomcat -->
<dependency>
    <groupId>org.jboss.weld.servlet</groupId>
    <artifactId>weld-servlet</artifactId>
    <version>1.1.10.Final</version>
</dependency>
```

Additionally, the following listener must be added to the WEB-INF/web.xml descriptor:

```
<!-- Required only for Tomcat -->
<listener>
    <listener-class>org.jboss.weld.environment.servlet.Listener</listener-
class>
</listener>
```

Next we'll discuss configuring the Liferay CDI Portlet Bridge.

4.1.8.2. Configuring the Liferay CDI Portlet Bridge

The Liferay CDI Portlet Bridge makes it possible to use CDI with your JSF portlets on Liferay. Your JSF portlet projects must include the Liferay CDI Portlet Bridge as a dependency.

For example, to specify the bridge dependency in a Maven project for Liferay 6.2, you'd add the following <dependency> to your POM's <dependencies> element:

```
<dependency>
    <groupId>com.liferay.cdi</groupId>
    <artifactId>cdi-portlet-bridge-shared</artifactId>
    <version>6.2.0.2</version>
```

```
</dependency>
The WEB-INF/portlet.xml descriptor of the portlet must include the following markup:
<filter>
    <filter-name>CDIPortletFilter</filter-name>
    <filter-class>com.liferay.cdi.portlet.bridge.CDIPortletFilter</filter-
class>
    <lifecycle>ACTION_PHASE</lifecycle>
    <lifecycle>EVENT_PHASE</lifecycle>
    <lifecycle>RENDER_PHASE</lifecycle>
    <lifecycle>RESOURCE_PHASE</lifecycle>
</filter>
<filter-mapping>
    <filter-name>CDIPortletFilter</filter-name>
    <portlet-name>my-portlet-name</portlet-name>
</filter-mapping>
```

Additionally, the portlet's WEB-INF/web.xml descriptor must include the following markup:

```
<filter>
    <filter-name>CDICrossContextFilter</filter-name>
    <filter-
class>com.liferay.cdi.portlet.bridge.CDICrossContextFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>CDICrossContextFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
<listener>
    <listener-
class>com.liferay.cdi.portlet.bridge.CDIContextListener</listener-class>
</listener>
```



Tip: The Liferay Faces Project features the jsf2-cdi-portlet demo (which is a variant of the jsf2-portlet demo). It's a good idea download and deploy the jsf2-cdi-portlet demo in your development environment in order to verify that CDI functions properly.

Now that everything is configured, you are ready to begin development with CDI.

4.1.8.3. Understanding CDI in JSF Annotations

When developing portlets with CDI, it is possible to annotate Java classes as CDI managed beans with @Named with the following scopes:

CDI Annotation	Description
@ApplicationScoped	An @ApplicationScoped managed bean exists for the entire lifetime of the portlet application.
@ConversationScoped	A @ConversationScoped managed bean is created when Conversation.begin() is called and is scheduled for

CDI Annotation	Description
@FlowScoped	garbage collection when <code>Conversation.end()</code> is called. A @FlowScoped managed bean is created when a JSF 2.2 Flow begins and scheduled for garbage collection when a JSF 2.2 Flow completes.
@RequestScoped	A @RequestScoped managed bean exists during an <code>ActionRequest</code> , <code>RenderRequest</code> , or <code>ResourceRequest</code> . Beans that are created during the <code>ActionRequest</code> do not survive into the <code>RenderRequest</code> .
@SessionScoped	A @SessionScoped managed bean exists for the duration of the user session.

In addition to CDI scope annotations, it's important to understand JSF 2 annotations and their equivalence to CDI annotations.

JSF Annotation	Equivalent CDI Annotation
<code>javax.faces.ManagedBean</code>	<code>javax.inject.Named</code>
<code>javax.faces.ApplicationScoped</code>	<code>javax.enterprise.context.ApplicationScoped</code>
<code>javax.faces.RequestScoped</code>	No such equivalent, since <code>javax.enterprise.context.RequestScope</code> does not span portlet lifecycle phases.
<code>javax.faces.SessionScoped</code>	<code>javax.enterprise.context.SessionScoped</code>
<code>javax.faces.ManagedProperty</code> (corresponding setter method required)	<code>javax.inject.Inject</code> (corresponding setter method not required)

Now that we have discussed JSF portlet development with CDI, let's take a look at some UI component tags included with Liferay Faces Bridge.

4.1.9. Using Liferay Faces Bridge JSF Component Tags

Although the JSR 329 standard does not define any JSF components that bridge implementations are required to provide, Liferay Faces Bridge comes with a handful of components that are helpful to use in JSF portlets.

Because Liferay Faces has several active versions (targeting different versions of JSF, Liferay Portal, etc.), there are several versions of the project View Declaration Language (VDL) documentation for these tags. The VDL documentation can be found at the following addresses:

- The VDL documentation for Liferay Faces 2.1 can be found at <http://docs.liferay.com/faces/2.1/vdldoc/>.
- The VDL documentation for Liferay Faces 3.0-legacy can be found at <http://docs.liferay.com/faces/3.0-legacy/vdldoc/>.
- The VDL documentation for Liferay Faces 3.0 can be found at <http://docs.liferay.com/faces/3.0/vdldoc/>.
- The VDL documentation for Liferay Faces 3.1 can be found at

- http://docs.liferay.com/faces/3.1/vdldoc/.
- The VDL documentation for Liferay Faces 3.2 can be found at http://docs.liferay.com/faces/3.2/vdldoc/.

Liferay Faces Bridge provides the following UI component tags under the bridge and portlet namespaces for the Bridge and Portlet 2.0 tags respectively. Let's look at the Bridge tags first.

4.1.9.1. Bridge UIComponent Tags

Liferay Faces Bridge provides the following bridge-specific UIComponent tags as part of its component suite.

The bridge:inputFile tag renders an HTML <input type="file"/> tag, providing file upload capability.

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
         xmlns:bridge="http://liferay.com/faces/bridge"
         xmlns:f="http://java.sun.com/jsf/core"
         xmlns:h="http://java.sun.com/jsf/html"
         xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:head />
    <h:body>
        <h:form>
            <bridge:inputFile
                fileUploadListener="#{backingBean.handleFileUpload}" multiple="multiple" />
            <h:commandButton value="Submit" />
        </h:form>
    </h:body>
</f:view>
```

Here's a code snippet from a class that imports the FileUploadEvent class and implements handling the file upload:

```
import com.liferay.faces.bridge.event.FileUploadEvent;
...
@ManagedBean(name = "backingBean")
@ViewScoped
public class ApplicantBackingBean implements Serializable {

    public void handleFileUpload(FileUploadEvent fileUploadEvent)
        throws Exception {
        UploadedFile uploadedFile = fileUploadEvent.getUploadedFile();
        System.out.println("Uploaded file:" + uploadedFile.getName());
    }
}
```



Note: Usage of this tag requires the Apache commons-fileupload and commons-io dependencies. See the Demo JSF2 Portlet for more details.

Next, let's learn about the Portlet UIComponent tags available in Liferay Faces Bridge.

4.1.9.2. Portlet 2.0 UIComponent Tags

Liferay Faces Bridge provides the following Portlet 2.0 UIComponent tags as part of its component suite.



Note: Although JSP tags are provided by the portlet container implementation, Liferay Faces Bridge provides these tags in order to support their usage within Facelets.

The first tag we'll look at is `portlet:actionURL`.

If the `var` attribute is present, the `portlet:actionURL` tag introduces an EL variable that contains a `javax.portlet.ActionURL`, adequate for postbacks. Otherwise, the URL is written to the response.

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
         xmlns:f="http://java.sun.com/jsf/core"
         xmlns:h="http://java.sun.com/jsf/html"
         xmlns:portlet="http://java.sun.com/portlet_2_0"
         xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:head />
    <h:body>
        <h:form>
            <portlet:actionURL var="myActionURL" >
                <portlet:param name="foo" value="1234" />
            </portlet:actionURL>
            <h:outputText var="actionURL=#{myActionURL}" />
        </h:form>
    </h:body>
</f:view>
```

Next, we'll look at an example of the `portlet:namespace` tag.

If the `var` attribute is present, the `portlet:namespace` tag introduces an EL variable that contains the portlet namespace. Otherwise, the namespace is written to the response.

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
         xmlns:f="http://java.sun.com/jsf/core"
         xmlns:h="http://java.sun.com/jsf/html"
         xmlns:portlet="http://java.sun.com/portlet_2_0"
         xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:head />
    <h:body>
        <h:form>
            <portlet:namespace var="mynamespace" />
            <h:outputText var="namespace=#{mynamespace}" />
        </h:form>
    </h:body>
</f:view>
```

```
</h:body>
</f:view>
```

The `portlet:param` tag is up next.

The `portlet:param` tag lets you add a request parameter name=value pair when nested inside `portlet:actionURL`, `portlet:renderURL`, or `portlet:resourceURL` tags.

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:portlet="http://java.sun.com/portlet_2_0"
    xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:head />
    <h:body>
        <h:form>
            <portlet:actionURL>
                <portlet:param name="foo" value="1234" />
            </portlet:actionURL>
        </h:form>
    </h:body>
</f:view>
```

The next tag we'll look at is the `portlet:renderURL` tag.

If the `var` attribute is present, the `portlet:renderURL` tag introduces an EL variable that contains a `javax.portlet.PortletURL`, adequate for rendering. Otherwise, the URL is written to the response.

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:portlet="http://java.sun.com/portlet_2_0"
    xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:head />
    <h:body>
        <h:form>
            <portlet:renderURL var="myRenderURL">
                <portlet:param name="foo" value="1234" />
            </portlet:renderURL>
            <h:outputText var="actionURL=#{myRenderURL}" />
        </h:form>
    </h:body>
</f:view>
```

Finally, we'll look at the `portlet:resourceURL` tag.

If the `var` attribute is present, the `portlet:resourceURL` tag introduces an EL variable that contains a `javax.portlet.ActionURL`, adequate for obtaining resources. Otherwise, the URL is written to the response.

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
```

```

xmlns:portlet="http://java.sun.com/portlet_2_0"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head />
<h:body>
    <h:form>
        <portlet:resourceURL var="myResourceURL">
            <portlet:param name="foo" value="1234" />
        </portlet:resourceURL>
        <h:outputText var="actionURL=#{myResourceURL}" />
    </h:form>
</h:body>
</f:view>

```

Now that we've introduced you to some of Liferay Faces Bridge's UIComponent tags, let's explore how to dynamically add JSF portlets to portal pages.

4.1.10. Dynamically Adding JSF Portlets to Liferay Portal (Runtime Portlets)

Liferay Portal lets you add portlets dynamically to portal pages using several approaches:

- Inside the FreeMarker template or Velocity template of a theme with `$theme.runtime()`
- Inside a layout template with `$processor.processPortlet()`
- Inside a JSP with `<liferay-portlet:runtime />`

Unfortunately, as described in FACES-244, dynamically adding JSF portlets doesn't work very well. It's actually not limited to JSF portlets-- this problem happens with any portlet that needs to add JS/CSS resources to the `<head>...</head>` section of the portal page. Since JSF portlets require the `jsf.js` resource to perform Ajax requests, the `jsf.js` resource must be loaded when the portal page is initially rendered.

There are two workarounds:

1. For plain JSF portlets, add a `<link />` element for the `jsf.js` resource in the `<head>...</head>` section of the `portal_normal.vm` or `portal_normal.ftl` file in the theme. The first few lines of `jsf.js` prevent double-instantiation in case it gets included multiple times on a page. This can occur when a JSF portlet is dynamically included and another JSF portlet is added statically. Unfortunately this approach doesn't work for PrimeFaces, since `primefaces.js` does not prevent double-instantiation.

2. Use an IFrame:

```

<div id="${request.portlet-namespace}my_runtime_portlet">
    <script type="text/javascript">
        AUI().use('liferay-portlet-url', 'aui-resize-iframe', function(A) {
            var portletURL = Liferay.PortletURL.createRenderURL();
            portletURL.setPortletId('1_WAR_mypluginportlet');
            portletURL.setPlid(themeDisplay.getPlid());
            var html = '<iframe frameborder="0" id="${request.portlet-
namespace}my_runtime_portlet_frame" src="' + portletURL.toString() + '"'
            scrolling="no" width="100%"></iframe>';
            A.one('#${request.portlet-
namespace}my_runtime_portlet').append(html);
    </script>

```

```

        A.one('#${request.portlet-
namespace}my_runtime_portlet_frame').plug(A.Plugin.ResizeIframe);
    });
</script>
</div>

```

In order to avoid the *You do not have the roles required to access this portlet* error message, add the following directive to the WEB-INF/liferay-portlet.xml descriptor:

```
<add-default-resource>true</add-default-resource>
```

Alternatively, you can place the portlet alone on a hidden portal page and then use a portlet URL referring to the plid of the hidden portal page. This approach is more appropriate for portlets that perform security-sensitive actions.

Note, when an end-user dynamically adds any JSF 2 portlet to a portal page, the JSF 2 standard jsf.js JavaScript code is not automatically executed. In order for the jsf.js to be executed, the page must be fully refreshed.

As a workaround, Liferay Portal provides configuration parameters that allow the developer to specify that a full page refresh is required. Specifying this ensures that JSF 2 is properly initialized. You specify the required <render-weight> and <ajaxable> parameter elements in the WEB-INF/liferay-portlet.xml configuration file.

```

<liferay-portlet-app>
    <portlet>
        <portlet-name>my_portlet</portlet-name>
        <instanceable>false</instanceable>
        <render-weight>1</render-weight>
        <ajaxable>false</ajaxable>
    </portlet>
</liferay-portlet-app>

```

Now, you know the options you have in dynamically adding your JSF portlets at runtime.

Next, we'll discuss extension of Liferay Faces Bridge with Factory Wrappers.

4.1.11. Extending Liferay Faces Bridge Using Factory Wrappers

Liferay Faces Bridge has several abstract classes that serve as contracts for defining factories:

- BridgeContextFactory.java
- BridgePhaseFactory.java
- BridgeRequestScopeFactory.java
- BridgeRequestScopeFactory.java
- BridgeRequestScopeCacheFactory.java
- BridgeRequestScopeCacheFactory.java
- BridgeRequestScopeManagerFactory.java
- BridgeWriteBehindSupportFactory.java
- BridgeURLFactory.java
- IncongruityContextFactory.java
- PortletContainerFactory.java
- PortletContainerFactory.java

- UploadedFileFactory.java

These factories are defined using the standard JSF <factory-extension> element in faces-config.xml. The *default implementations* of these factories are defined in the bridge's META-INF/faces-config.xml file.

The bridge features an *extension mechanism* that enables you to decorate any of these factories using a META-INF/faces-config.xml descriptor (inside a JAR), or a WEB-INF/faces-config.xml descriptor (inside a portlet WAR). This mechanism enables you to plug in your own factory implementations to decorate (wrap) the default implementations, using a FactoryWrapper.

4.1.11.1. Wrapping the BridgeContextFactory with a Custom BridgeContext

This *tutorial* for Liferay Faces Bridge shows you how to wrap the BridgeContextFactory class, so that it returns a custom BridgeContext instance by overriding one of the methods to provide custom functionality.

1. Create a wrapper class for the BridgeContext that overrides the getResponseNamespace() method:

```
package com.mycompany.myproject;

public class BridgeContextCustomImpl extends BridgeContextWrapper {

    private BridgeContext wrappedBridgeContext;

    public BridgeContextCustomImpl(BridgeContext bridgeContext) {

        this.wrappedBridgeContext = bridgeContext;
        BridgeContext.setCurrentInstance(this);
    }

    @Override
    public String getResponseNamespace() {
        // return value based on custom algorithm.
    }

    @Override
    public BridgeContext getWrapped() {
        return wrappedBridgeContext;
    }
}
```

2. Create a wrapper class for the BridgeContextFactory:

```
package com.mycompany.myproject;

public class BridgeContextFactoryCustomImpl extends BridgeContextFactory {

    private BridgeContextFactory wrappedBridgeContextFactory;
```

```

public BridgeContextFactoryCustomImpl(
    BridgeContextFactory bridgeContextFactory) {

    this.wrappedBridgeContextFactory = bridgeContextFactory;
}

public BridgeContextFactory getWrapped() {
    return wrappedBridgeContextFactory;
}

@Override
public BridgeContext getBridgeContext(
    BridgeConfig bridgeConfig, BridgeRequestScope bridgeRequestScope,
    PortletConfig portletConfig, PortletContext portletContext,
    PortletRequest portletRequest, PortletResponse portletResponse,
    Bridge.PortletPhase portletPhase, PortletContainer portletContainer,
    IncongruityContext incongruityContext) {

    BridgeContext wrappedBridgeContext =
        wrappedBridgeContextFactory.getBridgeContext(
            bridgeConfig, bridgeRequestScope, portletConfig, portletContext,
            portletRequest, portletResponse, portletPhase, portletContainer,
            incongruityContext);

    BridgeContext bridgeContext =
        new BridgeContextCustomImpl(wrappedBridgeContext);

    return bridgeContext;
}
}

```

3. In the portlet's WEB-INF/faces-config.xml, specify the custom factory:

```

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:bridge="http://www.liferay.com/xml/ns/liferay-faces-bridge-2.0-
    extension"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
    <factory>
        <factory-extension>
            <bridge:bridge-context-factory>
                com.mycompany.myproject.BridgeContextFactoryCustomImpl
            </bridge:bridge-context-factory>
        </factory-extension>
    </factory>
</faces-config>

```

4. Rebuild and re-deploy the portlet.

That's all you need to do to implement and deploy a `BridgeContextFactory` wrapper. Next, we'll take a detailed look at how Liferay Faces Bridge satisfies the portlet bridge specifications and at some of the bridge's configuration options.

4.2. Understanding Liferay Faces Bridge

Liferay Faces Bridge is a JAR that you can add as a dependency to your portlet WAR projects, in order to deploy JSF web applications as portlets within JSR 286 (Portlet 2.0) compliant portlet containers, like Liferay Portal 5.2, 6.0, 6.1, and 6.2.

The Liferay Faces Bridge project home page can be found at <http://www.liferay.com/community/liferay-projects/liferay-faces/bridge>.

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.

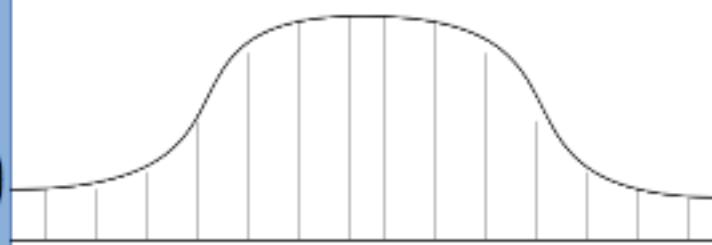
After the JSR 314 EG released JSF 2.0 in 2009 and JSF 2.1 in 2010, it became evident that a Portlet Bridge 3.0 standard would be beneficial. At the time of this writing, the JCP has not formed such an EG. In the meantime, Liferay developed *Liferay Faces Bridge*, which targets Portlet 2.0 and JSF 1.2/2.1/2.2.

Liferay Faces Bridge is an implementation of the JSR 329 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, let's consider the responsibilities required of the portlet bridge.

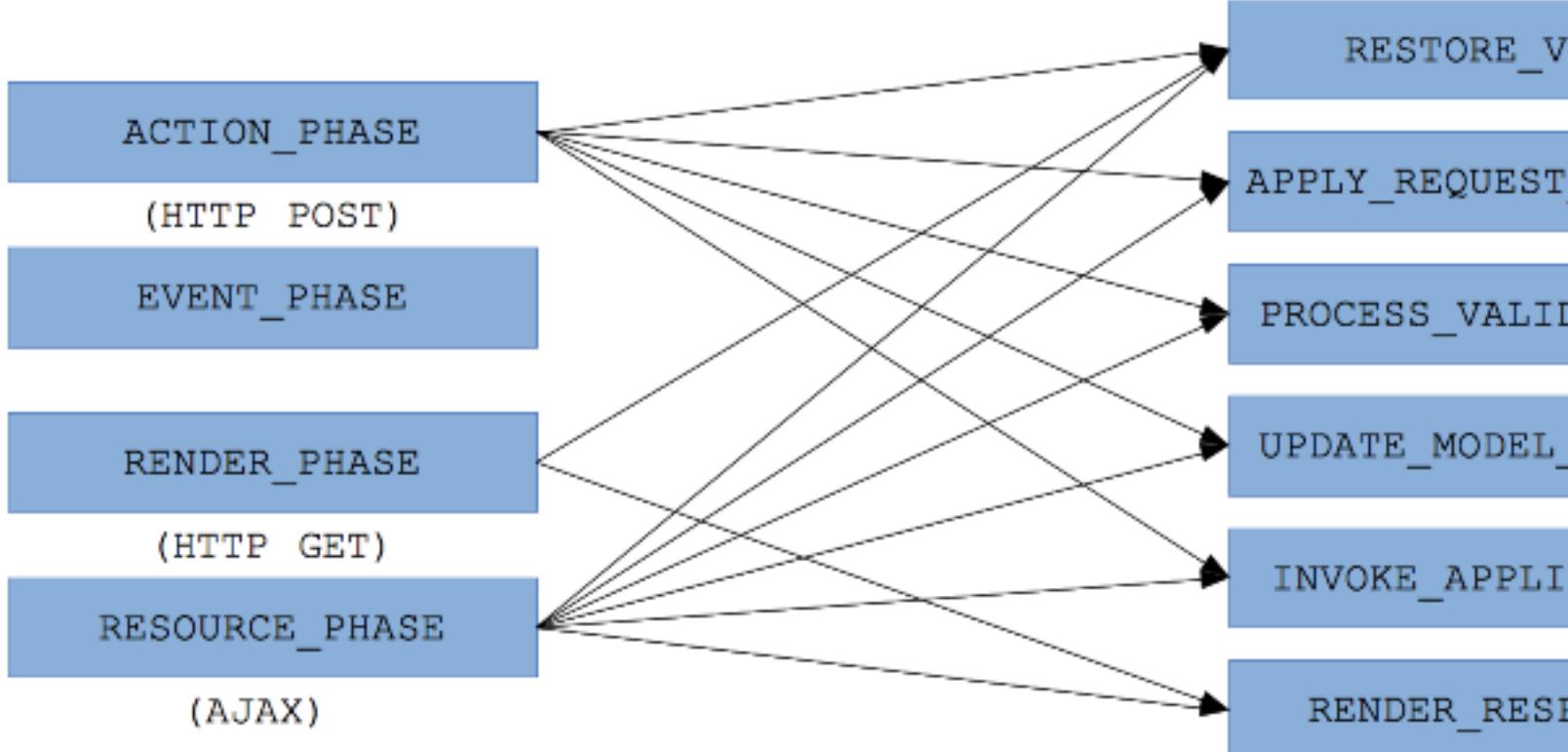
A JSF portlet bridge aligns the correct phases of the JSF lifecycle with each phase of the portlet lifecycle. For instance, if a browser sends an HTTP GET request to a portal page with a JSF portlet in it, the `RENDER_PHASE` is performed in the portlet's lifecycle. The JSF portlet bridge then initiates 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.

JSR 286 Portlet 2.0 Lifecycle



JSF Portlet Bridge

JSR 3... JSF 2... Lifecycle



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.

With the main aspects of JSF portlet bridges described, let's discuss the configuration of Liferay Faces Bridge.

4.2.1. Configuring Liferay Faces Bridge

The JSR 329 standard 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.

Let's consider configuring the Bridge Request Scope behavior first.

4.2.1.1. Configuring Bridge Request Scope Behavior

One of the key requirements in creating a JSF portlet bridge is managing JSF request-scoped data within the portlet lifecycle. This is normally referred to as the *Bridge Request Scope* by JSR 329. The lifespan of the Bridge Request Scope works like this:

1. ActionRequest/EventRequest: BridgeRequestScope begins.
2. RenderRequest: BridgeRequestScope is preserved.
3. Subsequent RenderRequest: BridgeRequestScope is reused.
4. Subsequent ActionRequest/EventRequest: BridgeRequestScope ends, and a new BridgeRequestScope begins.
5. If the session expires or is invalidated, then similar to the PortletSession scope, all BridgeRequestScope instances associated with the session are made available for garbage collection by the JVM.

The main use-case for having the `BridgeRequestScope` preserved in Step 2 (above) is for *re-rendering* portlets. Let's consider an example to help illustrate this use-case.

Let's say two or more JSF portlets are placed on a portal page (Portlets X and Y), and those portlets are *not* using `f:ajax` for form submission. In such a case, if the user were to submit a form (via full ActionRequest postback) in Portlet X, and then submit a form in Portlet Y, then Portlet X should be re-rendered with its previously submitted form data.

With the advent of JSF 2.x and Ajax, there were four drawbacks for continuing to support this use-case as the default behavior:

- Request-scoped data is basically semi-session-scoped in nature, because the `BridgeRequestScope` is preserved (even though the user might NEVER click the Submit button again).
- `BridgeRequestScope` can't be stored in the `PortletSession` because the data is request-scoped in nature, and the data stored in the scope isn't guaranteed to be `Serializable` for replication. Therefore, it doesn't really work well in a clustered deployment.
- The developer might have to specify the `javax.portlet.faces.MAX_MANAGED_REQUEST_SCOPES <init-param>`

in the WEB-INF/web.xml descriptor in order to tune the memory settings on the server.

As result, Liferay Faces Bridge was designed for JSF 2.x, and keeps Ajax in mind. The Liferay Faces Bridge makes the following assumptions:

- Developers are not primarily concerned about the *re-rendering of portlets* use-case mentioned above.
- Developers don't want any of the drawbacks mentioned above.
- Developers are making heavy use of the f:ajax tag and submitting forms via Ajax with their modern-day portlets.
- Developers want to do as little configuration as possible and don't want to be forced to add anything to the WEB-INF/web.xml descriptor.

Consequently, the default behavior of Liferay Faces Bridge is to cause the BridgeRequestScope to end at the end of the RenderRequest.

If you prefer the standard behavior over Liferay Faces Bridge's default behavior, then you can place the following option in your portlet's WEB-INF/web.xml descriptor:

```
<!--  
The default value of the following context-param is false, meaning that  
Liferay Faces Bridge will cause the BridgeRequestScope to end after the  
RENDER_PHASE of the portlet lifecycle. Setting the value to true will cause  
Liferay Faces Bridge to cause the BridgeRequestScope to last until the next  
ACTION_PHASE or EVENT_PHASE of the portlet lifecycle.  
-->  
<context-param>  
    <param-name>com.liferay.faces.bridge.bridgeRequestScopePreserved</param-  
name>  
    <param-value>true</param-value>  
</context-param>  
  
<!--  
The default value of the following context-param is 100. It defines the  
maximum number of BridgeRequestScope instances to keep in memory on the  
server if the bridgeRequestScopePreserved option is true.  
-->  
<context-param>  
    <param-name>javax.portlet.faces.MAX_MANAGED_REQUEST_SCOPES</param-name>  
    <param-value>2000</param-value>  
</context-param>
```

Alternatively, the com.liferay.faces.bridge.bridgeRequestScopePreserved value can be specified on a portlet-by-portlet basis in the WEB-INF/portlet.xml descriptor.

4.2.1.2. Using PreDestroy and BridgePreDestroy Annotations

When JSF developers want to perform cleanup on managed-beans before they are destroyed, they typically annotate a method inside the bean with the @PreDestroy annotation. However, section 6.8.1 of the JSR 329 standard discusses the need for the @BridgePreDestroy and @BridgeRequestScopeAttributeAdded annotations in the bridge API.



Note: For an in-depth discussion of this issue, please refer to <http://issues.liferay.com/browse/FACES-146>.

In order to explain this requirement, it is necessary to make a distinction between *local* portals and *remote* portals. Local portals invoke portlets that are deployed in the same (local) portlet container. Remote portals invoke portlets that are deployed elsewhere via WSRP (Web Services for Remote Portlets). The @BridgePreDestroy and

@BridgeRequestScopeAttributeAdded annotations were introduced into the JSR 329 standard primarily to support WSRP in remote portals. That being the case, the standard indicates that developers should always use @BridgePreDestroy instead of @PreDestroy. Liferay Faces Bridge however takes a different approach: rather than assuming the remote portal use-case, Liferay Faces Bridge assumes the local portal use-case. When developing with a local portal like Liferay, Liferay Faces Bridge ensures that the standard @PreDestroy annotation works as expected. This means there is no reason to use the @BridgeRequestScope annotation with a local portal when using Liferay Faces Bridge. Developers must manually configure Liferay Faces Bridge via the WEB-INF/web.xml descriptor in order to leverage the @BridgePreDestroy and @BridgeRequestScopeAttributeAdded annotations for WSRP.

<!--

The default value of the following context-param is false, meaning that Liferay Faces Bridge will invoke methods annotated with @PreDestroy over those annotated with @BridgePreDestroy. Setting the value of the following context-param instructs Liferay Faces Bridge to prefer the @BridgePreDestroy annotation over the standard @PreDestroy annotation in order to support a WSRP remote portal environment.

-->

```
<context-param>
    <param-name>com.liferay.faces.bridge.preferPreDestroy</param-name>
    <param-value>false</param-value>
</context-param>
```

<!--

The following listener is required to support the @BridgeRequestScopeAttributeAdded annotation in a WSRP remote portal environment.

-->

```
<listener>
    <listener-class>com.liferay.faces.bridge.servlet.BridgeRequestAttributeListener</listener-class>
</listener>
```

Alternatively, the com.liferay.faces.bridge.preferPreDestroy value can be specified on a portlet-by-portlet basis in the WEB-INF/portlet.xml descriptor.

4.2.1.3. Configuring the Portlet Container Abilities

Liferay Faces Bridge can be run in a variety of portlet containers (Liferay, Pluto, etc.) and is

aware of some of the abilities (or limitations) of these containers. Liferay Faces Bridge enables you to configure the abilities of the portlet container in the WEB-INF/web.xml descriptor.

```
<!--  
The default value of the following context-param depends on which portlet  
container the bridge is running in. The value determines whether or not the  
bridge resource handler will attempt to set the status code of downloaded  
resources to values like HttpServletResponse.SC_NOT_MODIFIED.  
-->  
<context-param>  
    <param-  
name>com.liferay.faces.bridge.containerAbleToSetHttpStatusCode</param-name>  
    <param-value>true</param-value>  
</context-param>
```

By configuring portlet container capabilities, you can take advantage of your portlet container's specific strengths while using Liferay Faces Bridge.

4.2.1.4. Configuring Portlet Namespace Optimization

The JSR 329 standard requires the bridge implementation to prepend the portlet namespace to every JSF view component's id attribute. This distinguishes the component when there are multiple JSF portlets on a portal page that contain similar component hierarchies and naming. Also, the JSR 329 standard indicates that the bridge implementation of the `ExternalContext.encodeNamesapce(String)` method is to prepend the value of `javax.portlet.PortletResponse.getNamespace()` to the specified String. The problem is that since the value returned by `getNamespace()` can be a lengthy String, the size of the rendered HTML portal page can become unnecessarily large. This can be especially non-performant when using the `f:ajax` tag in a Facelet view to perform partial-updates to the browser's DOM.

Liferay Faces Bridge has a built-in optimization that minimizes the value returned by the `ExternalContext.encodeNamesapce(String)` method, while still guaranteeing uniqueness.

If you don't want to leverage the namespace optimization and instead want to leverage the default behavior specified by JSR 329, you must set this value to false in the WEB-INF/web.xml descriptor:

```
<!--  
The default value of the following context-param is true, meaning that  
Liferay Faces Bridge will optimize the portlet namespace. Setting the value  
of the following context-param to false disables the optimization.  
-->  
<context-param>  
    <param-name>com.liferay.faces.bridge.optimizePortletNamespace</param-  
name>  
    <param-value>false</param-value>  
</context-param>
```



Note: Due to strict namespacing requirements introduced in Liferay Portal 6.2, the

namespace optimization feature only works in Liferay Portal 5.2, 6.0, and 6.1.

4.2.1.5. Configuring XML Entity Validation

Liferay Faces Bridge gives you the option of enabling or disabling XML validation for all faces-config.xml file entities. By default, the validation is disabled.

To enable XML validation for all faces-config.xml file entities, you can set the option to true in the WEB-INF/web.xml descriptor:

```
<!-- The default value of the following context-param is false. -->
<context-param>
    <param-name>com.liferay.faces.bridge.resolveXMLEntities</param-name>
    <param-value>true</param-value>
</context-param>
```

4.2.1.6. Configuring Resource Buffer Size

Liferay Faces Bridge lets you set the size of the buffer used to load resources into memory as file contents are copied to the response. The default value of this option is 1024 (1KB).

```
<!-- The default value of the following context-param is 1024. -->
<context-param>
    <param-name>com.liferay.faces.bridge.resourceBufferSize</param-name>
    <param-value>4096</param-value>
</context-param>
```

Alternatively, you can specify the

com.liferay.faces.bridge.resourceBufferSize value on a portlet-by-portlet basis in the WEB-INF/portlet.xml descriptor.

4.2.1.7. Configuring Distinct Request Scoped Managed Beans

Liferay Portal gives you the ability to specify whether or not request attributes are shared among portlets, using the <private-request-attributes> option in the WEB-INF/liferay-portlet.xml descriptor. The default value of this option is true, meaning that request attributes are NOT shared among portlets.

```
<liferay-portlet-app>
    <portlet>
        ...
        <private-request-attributes>false</private-request-attributes>
    </portlet>
    ...
</liferay-portlet-app>
```

However, this non-shared feature only works for request attributes that are present in the request map and that have a non-null value. This can cause a problem for JSF managed-beans in request scope. Specifically, the problem arises when a portal page has two or more portlets that have a

request scope managed bean with the same name.

For example, say Portlet X and Portlet Y each have a class named BackingBean annotated with @RequestScoped @ManagedBean. When the JSF runtime is asked to resolve an EL-expression # {backingBean}, there is no guarantee that the correct instance will be resolved. In order to solve this problem, Liferay Faces Bridge provides a configuration option that can be specified in WEB-INF/web.xml. It causes request-scoped managed beans to be distinct for each portlet.

```
<!-- The default value of the following context-param is false. -->
<context-param>
    <param-
name>com.liferay.faces.bridge.distinctRequestScopedManagedBeans</param-name>
        <param-value>true</param-value>
</context-param>
```

To ensure that @RequestScoped managed beans are resolved correctly for each portlet, set this value to true.

4.2.1.8. Configuring View Parameters

In the case of a portlet RenderRequest, Section 5.2.6 of the JSR 329 Spec requires that the bridge ensure that only the RESTORE_VIEW and RENDER_RESPONSE phases of the JSF lifecycle execute. In addition, Section 6.4 requires that a PhaseListener be used to skip the APPLY_REQUEST_VALUES, PROCESS_VALIDATIONS, UPDATE_MODEL_VALUES, and INVOKE_APPLICATION phases. These requirements are valid for JSF 1.x, but for JSF 2.x *View Parameters*, the presence of f:metadata and f:viewParam in a Facelet view, makes the entire JSF lifecycle run.

Liferay Faces Bridge enables support for View Parameters by default, but provides a configuration option in the WEB-INF/web.xml descriptor that lets you disable the feature.

```
<!-- The default value of the following context-param is true. -->
<context-param>
    <param-name>com.liferay.faces.bridge.viewParametersEnabled</param-name>
        <param-value>false</param-value>
</context-param>
```

If it is necessary to use the JSF 1.x version of this feature, then this parameter should be set to false.

Now that we've discussed JSF portlet bridge standards and Liferay Faces Bridge configuration options, let's learn how Liferay Faces Portal lets you leverage Liferay Portal's utilities and component tags.

4.3. Leveraging Liferay Utilities with Liferay Faces Portal

Let's first consider the Liferay Portal utilities available for you to use with your JSF portlets.

Since you're integrating your JSF portlet with Liferay Portal, you'll want to know how to access different parts of Liferay's development framework. In this section, we'll show you some of the key aspects of Liferay Portal that you can access via Liferay Faces Portal.

4.3.1. Using the LiferayFacesContext

LiferayFacesContext is an abstract class that extends the JSF FacesContext abstract class. Because of this, it supplies all the same method signatures. The LiferayFacesContext implements the delegation design pattern for methods defined by FacesContext by first calling FacesContext.getCurrentInstance() and then delegating to corresponding methods.

4.3.2. Leveraging the Current Theme

Liferay Faces Portal offers several features to help you access and use the current Liferay theme. Liferay Faces Portal provides the LiferayFacesContext.getThemeDisplay() method at the Java level and the liferay.themeDisplay EL variable at the Facelet level, for accessing the Liferay ThemeDisplay object.

Liferay Faces Portal provides the liferay-ui:icon Facelet composite component tag that encapsulates an HTML img tag whose src attribute contains a fully qualified URL to an icon image in the current Liferay theme. Additionally, Liferay Faces Portal provides the liferay.themeImagesURL and liferay.themeImageURL Facelet composite component tags for gaining access to theme image icons.

4.3.3. Giving Feedback to Users with Validation Messages

Most of the standard JSF HTML component tags render themselves as HTML markup such as <label />, <input />, , etc. and assume the current Liferay theme thanks to the power of CSS. However, the h:messages and h:message tag will not assume the current Liferay theme unless the following JSR 286 standard CSS class names portlet-msg-error, portlet-msg-info, and portlet-msg-warn are applied:

```
<h:messages errorClass="portlet-msg-error" fatalClass="portlet-msg-error"  
infoClass="portlet-msg-info" warnClass="portlet-msg-warn" />
```

As a convenience, Liferay Faces Portal provides the liferay-ui:message Facelet composite component tag that encapsulates the h:message tag. The liferay-ui:message tag automatically applies the JSR 286 standard class names, as shown above.



Note: When running as a portlet, the ICEfaces ice:messages and ice:message component tags automatically apply the JSR 286 standard class names too.

Additionally, the ice:dataTable component tag applies the following JSR 286 standard class names for alternating table rows:

- portlet-section-alternate
- portlet-section-body

Next, we'll look at using Liferay Faces Portal's language capabilities with JSF Portlets.

4.3.4. Leveraging the Portal User's Locale

By default, the Locale that is normally used to present internationalized JSF views is based on the web-browser's locale settings. In order to use the portal user's language preference, Liferay Faces Portal automatically registers the `LiferayLocalePhaseListener`. This phase listener modifies the locale inside the `UIViewRoot`, based on the user's language preference returned by the `User.getLocale()` method.

Now that you're familiar with some of the key utilities that you can access through Liferay Faces Portal, let's look at the `UIComponent` and composite component tags that you can leverage through Liferay Faces Portal.

4.4. Using Liferay Portal `UIComponent` and Composite Component-Tags

Liferay Faces Portal provides a set of Facelet `UIComponent` and Facelet Composite Component tags as part of its component suite.

Because Liferay Faces has several active versions (targeting different versions of JSF, Liferay Portal, etc.), there are several versions of the project's View Declaration Language (VDL) documentation for these tags. The VDL documentation can be found at the following addresses:

- The VDL documentation for Liferay Faces 2.1 can be found at <http://docs.liferay.com/faces/2.1/vdldoc/>.
- The VDL documentation for Liferay Faces 3.0-legacy can be found at <http://docs.liferay.com/faces/3.0-legacy/vdldoc/>.
- The VDL documentation for Liferay Faces 3.0 can be found at <http://docs.liferay.com/faces/3.0/vdldoc/>.
- The VDL documentation for Liferay Faces 3.1 can be found at <http://docs.liferay.com/faces/3.1/vdldoc/>.
- The VDL documentation for Liferay Faces 3.2 can be found at <http://docs.liferay.com/faces/3.2/vdldoc/>.

Liferay Faces Portal provides the following `UIComponent` tags under the `liferay-ui` and `liferay-security` tags.

4.4.1. The `liferay-ui:input-editor` tag

The `liferay-ui:input-editor` tag renders a text area that lets you enter rich text such as bold, italic, and underline. The renderer relies on the CKEditor™ to provide the rich text editing area. Since Liferay bundles the CKEditor™ JavaScript and related images with the portal, the portlet developer does not need to include it with the portlet.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<f:view xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:liferay-ui="xmlns:liferay-ui='http://liferay.com/faces/ui'">
```

```

<h:form>
    <liferay-ui:input-editor id="comments"
value="#{modelManagedBean.comments}" />
</h:form>

</f:view>

```



Note: Prior to Liferay 6.0 SP2 (6.0.12), the rich text area was rendered as an `<iframe>`. But due to incompatibilities with IE, the rich text area HTML markup is now rendered inline with the portal page. Liferay Faces Portal automatically detects the version of Liferay and renders the rich text area accordingly. However, if you are using Liferay 6.0 (6.0.10) or Liferay 6.0 SP1 (6.0.11) and have received an inline patch from Liferay Support, then you'll need to add the following context parameter to the portlet's WEB-INF/web.xml descriptor:

```

<context-param>
    <param-name>com.liferay.faces.portal.inlineInputEditor</param-name>
    <param-value>true</param-value>
</context-param>

```

If you're using ICEfaces, then the inline version of `liferay-ui:input-editor` exposes an inefficiency in the Direct2DOM™ (DOM-diff) algorithm. Typing a single character in the rich text area causes ICEfaces to detect a DOM-diff, causing the entire `liferay-ui:input-editor` to be replaced in the browser's DOM when the form is submitted via Ajax. To workaround this problem, use the JSF 2.x `f:ajax` component to optimize/control which parts of the JSF component tree are DOM-diffed by ICEfaces. For example, you could apply the `f:ajax` component like this:

```

<h:panelGroup id="feedback">
    <h:messages globalOnly="true" layout="table" />
</h:panelGroup>
<h:panelGroup id="editor">
    <liferay-ui:input-editor value="#{modelBean.text}" />
</h:panelGroup>
<h:commandButton>
    <f:ajax execute="@form" render="feedback" />
</h:commandButton>

```

Next, we'll look at the `liferay-ui` prefixed composite component tags.

4.5. Using Liferay Composite Component Tags

4.5.1. The `liferay-ui:ice-info-data-paginator` tag

The `liferay-ui:ice-info-data-paginator` encapsulates an ICEfaces 3.1 `ice:dataPaginator` tag that renders pagination information for an associated `ice:dataTable`. The navigation information matches the internationalized Liferay "showing-x-x-of-x-results" message. Since ICEfaces 4.0 removed support for `ice:dataPaginator`,

Liferay Faces 4.x no longer includes this feature.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<f:view xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ice="http://www.icesoft.com/icefaces/component"
    xmlns:liferay-ui="http://liferay.com/faces/ui">

    <liferay-ui:ice-info-data-paginator for="dataTable1" />
    <ice:dataTable id="dataTable1" value="#{modelManagedBean.rows}"
var="row">
        ...
    </ice:dataTable>

</f:view>
```

Next, we'll look at the liferay-ui:ice-nav-data-paginator composite component tag.

4.5.2. The liferay-ui:ice-nav-data-paginator tag

The liferay-ui:ice-info-data-paginator encapsulates an ICEfaces 3.1 ice:paginator tag that renders navigation controls for an associated ice:dataTable. The icons match the current Liferay theme. Since ICEfaces 4.0 has removed support for ice:paginator, Liferay Faces 4.x no longer includes this feature.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<f:view xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ice="http://www.icesoft.com/icefaces/component"
    xmlns:liferay-ui="http://liferay.com/faces/ui">

    <liferay-ui:ice-nav-data-paginator for="dataTable1" />
    <ice:dataTable id="dataTable1" value="#{modelManagedBean.rows}"
var="row">
        ...
    </ice:dataTable>

</f:view>
```

Next, we'll look at the liferay-ui:icon composite component tag.

4.5.3. The liferay-ui:icon tag

The liferay-ui:icon tag encapsulates an HTML img tag whose `src` attribute contains a fully qualified URL to an icon image in the current Liferay theme.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<f:view xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ice="http://www.icesoft.com/icefaces/component"
    xmlns:liferay-ui="http://liferay.com/faces/ui">
```

```
<liferay-ui:icon alt="#{i18n['delete']}
```

Next, we'll look at the liferay-security prefixed tags.

4.5.4. The liferay-security:permissionsURL tag

The liferay-security:permissionsURL tag renders an HTML anchor tag (hyperlink) to the Liferay Permissions screen for the associated resource.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<f:view xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:liferay-security="xmlns:liferay-
ui="http://liferay.com/faces/security">

    <h:form>
        <liferay-security:permissionsURL
            modelResource="myproject.model.Book"
            modelResourceDescription="Book"
            resourcePrimKey="#{modelManagedBean.book.bookId}" />
    </h:form>
</f:view>
```

Now that we have discussed the `LiferayFacesContext`, Liferay Faces Portal theme integration, and Liferay language integration, let's learn what UI components are available to use from Liferay Faces Alloy.

4.6. Leveraging AlloyUI Components with Liferay Faces Alloy

Liferay Faces Alloy is a .jar file that you add as a dependency in your JSF portlet project to leverage AlloyUI. Liferay Faces Alloy provides a way to use AlloyUI in a typical JSF development fashion. It provides a set of Facelet UIComponent and Facelet Composite Component tags as part of its component suite.

The Liferay Faces Alloy project home page can be found at
<http://www.liferay.com/community/liferay-projects/liferay-faces/alloy>.

Because Liferay Faces has several active versions (targeting different versions of JSF, Liferay Portal, etc.), there are several versions of the project's View Declaration Language (VDL) documentation for these tags. The VDL documentation can be found at the following addresses:

- The VDL documentation for the Liferay Faces 2.1 can be found at
<http://docs.liferay.com/faces/2.1/vdldoc/>.
- The VDL documentation for the Liferay Faces 3.0-legacy can be found at
<http://docs.liferay.com/faces/3.0-legacy/vdldoc/>.
- The VDL documentation for the Liferay Faces 3.0 can be found at
<http://docs.liferay.com/faces/3.0/vdldoc/>.
- The VDL documentation for the Liferay Faces 3.1 can be found at

<http://docs.liferay.com/faces/3.1/vdldoc/>.

- The VDL documentation for Liferay Faces 3.2 can be found at <http://docs.liferay.com/faces/3.2/vdldoc/>.

To see a sample JSF porlet that uses Liferay Faces Alloy, check out the JSF2 Registration Portlet. Next, let's get a handle on the versioning scheme used for Liferay Faces.

4.7. ***Understanding the Liferay Faces Version Scheme***

Liferay Faces follows a Major1.Major2.Minor-Type versioning scheme:

Major1:

- 1 = Portlet 1.0 Bridge for JSF 1.2 (JSR 301) (*Note that Liferay Faces does not support Portlet 1.0*)
- 2 = Portlet 2.0 Bridge for JSF 1.2 (JSR 329)
- 3 = Portlet 2.0 Bridge for JSF 2.1 (JSR TBA)
- 4 = Portlet 2.0 Bridge for JSF 2.2 (JSR TBA)

Major2:

- 0-legacy = Liferay 5.2
- 0 = Liferay 6.0
- 1 = Liferay 6.1
- 2 = Liferay 6.2

Minor:

- May contain bug fixes, improvements, and new features.

Type:

- GA (General Availability)
- RC (Release Candidate)
- BETA (Beta Quality)
- ALPHA (Alpha Quality)

Examples:

- 3.1.0-ga1: First GA release for JSF 2.0/2.1 for use with Liferay 6.1.x
- 3.1.1-ga2: Second GA release for JSF 2.0/2.1 for use with Liferay 6.1.x



Note: Some permutations of this versioning scheme are not supported, see table below for specific information on supported versions of JSF and Liferay Portal.

The following table displays the Liferay Faces version and its compatible Liferay Portal version and its compatible JSF version:

Liferay Faces Version	JSF Version (Major1)	Liferay Portal Version (Major2)
2.1.x	1.2	6.1
3.0.x-legacy	2.1	5.2
3.0.x	2.1	6.0
3.1.x	2.1	6.1

Liferay Faces Version	JSF Version (Major1)	Liferay Portal Version (Major2)
3.2.x	2.1	6.2
4.1.x	2.2	6.1
4.2.x (master)	2.2	6.2

While Liferay Faces Bridge is theoretically compatible with any portal that implements the Portlet 2.0 standard, it has been carefully tested for use with Liferay Portal versions 5.2, 6.0, 6.1, and 6.2 and has several optimizations that provide increased performance on Liferay.

If you've developed portlets that use the PortletFaces Bridge, you'll need to migrate them to Liferay Faces in order to deploy them using the Liferay Faces Bridge--don't worry, it's very straightforward. In the next section, we'll show you how easy it is to migrate to Liferay Faces.

4.8. *Migrating to Liferay Faces*

The Liferay Faces project originated from the <http://portletfaces.org> community website. On April 3, 2012 Liferay announced that it would be assuming leadership for the portletfaces.org community. Consequently, projects at portletfaces.org were repackaged under the Liferay Faces umbrella project and underwent the following name changes:

- AlloyFaces → Liferay Faces Alloy
- PortletFaces Bridge → Liferay Faces Bridge
- LiferayFaces → Liferay Faces Portal

Throughout this section, we'll cover the various replacements for old classes and namespaces.

4.8.1. **Migrating BridgeRequestAttributeListener**

PortletFaces Bridge provided a class named

`org.portletfaces.bridge.servlet.BridgeRequestAttributeListener` but
Liferay Faces Bridge uses

`com.liferay.faces.bridge.servlet.BridgeRequestAttributeListener`.

In order to migrate to the new class, you will need to refactor to the new package namespace, as a deprecated class has not been provided.

```
<!-- PortletFaces Bridge BridgeRequestAttributeListener -->
<web-app>
    <listener>
        ...
        <listener-class>
            org.portletfaces.bridge.servlet.BridgeRequestAttributeListener
        </listener-class>
        ...
    </listener>
</web-app>

<!-- Liferay Faces Bridge GenericFacesPortlet -->
<web-app>
    <listener>
        ...
        <listener-class>
            com.liferay.faces.bridge.servlet.BridgeRequestAttributeListener
        </listener-class>
    </listener>
</web-app>
```

```
...
</listener>
</web-app>
```

Next, we'll look at the migration of configuration option names.

4.8.2. Migrating Configuration Option Names

PortletFaces Bridge provided several configuration options for use within the WEB-INF/web.xml and WEB-INF/portlet.xml descriptors. In order to ease migration, the configuration option names have been reproduced in the Liferay Faces project. It is recommended that the new configuration option names be used, as shown in the following listing:

- org.portletfaces.bridge.containerAbleToSetHttpStatusCode → com.liferay.faces.bridge.containerAbleToSetHttpStatusCode
- org.portletfaces.bridgeRequestScopePreserved → com.liferay.faces.bridge.bridgeRequestScopePreserved
- org.portletfaces.bridge.optimizePortletNamespace → com.liferay.faces.bridge.optimizePortletNamespace
- org.portletfaces.bridge.preferPreDestroy → com.liferay.faces.bridge.preferPreDestroy
- org.portletfaces.bridge.resolveXMLEntities → com.liferay.faces.bridge.resolveXMLEntities
- org.portletfaces.bridge.resourceBufferSize → com.liferay.faces.bridge.resourceBufferSize

Next, we'll explain how file upload classes have changed between the PortletFaces Bridge and the Liferay Faces Bridge.

4.8.3. Migrating File Upload

PortletFaces Bridge provided classes named org.portletfaces.bridge.component.HtmlInputFile and org.portletfaces.bridge.component.UploadedFile, but Liferay Faces Bridge uses com.liferay.faces.bridge.component.HtmlInputFile and com.liferay.faces.bridge.component.UploadedFile, respectively. In order to migrate to the new classes, you will need to refactor to the new package namespace, as deprecated classes have not been provided.

```
// PortletFaces Bridge package name:
import org.portletfaces.bridge.component.HtmlInputFile;

// Liferay Faces Bridge package name:
import com.liferay.faces.bridge.component.HtmlInputFile;

// PortletFaces Bridge package name:
import org.portletfaces.bridge.component.UploadedFile;
```

```
// Liferay Faces Bridge package name:  
import com.liferay.faces.bridge.component UploadedFile;  
Next, we'll look at migrating Facelet Tags to Liferay Faces.
```

4.8.4. Migrating Facelet Tag Library Namespaces

The projects at portletfaces.org provided several UIComponents and Composite Components for use within Facelet views. The tag library documentation for these components has been migrated to VDL documentation for each version of the Liferay Faces Bridge:

- The VDL documentation for the Liferay Faces 2.1 can be found at <http://docs.liferay.com/faces/2.1/vdldoc/>.
- The VDL documentation for the Liferay Faces 3.0-legacy can be found at <http://docs.liferay.com/faces/3.0-legacy/vdldoc/>.
- The VDL documentation for the Liferay Faces 3.0 can be found at <http://docs.liferay.com/faces/3.0/vdldoc/>.
- The VDL documentation for the Liferay Faces 3.1 can be found at <http://docs.liferay.com/faces/3.1/vdldoc/>.
- The VDL documentation for Liferay Faces 3.2 can be found at <http://docs.liferay.com/faces/3.2/vdldoc/>.

Each link for the VDL documentation contains information about all the aui, aui-cc, bridge, liferay-ui, liferay-util, and liferay-security tags for that version of the Liferay Faces Bridge, so each VDL documentation link basically holds the entirety of the PortletFaces tag library documentation (with the exception of certain tags which are excluded from certain versions of the Liferay Faces Bridge).

4.8.5. Migrating GenericFacesPortlet

PortletFaces Bridge provided its own `org.portletfaces.bridge.GenericFacesPortlet` class but the Liferay Faces Bridge uses the JSR 329 standard

`javax.portlet.faces.bridge.GenericFacesPortlet` class. In order to ease migration, the old class still exists in Liferay Faces Bridge although it has been deprecated. It is recommended that the standard class name be used in all `WEB-INF/portlet.xml` `portlet-class` entries.

```
<!-- PortletFaces Bridge GenericFacesPortlet -->  
<portlet-app>  
    <portlet>  
        ...  
        <portlet-class>  
            org.portletfaces.bridge.GenericFacesPortlet  
        </portlet-class>  
        ...  
    </portlet>  
</portlet-app>  
  
<!-- Liferay Faces Bridge GenericFacesPortlet -->
```

```
<portlet-app>
  <portlet>
    ...
    <portlet-class>
      javax.portlet.faces.GenericFacesPortlet
    </portlet-class>
    ...
  </portlet>
</portlet-app>
```

Next we'll look at the new version of the LiferayFacesContext.

4.8.6. Migrating LiferayFacesContext

PortletFaces provided a class named

`org.portletfaces.liferay.faces.context.LiferayFacesContext` class but Liferay Faces Portal uses the

`com.liferay.faces.portal.context.LiferayFacesContext` class. In order to ease migration, the old class still exists in Liferay Faces Portal although it has been deprecated. It is recommended that the standard class name be used instead.

```
// LiferayFaces package name:
import org.portletfaces.liferay.faces.context.LiferayFacesContext;
```

```
// Liferay Faces Portal package name:
import com.liferay.faces.portal.context.LiferayFacesContext;
```

The next section explains some of the changes in Logging between PortletFaces and Liferay Faces.

4.8.7. Migrating Logging

The PortletFaces-Logging project at portletfaces.org has been moved into the Liferay Faces Bridge codebase. In order to keep using this logging API in your portlets, you will need to refactor to the new package namespace, as deprecated classes have not been provided.

```
// PortletFaces-Logging package names:
import org.portletfaces.logging.LoggerFactory;
import org.portletfaces.logging.Logger;
```

```
// Liferay Faces Bridge package names:
import com.liferay.faces.util.logging.LoggerFactory;
import com.liferay.faces.util.logging.Logger;
```

The last migration we'll look at is Portlet Preferences.

4.8.8. Migrating Portlet Preferences

PortletFaces Bridge provided its own

`org.portletfaces.bridge.preference.Preference` class but Liferay Faces Bridge uses the JSR 329 standard `javax.portlet.faces.preference.Preference` class. In order to migrate to the standard class, you will need to refactor to the new package namespace as deprecated classes have not been provided.

```
// PortletFaces Bridge package name:
```

```
import org.portletfaces.bridge.preference.Preference;  
  
// Liferay Faces Bridge package name:  
import javax.portlet.faces.preference.Preference;
```

And those are all the changes necessary to migrate projects from the PortletFaces Bridge to the Liferay Faces Bridge.

4.9. *Migrating From Liferay Faces 3.1 to Liferay Faces 3.2/4.2*

Liferay Faces 3.2 and 4.2 are compatible with Liferay Portal 6.2 (see the Liferay Faces Version Scheme for more info on Liferay Portal compatibility). Migrating to Liferay Faces 3.2/4.2 from Liferay Faces 3.1 requires a few changes to ensure your projects continue working correctly.

In this section, we'll cover the following migration topics:

- Migrating Liferay Faces Alloy Tags for Liferay Faces 3.2/4.2
- Migrating the `liferay-portlet.xml` File for Liferay Faces 3.2/4.2

First, let's explore Liferay Faces Alloy tag migration.

4.9.1. *Migrating Liferay Faces Alloy 3.1 Tags to Liferay Faces Alloy 3.2/4.2 Tags*

AlloyUI was upgraded from AlloyUI 1.5 to AlloyUI 2.0 between Liferay Portal 6.1 and Liferay Portal 6.2. The AlloyUI changes include the deprecation and removal of some JavaScript functionality, and the addition of some JavaScript widgets. As a result, in Liferay Faces 3.2 and 4.2, corresponding Liferay Faces Alloy tags are deprecated and some are replaced. We'll go over the Liferay Faces Alloy tag changes that you'll need to accommodate in your Facelets.

First, let's look at the changes to the `aui` namespace tags.

4.9.1.1. *Changes to the Liferay Faces Alloy aui Tags*

The following table identifies the `aui` tags that are deprecated in 3.2/4.2 and each tag's replacement, if a replacement exists.

Tag Deprecations in the `aui` Namespace

Deprecated <code>aui</code> Tag	Replacement <code>aui</code> Tag
<code>aui:column</code>	<code>aui:row</code> and <code>aui:col</code>
<code>aui:layout</code>	No Replacement



Note: The `aui:column` tag must be replaced by an `aui:col` tag that is nested within an `aui:row` tag.

Also, because `aui:col` has completely different attributes than `aui:column`, you must account for the `aui:col` attributes. For more information on the `aui:col` tag, see the VDLdocs for Liferay Faces 3.2.

Next, we'll look at the changes to the tags in the `aui-cc` namespace.

4.9.1.2. Changes to the Liferay Faces Alloy `aui-cc` Tags

Below is a table of the `aui-cc` tags that are deprecated or removed with respect to the Liferay Faces 3.2, 4.1, and 4.2 releases:

Tag Deprecations in the `aui-cc` Namespace

Deprecated <code>aui-cc</code> Tag
<code>aui-cc:button</code>
<code>aui-cc:input</code>
<code>aui-cc:select</code>
<code>aui-cc:message</code> (REMOVED in 4.1/4.2)
<code>aui-cc:messages</code> (REMOVED in 4.1/4.2)



Note: The `aui-cc:message` and `aui-cc:messages` tags have been completely removed in 4.1/4.2 because no analogous tags in Liferay Portal exist for them and their functionality is already implemented in the bridge's for `h:message` and `h:messages` tags, respectively.

Now that we've learned the AlloyUI related migration changes, we'll look at the modifications necessary to use Liferay Faces 3.1 portlets in Liferay Portal 6.2.

4.9.2. Migrating the `liferay-portlet.xml` File for Liferay Faces 3.2/4.2

Liferay Portal 6.2 has two compatible Liferay Faces Versions: 3.2 and 4.2. We provide Liferay Faces 3.2 for compatibility with JSF 2.1 and provide Liferay Faces 4.2 for compatibility with JSF 2.2 (see the Liferay Faces Version Scheme for more info on Liferay Portal and JSF compatibility). If you are currently using Liferay Faces 3.1 and are interested in upgrading from Liferay Portal 6.1 to 6.2, but aren't interested in compatibility with JSF 2.2, you should upgrade from Liferay Faces 3.1 to 3.2. But, if you want to use JSF 2.2 in addition to Liferay 6.2, you must upgrade to Liferay Faces 4.2.



Note: This guide only addresses upgrading as it relates to Liferay Portal. Upgrading from JSF 2.1 to 2.2 may require additional changes. For information on upgrading to JSF 2.2 you should check out JSF specific upgrade guides.

Liferay Portal 6.2 can enforce namespacing of portlet request parameters. But you must turn this off for your JSF portlets by specifying `<requires-namespaced-parameters>false</requires-namespaced-parameters>` in each `<portlet>` element of your portlet project's `WEB-INF/liferay-portlet.xml` file. Here's a snippet

that specifies this descriptor:

```
<liferay-portlet-app>
  <portlet>
    ...
    <requires-namespaced-parameters>false</requires-namespaced-
parameters>
    ...
  </portlet>
  ...
</liferay-portlet-app>
```

Turning off the parameter namespace requirement is all you need to do to upgrade your JSF portlets to Liferay Faces 3.2 or 4.2, for use in Liferay Portal 6.2.

As an example JSF portlet that runs on Liferay Portal 6.2, check out the demo JSF2-portlet and its `liferay-portlet.xml` file.

Next, we'll show you how to build the Liferay Faces project.

4.10. Building Liferay Faces From Source

You may have several reasons for downloading and building Liferay Faces from its project source code: - To try out the latest cutting edge changes - To investigate a suspected bug - To learn how Liferay Faces is implemented

Whatever your reasons may be, we're happy to show you how to access the Liferay Faces source code and build it.

We'll start with installing the `liferay-faces` project.

4.10.1. Installing the `liferay-faces` Project

It's important to install the version of Liferay Faces that you want. So, it's a good idea to check the Liferay Faces Version Scheme to confirm the version of Liferay Faces.

You can either install the project by cloning it from GitHub or by downloading it as a `.zip` file. We'll demonstrate both options.

Cloning the project from GitHub

Cloning the project, requires that you set up Git on your machine. Once you've set up Git, you can download the `liferay-faces` project from GitHub and work with a particular branch of the project, following these instructions:

1. Execute the following command from your terminal:

```
git clone https://github.com/liferay/liferay-faces.git
```

2. Navigate into that directory by executing `cd liferay-faces`.

3. Checkout the branch (master is the default branch) you want to use.

For example, to use the first milestone release of version 4.2.0, execute the following command:

```
git checkout 4.2.0-m1
```

Downloading the project as a .zip file

To download the liferay-faces project as a .zip file, follow these instructions:

1. Visit the Liferay Faces project page, <https://github.com/liferay/liferay-faces>.
2. Click on the *branch* drop-down menu and select the branch or tag for the version of the liferay-faces project that you'd like to use.
3. Click on the *Download Zip* button to download the [branch/tag name].zip file for that branch or tag.
4. Extract the .zip file contents to a location on your machine.
5. In a terminal window, navigate into the liferay-faces project's root directory:

```
cd liferay-faces-[version]
```

Now that you've installed the liferay-faces project, you can configure your environment for building the project.

4.10.2. Building Liferay Faces with Maven

Maven is required to build the liferay-faces project. You can download Maven from <http://maven.apache.org/download.cgi>. We recommend putting your Maven installation's bin directory in your system's \$PATH, so you can run the Maven executable (mvn) easily from your terminal.

1. Copy the externalLiferayFacesRepositories <profile> from settings.xml into your local \$HOME/.m2/settings.xml file. If you do not already have a settings.xml file in your Maven configuration, create a settings.xml file in your \$HOME/.m2 folder and copy the contents of the settings.xml file into it.
2. Build the source with Maven by executing the following command:

```
mvn clean package
```

Maven builds the following Liferay Faces artifacts:

- alloy/target/liferay-faces-alloy-[version].jar
- bridge-api/target/liferay-faces-bridge-api-[version].jar
- bridge-impl/target/liferay-faces-bridge-impl-[version].jar
- portal/target/liferay-faces-portal-[version].jar
- util/target/liferay-faces-util-[version].jar

That's it; you've built Liferay Faces from source!

In the next section, we'll reflect on what we've learned about developing JSF portlets with Liferay Faces.

4.11. Summary

You've come a long way since the initial sections on developing JSF portlets in Liferay Faces. You jumped right in with creating and deploying a JSF portlet using Liferay IDE. Then, you enhanced your portlet by implementing portlet preferences, internationalizing your portlet content, enabling your portlets to communicate with IPC, and context-scoping your managed beans with CDI. You learned how the Liferay Faces Bridge provides you with means to develop and deploy your JSF web apps as portlets.

You discovered how easy it was to use some of Liferay Portal's most common utilities and UI components via Liferay Faces Portal. And you surveyed the powerful AlloyUI components available through Liferay Faces Alloy. And let's not forget that Liferay Faces lets you use today's most popular JSF UI component suites including PrimeFaces, ICEFaces, and RichFaces. Liferay Faces provides you with the best options for creating JSF portlets on any JSR 286 (Portlet 2.0) compliant portlet container; and using Liferay IDE/Developer Studio, you can develop your JSF apps quickly and easily.

5. Generating Your Service Layer

The word *service* can mean many specific things, but the general dictionary definition states that it's "an act of helpful activity." Everyone has experienced this in some way. Whether it's an act of kindness from a friend or stranger or a service you pay for, in all instances, you have a need, and the service provides for that need.

Data-driven applications by their nature need access to a service for storing and retrieving their data. In a well-designed application, the application asks for data, and the service fetches it. The application can then display this data to the user, who reads it or modifies it. If the data is modified, the application passes it back to the service, and the service stores it. The application doesn't need to know anything about *how* the service does what it does. It trusts the service to handle storing and retrieving the data, freeing the application to be as robust as it needs to be.

This is what is called *loose coupling*, and it's a hallmark of good application design. If your application's service layer is self-contained, then you can swap it out for a better service layer when something more robust comes along, and you won't have to modify the application to take advantage of it.

Well, something more robust has come along, and it's called Service Builder. Using the Object-Relational Mapping engine provided by Hibernate along with a sophisticated code generator, Service Builder can help you implement your service layer in a fraction of the time it would normally take. But this isn't just any ordinary service layer: Service Builder also optionally helps you with remote services in multiple protocols, such as JSON and SOAP. And if you need to do something really funky with the database, it gets out of your way and lets you use whatever SQL queries you want.

Intrigued? We hope so. We'll cover the following topics:

- What is Service Builder?
- Defining Your Object-Relational Map
- Generating Services

- Writing Local Service Classes
- Calling Local Services
- Understanding the Service Builder-generated Code
- Using Model Hints
- Writing Remote Service Classes
- Developing Custom SQL Queries
- Configuring `service.properties`

As you can see, there is a lot to cover, so let's start by describing Service Builder in more detail.

5.1. **What is Service Builder?**

Service Builder is a model-driven code generation tool built by Liferay that allows developers to define custom object models called entities. Service Builder generates a service layer through object-relational mapping (ORM) technology that provides a clean separation between your object model and code for the underlying database. This frees you to add the necessary business logic for your application. Service Builder takes an XML file as input and generates the necessary model, persistence, and service layers for your application. These layers provide a clean separation of concerns. Service Builder generates most of the common code needed to implement create, read, update, delete, and find operations on the database, allowing you to focus on the higher level aspects of service design. In this section, we'll discuss some of the main benefits of using Service Builder:

- Integration with Liferay
- Automatically generated model, persistence, and service layers
- Automatically generated local and remote services
- Automatically generated Hibernate and Spring configurations
- Support for generating finder methods for entities and finder methods that account for permissions
- Built-in entity caching support
- Support for custom SQL queries and dynamic queries
- Saved development time

Liferay uses Service Builder to generate all of its internal database persistence code. In fact, all of Liferay's services, both local and remote, are generated by Service Builder. Additionally, the plugins' services in Liferay's Plugins SDK are generated by Service Builder. Service Builder's use in Liferay Portal and Liferay plugins demonstrates it to be a robust and reliable tool. As we'll see throughout this chapter, Service Builder is easy to use and can save developers *lots* of development time. Although the number of files Service Builder generates can seem intimidating at first, developers only need to work with a few files in order to make customizations to their applications and add business logic.



Note: You don't have to use Service Builder for plugin or portlet development. It's entirely possible to develop Liferay plugins by writing custom code for database persistence using your persistence framework of choice, such as JPA or Hibernate.

One of the main ways Service Builder saves development time is by completely eliminating the need to write and maintain database access code. To generate a basic service layer, you only need to create a `service.xml` file and run Service Builder. This generates a new service `.jar` file for your project. The generated service `.jar` file includes a model layer, a persistence layer, a service layer, and related infrastructure. These distinct layers represent a healthy separation of concerns. The model layer is responsible for defining objects to represent your project's entities, the persistence layer is responsible for saving entities to and retrieving entities from the database, and the service layer is responsible for exposing CRUD and related methods for your entities as an API. The code Service Builder generates is database-agnostic, as is Liferay itself.

Each entity generated by Service Builder contains a model implementation class, a local service implementation class, and optionally a remote service implementation class. Customizations and business logic can be implemented in these classes; in fact, these are the only classes generated by Service Builder that are intended to be customized. Ensuring that all customizations take place in only a few classes makes Service Builder projects easy to maintain. The local service implementation class is responsible for calling the persistence layer to retrieve and store data entities. Local services contain the business logic and access the persistence layer. They can be invoked by client code running in the same Java Virtual Machine. Remote services usually have additional code for permission checking and are meant to be accessible from anywhere over the Internet or your local network. Service Builder automatically generates the code necessary to allow access to the remote services. The remote services generated by Service Builder include SOAP utilities and can be accessed via SOAP or JSON.

Another way Service Builder saves development time is by providing Spring and Hibernate configurations for your project. Service Builder uses Spring dependency injection for making service implementation classes available at runtime and uses Spring AOP for database transaction management. Service Builder also uses the Hibernate persistence framework for object-relational mapping. As a convenience to developers, Service Builder hides the complexities of using these technologies. Developers can take advantage of Dependency Injection (DI), Aspect Oriented Programming (AOP), and Object-Relational Mapping (ORM) in their projects without having to manually set up a Spring or Hibernate environment or make any configurations.

Another benefit of using Service Builder is that it provides support for generating finder methods. Finder methods retrieve entity objects from the database based on specified parameters. You just need to specify the kinds of finder methods to be generated in the `service.xml` configuration file and Service Builder does the rest. The generated finder methods allow you, for example, to retrieve a list of all entities associated with a certain site or a list of all entities associated with a certain site *and* a certain user. Service Builder not only provides support for generating this kind of simple finder method but also for finder methods that take Liferay's permissions into account. For example, if you are using Liferay's permissions system to protect access to your entities, Service Builder can generate a different kind of finder method that returns a list of entities that the logged-in user has permission to view.

Service Builder also provides built-in caching support. Liferay caches objects at three levels: *entity*, *finder*, and *Hibernate*. By default, Liferay uses Ehcache as an underlying cache provider

for each of these cache levels but this is configurable via portal properties. All you have to do to enable entity and finder caching for an entity in your project is to set the `cache-enabled=true` attribute of your entity's `<entity>` element in your `service.xml` configuration file. Please refer to the Liferay Clustering section of *Using Liferay Portal 6.2* for more details about Liferay caching.

Service Builder is a flexible tool. It automates many of the common tasks associated with creating database persistence code but it doesn't prevent you from creating custom SQL queries or custom finder methods. Service Builder allows developers to define custom SQL queries in an XML file and to implement custom finder methods to run the queries. This could be useful, for example, for retrieving specific pieces of information from multiple tables via an SQL join. Service Builder also supports retrieving database information via dynamic query.

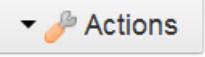
In summary, we encourage developers to use Service Builder for portlet and plugin development because it's a proven solution used by many Liferay plugins and by Liferay Portal itself. It generates distinct model, persistence, and service layers, local and remote services, Spring and Hibernate configurations, and related infrastructure without requiring any manual intervention by developers. It also allows basic SQL queries and finder methods to be generated and ones that filter results, taking Liferay's permissions into account. Service Builder also provides support for entity and query caching. Each of these features can save lots of development time, both initial development time and time that would have to be spent maintaining, extending, or customizing a project. Finally, Service Builder is not a restrictive tool: it allows custom SQL queries and finder methods to be added and it also supports dynamic query. Next, let's roll up our sleeves and learn how to use Service Builder.

5.2. Defining Your Object-Relational Map

In order to demonstrate how to use Service Builder, let's continue using the event-listing-portlet project that we created in Developing Apps with Liferay IDE. It's an example portlet project that Nose-ster, a fictitious organization, can use to schedule social events. We're using the event-listing-portlet project to manage and list these events. We need to add some entities, or model types, to represent Nose-ster's events and the locations where they hold their events. We'll define two entities: *events* and *locations*. The event entity represents a social event that can be scheduled for Nose-ster, while the location entity represents a location at which a social event can take place. Since an event must have a location, the event entity will reference a location entity as one of its attributes.

Event Listing Portlet

This is the Event Listing Portlet portlet in View mode.

Add Event				
Name	Description	Location	Date	
Tour Museo del Prado	Enjoy the "Captive Beauty" exhibition--"Hidden Beauty. From Fra Angelico to Fortuny."	Museo del Prado	01/11/2013 11:00 AM	
Muffaletta Mania!	Huge amazing sandwiches dressed with "the works!"	New Orleans	04/03/2014 03:30 PM	

If you'd like to examine the finished example project, it's a part of our *Dev Guide SDK* which you can browse at <https://github.com/liferay/liferay-docs/tree/master/devGuide/code/devGuide-sdk>. The project is in the SDK's portlets/event-listing-portlet folder.



Note: If you're looking for a fully-functional portlet application that can manage events, please use Liferay's Calendar portlet instead. The example described in this section is only intended to demonstrate how to use Service Builder. The Calendar portlet provides many more features than the simple example application described here. For information about the Calendar portlet, please refer to the chapter on Liferay's collaboration suite in *Using Liferay Portal 6.2*.

As with any portlet project, the event-listing-portlet project's Java sources lie in its docroot/WEB-INF/src folder. Notice that Liferay IDE's portlet wizard created the `EventListingPortlet.java` and `LocationListingPortlet.java` files in the `com.nostester.portlet.eventlisting` package. We'll add some business logic to these portlet classes after using Service Builder to create a service layer for our event and location entities.

The first step in using Service Builder is to define your model classes and their attributes in a `service.xml` file in your project's docroot/WEB-INF folder. In Service Builder terminology, your model classes (events and locations) are called entities. We've kept the requirements for our event and location entities fairly simple. Events should have the following

attributes:

Event Attributes

Attribute	Type	Attribute Description
name	String	The name of the event
description	String	A description of the event
date	Date	The date and time the event takes place
locationId	long	An event takes place at a location and we use a location Id to specify the location

Locations should have the following attributes:

Location Attributes

Attribute	Type	Attribute Description
name	String	The name of the location
description	String	A description of the location
streetAddress	String	The street address of the location
city	String	The city of the location
stateOrProvince	String	The state or province of the location
country	String	The country of the location

Service Builder defines a single file called `service.xml` for describing entities. Once you create the file, you can then define your entities. We'll walk you through the whole process for the entities we've defined above, using Liferay IDE, which makes it easy. It'll only take seven steps to do it:

1. Create the `service.xml` file in your project's docroot/WEB-INF folder, if one does not already exist there.
2. Define global information for the service.
3. Define service entities.
4. Define the columns (attributes) for each service entity.
5. Define relationships between entities.
6. Define a default order for the entity instances to be retrieved from the database.
7. Define finder methods that retrieve objects from the database based on specified parameters.

Let's start creating our service by using Liferay IDE to create your `service.xml` file.

5.2.1. Creating the `service.xml` File

To define a service for your portlet project, you must create a `service.xml` file. The DTD (Document Type Declaration) file http://www.liferay.com/dtd/liferay-service-builder_6_2_0.dtd specifies the format and requirements of the XML to use. You can create your `service.xml` file manually, following the DTD, or you can use Liferay IDE. Liferay IDE helps you build the `service.xml` file piece-by-piece, taking the guesswork out of creating XML that adheres to

the DTD. For our tutorial, we'll use Liferay IDE to build the `service.xml` file.

If a default `service.xml` file already exists in your `docroot/WEB-INF/src` folder, check to see if it has an `<entity />` element named "Foo". If it has the Foo entity, remove the entire `<entity name="Foo" ...> ... </entity>` element. The project wizard creates the Foo entity as an example, but it's of no use to us in this exercise.

If you don't already have a `service.xml` file it's easy to create one using Liferay IDE. Simply select your `event-listing-portlet` project in the Package Explorer and then select `File → New → Liferay Service Builder`. Liferay IDE creates a `service.xml` file in your `docroot/WEB-INF/src` folder and displays the file in *Overview* mode.

Liferay IDE also provides a *Diagram* mode and a *Source* mode to give you different perspectives of the service information in your `service.xml` file. Diagram mode is helpful for creating and visualizing relationships between service entities. Source mode brings up the `service.xml` file's raw XML content in the editor. You can switch between these modes as you wish. Since *Overview* mode facilitates creating service elements, we'll use it while creating our service.

Let's start filling out the global information for our service.

5.2.2. Defining Global Service Information

A service's global information applies to all of its entities, so let's specify this information first. Select the *Service Builder* node in the upper left corner of the *Overview* mode of your `service.xml` file. The main section of the view now shows the Service Builder form in which we can enter our service's global information. The fields include the service's package path, author, and namespace options. Here are the values we'll use for our example service:

- **Package path:** `com.nosester.portlet.eventlisting`
- **Auto namespace tables:** `no`
- **Author:** [your name]
- **Namespace:** `Event`

The *package path* specifies the package in which the service and persistence classes are generated. The package path we defined above ensures that the service classes are generated in the `com.nosester.portlet.eventlisting` package under the `docroot/WEB-INF/service` folder. The persistence classes are generated in a package of that name under the `docroot/WEB-INF/src` folder. The complete file paths for the service and persistence classes are `docroot/WEB-INF/service/com/nosester/portlet/eventlisting` and `docroot/WEB-INF/src/com/nosester/portlet/eventlisting`, respectively. Please refer to next section, *Generating the Services*, for a description of the contents of these packages.

Service Builder uses the service *namespace* in naming the database tables it generates for the service. Enter `Event` as the namespace for your example service. Service Builder uses the namespace in the following SQL scripts it generates in your `docroot/WEB-INF/sql` folder:

- `indexes.sql`
- `sequences.sql`

- `tables.sql`

Liferay Portal uses these scripts to create database tables for all the entities defined in the `service.xml` file. Service Builder prepends the namespace to the database table names. Since our namespace value is `Event`, the names of the database tables created for our entities start with `Event_` as their prefix. The namespace for each Service Builder project must be unique. Separate plugins should use separate namespaces and should not use a namespace already used by Liferay (such as `Users` or `Groups`). Check the table names in Liferay's database if you're wondering which namespaces are already in use.

As the last piece of global information, enter your name as the service's *author* in your `service.xml` file. Service Builder adds `@author` annotations with the specified name to all of the generated Java classes and interfaces. Save your `service.xml` file to preserve the information you added. Next, we'll add entities for your service's events and locations.

5.2.3. Defining Service Entities

Entities are the heart and soul of a service. Entities represent the map between the model objects in Java and the fields and tables in your database. Once your entities are defined, Service Builder handles the mapping automatically, giving you a facility for taking Java objects and persisting them. For this example, you'll create two entities--one for events and one for locations.

Here's a summary of the information we'll enter for the `Event` entity:

- **Name:** `Event`
- **Local service:** yes
- **Remote service:** yes

And here's what we'll enter for the `Location` entity:

- **Name:** `Location`
- **Local service:** yes
- **Remote service:** yes

To create these entities, select the *Entities* node under the Service Builder node in the outline on the left side of the `service.xml` editor in Overview mode. In the main part of the view, notice that the *Entities* table is empty. Create an entity by clicking on the *Add Entity* icon (a green plus sign) to the right of the table. Enter `Event` for your entity's name and select both the *Local Service* and the *Remote Service* options. Create a second entity named `Location` and select the *Local Service* and the *Remote Service* options for it too.

The screenshot shows the Liferay Service Builder interface. The title bar says '*service.xml'. The main area has two tabs: 'Outline' and 'Entities'. The 'Entities' tab is selected, showing a table with three columns: 'Name', 'Local Service', and 'Remote Service'. There is one row for 'Event', with checked boxes in both 'Local Service' and 'Remote Service' columns. To the right of the table is a toolbar with icons for search, refresh, and export. Below the toolbar is a large red box highlighting the 'Add Entity' button, which has a green plus sign icon. On the left, there's a tree view under 'Service Builder' with nodes for 'Entities', 'Event', 'Columns', 'Order', and 'Finders'. A filter input field is also present.

An entity's name is used to name the database table for persisting instances of the entity. The actual name of the database table is prefixed with the namespace; for our example, we'll have one database table named `Event_Event` and another named `Event_Location`.

Setting the `local service` attribute to `true` instructs Service Builder to generate local interfaces for our entity's service. The default value for local service is `false`. The design of this portlet, however, dictates that we be able to call the service, and it resides in our portlet's `.war` file. Our portlet will be deployed to our Liferay server. So the service will be local from our Liferay server's point of view.

Setting the `remote service` attribute to `true` instructs Service Builder to generate remote interfaces for the service. The default value for remote service is `true`. We could build a fully-functional event listing application without generating remote services, so we could set local service to `true` and remote service to `false` for both of our entities. Since, however, we want to demonstrate how to use the web services that Service Builder generates for our entities, we'll set both local service and remote service to `true`.



Tip: Suppose you have an existing DAO service for an entity built using some other framework such as JPA. You can set local service to `false` and remote service to `true` so that the methods of your remote `-Impl` class can call the methods of your existing DAO. This enables your entity to integrate with Liferay's permission-checking system and provides access to the web service APIs generated by Service Builder. This is a very handy, quite powerful, and often used feature of Liferay.

Now that we've created our Event and Location entities, let's describe their attributes using entity *columns*.

5.2.4. Defining the Columns (Attributes) for Each Service Entity

Each entity is described by its columns, which represent an entity's attributes. These attributes map on the one side to fields in a table and on the other side to attributes of a Java object. To add attributes for the Event entity, you need to drill down to its columns in the Overview mode outline of the `service.xml` file. From the outline, expand the *Entities* node and expand the new *Event* entity node. Then select the *Columns* node. Liferay IDE displays a table of the Event entity's columns.

Service Builder creates a database field for each column we 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 accessor methods in the model class for these attributes. The column's Name specifies the name used in the getters and setters that are created for the entity's Java field. The column's Type indicates the Java type of this field for the entity. If a column's Primary (i.e., primary key) attribute value is set to `true`, then the column becomes part of the primary key for the entity. An entity's primary key is a unique identifier for the entity. If only one column has Primary set to `true`, then that column represents the entire primary key for the entity. This is the case in our example. However, it's possible to use multiple columns as the primary key for an entity. In this case, the combination of columns makes up a compound primary key for the entity.

Similar to the way you used the form table for adding entities, add attribute columns for the entities as follows:

Event attribute columns

Name	Type	Primary
eventId	long	yes
name	String	no
description	String	no
date	Date	no

Location attribute columns

Name	Type	Primary
locationId	long	yes
name	String	no
description	String	no
streetAddress	String	no
city	String	no
stateOrProvince	String	no
country	String	no

Create each attribute by clicking the *add* icon. Then fill in the name of the attribute, select its type, and specify whether it is a primary key for the entity. While your cursor is in a column's *Type* field, an option icon appears. Click this icon to select the appropriate type for the column. Create a column for each attribute of your Event entity. Repeat the steps to create columns for each attribute of your Location entity.

In addition to columns for your entity's primary key and attributes, we recommend including columns for site ID and portal instance ID. They allow your portlet to support the multi-tenancy features of Liferay, so that each portal instance and each site in a portal instance can have independent sets of portlet data. To hold the site's ID, add a column called `groupId` of type `long`. To hold the portal instance's ID, add a column called `companyId` of type `long`. Add both of these columns to your Event and Location entities.

Portal and site scope columns

Name	Type	Primary
<code>companyId</code>	<code>long</code>	no
<code>groupId</code>	<code>long</code>	no

We'll also want to know who owns each entity instance. To keep track of that, add a column called `userId` of type `long`.

User column

Name	Type	Primary
<code>userId</code>	<code>long</code>	no

Lastly, add columns to help audit both of the Event and Location entities. Add a column named `createDate` of type `Date` to note the date an entity instance was created. And add a column named `modifiedDate` of type `Date` to track the last time an entity instance was modified.

Audit columns

Name	Type	Primary
<code>userId</code>	<code>long</code>	no
<code>createDate</code>	<code>Date</code>	no
<code>modifiedDate</code>	<code>Date</code>	no

Great! Our entities are set with the columns that not only represent their attributes, but also support multi-tenancy and entity auditing. Next, we'll specify the relationship between our Event entity and Location entity.

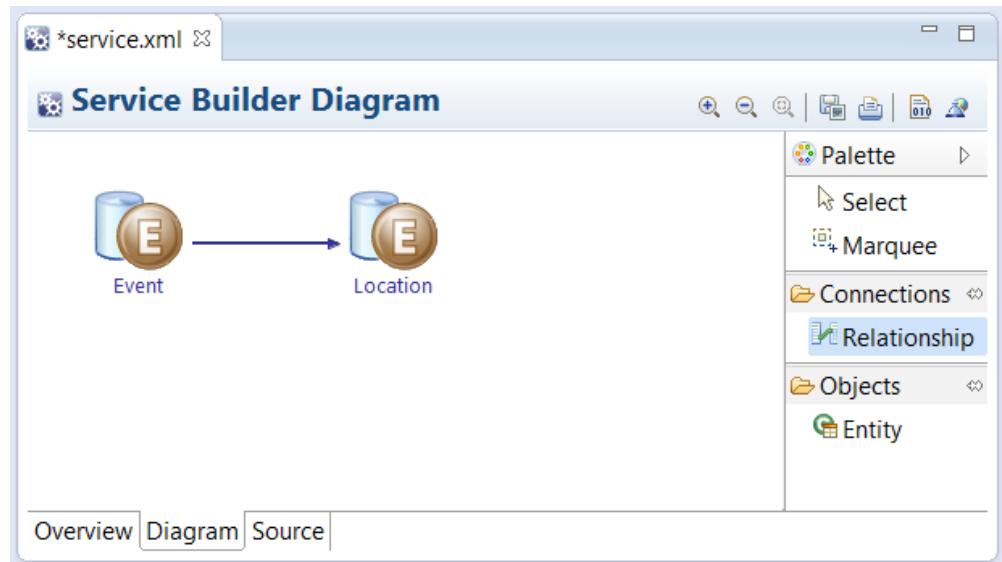
5.2.5. Defining Relationships Between Service Entities

Often you'll want to reference one type of entity in the context of another entity. That is, you'll want to *relate* the entities. We'll show you how to do this in our example Event Listing project.

As we mentioned earlier for the example, each event must have a location. Therefore, each Event entity must relate to a Location entity. The good news is that Liferay IDE's Diagram mode for `service.xml` makes relating entities easy. First, select Diagram mode for the `service.xml` file. Then select the *Relationship* option under *Connections* in the palette on the right side of the view. This relationship tool helps you draw relationships between entities in the diagram. Click the *Event* entity and move your cursor over the Location entity. Liferay IDE draws a dashed line from the Event entity to the cursor. Click the *Location* entity to complete drawing the relationship. Liferay IDE turns the dashed line into a solid line, with an arrow pointing to the Location entity. Save the `service.xml` file.

Congratulations! You've related the entities. Their relationship should show in Diagram mode

and look similar to that of the figure below.



Switch to *Source* mode in the editor for your service.xml file and note that Liferay IDE created a column element in the Event entity to hold the ID of the Location entity instance rerefERENCE:

```
<column
```

```
name="locationId" type="long"></column>
```

Now that our entity columns are in place, let's specify the default order in which the entity instances are retrieved from the database.

5.2.6. Defining Ordering of Service Entity Instances

Often, you want to retrieve multiple instances of a given entity and list them in a particular order. Liferay lets you specify the default order of the entities in your service.xml file.

Say you want to return Event entities in order by date, earliest to latest, and you want to return Location entities alphabetically by name. It's easy to specify these default orderings using Liferay IDE. Switch back to *Overview* mode in the editor for your service.xml file. Then select the *Order* node under the Event entity node in the outline on the left side of the view. The IDE displays a form for ordering the Event entity. Select the *Specify ordering* checkbox to show the form for specifying the ordering. Create an order column by clicking the *add icon* (a green plus sign) to the right of the table. Enter *date* for the column name to use in ordering the Event entity. Click the *Browse icon* to the right of the *By* field and choose the *asc* option. This orders the Event entity by ascending date. To specify ordering for Location entity instances, follow similar steps but specify *name* as the column and *asc* as the *select by* value.

The last thing do is define the finder methods for retrieving their instances from the database.

5.2.7. Defining Service Entity Finder Methods

Finder methods retrieve entity objects from the database based on specified parameters. You'll probably want to create at least one finder method for each entity you create in your services. Service Builder generates several methods based on each finder you create for an entity. It creates methods to fetch, find, remove, and count entity instances based on the finder's parameters.

For our example, we want to find Event and Location entities per site. We'll specify these finders using Liferay IDE's Overview mode of `service.xml`. Select the *Finders* node under the Event entity node in the outline on the left side of the screen. The IDE displays an empty *Finders* table in the main part of the view. Create a new finder by clicking the *add icon* (a green plus sign) to the right of the table. Name the finder *GroupId* and enter *Collection* as its return type. We use the Java camel-case naming convention in naming finders since the finder's name is used to name the methods Service Builder creates. The IDE creates a new *GroupId* node under the *Finders* node in the outline. We'll specify the finder column for this group ID node next.

Under the new *GroupId* node, the IDE created a *Finder Columns* node. Select *Finder Columns* node to specify the columns for our finder's parameters. Create a new finder column by clicking the *add icon* (a green plus sign) and specifying *groupId* as the column's name. Keep in mind that you can specify multiple parameters (columns) for a finder; this first example is kept simple. Follow similar steps to create a finder to retrieve Location entities by `groupId`. Save the `service.xml` file to preserve the finders you defined.

When you run Service Builder, it generates finder-related methods (`fetchByGroupId`, `findByGroupId`, `removeByGroupId`, `countByGroupId`) for the Event and Location entities in `-Persistence` and `-PersistenceImpl` classes. The first of these classes is the interface; the second is its implementation. The Event and Location entity's finder methods are generated in the `-Persistence` classes found in your `/docroot/WEB-INF/service/com/nosester/portlet/eventlisting/service/persistence` folder and the `-PersistenceImpl` classes found in your `/docroot/WEB-INF/src/com/nosester/portlet/eventlisting/service/persistence` folder.

Terrific! You've created the example service and its Event and Location entities for the Event Listing example project.

We've made the source code for the service and the entire Event Listing example project available in the *Dev Guide SDK* which you can browse at <https://github.com/liferay/liferay-docs/tree/master/devGuide/code/devGuide-sdk>. The project is in the SDK's `portlets/event-listing-portlet`. folder.

We've also listed the `service.xml` content here for your convenience. We've added some comments to highlight the service's various elements. Other than that, your `service.xml` file's contents should look similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 6.2.0//EN"
"http://www.liferay.com/dtd/liferay-service-builder_6_2_0.dtd">
<service-builder package-path="com.nosester.portlet.eventlisting">
    <author>Joe Bloggs</author>
    <namespace>Event</namespace>

    <entity name="Event" local-service="true" remote-service="true">
        <!-- PK fields -->
        <column name="eventId" type="long" primary="true" />
```

```

<!-- Audit fields -->

<column name="companyId" type="long" />
<column name="groupId" type="long" />
<column name="userId" type="long" />
<column name="createDate" type="Date" />
<column name="modifiedDate" type="Date" />

<!-- Other fields -->

<column name="name" type="String" />
<column name="description" type="String" />
<column name="date" type="Date" />
<column name="locationId" type="long" />

<!-- Order -->

<order by="asc">
    <order-column name="date" />
</order>

<!-- Finder methods -->

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

<entity name="Location" local-service="true" remote-service="true">

    <!-- PK fields -->

    <column name="locationId" type="long" primary="true" />

    <!-- Audit fields -->

    <column name="companyId" type="long" />
    <column name="groupId" type="long" />
    <column name="userId" type="long" />
    <column name="createDate" type="Date" />
    <column name="modifiedDate" type="Date" />

    <!-- Other fields -->

    <column name="name" type="String" />
    <column name="description" type="String" />
    <column name="streetAddress" type="String" />
    <column name="city" type="String" />
    <column name="stateOrProvince" type="String" />
    <column name="country" type="String" />

    <!-- Order -->

```

```

<order by="asc">
    <order-column name="name" />
</order>

<!-- Finder methods -->

<finder name="GroupId" return-type="Collection">
    <finder-column name="groupId" />
</finder>
</entity>
</service-builder>

```

Now that you've specified the service for the Event Listing example project, let's *build* the service by running Service Builder. Then we'll look at the code Service Builder generates.

5.3. Generating Services

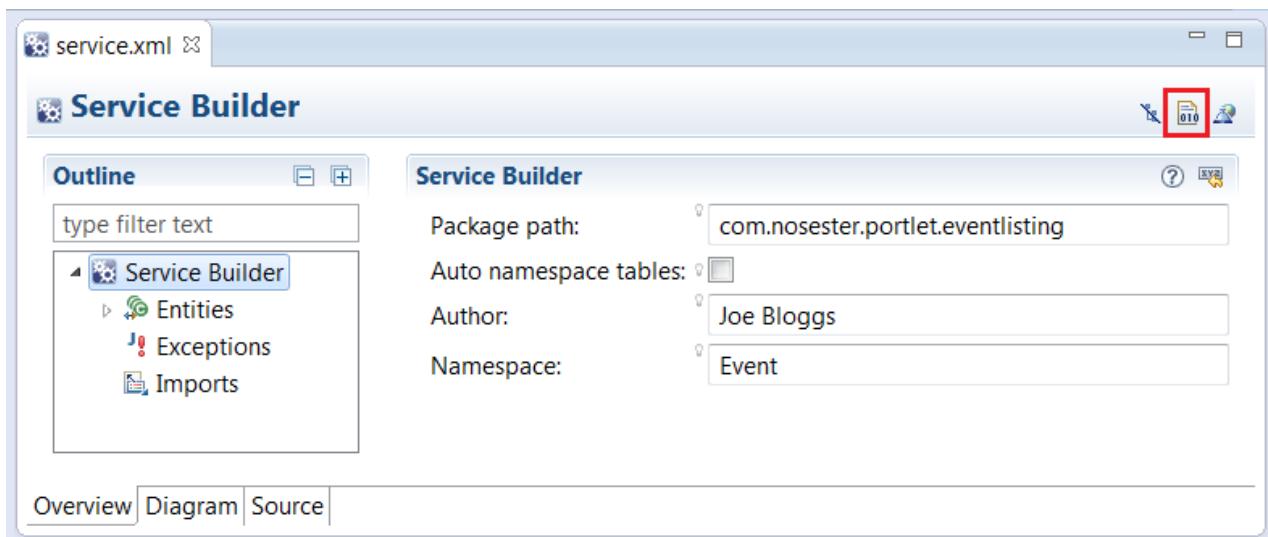
To build a service from a `service.xml` file, you can use *Liferay IDE*, *Liferay Developer Studio*, or use a terminal window. Next, you'll generate the service for the example Event Listing example project you've been developing throughout this chapter. The project resides in the `portlets/event-listing-portlet` folder of your Plugins SDK.



Note: On Windows, your Liferay Portal instance and your Plugins SDK must be on the same drive in order to build services. E.g., if your Liferay Portal instance is on your C :\ drive, your Plugins SDK must also be on your C :\ drive in order for Service Builder to be able to run successfully.

Using Liferay IDE or Developer Studio: From the Package Explorer, open the `service.xml` file from your `event-listing-portlet/docroot/WEB-INF` folder. By default, the file opens up in the Service Builder Editor. Make sure you are in Overview mode. Then click the *Build Services* button near the top-right corner of the view. The Build Services button has an image of a document with the numerical sequence *010* in front of it.

Make sure to click the *Build Services* button and not the *Build WSDD* button that appears next to it. Building the WSDDs won't hurt anything, but you'll generate files for the remote service instead of the local one. For information about WSDDs (web service deployment descriptors), please refer to the section on remote Liferay services later in this chapter.



After running Service Builder, the Plugins SDK prints messages listing the generated files and a message stating `BUILD SUCCESSFUL`. More information about these files appears below.

Using the terminal: Open a terminal window, navigate to your `portlets/event-listing-project-portlet` directory and enter this command:

```
ant build-service
```

When the service has been successfully generated, a `BUILD SUCCESSFUL` message appears in your terminal window. You should also see that a large number of files have been generated in your project. These files include a model layer, service layer, and persistence layer. Don't worry about the number of generated files--developers never have to customize more than three of them. We'll examine the files Service Builder generated for your Event entity after we add some custom service methods to `EventLocalServiceImpl` and call them from `EventPortlet`.

Now let's add some local service methods to `EventLocalServiceImpl` and learn how to call them. Later in this chapter, we'll add some remote service methods to `EventServiceImpl` and learn how to call those, too.

5.4. Writing Local Service Classes

The heart of your service is its `-LocalServiceImpl` class, where you put core business logic for working with your model. Throughout this chapter, you've been constructing services for the Nose-ster Event Listing example portlet project. Start with your services by examining the initial service classes Service Builder generated for it.

Note that Service Builder created an `EventLocalService` class which is the interface for the local service. It contains the signatures of every method in `EventLocalServiceBaseImpl` and `EventLocalServiceImpl`. `EventLocalServiceBaseImpl` contains a few automatically generated methods providing common functionality. Since the `EventLocalService` class is generated, you should never modify it. If you do, your changes

will be overwritten the next time you run Service Builder. Instead, all custom code should be placed in `EventLocalServiceImpl`.

Open the `EventLocalServiceImpl.java` file in your `/docroot/WEB-INF/src/com/nosester/portlet/eventlisting/service/impl/` folder.

Add the following database interaction methods to the `EventLocalServiceImpl` class:

```
public Event addEvent(
    long userId, long groupId, String name, String description,
    int month, int day, int year, int hour, int minute, long locationId,
    ServiceContext serviceContext)
throws PortalException, SystemException {

    User user = userPersistence.findByPrimaryKey(userId);

    Date now = new Date();

    long eventId = counterLocalService.increment(Event.class.getName());

    Event event = eventPersistence.create(eventId);

    event.setName(name);
    event.setDescription(description);

    Calendar dateCal = CalendarFactoryUtil.getCalendar(
        user.getTimeZone());
    dateCal.set(year, month, day, hour, minute);
    Date date = dateCal.getTime();
    event.setDate(date);

    event.setLocationId(locationId);

    event.setGroupId(groupId);
    event.setCompanyId(user.getCompanyId());
    event.setUserId(user.getUserId());
    event.setCreateDate(serviceContext.getCreateDate(now));
    event.setModifiedDate(serviceContext.getModifiedDate(now));

    super.addEvent(event);

    // Resources

    resourceLocalService.addResources(
        event.getCompanyId(), event.getGroupId(), event.getUserId(),
        Event.class.getName(), event.getEventId(), false,
        true, true);

    return event;
}

public Event deleteEvent(Event event) throws SystemException {

    return eventPersistence.remove(event);
```

```

}

public Event deleteEvent(long eventId)
    throws PortalException, SystemException {
    Event event = eventPersistence.findByPrimaryKey(eventId);
    return deleteEvent(event);
}

public Event getEvent(long eventId)
    throws SystemException, PortalException {
    return eventPersistence.findByPrimaryKey(eventId);
}

public List<Event> getEventsByGroupId(long groupId) throws SystemException {
    return eventPersistence.findByGroupId(groupId);
}

public List<Event> getEventsByGroupId(long groupId, int start, int end)
    throws SystemException {
    return eventPersistence.findByGroupId(groupId, start, end);
}

public int getEventsCountByGroupId(long groupId) throws SystemException {
    return eventPersistence.countByGroupId(groupId);
}

public Event updateEvent(
    long userId, long eventId, String name, String description,
    int month, int day, int year, int hour, int minute,
    long locationId, ServiceContext serviceContext)
    throws PortalException, SystemException {
    User user = userPersistence.findByPrimaryKey(userId);
    Date now = new Date();
    Event event = EventLocalServiceUtil.fetchEvent(eventId);
    event.setModifiedDate(serviceContext.getModifiedDate(now));
    event.setName(name);
    event.setDescription(description);
    Calendar dateCal = CalendarFactoryUtil.getCalendar(
        user.getTimeZone());
    dateCal.set(year, month, day, hour, minute);
    Date date = dateCal.getTime();
    event.setDate(date);
}

```

```

        event.setLocationId(locationId);

        super.updateEvent(event);

        return event;
    }
}

```

Remember to import the required classes. For your convenience, you can copy the following imports into your class:

```

import java.util.Calendar;
import java.util.Date;
import java.util.List;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.util.CalendarFactoryUtil;
import com.liferay.portal.model.User;
import com.liferay.portal.service.ServiceContext;

```

```

import com.nosester.portlet.eventlisting.model.Event;
import com.nosester.portlet.eventlisting.service.EventLocalServiceUtil;

```

In order to add an Event to the database, you need an ID for the Event. Liferay provides a counter service which you call to obtain a unique ID for each new Event entity. It's possible to use the `increment` method of Liferay's `CounterLocalServiceUtil` class but Service Builder already makes a `CounterLocalService` instance available to `EventLocalServiceBaseImpl` via Spring by dependency injection. Since your `EventLocalServiceImpl` class extends `EventLocalServiceBaseImpl`, you can access this `CounterLocalService` instance. See `EventLocalServiceBaseImpl` for a list of all the beans that Spring makes available for use. These include the following beans:

- `eventLocalService`
- `eventPersistence`
- `locationLocalService`
- `locationPersistence`
- `counterLocalService`
- `resourceLocalService`
- `resourceService`
- `resourcePersistence`
- `userLocalService`
- `userService`
- `userPersistence`

You can use either the injected class's `increment` method or you can call Liferay's `CounterLocalService`'s `increment` method directly.

```
long eventId = counterLocalService.increment(Event.class.getName());
```

We use the generated `eventId` as the ID for the new Event:

```
Event event = eventPersistence.create(eventId);
```

`eventPersistence` is one of the Spring beans injected into `EventLocalServiceBaseImpl` by Service Builder.

Next, we set the attribute fields that we specified for the Event. First, we set the name and

description of the Event. Then we use the date and time values to construct the Event's date. Lastly, we associate a location with the Event.

Then we assign values to the audit fields. First, we set the group, or scope, of the entity. In this case the group is the site. Then we set the company and user. The company represents the portal instance. We set the createDate and modifiedDate of our Event to the current time. After that, we call the generated addEvent method of EventLocalServiceBaseImpl with our Event. Lastly, we add the Event as a resource so that we can apply permissions to it later. We'll cover the details of adding resources in the Asset Framework section.

Since events require event locations, let's implement the local services for the Location entity, too. Open your LocationLocalServiceImpl.java file in your /docroot/WEB-INF/src/com/nosester/portlet/eventlisting/service/impl/ folder and add the following methods:

```
public Location addLocation(
    long userId, long groupId, String name, String description,
    String streetAddress, String city, String stateOrProvince,
    String country, ServiceContext serviceContext)
throws PortalException, SystemException {

    User user = userPersistence.findByPrimaryKey(userId);

    Date now = new Date();

    long locationId =
        counterLocalService.increment(Location.class.getName());

    Location location = locationPersistence.create(locationId);

    location.setName(name);
    location.setDescription(description);
    location.setStreetAddress(streetAddress);
    location.setCity(city);
    location.setStateOrProvince(stateOrProvince);
    location.setCountry(country);

    location.setGroupId(groupId);
    location.setCompanyId(user.getCompanyId());
    location.setUserId(user.getUserId());
    location.setCreateDate(serviceContext.getCreateDate(now));
    location.setModifiedDate(serviceContext.getModifiedDate(now));

    super.addLocation(location);

    return location;
}

public Location deleteLocation(Location location)
throws SystemException {

    return locationPersistence.remove(location);
}
```

```

public Location deleteLocation(long locationId)
    throws PortalException, SystemException {
    Location location = locationPersistence.fetchByPrimaryKey(locationId);
    return deleteLocation(location);
}

public List<Location> getLocationsByGroupId(long groupId)
    throws SystemException {
    return locationPersistence.findByGroupId(groupId);
}

public List<Location> getLocationsByGroupId(
    long groupId, int start, int end)
    throws SystemException {
    return locationPersistence.findByGroupId(groupId, start, end);
}

public int getLocationsCountByGroupId(long groupId) throws SystemException {
    return locationPersistence.countByGroupId(groupId);
}

public Location updateLocation(
    long userId, long locationId, String name, String description,
    String streetAddress, String city, String stateOrProvince,
    String country, ServiceContext serviceContext)
    throws PortalException, SystemException {
    User user = userPersistence.findByPrimaryKey(userId);
    Date now = new Date();
    Location location = locationPersistence.fetchByPrimaryKey(locationId);
    location.setName(name);
    location.setDescription(description);
    location.setStreetAddress(streetAddress);
    location.setCity(city);
    location.setStateOrProvince(stateOrProvince);
    location.setCountry(country);
    location.setModifiedDate(serviceContext.getModifiedDate(now));
    super.updateLocation(location);
    return location;
}

```

Make sure to add the following imports:

```
import java.util.Date;
```

```

import java.util.List;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.model.User;
import com.liferay.portal.service.ServiceContext;

import com.nosester.portlet.eventlisting.model.Location;
import
com.nosester.portlet.eventlisting.service.base.LocationLocalServiceBaseImpl;
Your local service implementations for events and locations are ready for action.

```

Before you can use any custom methods that you added to the `EventLocalServiceImpl` and `LocationLocalServiceImpl` classes, you must add their signatures to the `EventLocalService` and `LocationLocalService` interfaces by running Service Builder again.

Using Developer Studio: As we did before, open your `service.xml` file and make sure you are in the *Overview* mode. Then, select *Build Services*.

Using the terminal: Navigate to the root directory of your portlet in the terminal and run:

```
ant build-service
```

Service Builder looks through `EventLocalServiceImpl` and automatically copies the signatures of each method into the interface. You can now add a new Event to the database by making the following call:

```
EventLocalServiceUtil.addEvent(event);
```

Service Builder generates the `addEvent` method in the `EventLocalServiceUtil` utility class. In addition to all the Java classes, Service Builder also generates a `service.properties` file which we'll discuss later. Next, let's call our newly implemented local service.

5.5. Calling Local Services

Once Service Builder has generated your portlet project's services, you can call them in your project's -Portlet classes. You can call any methods in your `EventLocalServiceUtil` or `LocationLocalServiceUtil` static utility classes from `EventListingPortlet` and `LocationListingPortlet`. For example, you want the Event Listing Portlet to perform create, read, update, and delete (CRUD) operations on Events and the Location Listing Portlet to perform CRUD operations on Locations. To this end, you'll create the following methods for `EventListingPortlet` and similar ones for `LocationListingPortlet`:

- `addEvent`
- `updateEvent`
- `deleteEvent`

Create file `EventListingPortlet.java` in your `docroot/WEB-INF/src/com/nosester/portlet/eventlisting` folder, if it doesn't already exist. Replace the contents of `EventListingPortlet.java` with the following code:

```
package com.nosester.portlet.eventlisting;
```

```

import java.util.Calendar;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.log.Log;
import com.liferay.portal.kernel.log.LogFactoryUtil;
import com.liferay.portal.kernel.util.ParamUtil;
import com.liferay.portal.service.ServiceContext;
import com.liferay.portal.service.ServiceContextFactory;
import com.liferay.util.bridges.mvc.MVCPortlet;
import com.nosester.portlet.eventlisting.model.Event;
import com.nosester.portlet.eventlisting.service.EventLocalServiceUtil;

public class EventListingPortlet extends MVCPortlet {

    public void addEvent(ActionRequest request, ActionResponse response)
        throws Exception {

        _updateEvent(request);

        sendRedirect(request, response);
    }

    public void deleteEvent(ActionRequest request, ActionResponse response)
        throws Exception {

        long eventId = ParamUtil.getLong(request, "eventId");

        EventLocalServiceUtil.deleteEvent(eventId);

        sendRedirect(request, response);
    }

    public void updateEvent(ActionRequest request, ActionResponse response)
        throws Exception {

        _updateEvent(request);

        sendRedirect(request, response);
    }

    private Event _updateEvent(ActionRequest request)
        throws PortalException, SystemException {

        long eventId = ParamUtil.getLong(request, "eventId");
        String name = ParamUtil.getString(request, "name");
        String description = ParamUtil.getString(request, "description");
        long locationId = ParamUtil.getLong(request, "locationId");

        int year = ParamUtil.getInteger(request, "dateYear");
        int month = ParamUtil.getInteger(request, "dateMonth");
    }
}

```

```

        int day = ParamUtil.getInteger(request, "dateDay");
        int hour = ParamUtil.getInteger(request, "dateHour");
        int minute = ParamUtil.getInteger(request, "dateMinute");
        int amPm = ParamUtil.getInteger(request, "dateAmPm");

        if (amPm == Calendar.PM) {
            hour += 12;
        }

        ServiceContext serviceContext = ServiceContextFactory.getInstance(
            Event.class.getName(), request);

        Event event = null;

        if (eventId <= 0) {
            event = EventLocalServiceUtil.addEvent(
                serviceContext.getUserId(), serviceContext.getScopeGroupId(),
                name, description, month, year, hour, minute,
                locationId,
                serviceContext);
        }
        else {
            event = EventLocalServiceUtil.getEvent(eventId);

            event = EventLocalServiceUtil.updateEvent(
                serviceContext.getUserId(), eventId, name, description,
                month,
                day, year, hour, minute, locationId, serviceContext);
        }

        return event;
    }

    private static Log _log =
LogFactoryUtil.getLog(EventListingPortlet.class);
}

```

The Event Listing Portlet's addEvent, updateEvent, and deleteEvent methods now call the appropriate methods from EventLocalServiceUtil. Liferay's ParamUtil getter methods such as getLong and getString return default values like 0 or "" if the specified request parameter is not available from the portlet request. When adding a new event, for example, no event ID is available so ParamUtil.getLong("request", "eventId") returns 0. The Event portlet's addEvent method calls EventLocalServiceUtil's addEvent method. The event ID for the new event is generated at the service layer in the addEvent method that you added to the EventLocalServiceImpl class. The EventLocalServiceUtil generated for us by Service Builder contains various CRUD methods:

- createEvent
- addEvent
- deleteEvent
- updateEvent

- `fetchEvent`
- `getEvent`

The methods listed in the figure below are all generated by Service Builder and can be called by `EventListingPortlet`.

```
▲ G EventLocalServiceUtil
  □ S _service : EventLocalService
  △ S addEvent(Event) : Event
  △ S addEvent(long, long, String, String, int, int, int, int, int, long, ServiceContext) : Event
  △ S clearService() : void
  △ S createEvent(long) : Event
  △ S deleteEvent(Event) : Event
  △ S deleteEvent(long) : Event
  △ S dynamicQuery() : DynamicQuery
  △ S dynamicQuery(DynamicQuery) : List
  △ S dynamicQuery(DynamicQuery, int, int) : List
  △ S dynamicQuery(DynamicQuery, int, int, OrderByComparator) : List
  △ S dynamicQueryCount(DynamicQuery) : long
  △ S fetchEvent(long) : Event
  △ S getBeanIdentifier() : String
  △ S getEvent(long) : Event
  △ S getEvents(int, int) : List<Event>
  △ S getEventsById(long) : List<Event>
  △ S getEventsById(long, int, int) : List<Event>
  △ S getEventsCount() : int
  △ S getEventsCountById(long) : int
  △ S getPersistedModel(Serializable) : PersistedModel
  △ S getService() : EventLocalService
  △ S invokeMethod(String, String[], Object[]) : Object
  △ S setBeanIdentifier(String) : void
  △ S updateEvent(Event) : Event
  △ S updateEvent(Event, boolean) : Event
  △ S updateEvent(long, long, String, String, int, int, int, int, int, long, ServiceContext) : Event
```

Portlet classes should have access only to the – `LocalServ iceUtil` classes. The – `LocalServ iceUtil` classes, in turn, call their injected – `LocalServ iceImpl` classes. Notice in the figure above that the `EventLoca lServiceU til` utility class has a private instance variable called `_service`.

The `_service` instance variable of type `EventLocalService` gets an instance of `EventLocalServiceImpl` at runtime via dependency injection. So all the methods of the `EventLocalServiceUtil` utility class internally call corresponding methods of the `EventLocalServiceImpl` class at runtime.

Let's implement the `LocationListingPortlet` class with methods similar to the ones we implemented in the `EventListingPortlet` class. Create file `LocationListingPortlet.java` in your `docroot/WEB-INF/src/com/nosester/portlet/eventlisting` folder, if it doesn't already exist. Open your `LocationListingPortlet.java` file and replace its contents with the following code:

```

package com.nosester.portlet.eventlisting;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.log.Log;
import com.liferay.portal.kernel.log.LogFactoryUtil;
import com.liferay.portal.kernel.util.ParamUtil;
import com.liferay.portal.service.ServiceContext;
import com.liferay.portal.service.ServiceContextFactory;
import com.liferay.util.bridges.mvc.MVCPortlet;
import com.nosester.portlet.eventlisting.model.Location;
import com.nosester.portlet.eventlisting.service.LocationLocalServiceUtil;

public class LocationListingPortlet extends MVCPortlet {

    public void addLocation(ActionRequest request, ActionResponse response)
        throws Exception {

        _updateLocation(request);

        sendRedirect(request, response);
    }

    public void deleteLocation(ActionRequest request, ActionResponse response)
        throws Exception {

        long locationId = ParamUtil.getLong(request, "locationId");

        LocationLocalServiceUtil.deleteLocation(locationId);

        sendRedirect(request, response);
    }

    public void updateLocation(ActionRequest request, ActionResponse response)
        throws Exception {

        _updateLocation(request);

        sendRedirect(request, response);
    }

    private Location _updateLocation(ActionRequest request)
        throws PortalException, SystemException {

        long locationId = (ParamUtil.getLong(request, "locationId"));
        String name = (ParamUtil.getString(request, "name"));
        String description = (ParamUtil.getString(request, "description"));
        String streetAddress = (ParamUtil.getString(request,
            "streetAddress"));
    }
}

```

```

        String city = (ParamUtil.getString(request, "city"));
        String stateOrProvince = (ParamUtil.getString(request,
    "stateOrProvince"));
        String country = (ParamUtil.getString(request, "country"));

        ServiceContext serviceContext = ServiceContextFactory.getInstance(
            Location.class.getName(), request);

        Location location = null;

        if (locationId <= 0) {

            location = LocationLocalServiceUtil.addLocation(
                serviceContext.getUserId(), serviceContext.getScopeGroupId(),
name, description,
                streetAddress, city, stateOrProvince, country,
serviceContext);
        }
        else {
            location = LocationLocalServiceUtil.getLocation(locationId);

            location = LocationLocalServiceUtil.updateLocation(
                serviceContext.getUserId(), locationId, name,
                description, streetAddress, city, stateOrProvince,
country,
                serviceContext);
        }

        return location;
    }

    private static Log _log =
LogFactoryUtil.getLog(LocationListingPortlet.class);
}

```

We've demonstrated how to call the local services generated by Service Builder in our project's - Portlet classes. Next, let's learn how to how to call Liferay's local services.

5.6. *Understanding the Service Builder-generated Code*

Now let's examine the files Service Builder generated for your Event entity. Note that the files listed under Local Service and Remote Service below are only generated for an entity that has both `local-service` and `remote-service` attributes set to `true`. Service Builder generates services for these entities in two locations in your project. These locations use the package path that you specified in your `service.xml` file:

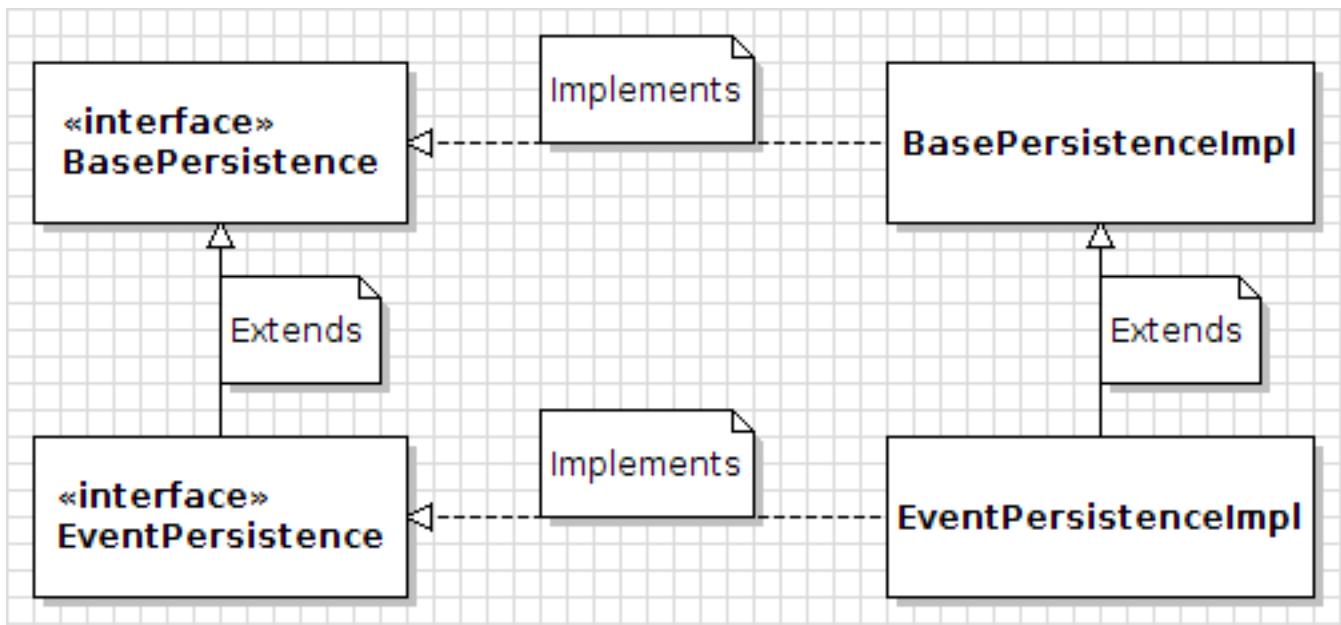
- `docroot/WEB-INF/service/com/nosester/portlet/eventlisting`
- `docroot/WEB-INF/src/com/nosester/portlet/eventlisting`

The `docroot/WEB-INF/service/com/nosester/portlet/eventlisting/` package contains utility classes and interfaces for the Event Listing project. All the classes and interfaces in the service folder are packaged in a `.jar` file called `event-listing-`

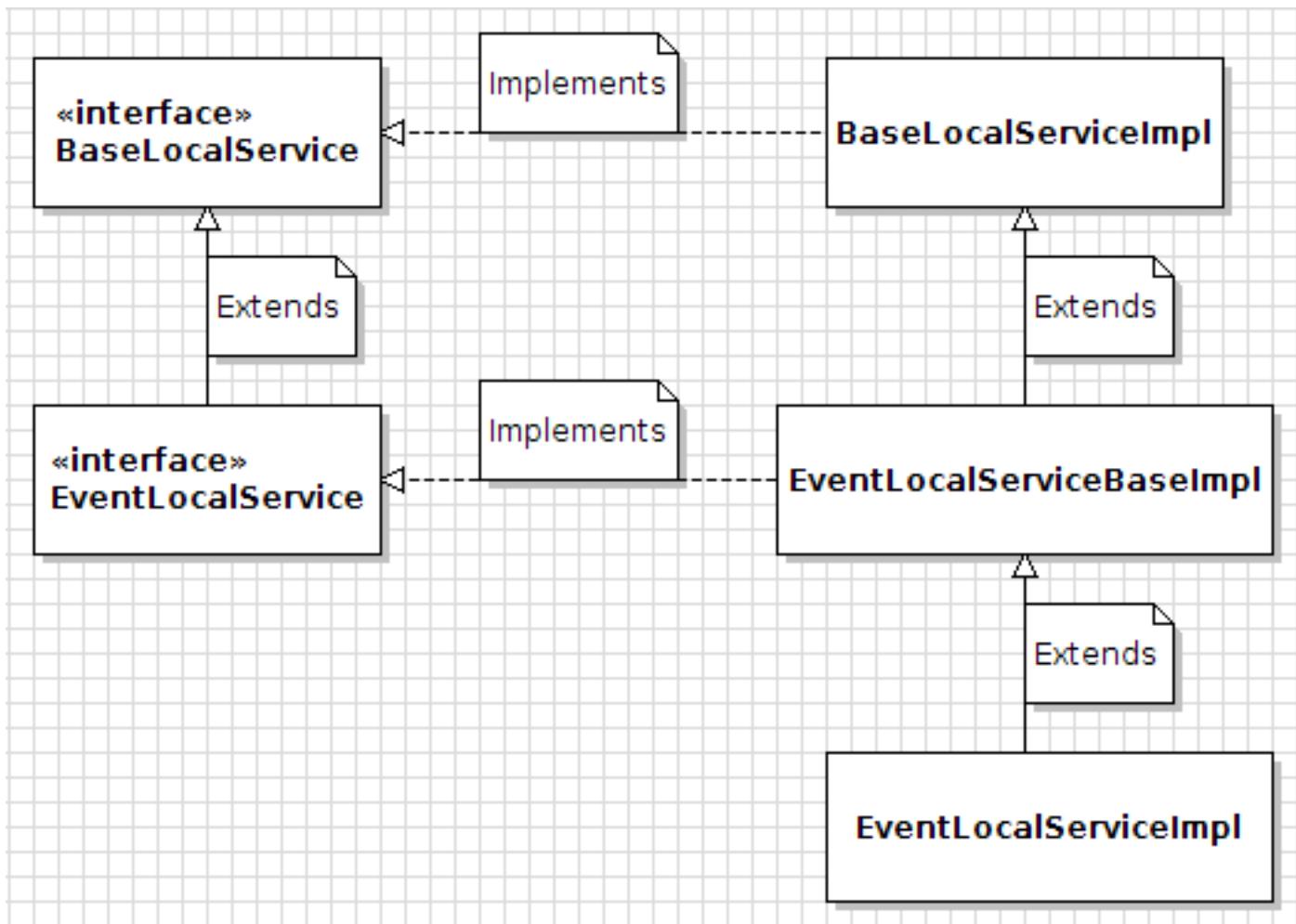
`project-portlet-service.jar`, in your `docroot/WEB-INF/lib` folder. This `.jar` file is generated whenever you run Service Builder. It's possible to place this `.jar` file on your application server's global classpath to make your project's services available to other projects. This allows a portlet in different project, for example, to create, update, and delete Events and Locations. Of course, you should consider the security implications of placing your project's service `.jar` file on your application server's global classpath: do you *really* want to allow other plugins to access your project's services?

The `docroot/WEB-INF/src/com/nosester/portlet/eventlisting` package contains the implementation of the interfaces defined in the `docroot/WEB-INF/service/com/nosester/portlet/eventlisting` package. It belongs to the Event Listing project's classpath but is not available outside the Event Listing project. Service Builder generates classes and interfaces belonging to the persistence layer, service layer, and model layer in the `docroot/WEB-INF/service/com/nosester/portlet/eventlisting` and `docroot/WEB-INF/src/com/nosester/portlet/eventlisting` packages. Let's look at the classes and interfaces generated for Events. The ones generated for Locations are similar. You won't have to customize more than three of these classes for each entity: `-LocalServiceImpl`, `-ServiceImpl`, and `-ModelImpl`.

- Persistence
 - `EventPersistence`: Event persistence interface that defines CRUD methods for the Event entity such as `create`, `remove`, `countAll`, `find`, `findAll`, etc.
 - `EventPersistenceImpl`: Event persistence implementation class that implements `EventPersistence`.
 - `EventUtil`: Event persistence utility class that wraps `EventPersistenceImpl` 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 `EventLocalServiceUtil` or `EventServiceUtil` instead.

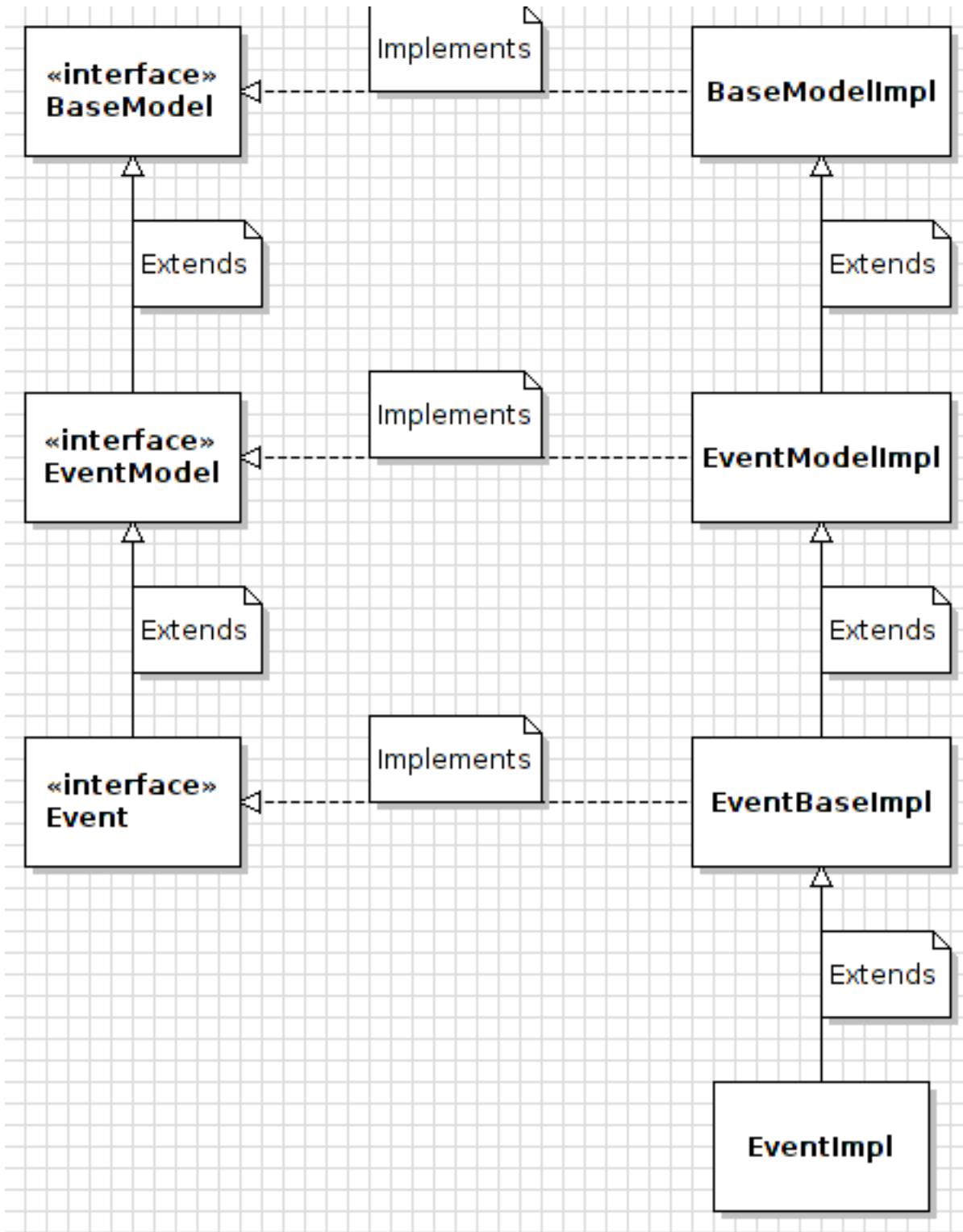


- Local Service (only generated for an entity if an entity's `local-service` attribute is set to `true` in `service.xml`)
 - `EventLocalService`: Event local service interface.
 - `EventLocalServiceImpl` (**LOCAL SERVICE IMPLEMENTATION**): Event local service implementation. This is the only class in the local service that you should modify manually. You can add custom business logic here. For any custom methods added here, Service Builder adds corresponding methods to the `EventLocalService` interface the next time you run it.
 - `EventLocalServiceBaseImpl`: Event 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`
 - `EventLocalServiceUtil`: Event local service utility class which wraps `EventLocalServiceImpl` and serves as the primary local access point to the service layer.
 - `EventLocalServiceWrapper`: Event local service wrapper which implements `EventLocalService`. This class is designed to be extended and it allows developers to customize the local Event services. Customizing services should be done via a hook plugin.



- Remote Service (only generated for an entity if an entity's `remote-service` attribute is set to `true` in `service.xml`)
 - `EventService`: Event remote service interface.
 - `EventServiceImpl` (**REMOTE SERVICE IMPLEMENTATION**): Event 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 `EventService` interface the next time you run it.
 - `EventServiceBaseImpl`: Event remote service base implementation. This is an abstract class. `@abstract`
 - `EventServiceUtil`: Event remote service utility class which wraps `EventServiceImpl` and serves as the primary remote access point to the service layer.
 - `EventServiceWrapper`: Event remote service wrapper which implements

- `EventService`. This class is designed to be extended and it allows developers to customize the remote Event services. Customizing services should be done in a hook plugin. `EventServiceImpl`
- `EventServiceSoap`: Event SOAP utility which the remote `EventServiceUtil` remote service utility can access. `EventServiceUtil`
 - `EventSoap`: Event SOAP model, similar to `EventModelImpl`. `EventSoap` is serializable; it does not implement `Event`.
 - Model
 - `EventModel`: Event base model interface. This interface and its `EventModelImpl` implementation serve only as a container for the default property accessors generated by Service Builder. Any helper methods and all application logic should be added to `EventImpl`.
 - `EventModelImpl`: Event base model implementation.
 - `Event`: Event model interface which extends `EventModel`.
 - `EventImpl`: (**MODEL IMPLEMENTATION**)Event 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 `Event` interface the next time you run it.
 - `EventWrapper`: Event wrapper, wraps `Event`.



Each file that Service Builder generates is assembled from an associated Freemarker template. You can find Service Builder's Freemarker templates in the

com.liferay.portal.tools.servicebuilder.dependencies package of Liferay's portal-impl/src folder. For example, if you want to find out how a `-ServiceImpl.java` file is generated, just look at the `service_impl.ftl` template.

Of all the classes generated by Service Builder, only three should be manually modified: `EventLocalServiceImpl`, `EventServiceImpl` and `EventImpl`. If you manually modify the other classes, your changes will be overwritten the next time you run Service Builder. Now that we can access the location and event entities with the local services, let's build our UI.

5.7. ***Creating User Interfaces for Service Builder Portlets***

We'll build a UI for the Location Listing Portlet that we've been developing in the Event Listing example project. The Location Listing Portlet's location entity is easy to work with because it has no dependencies on other entities (unlike the event entity, which depends on the location entity). We'll build a UI that allows us to do the following activities with the Location Listing Portlet:

- List the current locations
- Add new locations
- Edit locations
- Delete locations

It's always nice to see our current entities. So let's build a view that lists the entities. The figure

Name	Description	Street Address	City	State/Province	Country	
Hill Country	Bring your lunar rover.					
Sea of Tranquility	A great place to kick back and relax.	101 Delsea Drive	Seaside			

below shows what we want the Location Listing Portlet to look like, filled with locations:

Since this part of our UI will give us a "view" of our location entity instances, we'll implement it in a JSP file named `view.jsp` that we'll keep in folder

docroot/html/locationlisting. If you don't have this folder or the view.jsp file in it, create them. Then, open the view.jsp file and replace its contents with the following JSP code:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>

<%@ page import="com.liferay.portal.util.PortalUtil" %>
<%@ page import="com.nosester.portlet.eventlisting.model.Location" %>
<%@ page import="com.nosester.portlet.eventlisting.service.LocationLocalServiceUtil" %>

<liferay-theme:defineObjects />
<portlet:defineObjects />

This is the <b>Location Listing Portlet</b> in View mode.

<liferay-ui:search-container emptyResultsMessage="There are no locations to
display">
    <liferay-ui:search-container-results
        results="<%=
LocationLocalServiceUtil.getLocationsByGroupId(scopeGroupId,
searchContainer.getStart(), searchContainer.getEnd()) %>"%
        total="<%=
LocationLocalServiceUtil.getLocationsCountByGroupId(scopeGroupId) %>"%
    />

    <liferay-ui:search-container-row
        className="com.nosester.portlet.eventlisting.model.Location"
        keyProperty="locationId"
        modelVar="location" escapedModel="<%=
        true %>"%
    >
        <liferay-ui:search-container-column-text
            name="name"
            value="<%=
            location.getName() %>"%
        />

        <liferay-ui:search-container-column-text
            name="description"
            value="<%=
            location.getDescription() %>"%
        />

        <liferay-ui:search-container-column-text
            name="streetAddress"
            value="<%=
            location.getStreetAddress() %>"%
        />

        <liferay-ui:search-container-column-text
            name="city"
```

```

        value="<%=\ location.getCity() %>"
```

```

    />
```

```

<liferay-ui:search-container-column-text
    name="stateOrProvince"
    value="<%=\ location.getStateOrProvince() %>"
```

```

    />
```

```

<liferay-ui:search-container-column-text
    name="country"
    value="<%=\ location.getCountry() %>"
```

```

    />
</liferay-ui:search-container-row>
```

```

<liferay-ui:search-iterator />
```

```

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

Let's look at this code in detail, starting with the JSP directives. The taglib directives tell your JSP where to find tags you're using and what namespace prefix to use for them. For example, the `<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>` directive specifies the location of Liferay's theme tag library descriptor (TLD) and the namespace prefix to use with its tags. In our JSP, we've also specified directives for the Portlet 2.0 and Liferay UI taglibs. Lastly, we specified directives to import the `PortalUtil`, `Location`, and `LocationLocalServiceUtil` classes, as we're using in the JSP.

Below the page import directives, we have a couple interesting tags: `<liferay-theme:defineObjects />` and `<portlet:defineObjects />`. These tags give our JSP access to various variables from the context of the request objects and the portal's theme.

After outputting text that identifies the portlet, we get into the real "meat" of our JSP. We use Liferay's `<liferay-ui:search-container>` tag to return location entities from the database and list them in a table. The search container calls `LocationLocalServiceUtil` to get the locations. It then renders one instance of the `com.nosester.portlet.eventlisting.model.Location` class per table row. Each location is identified by its `locationId` and each of a location's attributes are rendered in columns via the `<liferay-ui:search-container-column-text>` tags.

Location Listing Portlet

This is the Location Listing Portlet in View mode.

There are no locations to display

Redeploy the portlet to catch a glimpse of the current locations.

We'll need to create a JSP that enables us to add locations. Let's name this JSP `edit_location.jsp`, as we'll use it for modifying locations, as well as adding them. In the `docroot/html/locationlisting` folder, create the `edit_location.jsp` file. Copy the following code into it:

```
<%@ 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/ui" prefix="liferay-ui" %>

<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>

<%@ page import="com.nosester.portlet.eventlisting.model.Location" %>
<%@ page import="com.nosester.portlet.eventlisting.service.LocationLocalServiceUtil" %>

<%
    Location location = null;

    long locationId = ParamUtil.getLong(request, "locationId");

    if (locationId > 0) {
        location = LocationLocalServiceUtil.getLocation(locationId);
    }

    String redirect = ParamUtil.getString(request, "redirect");
%>

<aui:model-context bean="<%= location %>" model="<%= Location.class %>" />
<portlet:renderURL var="viewLocationURL" />
<portlet:actionURL name='<%= location == null ? "addLocation" : "updateLocation" %>' var="editLocationURL" windowState="normal" />

<liferay-ui:header
    backURL="<%= viewLocationURL %>"
    title='<%= (location != null) ? location.getName() : "New Location" %>' />

<aui:form action="<%= editLocationURL %>" method="POST" name="fm">
    <aui:fieldset>
        <aui:input name="redirect" type="hidden" value="<%= redirect %>" />

        <aui:input name="locationId" type="hidden" value='<%= location == null ? "" : location.getLocationId() %>' />

        <aui:input name="name" />

        <aui:input name="description" />

        <aui:input name="streetAddress" />

        <aui:input name="city" />
```

```

<aui:input name="stateOrProvince" />

<aui:input name="country" />

</aui:fieldset>

<aui:button-row>
    <aui:button type="submit" />

    <aui:button onClick="<% viewLocationURL %>" type="cancel" />
</aui:button-row>
</aui:form>

```

As with the `view.jsp`, we specify directives to access taglibs and import classes that we're using in our JSP code.

Then we check for a location ID in the request. If a location ID is present, we'll edit that location. Otherwise, we'll create a new location to populate. Using the `<portlet:actionURL>` tag, we direct the portlet to edit an existing location or add a new location. The action uses the content from our JSP's form to modify the location. Lastly, before we go into the form code, we print a header for our portlet, either displaying the name of the existing location or displaying text to indicate that we're populating a new location's fields.

We use AlloyUI's `<aui:form>` tag to present a form for the user to fill in for the location and submit. It presents various location fields via `<aui:input>` tags. Each one uses a name that the `LocationListingPortlet` class references. Lastly, we include a button for users to submit the form to the portlet. The attributes specified for the button redirect control to the `view.jsp` via the `viewLocationURL` variable.

Now that we've implemented the `edit_location.jsp`, we must provide a way for users to get to it. Let's add a button to the `view.jsp`, that redirects control to the `edit_location.jsp`:

1. Open your location's `view.jsp` file and add the following code above the line that starts with the `<liferay-ui:search-container>` tag:

```

<%
    String redirect = PortalUtil.getCurrentURL(renderRequest);
%>

<aui:button-row>
    <portlet:renderURL var="addLocationURL">
        <portlet:param name="mvcPath"
value="/html/locationlisting/edit_location.jsp" />
        <portlet:param name="redirect" value="<% redirect %>" />
    </portlet:renderURL>

    <aui:button onClick="<% addLocationURL.toString() %>" value="add-
location" />
</aui:button-row>

```

2. Add the following directive to the top of the `view.jsp`, to import AlloyUI's `aui` taglib:

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

This code adds a button that redirects the user to the `edit_location.jsp` via the MVC path value `/html/locationlisting/edit_location.jsp`. It also includes the current URL, the URL of the `view.jsp`, with the request so that the `edit_location.jsp` can redirect the user back to the view after the user adds the new location.

Now that we've implemented a JSP for adding/editing locations and we've provided a button for users to click on to access that JSP, we must redeploy the portlet to update the portal with our portlet's changes.

Redeploy the portlet and add a new location.



New Location

Name

Description

Street Address

City

state-or-province

Country

Save

Cancel

After you've added a location or two, you can see how nicely each location is displayed in the

Name	Description	Street Address	City	State/Province	Country
Hill Country	Bring your lunar rover.				
Sea of Tranquility	A great place to kick back and relax.	101 Delsea Drive	Seaside		

view.jsp.

As we've mentioned before, we not only want to add new locations, but we also want to edit *existing* locations. In addition, we want to be able to delete locations. Let's provide a way for users to edit and delete individual locations. In the view, we'll display a dropdown button that lets a user edit or delete the applicable location. We'll create a JSP to handle each action request and send each request to the portlet.

Let's start by creating an action JSP called `location_actions.jsp`. Create a file called `location_actions.jsp` in the `/html/locationlisting` folder. Then copy the following contents into that JSP:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>
<%@ page import="com.liferay.portal.kernel.dao.search.ResultRow" %>
<%@ page import="com.liferay.portal.kernel.util.WebKeys" %>
<%@ page import="com.liferay.portal.util.PortalUtil" %>
<%@ page import="com.nosester.portlet.eventlisting.model.Location" %>
<portlet:defineObjects />
<%
    ResultRow row = (ResultRow) request
        .getAttribute(WebKeys.SEARCH_CONTAINER_RESULT_ROW);
```

```

        Location location = (Location) row.getObject();

        long groupId = location.getGroupId();
        String name = Location.class.getName();
        long locationId = location.getLocationId();

        String redirect = PortalUtil.getCurrentURL(renderRequest);
%>

<liferay-ui:icon-menu>
    <portlet:renderURL var="editURL">
        <portlet:param name="mvcPath"
value="/html/locationlisting/edit_location.jsp" />
        <portlet:param name="locationId" value="<%=
String.valueOf(locationId) %>" />
        <portlet:param name="redirect" value="<%= redirect %>" />
    </portlet:renderURL>

    <liferay-ui:icon image="edit" url="<%=
editURL.toString() %>" />

    <portlet:actionURL name="deleteLocation" var="deleteURL">
        <portlet:param name="locationId" value="<%=
String.valueOf(locationId) %>" />
        <portlet:param name="redirect" value="<%= redirect %>" />
    </portlet:actionURL>

    <liferay-ui:icon-delete url="<%=
deleteURL.toString() %>" />
</liferay-ui:icon-menu>

```

The above code from the `location_actions.jsp` extracts the location information from the request object. Then it provides an icon for editing the location and an icon for deleting the location. The user is redirected to the `edit_location.jsp` if the user clicks on the edit icon. If the user clicks on the delete icon, a request is sent to the portlet to delete the location and the user is redirected to the `view.jsp`. Now that the `location_actions.jsp` is implemented, let's link it to the `view.jsp`.

We'll fit the `view.jsp` with a container to render the edit and delete icons we implemented in the `location_actions.jsp`. To do so, add the following element in the `view.jsp`, just before the closing `</liferay-ui:search-container-row>` tag:

```

<liferay-ui:search-container-column-jsp
    align="right"
    path="/html/locationlisting/location_actions.jsp"
/>

```

The `<liferay-ui:search-container-column-jsp>` tag displays the *Actions* button, which presents the user with the action icons implemented in the `location_actions.jsp`.

Now that you have your *Actions* button in place, redeploy your portlet and refresh its view in your browser. Notice the fancy *Actions* button for each location. Try editing a location and deleting a location.

Name	Description	Street Address	City	State/Province	Country	
Hill Country	Bring your lunar rover.					
Sea of Tranquility	A great place to kick back and relax.	101 Delsea Drive	Seaside			<div style="background-color: #0072BC; color: white; padding: 5px 10px;"> Edit </div> <div style="background-color: white; border: 1px solid #ccc; padding: 5px 10px; margin-top: 5px;"> X Delete </div>

Great job creating the Location Listing Portlet's UI!

In case you're interested, the finished event-listing-portlet example project, including the UIs of the Location Listing Portlet and Event Listing Portlet, is available in the Dev Guide SDK in the SDK's portlets/event-listing-portlet folder.

Next, let's learn how to call core Liferay services. Calling core Liferay services in your portlet is just as easy as calling your project's services you generated via Service Builder.

5.8. *Calling Liferay Services*

Every service provides a local interface to clients running in the same JVM as Liferay Portal. These are called by use of the -ServiceUtil classes. These classes mask the complexity of service implementations. The core Liferay services that are provided as part of Liferay Portal were generated by the same Service Builder tool that we used in our project. Let's invoke a Liferay service using its -ServiceUtil class. The following JSP code snippet demonstrates how to get a list of the most recent bloggers from an organization.

```
<%@ page
import="com.liferay.portlet.blogs.service.BlogsStatsUserLocalServiceUtil" %>
<%@ page
import="com.liferay.portlet.blogs.util.comparator.StatsUserLastPostDateComparator" %>
...
<%@
```

```
List statsUsers = BlogsStatsUserLocalServiceUtil.getOrganizationStatsUsers(  
    organizationId, 0, max, new StatsUserLastPostDateComparator());  
%>
```

This JSP code invokes the static method `getOrganizationStatsUsers()` from the `-LocalServiceUtil` class `BlogsStatsUserLocalServiceUtil`.

In addition to the services you create using Service Builder, your portlets can also access a variety of services built into Liferay. These include the following services:

- `UserService` - for accessing, adding, authenticating, deleting, and updating users.
- `OrganizationService` - for accessing, adding, deleting, and updating organizations.
- `GroupService` - for accessing, adding, deleting, and updating groups.
- `CompanyService` - for accessing, adding, checking, and updating companies.
- `ImageService` - for accessing images.
- `LayoutService` - for accessing, adding, deleting, exporting, importing, and updating layouts.
- `PermissionService` - for checking permissions.
- `UserGroupService` - for accessing, adding, deleting, and updating user groups.
- `RoleService` - for accessing, adding, unassigning, checking, deleting, and updating roles.

For more information on these services, see the Liferay Portal CE Javadocs at <http://docs.liferay.com/portal/6.2/javadocs/> or the Liferay Portal EE Javadocs included in the Liferay Portal EE Documentation .zip file that you can download from the Customer Portal on <http://www.liferay.com>.

Next, you'll learn how to give Liferay portal instructions, or *hints*, for presenting your entity models in your portlet's view.

5.9. Using Model Hints

Now that you've created your model entities and implemented your business logic to create and modify those entities, you probably have some ideas for helping users input valid model entity data. For example, in the Event Listing project you've been working on throughout this chapter, you want users to create social events for the future, not for the past. And it would be nice to give users a nice text editor to fill in their descriptions. Wouldn't it be great to specify these customizations from a single place in your portal project? Good news! Service Builder lets you specify this information as *model hints* in a single file called `portlet-model-hints.xml` in your project's `docroot/WEB-INF/src/META-INF` folder. Liferay calls them *model hints* because they suggest how entities should be presented to users and can also specify the size of database columns used to store the entities.

Model hints let you to configure how the AlloyUI tag library, `aui`, shows model fields. As Liferay Portal displays form fields in your application, it first checks the model hints you specified and customizes the form's input fields based on these hints.

Let's look at the model hints file that Service Builder generated for the Event Listing Portlet. Examine your project's `docroot/WEB-INF/src/META-INF/portlet-model-`

hints.xml file. If you've been following along in the previous sections, Service Builder created the portlet-model-hints.xml file with the following contents:

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

<model-hints>
    <model name="com.nosester.portlet.eventlisting.model.Event">
        <field name="eventId" type="long" />
        <field name="companyId" type="long" />
        <field name="groupId" type="long" />
        <field name="userId" type="long" />
        <field name="createDate" type="Date" />
        <field name="modifiedDate" type="Date" />
        <field name="name" type="String" />
        <field name="description" type="String" />
        <field name="date" type="Date" />
        <field name="locationId" type="long" />
    </model>
    <model name="com.nosester.portlet.eventlisting.model.Location">
        <field name="locationId" type="long" />
        <field name="companyId" type="long" />
        <field name="groupId" type="long" />
        <field name="userId" type="long" />
        <field name="createDate" type="Date" />
        <field name="modifiedDate" type="Date" />
        <field name="name" type="String" />
        <field name="description" type="String" />
        <field name="streetAddress" type="String" />
        <field name="city" type="String" />
        <field name="stateOrProvince" type="String" />
        <field name="country" type="String" />
    </model>
</model-hints>
```

The root-level element is model-hints. All your model entities are represented by model sub-elements of the model-hints element. Each model element must have a name attribute specifying the fully-qualified model class name. Each model has field elements representing its model entity's columns. Lastly, each field element must have a name and a type. Each field element's names and types correspond to the names and types specified for each entity's columns in your project's service.xml file. Service Builder generates all these elements for you, based on service.xml file.

To add hints to a field, add a hint tag inside its field tag. For example, you can add a display-width hint to specify the pixel width that should be used when displaying the field. The default pixel width is 350. To show a String field with 50 pixels, we could nest a hint element named display-width and give it a value of 50 for 50 pixels. Here's an example of using the display-width hint in a field element:

```
<field name="name" type="String">
    <hint name="display-width">50</hint>
</field>
```

In order to see the effect of a hint on a field, you must run Service Builder again and redeploy

your portlet project. Changing the `display-width` doesn't actually limit the number of characters that can be entered into the `name` field; it's just a way to control the width of the field 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 wanted the `name` field to persist up to 100 characters, you'd 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 portlet project after updating the `portlet-model-hints.xml` file.

So, we've mentioned a few different hints. It's about time we listed the portlet hints available to you. The following table describes the portlet model hints.

Model Hint Values and Descriptions

Name	Type	Description	Default
auto-escape	boolean	sets whether text values should be escaped via <code>HtmlUtil.escape</code>	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 for a field	(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 hide the characters input by the user	false
show-time	boolean	sets whether to show include time along with the date	true
upper-case	boolean	converts all characters to upper case	false
year-nullable	boolean	allows the year to be null in a date field	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

Name	Type	Description	Default
past			

Liferay Portal has its own model hints XML configuration file called `portal-model-hints.xml` which is in Liferay's `portal-impl/classes/META-INF` folder. Liferay's model hints configuration file contains many hint examples, so you can reference it when customizing your `portlet-model-hints.xml` file.

You can use the `default-hints` element to define a list of hints to be applied to every field of a model. For example, adding the following element inside a model element applies a `display-width` of 300 to each field:

```
<default-hints>
  <hint name="display-width">300</hint>
</default-hints>
```

You can define `hint-collection` elements inside the `model-hints` root-level element to define a list of hints to be applied 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="URL">
  <hint name="max-length">4000</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="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>
```

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 Event model's date field by using a `hint-collection` element that references the collection by its name:

```
<field name="date" type="Date">
  <hint-collection name="SEARCHABLE-DATE" />
</field>
```

As always, remember to run Service Builder and redeploy your project after updating your `portlet-model-hints.xml` file.

Now you can use a couple of model hints in the Event Listing Portlet and Location Listing Portlet. Start by giving users an editor for filling in their description fields. Since you want to apply the same hint to both the event and location entities, define it as a hint collection. Then you can reference the hint collection in them.

Define the following hint collection just below the `model-hints` root element in the `portlet-model-hints.xml` file:

```
<hint-collection name="DESCRIPTION-TEXTAREA">
    <hint name="display-height">105</hint>
    <hint name="display-width">500</hint>
    <hint name="max-length">4000</hint>
</hint-collection>
```

Then replace the event and location description fields' entities with a reference to the hint collection, as demonstrated below:

```
<field name="description" type="String">
    <hint-collection name="DESCRIPTION-TEXTAREA" />
</field>
```

Great! Now rebuild your service using Service Builder, redeploy your portlet project, and add or edit an event using the portlet. Check that the size of the description text area has changed as specified in your model hints.

Well, you've learned the art of persuasion through Liferay's model hints. Now, not only can you influence how your model's input fields are displayed but 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. Looks like you've picked up on the "hints" on how Liferay model hints help portlet data!

Next, let's find out how to implement a remote service.

5.10. Writing Remote Service Classes

Many default Liferay services are available as web services. Liferay exposes its web services via SOAP and JSON web services. If you're running Liferay locally on port 8080, visit the following URL to browse Liferay's default SOAP web services:

`http://localhost:8080/api/axis`

To browse Liferay's default JSON web services, visit this URL:

`http://localhost:8080/api/jsonws/`

These web services APIs can be accessed by many different kinds of clients, including non-portlet and even non-Java clients. You can use Service Builder to generate similar remote services for your projects' custom entities. When you run Service Builder with the `remote-service` attribute set to `true` for an entity, all the classes, interfaces, and files required to support both SOAP and JSON web services are generated for that entity. Service Builder generates methods that call existing services, but it's up to you to implement the methods that are exposed remotely. Let's use Service Builder to generate remote services for the Event Listing example project. You'll implement a few methods for the Event Listing Portlet that can be called remotely via SOAP and JSON web services.

Remember: local service methods are implemented in `EventLocalServiceImpl`. Similarly, you'll implement remote service methods in `EventServiceImpl`. Add the following methods to the `EventServiceImpl` class:

```
public Event addEvent(
    long groupId, String name, String description,
    int month, int day, int year, int hour, int minute, long locationId,
```

```

        ServiceContext serviceContext)
throws PortalException, SystemException {

    EventListingPermission.check(
        getPermissionChecker(), groupId, EventListingActionKeys.ADD_EVENT);

    return EventLocalServiceUtil.addEvent(
        getUserId(), groupId, name, description, month, day, year, hour,
        minute, locationId, serviceContext);
}

public Event deleteEvent(long eventId)
throws PortalException, SystemException {

    EventPermission.check(getPermissionChecker(), eventId,
        EventListingActionKeys.DELETE_EVENT);

    return eventLocalService.deleteEvent(eventId);
}

public Event getEvent(long eventId)
throws PortalException, SystemException {

    EventPermission.check(getPermissionChecker(), eventId,
        EventListingActionKeys.VIEW);

    return EventLocalServiceUtil.getEvent(eventId);
}

public Event updateEvent(
    long userId, long eventId, String name, String description,
    int month, int day, int year, int hour, int minute, long locationId,
    ServiceContext serviceContext)
throws PortalException, SystemException {

    EventPermission.check(getPermissionChecker(), eventId,
        EventListingActionKeys.UPDATE_EVENT);

    return EventLocalServiceUtil.updateEvent(
        userId, eventId, name, description, month, day, year, hour, minute,
        locationId, serviceContext);
}

```

Each remote service method performs security checks to determine whether the caller has permission to add/update/delete events. Notice that the methods use three new classes:

- `EventListingActionKeys` is an extension of the `ActionKeys` class, providing constants specifying the types of actions your plugin's portlets perform.
- `EventPermission` is a helper class for checking whether the user is authorized to perform specific actions on the `Event` entity.
- `EventListingPermission` is a helper class for checking whether the user is authorized to add the instances of the plugin's specific entity types.

You must manually create all of these types of classes. You can create .java files for each of them and copy contents from the linked solution classes above into the respective source files you create. We cover Liferay's Security and Permissions framework later in this guide. To see how the Event Listing Portlet is integrated with Liferay's permissions system, browse the Event Listing example project available in the *Dev Guide SDK* at <https://github.com/liferay/liferay-docs/tree/master/devGuide/code/devGuide-sdk>. The project is in the SDK's portlets/event-listing-portlet folder.

Notice the calls to the eventLocalService field's addEvent, updateEvent, and deleteEvent methods. The eventLocalService field holds a Spring bean of type EventLocalServiceImpl that's injected into EventServiceImpl by Service Builder. See EventServiceImpl for a complete list of the Spring beans available in EventServiceImpl. These include the following:

- counterLocalService
- eventLocalService
- eventService
- eventPersistence
- locationLocalService
- locationPersistence
- locationService
- resourceLocalService
- resourcePersistence
- resourceService
- userLocalService
- userPersistence
- userService

Notice also that we modified the deleteEvent method of the EventServiceImpl class passing it an event ID as a parameter instead of an entire event object. The method is now ready to call as a remote web service.

After you finish adding imports to EventServiceImpl, save the class and run Service Builder again.

Liferay uses Apache Axis to make SOAP web services available. Axis requires a Web Service Deployment Descriptor (WSDD) to be generated in order to make the SOAP web services available. Liferay provides a build-wsdd Ant target that generates the WSDD. In Liferay IDE or Developer Studio, when viewing your service.xml file in Overview mode, there's a button in the top-right corner of the screen for calling the Build WSDD target. Liferay Portal makes your service's Web Services Definition Language (WSDL) available after you've built its WSDD and deployed your portlet project. Let's learn how to call your remote services next.

5.10.1. Calling Remote Services

Service Builder can expose your project's remote web services both via a JSON API and via SOAP. By default, running Service Builder with remote-service set to true for your entities generates a JSON web services API for your project. You can access your project's

JSON-based RESTful services via a convenient web interface. To view the JSON web services available for the Event Listing plugin, visit the following URL:

`http://localhost:8080/event-listing-portlet/api/jsonws`

Each entity's available operations are listed on the plugin's JSON web services API page. If you've been implementing the Nose-ster Event Listing example portlet used throughout this chapter, you'll be anxious to try out its remote web services. You can invoke JSON web services directly from your browser. For example, to bring up a test form for your Event entity's *delete-event* operation, visit the above URL and click on its *delete-event* link.



Event

[add-event](#)[add-event](#)[delete](#)[delete](#)[delete-event](#)[delete-event](#)[get-bean-identifier](#)[get-bean-identifier](#)[invoke-method](#)[invoke-method](#)[set-bean-identifier](#)[set-bean-identifier](#)[update](#)[update](#)

Location

[get-bean-identifier](#)

⚡ /event-listing-portlet

com.nosester.portlet.eventlisting.s

.Parameters

eventId long

Return Type

com.nosester.portlet.eventlisting.m

Exception

com.liferay.portal.kernel.exception

com.liferay.portal.kernel.exception

Execute

eventId

long

Invoke

The only parameter required for the `delete-event` operation is an event ID. Since there's no UI yet for the Event Listing application, you probably don't have any events in your database. But if you did, you could check for an event's ID in your `Event_Event` database table. Then you could enter the value into the `eventId` field in the test page's `Execute` section and click *Invoke* to delete that event. Liferay returns feedback from each invocation.

Finding a portlet's web services is easy with Liferay's JSON web service interface. Invoking a portlet's web services via Liferay's JSON web service interface is a great way to test them. You can also examine alternate equivalent methods of calling the SOAP and JSON web services via JavaScript, Curl, and URLs. Next, we'll consider how to implement custom SQL queries in your portlet, so you can easily leverage information from multiple entity types.

Service Builder can also make your project's web services available via SOAP using Apache Axis. After you've built your portlet project's WSDDs and deployed the project as a plugin, its services are available on the portal server. It's easy to list your Nose-ster Event Listing plugin's SOAP web services. You have two options: you can access them in Liferay IDE or you can open your browser to the URL for its web services.

To access your SOAP web services in Liferay IDE, right-select your portlet from under it's Liferay server in the *Servers* view. Then select *Test Liferay Web Services....*

If you'd rather view your SOAP web services from a browser, go to the following URL:

```
http://localhost:8080/event-listing-portlet/api/axis
```

Liferay Portal lists the services available for all your entities and provides links to their WSDL documents. Clicking on the WSDL link for the Event service takes you to the following URL:

```
http://localhost:8080/event-listing-
portlet/api/axis/Plugin_Event_EventService?wsdl
```

This WSDL document lists the Event entity's SOAP web services. Once the web service's WSDL is available, any SOAP web service client can access it.

5.11. **Developing Custom SQL Queries**

Service Builder's finder methods facilitate searching for entities by their attributes--their column values. Add the column as a parameter for the finder in your `service.xml` file, run Service Builder, and it generates the finder method in your persistence layer and adds methods to your service layer that invoke the finder. But what if you'd like to do more complicated searches that incorporate attributes from multiple entities?

For example, consider the Event Listing Portlet you've been developing in this chapter. Suppose you want to find an event based on its name, description, and location name. If you recall, the event entity refers to its location by the location's ID, not its name. That is, the event entity table, `Event_Event`, refers to an event's location by its long integer ID in the table's `locationId` column. But you need to access the *name* of the event's location. Of course, with SQL you can join the event and location tables to include the location name. But how would you incorporate custom SQL into your portlet? And how would you invoke the SQL from your service? Service Builder lets you do this by specifying the SQL as *Liferay custom SQL* and invoking it in your service via a *custom finder method*.

Liferay custom SQL is a Service Builder-supported method for performing complex and custom

queries against the database. Invoking custom SQL from a finder method in your persistence layer is straightforward. And Service Builder helps you generate the interfaces to your finder method. It's easy to do by following these steps:

1. Specify your custom SQL.
2. Implement your finder method.
3. Access your finder method from your service.

Next, you'll do exactly this to create and invoke custom SQL in your Event Listing Portlet.

5.11.1. Step 1: Specify Your Custom SQL

After you've tested your SQL, you must specify it in a particular file for Liferay to access it. Liferay's `CustomSQLUtil` class looks up custom SQL from a file called `default.xml` in your portlet project's `docroot/WEB-INF/src/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>
```

You can add a `custom-sql` element for every custom SQL query you'd like for your portlet, as long as each query has a unique ID. The convention we recommend using 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 in Step 2.

For this example, you'll use the following ID value for the query:

```
com.nosester.portlet.eventlisting.service.persistence.\EventFinder.findByEventNameEventDescriptionLocationName
```

Custom SQL must be wrapped in character data (CDATA) for the `sql` element. Importantly, the SQL must *not* be terminated with a semi-colon.

Following these rules, specify the custom SQL for this query by replacing the contents of the `default.xml` file with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<custom-sql>
  <sql
    id="com.nosester.portlet.eventlisting.service.persistence.EventFinder.\findByEventNameEventDescriptionLocationName">
    SELECT Event_Event.*
    FROM Event_Event
    INNER JOIN
      Event_Location ON Event_Event.locationId =
    Event_Location.locationId
    WHERE
      (Event_Event.name LIKE ?) AND
      (Event_Event.description LIKE ?) AND
```

```

        (Event_Location.name LIKE ?)
    </sql>
</custom-sql>
```

Make sure to delete the backslash (\ \) character from the end of the ID so that the finder method name `findByEventNameEventDescriptionLocationName` immediately follows the package path specified below:

```
com.nosester.portlet.eventlisting.service.persistence.
```

Now that you've specified some custom SQL, the next step is to implement the finder method. The method name for the finder should match the ID you just specified for the `sql` element.

5.11.2. Step 2: Implement Your Finder Method

It's time to implement the finder method to invoke the custom SQL query. This should be done in the service's persistence layer, since it's SQL invoked on a relational database. You'll rely on Service Builder to generate the interface for it. But before you do that, you need to create the implementation of the finder.

The first step is to create a `-FinderImpl` class in the service persistence package. Create a class called `EventFinderImpl` in the

```
com.nosester.portlet.eventlisting.service.persistence.impl package.
```

Make the class extend `BasePersistenceImpl<Event>`. Your

`EventFinderImpl.java` file should now have the following contents:

```
package com.nosester.portlet.eventlisting.service.persistence;

import com.liferay.portal.service.persistence.impl.BasePersistenceImpl;
import com.nosester.portlet.eventlisting.model.Event;

public class EventFinderImpl extends BasePersistenceImpl<Event> {
```

```
}
```

Run Service Builder to generate the `-Finder` interface and the `-Util` class for the finder. Service Builder generates the `EventFinder` interface and the `EventFinderUtil` utility class based on the `EventFinderImpl` class. Modify your `EventFinderImpl` class to have it implement the `EventFinder` interface you just generated:

```
public class EventFinderImpl extends BasePersistenceImpl<Event>
    implements EventFinder {
```

```
}
```

Now you can create our finder method in your `EventFinderImpl` class. Add the following finder method and static field to the `EventFinderImpl` class:

```
public List<Event> findByEventNameEventDescriptionLocationName(
    String eventName, String eventDescription, String locationName,
    int begin, int end) {

    Session session = null;
    try {
        session = openSession();
```

```

        String sql = CustomSQLUtil.get(
            FIND_BY_EVENTNAME_EVENTDESCRIPTON_LOCATIONNAME);

        SQLQuery q = session.createSQLQuery(sql);
        q.setCacheable(false);
        q.addEntity("Event_Event", EventImpl.class);

        QueryPos qPos = QueryPos.getInstance(q);
        qPos.add(eventName);
        qPos.add(eventDescription);
        qPos.add(locationName);

        return (List<Event>) 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_EVENTNAME_EVENTDESCRIPTON_LOCATIONNAME =
    EventFinder.class.getName() +
        ".findByNameEventDescriptionLocationName";

```

Remember to import the required classes. We've provided the imports here for your convenience:

```

import java.util.List;

import com.liferay.portal.kernel.dao.orm.QueryPos;
import com.liferay.portal.kernel.dao.orm.QueryUtil;
import com.liferay.portal.kernel.dao.orm.SQLQuery;
import com.liferay.portal.kernel.dao.orm.Session;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.service.persistence.impl.BasePersistenceImpl;
import com.liferay.util.dao.orm.CustomSQLUtil;

```

```

import com.nosester.portlet.eventlisting.model.Event;
import com.nosester.portlet.eventlisting.model.impl.EventImpl;

```

The custom finder method opens a new Hibernate session and uses Liferay's CustomSQLUtil.get(String id) method to get the custom SQL to use for the database query. The FIND_BY_EVENTNAME_EVENTDESCRIPTON_LOCATIONNAME 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 (findByNameEventDescriptionLocationName).

Awesome! Custom SQL is in place and your finder method is implemented. Next, you'll call the finder method from your service.

5.11.3. Step 3: Access Your Finder Method from Your Service

So far, you created a `-FinderImpl` class and generated a `-FinderUtil` utility class. However, your portlet class should not use the finder utility class directly; only a local or remote service implementation (i.e., `-LocalServiceImpl` or `-ServiceImpl`) in your plugin project should invoke the `-FinderUtil` class. This encourages a proper separation of concerns: the portlet classes invoke business logic of the services and the services in turn access the data model using the persistence layer's finder classes. So you'll add a method in the `-LocalServiceImpl` class that invokes the finder method implementation via the `-FinderUtil` class. Then you'll provide the portlet and JSPs access to this service method by rebuilding the service.

Add the following method to the `EventLocalServiceImpl` class:

```
public List<Event> findByEventNameEventDescriptionLocationName (String
eventName,
    String eventDescription, String locationName, int begin, int end)
throws SystemException {

    return EventFinderUtil.findByEventNameEventDescriptionLocationName (
        eventName, eventDescription, locationName, begin, end);
}
```

After you've added this method, run Service Builder to generate the interface and make this finder method available in the `EventLocalServiceUtil` class.

Now you can indirectly call the finder method from your portlet class or from a JSP by calling `EventLocalServiceUtil.findByEventNameEventDescriptionLocationName(...)`!

Congratulations on following the 3 step process in developing a custom SQL query and custom finder for your portlet!

Next you'll tour through the `service.properties` file that Service Builder generates.

5.12. Configuring `service.properties`

Service Builder generates a `service.properties` file in your project's `docroot/WEB-INF/src` folder. Liferay Portal uses the properties in this file to alter your service's database schema and load Spring configuration files to support deployment of your service. You should not modify this file, but rather make any necessary overrides in a `service-ext.properties` file in that same folder.

Here are some of the properties included in the `service.properties` file:

- `build.auto.upgrade`: This is `true` by default. This property determines whether or not Liferay should automatically apply changes to the database model when a new version of the plugin is deployed.
- `build.namespace`: This is the namespace you defined in `docroot/WEB-INF/service.xml`. Liferay distinguishes different plugins from each other using their namespaces.

- `build.number`: Liferay distinguishes different builds of your plugin. Each time a distinct build of your plugin is deployed to Liferay, Liferay increments this number.
- `build.date`: This is the time of the latest build of your plugin.
- `spring.configs`: This is a comma-delimited list of Spring configurations.
- `include-and-override`: The default value of this property defines `service-ext.properties` as an override file for `service.properties`.

It's sometimes useful to override the `build.auto.upgrade` property from `service.properties`. Setting `build.auto.upgrade=false` in your `service-ext.properties` file prevents Liferay from trying automatically to apply any changes to the database model when a new version of the plugin is deployed. This is needed in projects in which it is preferred to manually manage the changes to the database or in which the SQL schema has intentionally been modified manually after generation by Service Builder.

5.13. Summary

You've covered a lot of ground in this chapter. You learned how to map out your data model as entities to use in services. You used Service Builder and Liferay IDE's powerful editing modes to create services, relate service entities, and generate your implementation stubs. You implemented business logic and re-ran Service Builder to generate the corresponding interfaces. You found out that integrating your own SQL queries in your services was easy and that model hints enable you to specify service entity data limitations and to make entity display customizations. You also briefly implemented remote services but you merely scratched the surface of this topic.

In the next chapter, you'll learn how to find and invoke other remote Liferay services. You'll have an in-depth look at Liferay's service security layer. And last but not least, you'll dive deep into SOAP web services and JSON web services. So hold on tight, you're about to get served a big helping of Liferay services.

6. Accessing Services Remotely

You've created your portlet and built some terrific services. You're happy to brag to your colleagues about the awesome things your portlet does. Now folks are getting interested; they want to call your portlet's services. You wonder whether this will be difficult and you start asking yourself questions. How can I publish my services? How can my clients find my services? How can consumers call my services efficiently? No worries. We'll answer all of these questions on accessing remote services.

Here are the topics we'll cover in this chapter:

- Finding Services
- Invoking the API Remotely
- Service Security Layers
- SOAP Web Services
- JSON Web Services
- Authorizing Access to Services with OAuth

6.1. Finding Services

You can find Liferay's services by searching for them in the Javadocs:

<http://docs.liferay.com/portal/6.2/javadocs/>. Below, we'll show you how to search for portal services and portlet services.

Let's start by finding a portal service.

6.1.1. Finding Portal Services

Liferay's Javadocs are easy to browse and well-organized. Here's how to find the *Organization* services:

1. In your browser, open up the Javadocs: <http://docs.liferay.com/portal/6.2/javadocs/>
2. Under *Portal Services*, click the link for the `com.liferay.portal.service` package, since the services for the *Organization* entity belong to the *Portal* scope.
3. Find and click on the `-ServiceUtil` class (in this case, `OrganizationLocalServiceUtil`) in the *Class Summary* table or the *Classes* list at the bottom of the page.

That was easy! What if you want to find portlet services?

6.1.2. Finding Portlet Services

Searching for one of Liferay's built-in portlet services is also easy. Instead of clicking the link for the service package of the *portal*, click the link for the service package of the *portlet*. The portlet service packages use the naming convention `com.liferay.portlet.[portlet-name].service`, where `[portlet-name]` is replaced with the actual name of the portlet.

Here's how you find services for a user's blogs statistics:

1. In your browser, open the Javadocs: <http://docs.liferay.com/portal/6.2/javadocs/>
2. Under *Portlet Services*, click the link for the `com.liferay.portlet.blogs.service` package in the *Packages* frame, since the services are a part of the Blogs portlet.
3. Find and click on the `-ServiceUtil` class (in this case `BlogsStatsUserLocalServiceUtil`) in the *Class Summary* table or the *Classes* list.

Now you're ready to invoke Liferay services.

6.2. Invoking the API Remotely

Remote clients run outside of the portal JVM or on a remote machine but Liferay's remote services allow the portal's API to be called from outside the portal. Remote services can be invoked through non-Java languages like JavaScript and PHP and remote services can reply to calls with JSON objects. Note, however, that remote services are often harder to call than local services since contextual information that's usually available when making local service calls is

not available when making remote service calls. For example, the `ServiceContext` or `ThemeDisplay` objects are often available when you're making local service calls but not when you're making remote service calls.

Invoking remote services does require more overhead such as memory, network bandwidth, and processing than does invoking local services. Also, remote services require permission-checking to prevent just anyone from remotely invoking them. The remote services of Liferay's API perform security checks but these are not automatically generated by Service Builder; they're manually added by developers. For information on adding permission checks to a plugin's remote services, please refer to the previous chapter. If you'd like to *invoke* a plugin's remote services, make sure to check that the remote service methods perform permission checks. Unless you want to avoid permission checking, it's often a good idea to develop your clients (even if they're local) so they trigger the front-end security layer.

Liferay's API follows a Service Oriented Architecture (SOA). The API supports Java invocation and a variety of protocols including SOAP and JSON over HTTP. A limited set of *RESTful* web services, based on the AtomPub protocol, are also supported--see the Portal Atom Collections wiki by Igor Spasić for more details. You can also use the API through Remote Procedure Calls (RPC). You have many good options for leveraging Liferay's API.

Let's step back now and discuss the security layers of Liferay's *service oriented* architecture and how you can configure them.

6.3. Service Security Layers

Liferay's remote services sit behind a layer of security that by default allows only local connections. Access to the remote APIs must be enabled as a separate step in order to call them from a remote machine. Liferay's core web services require user authentication and authentication verification. We'll discuss this process later in this section. Lastly, regardless of whether the remote service is called from the same machine or via a web service, Liferay's standard security model comes into action: a user must have the proper permissions in Liferay's permissions system to access remote services.

The first layer of security a client encounters when calling a remote service is called *invoker IP filtering*. Imagine you have a batch job that runs on another machine in your network. This job polls a shared folder on your network and uploads documents to your site's *Documents and Media* portlet on a regular basis, using Liferay's web services. To get your batch job through the IP filter, the portal administrator has to allow the machine on which the batch job is running access to Liferay's remote service. For example, if your batch job uses the SOAP web services to upload the documents, the portal administrator must add the IP address of the machine on which the batch job is running to the `axis.servlet.hosts.allowed` property. A typical entry might look like this:

```
axis.servlet.hosts.allowed=192.168.100.100, 127.0.0.1, [SERVER_IP]
```

If the IP address of the machine on which the batch job is running is listed as an authorized host for the service, it's allowed to connect to Liferay's web services, pass in the appropriate user credentials, and upload the documents.



Note: The `portal.properties` file resides on the portal host machine and is controlled by the portal administrator. Portal administrators can configure security settings for the Axis Servlet, the Liferay Tunnel Servlet, the Spring Remoting Servlet, the JSON Servlet, the JSON Web Service Servlet, and the WebDAV Servlet. The `portal.properties` file (online version is available at <http://docs.liferay.com/portal/6.2/propertiesdoc/portal.properties.html>) describes these properties.

Next, if you're invoking the remote service via web services (e.g., JSON WS, old JSON, Axis, REST, etc.), a two step process of authentication and authentication verification is involved. Each call to a Liferay portal web services must be accompanied by a user authentication token. It's up to the web service caller to produce the token (e.g., through Liferay's utilities or through some third-party software). Liferay verifies that there is a Liferay user that matches the token. If the credentials are invalid, the web service invocation is aborted. Otherwise, processing enters into Liferay's user permission layer.

Liferay's user permission layer is the last Liferay security layer triggered when services are invoked remotely, and it's used for every object in the portal, whether accessing it locally or remotely. The user ID accessing the services remotely must have the proper permission to operate on the objects it's trying to access. A remote exception is thrown if the user ID isn't permitted. A portal administrator can grant users access to these resources. For example, suppose you created a Documents and Media Library folder called *Documents* in a site, created a role called *Document Uploaders*, and granted this role the rights to add documents to your new folder. If your batch job accesses Liferay's web services to upload documents into the folder, you have to call the web service using a user ID of a member of this role (or using the user ID of a user with individual rights to add documents to this folder, such as a portal administrator). If you don't, Liferay denies you access to the web service.

With remote services, you can specify the user credentials using HTTP basic authentication. Since those credentials are passed over the network unencrypted, we recommend using HTTPS whenever accessing these services on an untrusted network. Most HTTP clients let you specify the basic authentication credentials in the URL--this is very handy for testing.

Use the following syntax to call the AXIS web service using credentials. Make sure to remove the line escape character \ when entering your URL:

```
http://" + screenNameOrUserIdAsString + ":" + password + "@[server.com]:\" +  
[port]/api/axis/" + serviceName
```

The `screenNameOrUserIdAsString` should either be the user's screen name or the user's ID from the Liferay database. The portal's authentication type setting determines which one to use; we discuss this in more detail below. A user can find his or her ID by logging in as the user and accessing *My Account* from the Dockbar. On this interface, the user ID appears below the user's profile picture and above the birthday field.

Let's pretend that your portal's authentication type is set to be by *user ID* and that there's a user whose ID is 2 and whose password is `test`. You can access Liferay's remote Organization service with the following URL:

```
http://2:test@localhost:8080/api/axis/Portal_OrganizationService
```

As mentioned above, the authentication type specified for your Liferay Portal instance dictates the authentication type you'll use to access your web service. The portal administrator can set the portal's authentication type to any of the following:

- *email address*
- *screen name*
- *user ID*



Important: In order for authentication to work for remote service calls, the portal authentication type must be set either to *screen name* or *user ID*. Authentication using the *email address* authentication type is not supported for remote service calls.

You can set the authentication type via the Control Panel or via the `portal-ext.properties` file. To set the portal authentication type via the Control Panel, navigate to the Control Panel, click on *Portal Settings*, and then on *Authentication*. Under *How do users authenticate?*, make a selection. To set the portal authentication type via properties file, add the following lines to your Liferay instance's `portal-ext.properties` file and uncomment the line for the appropriate authentication type:

```
#company.security.auth.type=emailAddress  
#company.security.auth.type=screenName  
#company.security.auth.type=userId
```

Your Liferay Portal password policies (see the User Management chapter of *Using Liferay Portal 6.2*) should be reviewed, since they'll be enforced on your administrative ID as well. If the portal is enforcing password policies on its users (e.g., requiring them to change their passwords on a periodic basis), an administrative ID accessing Liferay's web services in a batch job will have its password expire too.

To prevent a password from expiring, a portal administrator can add a new password policy that doesn't enforce password expiration and add a specific administrative user ID to it. Then your batch job can run as many times as you need it to, without your administrative ID's password expiring.

To summarize, accessing Liferay remotely requires you to pass two layers of security checks:

- *IP permission layer*: The IP address must be pre-configured in the server's portal properties.
- *Authentication/verification layer (web services only)*: Liferay verifies that the caller's authorization token can be associated with a portal user.
- *User permission layer*: The user needs permission to access the related resources.

Next, let's talk about Liferay's SOAP web services.

6.4. SOAP Web Services

You can access Liferay's services via *Simple Object Access Protocol (SOAP)* over HTTP. The *packaging* protocol is SOAP and the *transport* protocol is HTTP.



Note: An authentication related token must accompany each Liferay web service invocation. For details, read the section on service security layers found earlier in this chapter.

As an example, let's look at the SOAP web service classes for Liferay's Company, User, and UserGroup portal services to execute the following:

1. List each user group to which user *test* belongs.
2. Add a new user group named *MyGroup*.
3. Add your portal's administrative user to the new user group. For demonstration purposes, we'll use an administrative user whose email address is `test@liferay.com`.

We'll use these SOAP related classes:

```
import com.liferay.portal.model.CompanySoap;
import com.liferay.portal.model.UserGroupSoap;
import com.liferay.portal.service.http.CompanyServiceSoap;
import com.liferay.portal.service.http.CompanyServiceSoapServiceLocator;
import com.liferay.portal.service.http.UserGroupServiceSoap;
import com.liferay.portal.service.http.UserGroupServiceSoapServiceLocator;
import com.liferay.portal.service.http.UserServiceSoap;
import com.liferay.portal.service.http.UserServiceSoapServiceLocator;
```

Can you see the naming convention for SOAP related classes? The classes above all have suffixes `-ServiceSoapServiceLocator`, `-ServiceSoap`, and `-Soap`. The `-ServiceSoapServiceLocator` class finds the `-ServiceSoap` by means of the service's URL you provide. The `-ServiceSoap` class is the interface to the services specified in the *Web Services Definition Language (WSDL)* file for each service. The `-Soap` classes are the serializable implementations of the models. Let's look at how to determine the URLs for these services.

You can see a list of the services deployed on your portal by opening your browser to the following URL:

```
http://[host]:[port]/api/axis
```



Note: Prior to Liferay 6.2, there were two different URLs for accessing remote Liferay services. `http://[host]:[port]/api/secure/axis` was for services requiring authentication and `http://[host]:[port]/api/axis` was for services that didn't require authentication. As of Liferay 6.2, all remote Liferay services require authentication and the `http://[host]:[port]/api/axis` URL is used to access them.

Here's the list of *secure* web services for UserGroup:

- `Portal_UserGroupService (wsdl)`
 - `addGroupUserGroups`
 - `addTeamUserGroups`

- › addUserGroup
- › deleteUserGroup
- › getUserGroup
- › getUserUserGroups
- › unsetGroupUserGroups
- › unsetTeamUserGroups
- › updateUserGroup



Note: Liferay's developers use a tool called *Service Builder* to expose their services via SOAP automatically. If you're interested in using Service Builder, read Generating Your Service Layer.

Each web service is listed with its name, operations, and a link to its WSDL file. The WSDL file is written in XML and provides a model for describing and locating the web service.

Here's a WSDL excerpt of the addUserGroup operation of UserGroup:

```
<wsdl:operation name="addUserGroup" parameterOrder="name description
publicLayoutSetPropertyId privateLayoutSetPropertyId">
    <wsdl:input message="impl:addUserGroupRequest" name="addUserGroupRequest"
    />
    <wsdl:outputMessage="impl:addUserGroupResponse"
name="assUserGroupResponse"
    />
</wsdl:operation>
```

To use the service, you pass in the WSDL URL along with your login credentials to the SOAP service locator for your service. We'll show you an example in the next section.

Next, let's invoke the web service!

6.4.1. SOAP Java Client

You can easily set up a Java web service client to access Liferay's remote services using Eclipse. Here's how:

In Eclipse, you can use the *New → Web Service Client* wizard to either create a new web service client project or add a client to an existing project. You need to add a new web service client to your project for each service that you need to consume in your client code. For our example, we'll build a web service client to invoke the portal's Company, User, and UserGroup services.

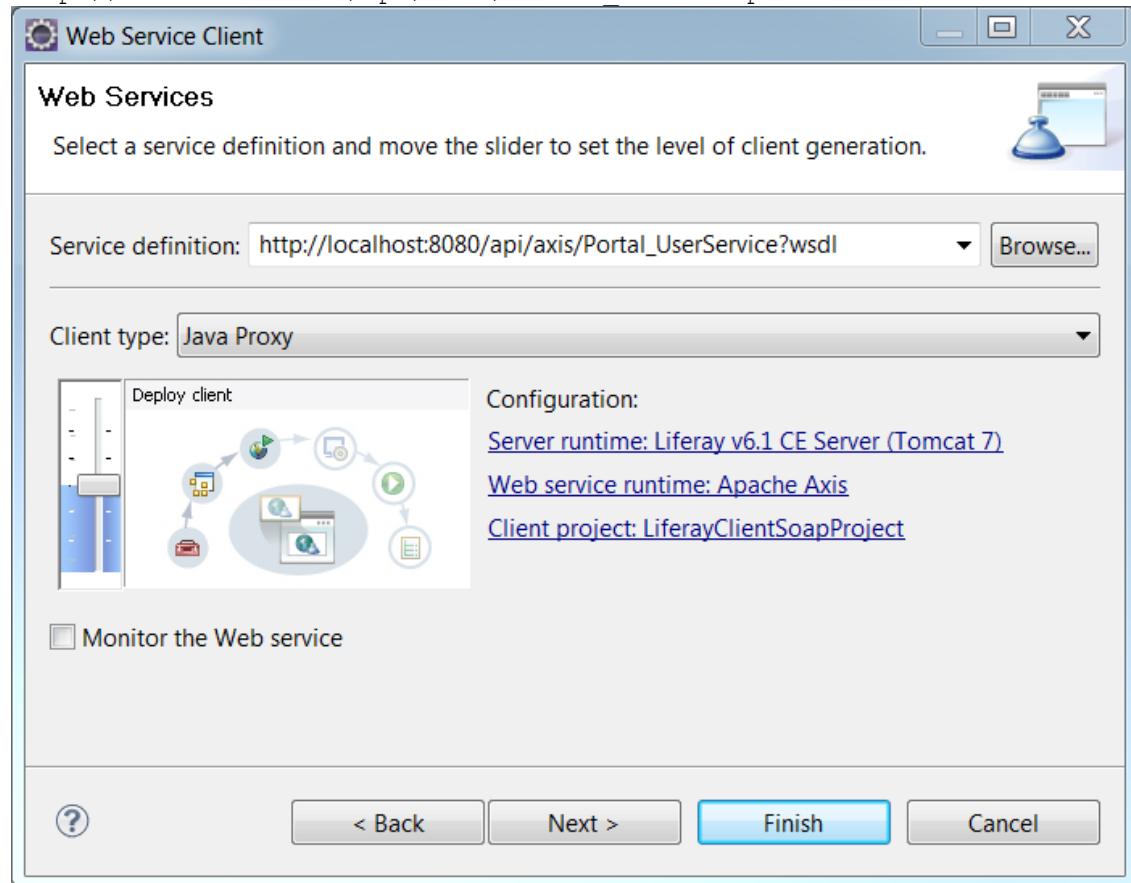
To create a new web service client project in Eclipse, click *File → New → Other...*, then expand the *Web Services* category. Select *Web Service Client*.

For each client you create, you're prompted to enter the service definition (WSDL) for the desired service. Since our example web service client will use Liferay Portal's Company, User, and UserGroup services, we'll need to enter the following WSDLs:

`http://localhost:8080/api/axis/Portal_CompanyService?wsdl`

http://localhost:8080/api/axis/Portal_UserService?wsdl

http://localhost:8080/api/axis/Portal_UserGroupService?wsdl



When you specify a WSDL, Eclipse automatically adds the auxiliary files and libraries required to consume that web service. Nifty!

After you've created your web service client project using one of the above WSDLs, you need to create additional clients in the project using the remaining WSDLs. To create an additional client in an existing project, right-click on the project and select *New → Other → Web Service Client*. Click *Next*, enter the WSDL, and complete the wizard.

The code below locates and invokes operations to create a new user group named *MyUserGroup* and add a user with the screen name *test* to it. Create a *LiferaySoapClient.java* file in your web service client project and add the following code to it. If you create this class in a package other than the one that's specified in the code below, replace the package with your package. To run the client from Eclipse, make sure that your Liferay server is running, right-click on the *LiferaySoapClient.java* class, and select *Run as Java application*. Check your console to check that your service calls succeeded.

```
package com.liferay.test;

import java.net.URL;

import com.liferay.portal.model.CompanySoap;
import com.liferay.portal.model.UserGroupSoap;
import com.liferay.portal.service.http.CompanyServiceSoap;
```

```

import com.liferay.portal.service.http.CompanyServiceSoapServiceLocator;
import com.liferay.portal.service.http.UserGroupServiceSoap;
import com.liferay.portal.service.http.UserGroupServiceSoapServiceLocator;
import com.liferay.portal.service.http.UserServiceSoap;
import com.liferay.portal.service.http.UserServiceSoapServiceLocator;

public class LiferaySoapClient {

    public static void main(String[] args) {

        try {
            String remoteUser = "test";
            String password = "test";
            String virtualHost = "localhost";

            String groupName = "MyUserGroup";

            String serviceCompanyName = "Portal_CompanyService";
            String serviceUserName = "Portal_UserService";
            String serviceUserGroupName = "Portal_UserGroupService";

            long userId = 0;

            // Locate the Company
            CompanyServiceSoapServiceLocator locatorCompany =
                new CompanyServiceSoapServiceLocator();

            CompanyServiceSoap soapCompany =
                locatorCompany.getPortal_CompanyService(
                    _getURL(remoteUser, password, serviceCompanyName,
                        true)));

            CompanySoap companySoap =
                soapCompany.getCompanyByVirtualHost(virtualHost);

            // Locate the User service
            UserServiceSoapServiceLocator locatorUser =
                new UserServiceSoapServiceLocator();
            UserServiceSoap userSoap = locatorUser.getPortal_UserService(
                _getURL(remoteUser, password, serviceUserName, true));

            // Get the ID of the remote user
            userId = userSoap.getUserIdByScreenName(
                companySoap.getCompanyId(), remoteUser);
            System.out.println("userId for user named " + remoteUser +
                " is " + userId);

            // Locate the UserGroup service
            UserGroupServiceSoapServiceLocator locator =
                new UserGroupServiceSoapServiceLocator();
            UserGroupServiceSoap usergroupsoap =
                locator.getPortal_UserGroupService(
                    _getURL(remoteUser, password, serviceUserGroupName,
                        true)));

```

```

// Get the user's user groups
UserGroupSoap[] usergroups = usergroupsoap.getUserUserGroups(
    userId);

System.out.println("User groups for userId " + userId + " ...");
for (int i = 0; i < usergroups.length; i++) {
    System.out.println("\t" + usergroups[i].getName());
}

// Adds the user group if it does not already exist
String groupDesc = "My new user group";
UserGroupSoap newUserGroup = null;

boolean userGroupAlreadyExists = false;
try {
    newUserGroup = usergroupsoap.getUserGroup(groupName);
    if (newUserGroup != null) {
        System.out.println("User with userId " + userId +
            " is already a member of UserGroup " +
            newUserGroup.getName());
        userGroupAlreadyExists = true;
    }
} catch (Exception e) {
    // Print cause, but continue
    System.out.println(e.getLocalizedMessage());
}

if (!userGroupAlreadyExists) {
    newUserGroup = usergroupsoap.addUserGroup(
        groupName, groupDesc);
    System.out.println("Added user group named " + groupName);

    long users[] = {userId};
    userSoap.addUserGroupUsers(newUserGroup.getUserGroupId(),
        users);
}

// Get the user's user groups
usergroups = usergroupsoap.getUserUserGroups(userId);

System.out.println("User groups for userId " + userId + " ...");
for (int i = 0; i < usergroups.length; i++) {
    System.out.println("\t" + usergroups[i].getName());
}
}

catch (Exception e) {
    e.getLocalizedMessage();
}
}

private static URL _getURL(String remoteUser, String password,
    String serviceName, boolean authenticate)
throws Exception {

```

```

    // Unauthenticated url
    String url = "http://localhost:8080/api/axis/" + serviceName;

    // Authenticated url
    if (authenticate) {
        url = "http://" + remoteUser + ":" + password
            + "@localhost:8080/api/axis/"
            + serviceName;
    }

    return new URL(url);
}

}

```

Running this client should produce output like the following example:

```

userId for user named test is 10196
User groups for user 10196 ...
java.rmi.RemoteException: No UserGroup exists with the key {companyId=10154,
name=MyUserGroup}
Added user group named
Added user to user group named MyUserGroup
User groups for user 10196 ...
    MyUserGroup

```

The output tells us the user had no groups, but was added to UserGroup MyUserGroup.

You might be thinking, "But an error was thrown! We did something wrong!" Yes, an error was thrown (`java.rmi.RemoteException:`), but we're sitting here as cool as an ice cream sandwich all the same. The exception was thrown simply because the `UserGroup` check was invoked before the `UserGroup` was created. Because the very next line of the output says `Added user group named....`, we're okay. Don't worry, be happy!

Here are a few things to note about this example:

- Authentication is done using HTTP Basic Authentication, which isn't appropriate for a production environment, since the password is unencrypted. It's simply used for convenience in this example. In production, you should set `company.security.auth.requires.https=false`. Please refer to Liferay's `portal.properties` file for more information.
- The screen name and password are passed in the URL as credentials.
- The name of the service (e.g. `Portal_UserGroupService`) is specified at the end of the URL. Remember that the service name can be found in the web service listing.

The operations `getCompanyByVirtualHost()`, `getUserByIdByScreenName()`, `getUserUserGroups()`, `addUserGroup()` and `addUserGroupUsers()` are specified for the `-ServiceSOAP` classes `CompanyServiceSoap`, `UserServiceSoap` and `UserGroupServiceSoap` in the WSDL files. Information on parameter types, parameter order, request type, response type, and return type are conveniently specified in the WSDL for each Liferay web service. It's all there for you!

Next, let's implement a web service client in PHP.

6.4.2. SOAP PHP Client

You can write your client in any language that supports web services invocation. Let's invoke the same operations we did when we created our Java client, this time using PHP and a PHP SOAP Client:

```
<?php
    $userGroupName = "MyUserGroup2";
    $userName = "test";
    $clientOptions = array('login' => $userName, 'password' => 'test');

    // Add user group
    $userGroupClient = new
        SoapClient(
            "http://localhost:8080/api/axis/Portal_UserGroupService?wsdl",
            $clientOptions);
    $userGroup = $userGroupClient->addUserGroup($userGroupName,
        "This user group was created by the PHP client! ");
    print ("User group ID is $userGroup->userGroupId ");

    // Add user to user group
    $companyClient = new SoapClient(
        "http://localhost:8080/api/axis/Portal_CompanyService?wsdl",
        $clientOptions);
    $company = $companyClient->getCompanyByVirtualHost("localhost");
    $userClient = new SoapClient(
        "http://localhost:8080/api/axis/Portal_UserService?wsdl",
        $clientOptions);
    $userId = $userClient->getUserByIdByScreenName($company->companyId,
        $userName);
    print ("User ID for $userName is $userId ");
    $users = array($userId);
    $userClient->addUserGroupUsers($userGroup->userGroupId, $users);

    // Print the user groups to which the user belongs
    $userGroups = $userGroupClient->getUserUserGroups($userId);
    print ("User groups for user $userId ... ");
    foreach($userGroups as $ug)
        print ("$ug->name, $ug->userGroupId ")
?>
```

Remember, you can implement a web service client in any language that supports using SOAP web services. Next, we'll explore Liferay's JSON web services.

6.5. JSON Web Services

JSON web services let you access portal service methods by exposing them as a JSON HTTP API. Service methods are made easily accessible using HTTP requests, both from JavaScript within the portal and from any JSON-speaking client.

We'll cover the following topics as we explore JSON Web Service functionality:

- Registration
- Configuration

- Invocation
- Results

Let's start by discussing how to register JSON web services.

6.5.1. Registering JSON Web Services

Liferay's developers use a tool called *Service Builder* to build services. When you build services with Service Builder, all remote-enabled services (i.e., `service.xml` entities with the property `remote-service="true"`) are exposed as JSON web services. When each `-Service.java` interface is created for a remote-enabled service, the `@JSONWebService` annotation is added on the class level of that interface. All of the public methods of that interface become registered and available as JSON web services.

The `-Service.java` interface source file should never be modified by the user. If you need, however, more control over its methods (e.g., hiding some methods and exposing others), you can configure the `-ServiceImpl` class. When the service implementation class (`-ServiceImpl`) is annotated with the `@JSONWebService` annotation, the service interface is ignored and the service implementation class is used for configuration in its place. In other words, `@JSONWebService` annotations in the service implementation override any JSON Web Service configuration in service interface.

That's it! When you start Liferay Portal, it scans service classes for annotations (more about scanning later). Each class that uses the `@JSONWebService` annotation is examined and its methods become exposed as JSON API. As explained previously, the `-ServiceImpl` configuration overrides the `-Service` interface configuration during registration.

Liferay Portal, however, does not scan all available classes for the annotations. Instead, it only scans services. More precisely, it scans all classes, including plugin classes, registered in the portal's application context. All classes that are available to the `BeanLocator` are scanned. Practically, this means that the portal scans all classes registered in its Spring context and the Spring context of its plugins. If you use Service Builder to build plugin services, the services are automatically registered to the Spring context and are made available to the `BeanLocator`. Moreover, this means that you can register *any* object in the Spring context of your plugin and the portal scans it for remote services! We are not forcing you to use Service Builder. We recommend using it because it easily does so many things with regards to your remote services.



Note: Liferay's developers use *Service Builder* to expose their services via JSON automatically. If you're interested in using Service Builder, read Generating Your Service Layer.

OK, now let's see how you can create a plugin with some remote services. Keep in mind that Liferay developers use the very same mechanism so that Liferay Portal's services come enabled out-of-the-box.

6.5.1.1. Registering Plugin JSON Web Services

Let's say you have a portlet named `SupraSurf` that has some services. And you decide to expose them as remote services. After enabling the `remote-service` attribute on its `SurfBoard` entity, you rebuild the services. Service Builder regenerates the `SurfBoardService` interface, adding the `@JSONWebService` annotation to it. This annotation tells the portal that the interface's public methods are to be exposed as JSON web services, making them a part of the plugin's JSON API.

By default, scanning of the portlet's services is disabled. To enable scanning, you need to add an appropriate filter definition in the portlet's `web.xml` file. Fortunately, Liferay provides a way to automatically add the filter. Just click the *Build WSDD* button in Liferay IDE while editing the `service.xml` file in *Overview* mode, or just invoke the `build-wsdd` Ant target. On building the WSDD, Liferay's Plugins SDK modifies the portlet's `web.xml` and enables the JSON web services for the plugin. Under the hood, the Plugins SDK registers the `SecureFilter` and the `JSONWebServiceServlet` for the plugin. You only need to enable JSON web services for your plugin once.

Let's deploy the `SupraSurf` portlet plugin on our portal server. If your server isn't running, start it up. Then deploy your plugin onto it.

To get some feedback from the portal on registering your plugin's services, configure the portal to log the plugin's informational messages (i.e., its `INFO ...` messages). See the section on Liferay's logging system in *Using Liferay Portal*.

Let's add a simple method to the plugin's services. Edit the `SurfBoardServiceImpl` class and add the following method:

```
public String helloWorld(String worldName) {  
    return "Hello world: " + worldName;  
}
```

Rebuild the services and deploy the plugin. Notice that the portal prints a message like the one below informing us that an action was configured for the portlet. This indicates that the service method is now registered as a JSON Web Web Service action!

```
INFO [JSONWebServiceActionsManagerImpl:117] Configured 1 actions for\  
/suprasurf-portlet
```

This same mechanism registers Liferay Portal's own service actions. They are conveniently enabled by default, so you don't have to configure them.

Next, let's learn how to form a mapped URL for the remote service so we can access it.

6.5.1.2. Mapping and Naming Conventions

You can form the mapped URL of an exposed service by following the naming convention below:

```
http://[server]:[port]/api/jsonws/[plugin-context-name.][service-class-  
name]/[service-method-name]
```

Let's look at the last three bracketed items more closely:

- `plugin-context-name` is the plugin's context name (e.g., `suprasurf-portlet`

in our example). For the portal's services, this part is not needed.

- `service-class-name` is generated from the service's class name in lower case, minus its `Service` or `ServiceImpl` suffix. For example, specify `surfboard` as the `plugin-context-name` for the `SurfBoardService` class.
- `service-method-name` is generated from the service's method name by converting its camel case to lower case and using dashes (-) to separate words.

We'll demonstrate these naming conventions by mapping a service method's URL using the naming conventions both on a created plugin service and on a portal service.

For our created service method, the URL looks like:

`http://localhost:8080/api/jsonws/suprasurf-portlet.surfboard/hello-world`

Note the context name part of the URL. For the portal, it's similar. Here's a portal service method we want to access:

```
@JSONWebService  
public interface UserService {  
    public com.liferay.portal.model.User getUserId(long userId) {...}
```

Here's is that portal service method's URL:

`http://localhost:8080/api/jsonws/user-service/get-user-by-id`

Each service method is bound to one HTTP method type. Any method with a name starting with `get`, `is`, or `has` is assumed to be a read-only method and is mapped as a *GET HTTP* method by default. All other methods are mapped as *POST HTTP* methods.

As you may have noticed, plugin services are accessed via the portal context. Conveniently, requests sent this way can leverage the user's authentication in his current portal session.

Next, we'll learn to how to *list* JSON web services available from our portal.

6.5.1.3. Listing Available JSON Web Services

To see which service methods are registered and available for use, open your browser to the following address:

`http://localhost:8080/api/jsonws`

The API page lists the portal's registered and exposed service methods. To get each method's details, click on the method name. You'll see the full signature of the method, all of its arguments, and a list of exceptions that can be thrown. For additional information about remote service methods, you can look up the method in Liferay Portal's Javadocs. Using a simple form from within your browser, you can even invoke the service method for testing purposes.

The same API page lists remote services of plugins, too. When multiple plugins with remote services enabled are deployed, the API page shows a select box with all available plugin context paths (including the portal's path). The select box facilitates switching between the plugins' list of remote services and the portal's list of remote services.

If you've been paying attention, you already know how to control registration by using the `@JSONWebService` annotation in your `-ServiceImpl` class. This overrides any configuration defined in the interface. What you might not know is that you can control the visibility of methods using annotations at the method level.

Let's find out how to ignore a specific method.

6.5.1.4. Ignoring a Method

To keep a method from being exposed as a service, annotate the method with the following option:

```
@JSONWebService(mode = JSONWebServiceMode.IGNORE)
```

Methods with this annotation don't become part of the JSON Web Service API.

Let's learn to define custom HTTP method names and URL names.

6.5.1.5. HTTP Method Name and URL

At the method level, you can define custom HTTP method names and URL names. Just use an annotation like this one:

```
@JSONWebService(value = "add-board-wow", method = "PUT")
public boolean addBoard()
```

In this example, the plugin's service method `addBoard` is mapped to URL method name `add-board-wow`. Its complete URL is now

```
http://localhost:8080/api/jsonws/suprasurf-
portlet.surfboard/add-board-wow
```

 and can be accessed using the HTTP PUT method.

If the URL name starts with a slash character (/), only the method name is used to form the service URL; the class name is ignored.

```
@JSONWebService("/add-something-very-specific")
public boolean addBoard()
```

Similarly, you can change the class name part of the URL, by setting the value in class-level annotation:

```
@JSONWebService("sbs")
public class SurfBoardServiceImpl extends SurfBoardServiceBaseImpl {
```

This maps all of the service's methods to a URL class name `sbs` instead of the default class name `surfboard`.

Next, we'll show you a different approach to exposing your methods as we discuss manual registration.

6.5.1.6. Manual Registration Mode

Up to now, it is assumed that you want to expose most of your service methods, while hiding some specific methods (the *blacklist* approach). Sometimes, however, you want the opposite: to explicitly specify only those methods that are to be exposed (the *whitelist* approach). This is possible, too, by specifying *manual mode* on the class-level annotation. Then it is up to you to annotate only those methods that you want exposed.

```
@JSONWebService(mode = JSONWebServiceMode.MANUAL)
public class SurfBoardServiceImpl extends SurfBoardServiceBaseImpl{
    ...
    @JSONWebService
    public boolean addBoard()
```

Now only the `addBoard` method and any other method annotated with `@JSONWebService` will be part of the JSON Web Service API; all other methods of this service will be excluded

from the API.

Next, let's look at portal configuration options that apply to JSON Web Services.

6.5.2. Portal Configuration of JSON Web Services

JSON web services are enabled on Liferay Portal by default. If you need to disable them, specify this portal property setting:

```
json.web.service.enabled=false
```

Now let's look at strict HTTP methods.

6.5.2.1. Strict HTTP Methods

All JSON web services are mapped to either GET or POST HTTP methods. If a service method name starts with get, is or has, the service is assumed to be read-only and is bound to the GET method; otherwise it's bound to POST.

By default, Liferay Portal doesn't check HTTP methods when invoking a service call; it works in *non-strict http method* mode, where services may be invoked using any HTTP method. If you need the strict mode, you can set it with this portal property:

```
jsonws.web.service.strict.http.method=true
```

When using strict mode, you must use the correct HTTP methods in calling service methods.

When strict HTTP mode is enabled, you still might have need to disable HTTP methods. We'll show you how next.

6.5.2.2. Disabling HTTP Methods

When strict HTTP method mode is enabled, you can filter web service access based on HTTP methods used by the services. For example, you can set the portal JSON web services to work in read-only mode by disabling HTTP methods other than GET. For example:

```
jsonws.web.service.invalid.http.methods=DELETE,POST,PUT
```

Now all requests that use HTTP methods from the list above are ignored.

Next, we'll show you how to restrict public access to exposed JSON APIs.

6.5.2.3. Controlling Public Access

Each service method knows if it can be executed by unauthenticated users and if a user has adequate permission for the chosen action. Most of the portal's read-only methods are open to public access.

If you're concerned about security, you can further restrict public access to exposed JSON APIs by explicitly stating which methods are *public* (i.e., accessible to unauthenticated users). Use the following property to specify your public methods:

```
jsonws.web.service.public.methods=*
```

The property supports wildcards, so if you specify `get*, has*, is*` on the right hand side of the `=` symbol, all read-only JSON methods will be publicly accessible. All other JSON methods will be secured. To disable access to *all* exposed methods, you can leave the right side of the `=`

symbol empty; to enable access to all exposed methods, specify *.

Read on to find out how to invoke JSON web services.

6.5.3. Invoking JSON Web Services

How you invoke a JSON web service depends on how you pass in its parameters. We'll discuss how to pass in parameters below, but first you need to understand how your invocation is matched to a method, especially when a service method is overloaded.

The general rule is that you provide the method name and *all* parameters for that service method--even if you only provide null.

It's important to provide all parameters, but it doesn't matter *how* you do it (e.g., as part of the URL line, as request parameters, etc.). The order of the parameters doesn't matter either.



Note: An authentication related token must accompany each Liferay web service invocation. For details, read the section on service security layers found earlier in this chapter.

Exceptions abound in life, and there's an exception to the rule that *all* parameters are required--when using numeric *hints* to match methods. Let's look at using hints next.

6.5.3.1. Using Hints

Adding numeric hints lets you specify how many method arguments a service has. If you don't specify an argument for a parameter, it's automatically passed in as null. Syntactically, you can add hints as numbers separated by a dot in the method name. Here's an example:

/foo/get-bar.2/param1/123/-param2

Here, the .2 is a numeric hint specifying that only service methods with two arguments will be matched; others will be ignored for matching.

There's an important distinction to make between matching using hints and matching without hints. When a hint is specified, you don't have to specify all of the parameters. Any missing arguments are treated as null. The previous example may be called like this:

/foo/get-bar.2/param1/123

In this example, param2 will automatically be set to null.

Find out how to pass parameters as part of the URL path next.

6.5.3.2. Passing Parameters as Part of a URL Path

You can pass parameters as part of the URL path. After the service URL, just specify method parameters in name-value pairs. Parameter names must be formed from method argument names by converting them from camelCase to names using all lower case and separated-by-dash. Here's an example of calling one of the portal's services:

`http://localhost:8080/api/jsonws/dlapp/get-file-entries/repository-id/\10172/folder-id/0`

Note, we've inserted line escape character \ in order to fit the example URL on this page.

You can pass parameters in any order; it's not necessary to follow the order in which the arguments specified in the method signatures.

When a method name is overloaded, the *best match* will be used. It chooses the method that contains the least number of undefined arguments and invokes it for you.

You can also pass parameters in a URL query, and we'll show you how next.

6.5.3.3. Passing Parameters as a URL Query

You can pass in parameters as request parameters. Parameter names are specified as is (e.g. camelCase) and are set equal to their argument values, like this:

```
http://localhost:8080/api/jsonws/dlapp/get-file-entries?repositoryId=\10172&folderId=0
```

Note, we've inserted line escape character \ in order to fit the example URL on this page.

As with passing parameters as part of a URL path, the parameter order is not important, and the *best match* rule applies for overloaded methods.

Now you know a few different ways to pass parameters. It's also possible to pass URL parameters in a mixed way. Some can be part of the URL path and some can be specified as request parameters.

Parameter values are sent as strings using the HTTP protocol. Before a matching Java service method is invoked, each parameter value is converted from a `String` to its target Java type. Liferay uses a third party open source library to convert each object to its appropriate common type. It's possible to add or change the conversion for certain types but we'll just cover the standard conversions process.

Conversion for common types (e.g., `long`, `String`, `boolean`) is straightforward. Dates can be given in milliseconds; locales can be passed as locale names (e.g. `en` and `en_US`). To pass in an array of numbers, send a `String` of comma-separated numbers (e.g. `String 4, 8, 15, 16, 23, 42` can be converted to `long[]` type). You get the picture!

In addition to the common types, arguments can be of type `List` or `Map`. To pass a `List` argument, send a JSON array. To pass a `Map` argument, send a JSON object. The conversion of these is done in two steps, ingeniously referred to below as *Step 1* and *Step 2*:

- *Step 1--JSON deserialization*: JSON arrays are converted into `List<String>` and JSON objects are converted to `Map<String, String>`. For security reasons, it is forbidden to instantiate any type within JSON deserialization.
- *Step 2--Generification*: Each `String` element of the `List` and `Map` is converted to its target type (the argument's generic Java type specified in the method signature). This step is only executed if the Java argument type uses generics.

As an example, let's consider the conversion of `String` array [`en`, `fr`] as JSON web service parameters for a `List<Locale>` Java method argument type:

- *Step 1--JSON deserialization*: The JSON array is deserialized to a `List<String>` containing `Strings en and fr`.

- *Step 2--Generification:* Each String is converted to the Locale (the generic type), resulting in the List<Locale> Java argument type.

Now let's see how to specify an argument as null.

6.5.3.4. Sending NULL Values

To pass a null value for an argument, prefix the parameter name with a dash. Here's an example:

```
.../dlsync/get-d-1-sync-update/company-id/10151/repository-id/10195/-last-access-date
```

The last-access-date parameter is interpreted as null. Although we have it last in the URL above, it's not necessary.

Null parameters don't have specified values. When a null parameter is passed as a request parameter, its value is ignored and null is used instead:

```
<input type="hidden" name="-last-access-date" value="" />
```

When using JSON RPC (see below), you can send null values explicitly, even without a prefix.

Here's an example:

```
"last-access-date":null
```

Now let's learn about encoding parameters.

6.5.3.5. Encoding Parameters

There's a difference between URL encoding and query (i.e. request parameters) encoding. The difference lies in how the space character is encoded. When the space character is part of the URL path, it's encoded as %20; when it's part of the query it's encoded as a plus sign (+).

All these encoding rules apply to ASCII and international (non-ASCII) characters. Since Liferay Portal works in UTF-8 mode, parameter values must be encoded as UTF-8 values. Liferay Portal doesn't decode request URLs and request parameter values to UTF-8 itself; it relies on the web server layer. When accessing services through JSON-RPC, encoding parameters to UTF-8 isn't enough--you need to send the encoding type in a Content-Type header (e.g. Content-Type : "text/plain; charset=utf-8").

For example, let's pass the value "Супер" ("Super" in Cyrillic) to some JSON Web Service method. This name first has to be converted to UTF-8 (resulting in array of 10 bytes) and then encoded for URLs or request parameters. The resulting value is the string %D0%A1%D1%83%D0%BF%D0%B5%D1%80 that can be passed to our service method. When received, this value is first going to be translated to an array of 10 bytes (URL decoded) and then converted to a UTF-8 string of the 5 original characters.

Did you know you can send files as arguments? Find out how next.

6.5.3.6. Sending Files as Arguments

Files can be uploaded using multipart forms and requests. Here's an example:

```
<form  
action="http://localhost:8080/api/jsonws/dlapp/add-file-entry"
```

```

method="POST"
enctype="multipart/form-data">
<input type="hidden" name="repositoryId" value="10172"/>
<input type="hidden" name="folderId" value="0"/>
<input type="hidden" name="title" value="test.jpg"/>
<input type="hidden" name="description" value="File upload example"/>
<input type="hidden" name="changeLog" value="v1"/>
<input type="file" name="file"/>
<input type="submit" value="addFileEntry(file)"/>
</form>

```

This is a common upload form that invokes the `addFileEntry` method of the `DLAppService` class.

Now we'll show you how to invoke JSON web services using JSON RPC.

6.5.3.7. JSON RPC

You can invoke JSON Web Service using JSON RPC. Most of the JSON RPC 2.0 specification is supported in Liferay JSON web services. One important limitation is that parameters must be passed in as *named* parameters. Positional parameters are not supported, as there are too many overloaded methods for convenient use of positional parameters.

Here's an example of invoking a JSON web service using JSON RPC:

```

POST http://localhost:8080/api/jsonws/dlapp
{
    "method": "get-folders",
    "params": {"repositoryId": 10172, "parentFolderId": 0},
    "id": 123,
    "jsonrpc": "2.0"
}

```

Let's talk about parameters that are made available to secure JSON web services by default.

6.5.3.8. Default Parameters

When accessing *secure* JSON web services (i.e., the user has to be authenticated), some parameters are made available to the web services by default. Unless you want to change their values to something other than their defaults, you don't have to specify them explicitly.

Here are the default parameters:

- `userId`: The id of authenticated user
- `user`: The full User object
- `companyId`: The users company
- `serviceContext`: The empty service context object

Let's find out about object parameters next.

6.5.3.9. Object Parameters

Most services accept simple parameters like numbers and strings. However, sometimes you might need to provide an object (a non-simple type) as a service parameter.

To create an instance of an object parameter, prefix the parameter with a plus sign, + and don't assign it any other parameter value. This is similar to when we specified a null parameter by prefixing the parameter with a dash symbol, -.

Here's an example:

```
/jsonws/foo/get-bar/zap-id/10172/start/0/end/1/+foo
```

To create an instance of an object parameter as a request parameter, make sure you encode the + symbol:

```
/jsonws/foo/get-bar?zapId=10172&start=0&end=1&%2Bfoo
```

Here's an alternative syntax:

```
<input type="hidden" name="+foo" value="" />
```

If a parameter is an abstract class or an interface, it can't be instantiated as such. Instead, a concrete implementation class must be specified to create the argument value. You can do this by specifying the + prefix before the parameter name followed by specifying the concrete implementation class. Check out this example:

```
/jsonws/foo/get-bar/zap-id/10172/start/0/end/1/+foo:com.liferay.impl.FooBean
```

Here's another way of doing it:

```
<input type="hidden" name="+foo:com.liferay.impl.FooBean" value="" />
```

The examples above specify that a com.liferay.impl.FooBean object, presumed to implement the class of the parameter named foo, is to be created.

You can also set a concrete implementation as a value. Here's an example:

```
<input type="hidden" name="+foo" value="com.liferay.impl.FooBean" />
```

In JSON RPC, here's what it looks like:

```
"+foo" : "com.liferay.impl.FooBean"
```

All the examples above specify a concrete implementation for the foo service method parameter.

Once you pass in an object parameter, you might want to populate the object. Find out how next.

6.5.3.10. 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 (see the ServiceContext section of the next chapter to find out more about this type). To make an appropriate call to JSONWS, you might need to set the serviceContext parameter's fields addGroupPermissions and scopeGroupId.

You can pass inner parameters by specifying them using dot notation. Just append the name of the parameter with a dot (i.e., a period, .), followed by the name of the inner parameter. For the ServiceContext inner parameters we mentioned above, you'd specify serviceContext.addGroupPermissions and serviceContext.scopeGroupId. They're recognized as inner parameters and their values are injected into existing parameters before the API service method is executed.

Inner parameters aren't counted as regular parameters for matching methods and are ignored during matching.



Tip: Use inner parameters with object parameters to set inner content of created parameter instances!

Next, let's see what values are returned when a JSON web service is invoked.

6.5.4. Returned Values

No matter how a JSON web service is invoked, it returns a JSON string that represents the service method result. Returned objects are *loosely* serialized to a JSON string and returned to the caller.

Let's look at some values returned from service calls. We'll create a `UserGroup` as we did in our SOAP web service client examples. To make it easy, we'll use the test form provided with the JSON web service in our browser.

1. Sign in to your portal as an administrator and then point your browser to the JSON web service method that adds a `UserGroup`:

```
http://127.0.0.1:8080/api/jsonws?signature=/usergroup/add-user-group-2-\name-description
```

Note, we've inserted line escape character \ in order to fit the example URL on this page.

Alternatively, navigate to it by starting at `http://127.0.0.1:8080/api/jsonws` then scrolling down to the section for `UserGroup`; click `add-user-group`.

2. In the `name` field, enter `UserGroup3` and set the description to an arbitrary value like `Created via JSON WS`.
3. Click `Invoke` and you'll get a result similar to the following:

```
{
  "addedByLDAPImport": false,
  "companyId": 10154,
  "createDate": 1382460167254,
  "description": "Created via JSON WS",
  "modifiedDate": 1382460167254,
  "name": "UserGroup3",
  "parentUserGroupId": 0,
  "userGroupId": 13901,
  "userId": 10198,
  "userName": "Test Test",
  "uuid": "1b18c73d-482a-4772-b6f4-a9253bbcbf92"
}
```

The returned String represents the `UserGroup` object you just created, serialized into a JSON string. To find out more about JSON strings, go to json.org.

Let's check out some common JSON WebService errors.

6.5.5. Common JSON Web Service Errors

While working with JSON web services, you may encounter errors. Let's look at the most common errors in the following subsections.

- Authenticated access required

If you see this error, it means you don't have permission to invoke the remote service. Double-check that you're signed in as a user with the appropriate permissions. If necessary, sign in as an administrator to invoke the remote service.

- Missing value for parameter

If you see this error, you didn't pass a parameter value along with the parameter name in your URL path. The parameter value must follow the parameter name, like in this example:

```
/api/jsonws/user/get-user-by-id/userId
```

The path above specifies a parameter named `userId`, but doesn't specify the parameter's value. You can resolve this error by providing the parameter value after the parameter name:

```
/api/jsonws/user/get-user-by-id/userId/173
```

- No JSON web service action associated

This error means no service method could be matched with the provided data (method name and argument names). This can be due to various reasons: arguments may be misspelled, the method name may be formatted incorrectly, etc. Since JSON web services reflect the underlying Java API, any changes in the respective Java API will automatically be propagated to the JSON web services. For example, if a new argument is added to a method or an existing argument is removed from a method, the parameter data must match that of the new method signature.

- Unmatched argument type

This error appears when you try to instantiate a method argument using an incompatible argument type.

- Judgment Day

We hope you never see this error. It means that Skynet has initiated a nuclear war and most of humanity will be wiped out; survivors will need to battle *Terminator* cyborgs. If you see this error and survive *Judgment Day*, we recommend joining the resistance--they'll likely need good developers to support the cause, especially those familiar with time travel.

Had you going there, didn't we?

Next, we'll show you how to optimize your use of JSON web services by using the *JSON Web Services Invoker*.

6.5.6. JSON Web Services Invoker

Using JSON web services is easy: you send a request that defines a service method and parameters, and you receive the result as JSON object. Below we'll show you why that's not optimal, and introduce a tool that lets you use JSON web services more efficiently and pragmatically.

Consider this example: You're working with two related objects: a `User` and its corresponding `Contact`. With simple JSON Web Service calls, you first call some user service to get the user object and then you call the contact service using the contact ID from the user object. You end up sending two HTTP requests to get two JSON objects that aren't even bound together; there's no

contact information in the user object (i.e. no `user.contact`). This approach is suboptimal for performance (sending two HTTP calls) and usability (manually managing the relationship between two objects). It'd be nicer if you had a tool to address these inefficiencies. Fortunately, the *JSON Web Service Invoker* does just that!

Liferay's JSON Web Service Invoker helps you optimize your use of JSON Web Services. In the following sections, we'll show you how.

6.5.6.1. A Simple Invoker Call

The Invoker is accessible from the following fixed address:

`http://[address]:[port]/api/jsonws/invoke`

It only accepts a `cmd` request parameter--this is the Invoker's command. If the command request parameter is missing, the request body is used as the command. So you can specify the command by either using the request parameter `cmd` or the request body.

The Invoker command is a plain JSON map describing how JSON web services are called and how the results are managed. Here's an example of how to call a simple service using the Invoker:

```
{  
    "/user/get-user-by-id": {  
        "userId": 123,  
        "param1": null  
    }  
}
```

The service call is defined as a JSON map. The key specifies the service URL (i.e. the service method to be invoked) and the key's value specifies a map of service parameter names (i.e. `userId` and `param1`) and their values. In the example above, the retrieved user is returned as a JSON object. Since the command is a JSON string, null values can be specified either by explicitly using the `null` keyword or by placing a dash before the parameter name and leaving the value empty (e.g. `"-param1": ''`).

The example Invoker calls functions exactly the same as the following standard JSON Web Service call:

`/user/get-user-by-id?userId=123&-param1`

Of course, JSON Web Service invoker handles calls to plugin methods as well:

```
{  
    "/suprasurf-portlet.surfboard/hello-world": {  
        "worldName": "Mavericks"  
    }  
}
```

The code above calls our plugin's remote service.

You can use variables to reference objects returned from service calls. Variable names must start with a dollar sign, `$`. In our previous example, the service call returned a user object you can assign to a variable:

```
{  
    "$user = /user/get-user-by-id": {  
        "userId": 123,  
    }
```

```
    }
}
```

The `$user` variable holds the returned user object. You can reference the user's contact ID using the syntax `$user.contactId`.

Next, see how you can use nested service calls to join information from two related objects.

6.5.6.2. Nesting Service Calls

With nested service calls, you can magically bind information from related objects together in a JSON object. You can call other services within the same HTTP request and nest returned objects in a convenient way. Here's the magic of a nested service call in action:

```
{
    "$user = /user/get-user-by-id": {
        "userId": 123,
        "$contact = /contact/get-contact-by-id": {
            "@contactId" : "$user.contactId"
        }
    }
}
```

This command defines two service calls: the contact object returned from the second service call is nested in (i.e. injected into) the user object, as a property named `contact`. Now we can bind the user and his or her contact information together!

Let's see what the Invoker did in the background when we used a single HTTP request to make the above nested service call:

- First, the Invoker called the Java service mapped to `/user/get-user-by-id`, passing in a value for the `userId` parameter.
- Next, the resulting user object was assigned to the variable `$user`.
- The nested service calls were invoked.
- The Invoker called the Java service mapped to `/contact/get-contact-by-id` by using the `contactId` parameter, with the `$user.contactId` value from the object `$user`.
- The resulting contact object was assigned to the variable `$contact`.
- Lastly, the Invoker injected the contact object referenced by `$contact` into the user object's property named `contact`.



Note: You must *flag* parameters that take values from existing variables. To flag a parameter, insert the `@` prefix before the parameter name.

Next, let's talk about filtering object properties so only those you need are returned when you invoke a service.

6.5.6.3. Filtering Results

Many of Liferay Portal's model objects are rich with properties. If you only need a handful of an

object's properties for your business logic, making a web service invocation that returns all of an object's properties is a waste of network bandwidth. With the JSON Web Service Invoker, you can define a *white-list* of properties: only the specific properties you request in the object will be returned from your web service call. Here's how you white-list the properties you need:

```
{  
    "$user(firstName, emailAddress) = /user/get-user-by-id": {  
        "userId": 123,  
        "$contact = /contact/get-contact-by-id": {  
            "@contactId" : "$user.contactId"  
        }  
    }  
}
```

In this example, the returned user object has only the `firstName` and `emailAddress` properties (it still has the `contact` property, too). To specify white-list properties, you simply place the properties in square brackets (e.g., `[whiteList]`) immediately following the name of your variable.

Let's talk about batching calls next.

6.5.6.4. Batching Calls

When we nested service calls earlier, the intent was to invoke multiple services with a single HTTP request. Using a single request for multiple service calls is helpful for gathering related information from the service call results, but it can also be advantageous to use a single request to invoke multiple unrelated service calls. The Invoker lets you batch service calls together to improve performance. It's simple: just pass in a JSON array of commands using the following format:

```
[  
    {/* first command */},  
    {/* second command */}  
]
```

The result is a JSON array populated with results from each command. The commands are collectively invoked in a single HTTP request, one after another.

By learning to leverage JSON web services in Liferay, you've added some powerful tools to your toolbox. Good job! Next, let's learn how to implement OAuth so you can access third-party services.

6.6. Authorizing Access to Services with OAuth

Suppose you wanted users to authenticate to your Liferay Portal plugin from a provider, like Twitter. You might think that you'd need to store the user's credentials (e.g., the user's Twitter account name and password), so you could pass them along with requests to the service provider and log them in. But this opens up a can of worms. The third party is a moving target: what happens when they modify their site, and your slick authenticator stops working? Additionally, you might receive some criticism from your users for asking them to give their Twitter credentials to you so you can use them to log in. Sounds like a hassle, right? This is where OAuth comes into play, taking a approach that is safe and simple.

OAuth delegates user authentication to the service provider. An OAuth-enabled plugin uses a token to prove it is authorized to access the user's third-party profile data and invoke authorized services. By implementing OAuth in your plugin, you get the best of both worlds--access to an outside service provider, and your users' trust that the plugin won't have access to their protected resources.

In addition, Liferay Portal instances can act as OAuth service providers: you can provide a means for your users to use their portal credentials to access other services that have OAuth configured. We refer to such portals as *Liferay Service Portals*. The OAuth framework lets Liferay Service Portal administrators specify well-defined service authorizations. Once authorized, the users can invoke the services via OAuth clients, such as the OAuth-enabled plugin that you'll learn about in this section.



Note: To learn more about the OAuth framework, Liferay OAuth app, registering your OAuth app, or activating it from a portal page, visit the OAuth section of *Using Liferay Portal*.

To access portal services using OAuth, you'll need to create a client that uses an OAuth cycle implementation, along with a user interface to lead your users through the cycle. In this section, you'll see an example of a portlet accessing JSON Web Services from a remote portal. Let's get started by first selecting and implementing services of an OAuth Client library.

6.6.1. Selecting an OAuth Client Library

In order for your portlet to use OAuth, it must have a reference to OAuth standards for authorization. You can offer your portlet an OAuth client library by specifying a single JAR file. In this example, Scribe is chosen as the OAuth library because it's available in Liferay Portal and can be easily included in a plugin. To use the Scribe OAuth client library, open your plugin's `liferay-plugin-package.properties` file and insert the `scribe.jar` file as a portal dependency jar:

```
portal-dependency-jars=\  
    scribe.jar
```

That's all you have to do! Your portlet now has access to Scribe's OAuth library. Next, you'll implement Scribe's OAuth service interface.

6.6.2. Configuring OAuth's Service Implementation

Now that your portlet can access an OAuth client library, you need to implement the OAuth services in your portlet. The following code demonstrates implementing a Scribe OAuth service API:

```
import org.scribe.builder.api.DefaultApi10a;  
...  
  
public class OAuthAPIImpl extends DefaultApi10a {  
  
    @Override
```

```

protected String getAccessTokenEndpoint() {
    if (Validator.isNull(_accessTokenEndpoint)) {
        _accessTokenEndpoint = OAuthUtil.buildURL(
            "oauth-portal-host", 80, "http",
            PortletPropsValues.OSB_LCS_PORTLET_OAUTH_ACCESS_TOKEN_URI);
    }

    return _accessTokenEndpoint;
}

@Override
protected String getRequestTokenEndpoint() {
    if (Validator.isNull(_requestTokenEndpoint)) {
        _requestTokenEndpoint = OAuthUtil.buildURL(
            "oauth-portal-host", 80, "http",
            PortletPropsValues.OSB_LCS_PORTLET_OAUTH_REQUEST_TOKEN_URI);
    }

    return _requestTokenEndpoint;
}

private String _accessTokenEndpoint;
private String _requestTokenEndpoint;
}

```

In this code snippet, the portlet provides the service platform's OAuth URLs to Scribe to acquire the access token and request token from the service provider. A *request token* is a value the portlet uses to obtain user authorization. It is exchanged for an *access token*. The access token is a value the portlet uses to gain access to protected resources on behalf of the user. The exchange of a request token for an access token replaces the need for supplying the user's service provider credentials.

In addition to the tokens, you'll also need to provide the callback URL so that the service platform can redirect the user's browser back to your portlet, once authentication and authorization is complete. The callback URL can be provided in an authorization request as a parameter, or it can be specified when registering your application through Liferay's OAuth Admin menu. Keep in mind that a callback URL provided via an authorization parameter overrides the callback setting specified in the OAuth Admin menu. You can specify the callback URL as an authorization parameter in your portlet's `portlet.properties` file. You'll see this process later. Here's a code snippet that uses the callback URL and request token in acquiring the OAuth Service:

```

public class OAuthUtil {

    public static String buildURL(
        String hostName, int port, String protocol, String uri) {
        ...
    }

    public static Token extractAccessToken(
        Token requestToken, String oAuthVerifier) {

```

```

Verifier verifier = new Verifier(oAuthVerifier);

OAuthService oAuthService = getOAuthService();

return oAuthService.getAccessToken(requestToken, verifier);
}

public static String getAuthorizeURL(
    String callbackURL, Token requestToken) {

    if (Validator.isNull(_authorizeRequestURL)) {
        authorizeRequestURL = buildURL(
            "oauth-portal-host", 80, "http",
            PortletPropsValues.OSB_LCS_PORTLET_OAUTH_AUTHORIZE_URI);

        if (Validator.isNotNull(callbackURL)) {
            authorizeRequestURL = HttpUtil.addParameter(
                authorizeRequestURL, "oauth_callback",
                callbackURL);
        }
    }

    _authorizeRequestURL.replace("{0}", requestToken.getToken());
}

public static OAuthService getOAuthService() {
    if (_oAuthService == null) {
        ServiceBuilder oAuthServiceBuilder = new ServiceBuilder();

        oAuthServiceBuilder.apiKey(
            PortletPropsValues.OSB_LCS_PORTLET_OAUTH_CONSUMER_KEY);
        oAuthServiceBuilder.apiSecret(
            PortletPropsValues.OSB_LCS_PORTLET_OAUTH_CONSUMER_SECRET);
        oAuthServiceBuilder.provider(OAuthAPIImpl.class);

        _oAuthService = oAuthServiceBuilder.build();
    }

    return _oAuthService;
}

public static Token getRequestToken() {
    OAuthService oAuthService = getOAuthService();

    return oAuthService.getRequestToken();
}

private static String _authorizeRequestURL;
private static OAuthService _oAuthService;
}

```

Besides authorizing the callback URL, you're also implementing methods to acquire the OAuth service, submit the request to that service, and obtain tokens from the service. By doing this, you

provide OAuth services to your portlet. You're not quite done yet; you still need to provide information about the OAuth platform you're accessing.

First, you need to specify the OAuth protocol context paths for your URLs. In the case of using Liferay Portal as a service platform, the default paths for the OAuth portlet are specified in the auth.public.paths portal property found in the Authentication Pipeline section of Portal's portal.properties file. The URLs specified here do not require authentication to access.

```
auth.public.paths=\n    /portal/oauth/access_token,\n    /portal/oauth/authorize,\n    /portal/oauth/request_token
```

You'll need to transfer these OAuth related constants to your portlet's portlet.properties file. Here's an example code snippet of what these property settings look like:

```
oauth.access.token.uri=/c/portal/oauth/access_token\noauth.authorize.uri=/c/portal/oauth/authorize?oauth_token={0}\noauth.consumer.key=42c56e22-d5a2-4003-86f4-cbc34b6de3e3\noauth.consumer.secret=793195c2936a85649042b24ed843a036\noauth.request.token.uri=/c/portal/oauth/request_token
```

Great! Now your OAuth services are implemented and OAuth constants are specified. Your portlet can now take part in the OAuth authorization process! You'll just need to set up a simple user interface to start the OAuth cycle. Let's do this next!

6.6.3. Creating a User Interface for Authentication

Your portlet's user interface must initiate the OAuth cycle the first time it accesses the OAuth platform for each specific user. Your portlet must render the OAuth authorization UI automatically when the portlet does not possess the access token and access secret. The JSP code snippet below initiates the OAuth authorization process:

```
<portlet:actionurl name="setupOAuth" var="setupOAuthURL">\n<%\nToken requestToken = OAuthUtil.getRequestToken();\n\nportletSession.setAttribute(Token.class.getName(), requestToken);\n%>\n<div class="button-container">\n    <a class="lcs-portal-link" href="<%=\nOAuthUtil.getAuthorizeURL(setupOAuthURL, requestToken) %><liferay-\nui:message key="authorize-access"/>\n    </a>\n</div>
```

On successfully getting authorization from the service provider, the OAuth platform redirects the user back to the callback URL, which in this case is a URL for the setupOAuth portlet action. This action method uses the request token to get the access token. It stores the token secret and the token itself. Here's a snippet of the portlet action method:

```
public void setupOAuth(\n    ActionRequest actionRequest, ActionResponse actionResponse)\nthrows Exception {\n\n    PortletSession portletSession = actionRequest.getPortletSession();
```

```

Token requestToken = (Token)portletSession.getAttribute(
    Token.class.getName());

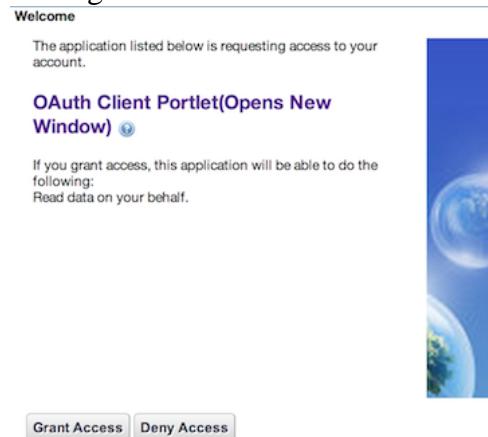
String oAuthVerifier = ParamUtil.getString(
    actionRequest, "oauth_verifier");

Token token = OAuthUtil.extractAccessToken(requestToken, oAuthVerifier);

// store token.getSecret() and token.getToken()
...
}

```

The figure below shows the OAuth authorization user interface.



On completing initial OAuth authorization via the UI and on the user revisiting the portlet instance thereafter, the portlet should render its normal UI. Once your portlet is granted access, the OAuth platform redirects the user back to the callback URL you specified during the portlet's registration.

Once you have the access token and access secret stored, your portlet can use them to access services such as JSON web services. Here's a simple code example for this scenario:

```

Token token = new Token(getAccessToken(), getAccessSecret());

String requestURL = OAuthUtil.buildURL(
    "oauth-portal-host", 80, "http",
    "/api/secure/jsonws/context.service/method/parms");

OAuthRequest oAuthRequest = new OAuthRequest(Verb.POST, requestURL);

OAuthService oAuthService = OAuthUtil.getOAuthService();

oAuthService.signRequest(token, oAuthRequest);

Response response = oAuthRequest.send();

if (response.getCode() == HttpServletResponse.SC_UNAUTHORIZED) {
    String value = response.getHeader("WWW-Authenticate");

    throw new CredentialException(value);
}

if (response.getCode() == HttpServletResponse.SC_OK) {
    // do something with results from response.getBody();
}

```

That's it! You've implemented an OAuth client library, created a service implementation, and developed a user interface to present the OAuth cycle. Of course, this example and its code

snippets are not compatible for all use cases, but they demonstrate configuring an OAuth-ready application for Liferay Portal.

6.7. Summary

In this chapter, you saw how easy it is to find and invoke Liferay remote web services. We also explained how Liferay's service security layers are used to protect your data and prevent unauthorized service calls. Then, we dove into SOAP web services and showed you how to create SOAP web service clients to invoke them. Lastly, we jumped into JSON web services. We learned how to register them, list them, and invoke them from Liferay JSON web service interface. Next, we learned about several different URL patterns and ways to pass JSON web service parameters in service calls. Lastly, we explored the world of OAuth and explained how to configure a Liferay application to use the OAuth platform. You see, here at Liferay, we aim to give you terrific service!

Next, we'll take a look at some of the powerful frameworks of Liferay Portal, learn how they work and how you can leverage them.

7. Using Liferay Frameworks

Picture a hot, summer day. You're on vacation, and you're just coming back from the beach after a day of frolicking on the sand and in the water. After all that activity, you're hungry. Time to grill up some burgers and dogs.

To grill hamburgers and hot dogs, you have to have a proper procedure and apparatus for accomplishing the task. The *procedure* would be called an algorithm in computer science terms. The *apparatus* is the framework.

You first need a grill. That grill should be equipped with a heating mechanism; in the case of most grills, that's either charcoal or propane gas. Obviously, it also has a metal frame, or grill, placed near the heat. You also need tools, such as a spatula, a plate to hold the meat, and if you're making chicken in addition to those burgers, a brush and some barbecue sauce.

All of these together form a *framework* for making grilled food on a hot summer day. All the tools you need are at your disposal; you just need to pick them up and get grilling! If the framework is already in place, it's obviously a lot easier (and more timely) to cook food than it would be if the framework weren't there. Just ask the cave men.

Liferay contains several frameworks that give you all the tools you need to perform various common tasks, such as handling permissions, letting users enter comments, categories, and tags, and other common tasks that Liferay doesn't make you have to write yourself.

This chapter covers the following topics:

- ServiceContext
- Security and Permissions
- Asset Framework
- Recycle Bin
- Message Bus
- Device Detection

Let's get cookin' with Liferay's `ServiceContext` class next.

7.1. ServiceContext

The `ServiceContext` class is a parameter class used for passing contextual information for a service. Using a parameter class lets you consolidate many different methods with different sets of optional parameters into a single, easier-to-use method. The class also aggregates information necessary for features used throughout Liferay's core portlets, such as permissioning, tagging, categorization, and more.

In this section we'll look at the Service Context fields, learn how to create and populate a Service Context, and learn to access Service Context data.

First we'll look at the fields of the `ServiceContext` class.

7.1.1. Service Context Fields

The `ServiceContext` class has many fields. The best field descriptions are found in the Javadoc:

<http://docs.liferay.com/portal/6.2/javadocs/com/liferay/portal/service/ServiceContext.html>.

Here we'll give you a helpful categorical listing of the fields:

- Actions:
 - `_command`
 - `_workflowAction`
- Attributes:
 - `_attributes`
 - `_expandoBridgeAttributes`
- Classification:
 - `_assetCategoryIds`
 - `_assetTagNames`
- IDs and Scope:
 - `_companyId`
 - `_portletPreferencesIds`
 - `_plid`
 - `_scopeGroupId`
 - `_userId`
 - `_uuid`
- Language:
 - `_languageId`
- Miscellaneous:
 - `_headers`
 - `_signedIn`
- Permissions:
 - `_addGroupPermissions`
 - `_addGuestPermissions`

- › `_deriveDefaultPermissions`
- › `_groupPermissions`
- › `_guestPermissions`
- Resources:
 - › `_assetEntryVisible`
 - › `_assetLinkEntryIds`
 - › `_createDate`
 - › `_indexingEnabled`
 - › `_modifiedDate`
- URLs, paths and addresses:
 - › `_currentURL`
 - › `_layoutFullURL`
 - › `_layoutURL`
 - › `_pathMain`
 - › `_portalURL`
 - › `_remoteAddr`
 - › `_remoteHost`
 - › `_userDisplayURL`

Are you wondering how the `ServiceContext` fields get populated? Good! We'll show you that next.

7.1.2. Creating and Populating a Service Context

Although all the `ServiceContext` class fields are optional, services that store any type of content need the scope group ID specified, at least. Here's a simple example of creating a `ServiceContext` instance and passing it as a parameter to a service API using Java:

```
ServiceContext serviceContext = new ServiceContext();
serviceContext.setScopeGroupId(myGroupId);
...
BlogsEntryServiceUtil.addEntry(..., serviceContext);
```

If you invoke the service from a servlet, a Struts action or any other front-end end class which has access to the `PortletRequest`, use one of the

`ServiceContextFactory.getInstance(...)` methods. These methods create the `ServiceContext` object and automatically fill it with all necessary values. The above example looks different if you invoke the service from a servlet:

```
ServiceContext serviceContext =
    ServiceContextFactory.getInstance(BlogsEntry.class.getName(),
portletRequest);
BlogsEntryServiceUtil.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 your expando bridge attributes are set on the `ServiceContext`.

Next, let's see an example of accessing information from a `ServiceContext`.

7.1.3. Accessing Service Context Data

We'll use code snippets from `BlogsEntryLocalServiceImpl.addEntry(..., ServiceContext)` to show you how to access information from a `ServiceContext` and comment on how the context information is being used.

As we mentioned, your service needs a scope group ID from your `ServiceContext`. The same holds true for the blogs entry service because the scope group ID provides the scope of the blogs entry (the entity being persisted). For the blogs entry, the scope group ID is used in the following way:

- It's used as the `groupId` for the `BlogsEntry` entity.
- It's used to generate a unique URL for the blog entry.
- It's used to set the scope for comments on the blog entry.

Here are the corresponding code snippets:

```
long groupId = serviceContext.getScopeGroupId();
...
entry.setGroupId(groupId);
...
entry.setUrlTitle(getUniqueUrlTitle(entryId, groupId, title));
...

// Message boards
```

```
if (PropsValues.BLOGS_ENTRY_COMMENTS_ENABLED) {
    mbMessageLocalService.addDiscussionMessage(
        userId, entry.getUserName(), groupId,
        BlogsEntry.class.getName(), entryId,
        WorkflowConstants.ACTION_PUBLISH);
}
```

Can `ServiceContext` be used to access the UUID of the blog entry? Absolutely! Can you use `ServiceContext` to set the time the blog entry was added? You sure can. See here:

```
entry.setUuid(serviceContext.getUuid());
...
```

```
entry.setCreateDate(serviceContext.getCreateDate(now));
```

Can `ServiceContext` be used in setting permissions on resources? You bet! When adding a blog entry, you can add new permissions or apply existing permissions for the entry, like this:

```
// Resources

if (serviceContext.isAddGroupPermissions() ||
    serviceContext.isAddGuestPermissions()) {

    addEntryResources(
        entry, serviceContext.isAddGroupPermissions(),
        serviceContext.isAddGuestPermissions());
}
```

```

    else {
        addEntryResources(
            entry, serviceContext.getGroupPermissions(),
            serviceContext.getGuestPermissions());
    }
}

ServiceContext helps apply categories, tag names, and the link entry IDs to asset entries
too.

// Asset

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);
}

WorkflowHandlerRegistryUtil.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.

Let's look at Liferay's permissions system next.

7.2. Security and Permissions

The Java portlet standard defines a simple security scheme using portlet roles and their mapping to portal roles. On top of this, Liferay provides a fine-grained permissions system that you can use to implement access security in your custom portlets. Here, we'll give an overview of the standard Java security system, Liferay's permission system, and how to use them in your own portlets.

7.2.1. JSR Portlet Security

The JSR specification defines a means to specify roles used by portlets in their `portlet.xml` definitions. The role names themselves, however, are not standardized, so when these portlets run in Liferay, you'll recognize familiar role names. For example, the Liferay Blogs portlet

definition references the *guest*, *power-user*, and *user* roles:

```
<portlet>
    <portlet-name>33</portlet-name>
    <display-name>Blogs</display-name>
    <portlet-class>com.liferay.portlet.StrutsPortlet</portlet-class>
    <init-param>
        <name>view-action</name>
        <value>/blogs/view</value>
    </init-param>
    <init-param>
        <name>config-jsp</name>
        <value>/html/portlet/blogs/configuration.jsp</value>
    </init-param>
    <expiration-cache>0</expiration-cache>
    <supports>
        <mime-type>text/html</mime-type>
    </supports>
    <resource-bundle>com.liferay.portlet.StrutsResourceBundle</resource-
bundle>
    <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>
    <supported-public-render-parameter>
        categoryId
    </supported-public-render-parameter>
    <supported-public-render-parameter>
        tag
    </supported-public-render-parameter>
</portlet>
```

Your `portlet.xml` roles need to be mapped to specific roles in the portal. That way the portal can 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.

To map the roles to the portal, you'll have to use a Liferay-specific configuration file called `liferay-portlet.xml`. For an example, see the mapping defined inside `liferay-portlet.xml` found in `portal-web/docroot/WEB-INF`:

```
<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 its database.

Once roles are mapped to the portal, you can use methods as defined in 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. Next we'll describe Liferay's permission system and how to use it in your portlets. We recommend using Liferay's permission system, because it offers a much more robust way of tailoring your application's permissions.

7.2.2. Liferay's Permission System

You can add permissions to your custom portlets using four easy steps (also known as DRAC):

1. Define all resources and their permissions.
2. Register all defined resources in the permissions system. This is also known as *adding resources*.
3. Associate the necessary permissions with resources.
4. Check permission before returning resources.

Before you start adding permissions to a portlet, make sure you understand these two critical terms used throughout this section:

- *Resource*: A generic term for any object represented in the portal. Examples of resources include portlets (e.g. Message Boards, Calendar, etc.), Java classes (e.g. Message Board Topics, Calendar Events, etc.), and files (e.g. documents, images, etc.).
- *Permission*: An action on a resource. For example, the *view* action with respect to *viewing the calendar portlet* is defined as a permission in Liferay.

It's important to know that permissions for *portlet* resources are implemented a little differently

than for other resources like Java classes and files. Below, we'll describe permission implementation for the *portlet* resource first, followed by the model resource.

The first step is to define your resources and the actions that can be defined on them. Let's use the Blogs portlet to demonstrate. Open the `blogs.xml` file in `portal-impl/src/resource-actions` and you'll see the following mapping of resources to actions:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC
"-//Liferay//DTD Resource Action Mapping 6.2.0//EN"
"http://www.liferay.com/dtd/liferay-resource-action-mapping_6_2_0.dtd">

<resource-action-mapping>
    <portlet-resource>
        <portlet-name>33</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>
                </guest-unsupported>
            </permissions>
        </portlet-resource>
        <portlet-resource>
            <portlet-name>161</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>
<model-resource>
    <model-name>com.liferay.portlet.blogs</model-name>
    <portlet-ref>
        <portlet-name>33</portlet-name>
        <portlet-name>161</portlet-name>
    </portlet-ref>
    <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 />
        <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>
<model-resource>
    <model-
name>com.liferay.portlet.blogs.model.BlogsEntry</model-name>
    <portlet-ref>
        <portlet-name>33</portlet-name>
        <portlet-name>161</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>
    </permissions>
</model-resource>

```

```

<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>
</resource-action-mapping>

```

Permissions in the blogs portlet are defined at two different levels: the portlet level and the model level. The portlet level defines permissions on the portlet as a whole. The model level defines permissions on the model layer of the application, as defined by the entities in the application. Each level coincides with a section of the `resource-actions` XML file (in this case, `blogs.xml`).

At the `<portlet-resource>` level, actions and default permissions are defined on the portlet itself. Changes to portlet level permissions are performed on a per-site basis, and define whether users can add the portlet to a page, edit the portlet's configuration, or view the portlet. All these actions are defined in the `<supports>` tag for the portlet resource's permissions. The default portlet-level permissions for members of the site are defined in the `<site-member-defaults>` tag. In the case of the Blogs portlet, site members can view any blog in the site. Similarly, default guest permissions are defined in the `<guest-defaults>` tag. The `<guest-unsupported>` tag contains permissions forbidden to guests. Here, guests can't be given permission to configure the portlet.

The `<model-resource>` section contains the next level of permissions, based on the *scope* of an individual instance of the portlet. A scope in Liferay refers to how widely the data from an instance of a portlet is shared. For example, if you place a Blogs portlet on a page in the guest site and place another Blogs portlet on another page in the same site, the two blogs share the same set of posts. That happens because portlets are given a site level scope by default. If you reconfigure one of the two Blogs and change its scope to be the current page, that Blogs portlet instance no longer shares content with the other instance (or any other Blogs instance in that site). A portlet instance's scope-based permissions can thus span an entire site or be restricted to a single page. If you set the scope to the page, it's possible to have multiple distinct Blog instances within a site, each with different permissions for site users. For example, a food site could have one blog open to posts from any site member, but also have a separate informational blog about the site itself restricted to posts from administrators.

After defining the portlet and portlet instance as resources, we need to define permissions on the models in the portlet. The model resource is surrounded by the `<model-resource>` tag. Inside the tag, we first define the model name; the `<model-name>` isn't the name of a Java class, but the fully qualified name of the portlet's package (e.g. the blog portlet's package `com.liferay.portlet.blogs`). This is the recommended convention for permissions that refer to an instance of the portlet as a whole. Permissions like the ability to *add* or *subscribe* to a blog entry are defined here, and affect the portlet at the instance level. The `<portlet-ref>` element comes next and contains a `<portlet-name>`. The value of `<portlet-name>`

references the name of the portlet to which the model resource belongs. Theoretically, a model resource can belong to multiple portlets referenced with multiple `<portlet-name>` elements, but this is uncommon. Like the portlet resource, the model resource lets you define a list of supported actions that require permission to perform. You must list all the performable actions that require a permission check. For a blog entry, users must belong to appropriate roles for permission to do the following:

- *Add comments* to an entry
- *Delete* an entry
- *Change the permission* setting of an entry
- *Update* an entry
- *View* an entry

As with a portlet resource, the `<site-member-defaults>` tag, `<guest-defaults>` tag, and `<guest-unsupported>` tag define default permissions for site members and guests, respectively, for *model resources*.

After defining resource permissions for your custom portlet, you need to refer Liferay to the `resource-actions` XML file that contains your definitions (e.g. `blogs.xml` for the Blogs portlet). For Liferay core, the `resource-actions` XML files are in the `portal/portal-impl/src/resource-actions` directory and the file named `default.xml` file refers to each of these files. This excerpt from `default.xml` references the resource permission definition files for all built-in Liferay portlets (including the blogs portlet):

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC
  "--//Liferay//DTD Resource Action Mapping 6.2.0//EN"
  "http://www.liferay.com/dtd/liferay-resource-action-mapping_6_2_0.dtd">

<resource-action-mapping>
  <resource file="resource-actions/portal.xml" />
  <resource file="resource-actions/announcements.xml" />
  <resource file="resource-actions/asset.xml" />
  <resource file="resource-actions/blogs.xml" />
  ...
</resource-action-mapping>
```

You should put your plugin's `resource-actions` XML files (e.g. `default.xml` and `blogs.xml`) in a directory in your project's classpath. Then create a properties file (typically named `portlet.properties`) for your portlet that references the file that specifies your `<resource-action-mapping>` element (e.g. `default.xml`). In this portlet properties file, create a property named `resource.actions.configs` with the relative path to your portlet's `resource-action mapping` file (e.g. `default.xml`) as its value. Here's what this property specification might look like:

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

Check out a copy of the Liferay source code from the Liferay Github repository to see an example of a portlet that defines its resources and permissions as we just described; start by looking at the definition files found in the `portal-impl/src/resource-actions` directory. For an example of defining permissions in the context of a portlet plugin, check out

plugins/trunk and look at the portlet sample-permissions-portlet. Next, we'll show you how to add resources.

7.2.3. Adding a Resource

After defining resources and actions, it's time to add resources into the permissions system. Resources are added at the same time entities are added to the database. Managing resources follows the same pattern you've seen throughout Liferay: there's a service to use. Adding resources is as easy as calling the `addResources(...)` method of the `ResourceLocalServiceUtil` class. Here's the signature of that method:

```
public void addResources(  
    long companyId, long groupId, long userId, String name,  
    String primKey, boolean portletActions,  
    boolean addGroupPermissions, boolean addGuestPermissions)
```

Each entity that requires access permission must be added as a resource every time it is stored. For example, every time a user adds a new entry to her blog, the `addResources(...)` method must be called to add the new entry object to the resource system. Here's an example of the call from the `BlogsEntryLocalServiceImpl` class:

```
resourceLocalService.addResources(  
    entry.getCompanyId(), entry.getGroupId(), entry.getUserId(),  
    BlogsEntry.class.getName(), entry.getEntryId(), false,  
    addGroupPermissions, addGuestPermissions);
```

In the `addResources(...)` method, the parameters `companyId`, `groupId`, and `userId` correspond to the portal instance, the site in which the entity is being saved, and the user who is saving it. Let's look more closely at the remaining parameters:

- The `name` parameter is the fully qualified Java class name for the entity being added.
- The `primKey` parameter is the primary key of the entity.
- The `portletActions` parameter should be set to `true` if you're adding portlet action permissions. In our example, it's `false` because we're adding a model resource, which should be associated with permissions related to the model action defined in `blogs.xml`.
- The `addGroupPermissions` and the `addGuestPermissions` parameters are inputs from the user. If set to `true`, `ResourceLocalService` adds the default permissions to the current group and the guest group for this resource, respectively.

You can let your users choose whether to add the default site permission and/or the guest permission for your custom portlet resources: Liferay has a custom JSP tag `<liferay-ui:input-permissions />` that you can use to quickly add that functionality. You just insert the tag into the appropriate JSP and the checkboxes appear on that page. Make sure that the tag is inside the appropriate `<form>` tags, and that's all there is to it.

When you remove entities from the database, it's good practice to remove permissions mapped directly to the entity. To prevent dead resources from taking up space in the `Resource_` database table, remember to remove them from the `Resource_` table when the resource no longer applies. Do this by calling the `deleteResource(...)` method of `ResourceLocalServiceService`. Here's an example of a blogs entry being removed:

```
resourceLocalService.deleteResource(  
    entry.getCompanyId(), BlogsEntry.class.getName(),  
    ResourceConstants.SCOPe_INDIVIDUAL, entry.getEntryId());
```

Great! Now that you know how to work with resource permissions, we'll show you how to provide a user interface for managing resource permissions.

7.2.4. Adding Permissions

On the portlet level, no code needs to be written in order to have the permission system work for your custom portlet. If you've defined any custom permissions (supported actions) in your configuration file's `portlet-resource` tag, they're automatically added to a list of permissions in Liferay's permissions UI. What good, however, are permissions that are available but can't be set by users?

To let a user set permissions on model resources, you must expose the permission interface to the user. Just 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.

This example demonstrates the use of both tags; it comes from the

`view_entry_content.jspf` file.

```
<liferay-security:permissionsURL  
    modelResource="<%=" BlogsEntry.class.getName() %>"  
    modelResourceDescription="<%=" entry.getTitle() %>"  
    resourcePrimKey="<%=" entry.getPrimaryKey().toString() %>"  
    var="entryURL"  
/>
```

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

For the first tag, specify the following attributes:

- `modelResource`: The fully qualified Java object class name. This class name gets translated into its more readable name as specified in `Language.properties`.
- `modelResourceDescription`: You can pass in anything that best describes this model instance. In this example, the blogs title was passed in.
- `resourcePrimKey`: The primary key of your model instance.
- `var`: Specifies the name of the variable to which the resulting URL string will be 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.

Next, we'll show you how to implement permissions checking.

7.2.5. Checking Permissions

The last major step toward implementing permissions for your custom portlet is to ensure the configured permissions are enforced. You'll do this by adding permission checks to your application. For example, your business layer can check for permission before deleting a resource, or your user interface can hide a button that adds an entity (e.g. a calendar event) if the user doesn't have permission.

You need to implement checking of any custom permissions you defined in your `resource-actions` XML file. In the Blogs portlet, one supported custom action is `ADD_ENTRY`. There are two places in the source code to check for this permission: in your JSP files and in the business logic. For the JSP files, you want to wrap certain elements in permission checks, so they only appear for users with the permission to perform those functions. For example, the presence of the Add Entry button is contingent on whether the user has permission to add a blog entry.

Here's the `ADD_ENTRY` action in a JSP file:

```
<%
if (permissionChecker.hasPermission(
    scopeGroupId, "com.liferay.portlet.blogs.model",
    scopeGroupId, "ADD_ENTRY") {
    // Show add entry button
}
%>
```

The second place to check for the add entry permission is in the business logic. If the check fails, a `PrincipalException` is thrown and the add entry request is aborted:

```
if (!permissionChecker.hasPermission(
    scopeGroupId, "com.liferay.portlet.blogs.model",
    scopeGroupId, "ADD_ENTRY")) {
    throw new PrincipalException();
}

blogsEntryLocalService.addEntry(...);
```

The `PermissionChecker` class has a method `hasPermission(...)` that checks whether a user making a resource request has the necessary access permission. If the user isn't signed in (guest user), it checks for guest permissions. Otherwise, it checks for user permissions. Let's quickly review the parameters of this method:

- `groupId`: Represents the scope where the permission check is performed. In Liferay, many scopes are available, including a specific site, organization, personal site of a user, or a page in a site. This is important because a user may be allowed to add blog entries in one site, but not in another. For resources that don't belong to a scope (extremely rare and unlikely), set the value of this parameter to 0. There are several ways you can obtain the `groupId` of the current scope:
 - JSP that uses the `<theme:defineObjects/>` tag: there's an implicit variable called `scopeGroupId`.
 - Business logic class: If you're using the `ServiceContext` pattern, you can obtain the `groupId` by using `serviceContext.getScopeGroupId()`. If you're not using the `ServiceContext` pattern, you can obtain `groupId` from the theme

display request object:

```
ThemeDisplay themeDisplay = (ThemeDisplay)
    request.getAttribute(WebKeys.THEME_DISPLAY);
long scopeGroupId = themeDisplay.getScopeGroupId();
```

- `name`: The name of the resource as specified in the XML file of the previous sections.
- `primKey`: The primary key of the resource. In this example the resource doesn't exist as an entry in the database, so we use the `groupId` again. If we were checking for a permission on a given blog entry, we'd use the primary key of that blog entry instead.
- `actionId`: The name of the action as it appears in the XML file. To simplify searching for usages, consider creating a helper class that has constants for all the actions defined.

In the examples above, we're assuming there's a variable called `permissionChecker` already available. Liferay automatically creates a `PermissionChecker` instance that has the necessary information from the user for every request. Liferay also caches the security checks to ensure good performance. There are several ways to obtain a permission checker:

- In a JSP that uses the `<theme:defineObjects/>` tag, there's an implicit variable called `permissionChecker`.
- With Service Builder, every service implementation class can access the `PermissionChecker` instance by using the method `getPermissionChecker()`.
- If you're not using Service Builder, `PermissionChecker` can be obtained from the theme display request object:

```
ThemeDisplay themeDisplay = (ThemeDisplay)
    request.getAttribute(WebKeys.THEME_DISPLAY);
PermissionChecker permissionChecker =
    themeDisplay.getPermissionChecker();
```

Next, you'll optimize permission checking by creating helper classes to do most of the heavy lifting.

7.2.6. Creating Helper Classes for Permission Checking

Helper classes streamline your code. They encapsulate the use of `permissionChecker` and the names of the resources for a specific portlet. This is especially useful when there are complex parent-child relationships, or if your permission logic calls for checking multiple action types.

`BlogsPermission` is an example of a permission helper class. Here's how

`BlogsPermission` is used in a JSP:

```
<%
if (BlogsPermission.contains(permissionChecker, scopeGroupId,
    ActionKeys.ADD_ENTRY)) {
    // show add entry button
}
%>
```

Now let's see how a `ServiceImpl` class, `BlogsEntryServiceImpl`, uses the `BlogsPermission` helper class. In the method

`BlogsEntryServiceImpl.addEntry(...)`, a call is made to check whether the incoming request has permission to add an entry. The check is done using the helper class `BlogsPermission`. If the check fails, it throws a `PrincipalException` and the add entry request aborts.

```

public BlogsEntry addEntry(
    String title, String description, String content,
    int displayDateMonth, int displayDateDay, int displayDateYear,
    int displayDateHour, int displayDateMinute, boolean allowPingbacks,
    boolean allowTrackbacks, String[] trackbacks, boolean smallImage,
    String smallImageURL, String smallImageFileName,
    InputStream smallImageInputStream, ServiceContext serviceContext)
throws PortalException, SystemException {

    BlogsPermission.check(
        getPermissionChecker(), serviceContext.getScopeGroupId(),
        ActionKeys.ADD_ENTRY);

    return blogsEntryLocalService.addEntry(
        getUserId(), title, description, content, displayDateMonth,
        displayDateDay, displayDateYear, displayDateHour, displayDateMinute,
        allowPingbacks, allowTrackbacks, trackbacks, smallImage,
        smallImageURL, smallImageFileName, smallImageInputStream,
        serviceContext);
}

```

Check out the parameters passed into the `check(...)` method. Again, the `getPermissionChecker()` method is readily available in all `ServiceImpl` classes. The blogs entry ID is available in the `serviceContext`, indicating that the permission check is against the Blogs portlet. `ActionKeys.ADD_ENTRY` is a static string used to indicate the action requiring the permission check. Likewise, you're encouraged to use custom portlet action keys.

Let's review what we've just covered. Implementing permissions into your custom portlet consists of four main steps:

1. Define any custom resources and actions.
2. Implement code to register (i.e., add) any newly created resources, such as a `BlogsEntry` object.
3. Provide an interface for the user to configure permissions.
4. Implement code to check permissions before returning resources or showing custom features.

The two major resources are portlets and entities. Liferay does most of the work for you by providing a configuration file and a system of resources that let you check permissions wherever you need to in your application.

You're now equipped to implement security in your custom Liferay portlets!

Next, let's learn how to create and use portal and portlet URLs.

7.3. *Creating Portlet URLs in JavaScript*

The `Liferay.PortletURL` JavaScript module allows developers to generate portlet URLs from their JavaScript code. This module has been updated since the first released version of Liferay 6.2. In Liferay 6.2.0 M5 (a Milestone release) and prior versions, developers could call

this module to generate URLs from their JavaScript in the following ways:

- `Liferay.PortletURL.createActionURL()`
- `Liferay.PortletURL.createRenderURL()`
- `Liferay.PortletURL.createResourceURL()`
- Invoking the URL `/c/portal/ [portlet URL]`

Each of these methods returned server generated URLs that included authentication tokens (the `p_auth` or `p_p_auth` URL parameters), whenever necessary. That behavior, however, posed a security risk. Please see LPS-34098 for details. In all Liferay EE 6.2 GA and CE 6.2 GA releases, and Milestone releases after Liferay 6.2.0 M5, these methods are updated so that they still work (i.e., they still return appropriate URLs) but they no longer include the authentication tokens in the returned URLs. This update has been distributed to EE customers as a fix pack.

If you experience problems using the above methods to generate URLs in Liferay 6.2.0 M6 or later versions, you might need to update your plugins. If you're an EE customer and have applied all available fix packs, you might also need to update your plugins. If you'd like to determine whether or not you need to update your plugins, use the following verification steps:

1. Search for `Liferay.PortletURL` or `/c/portal/portlet_url` in your code, portal content text, and templates.
2. If you don't find any occurrences, you don't need to update your plugins. In this case, you can safely set `portlet.url.generate.by.path.enabled=false` in your `portal-ext.properties` file. Setting this property to `false` prevents portlet URLs from being generated using `/c/portal/portlet_url`. This property is included in Liferay 6.2.0 M6 and later versions.

If you do find `Liferay.PortletURL` or `/c/portal/portlet_url` when you search, you will need to update your plugins. There are several ways to do it:

1. Refactor your code to call the JavaScript method

`Liferay.PortletURL.createURL`, passing the `baseURL` as a parameter, instead of using one of the updated methods. You can generate the `baseURL`, from a JSP for example, using the Java method `PortletURLFactoryUtil.create`.

This method is used, for example, in Liferay's `select_add_file_entry_type.jsp` JSP:

```
var portletURL = Liferay.PortletURL.createURL('<%=  
PortletURLFactoryUtil.create(request, portletId, themeDisplay.getPlid(),  
PortletRequest.RENDER_PHASE) %>');
```

Visit https://github.com/liferay/liferay-portal/blob/6.2.0-ga1/portal-web/docroot/html/portlet/document_library_display/select_add_file_entry_type.jsp#L56 to view the complete JSP.

2. Refactor your code to remove dependencies on the `Liferay.PortletURL` object and on `/c/portal/portlet_url` calls. When you're finished, set `portlet.url.generate.by.path.enabled=false`. Let's examine two examples:

Example 1 (before refactoring):

```

function(event) {
    var portletURL = Liferay.PortletURL.createRenderURL();
    portletURL.setParameter('groupId', '<%= scopeGroupId %>');
    portletURL.setParameter('struts_action',
    '/journal/select_document_library');
    portletURL.setPlid('<%= controlPanelPlid %>');
    portletURL.setPortletId('15');
    portletURL.setWindowState('pop_up');
    Liferay.Util.openWindow(
        {
            id: '<portlet:namespace />selectDocumentLibrary',
            uri: portletURL.toString()
        }
    );
}

```

Example 1 (after refactoring):

```

<%
LiferayPortletURL selectDocumentURL = portletURLFactory.create(request, "15",
controlPanelPlid, "RENDER_PHASE");
selectDocumentURL.setWindowState("pop_up");
selectDocumentURL.setParameter("groupId", "scopeGroupId");
selectDocumentURL.setParameter("struts_action",
"/journal/select_document_library");
%>
function(event) {
    Liferay.Util.openWindow(
        {
            id: '<portlet:namespace />selectDocumentLibrary',
            uri:'<%= selectDocumentURL %>'
        }
    );
}

```

In example 1, the code uses the `portletURLFactory` to create a `LiferayPortletURL` object, instead of using the `Liferay.PortletURL` object to create it.

Example 2 (before refactoring):

```

var <portlet:namespace />createContactURL = function(contact) {
    var portletURL = new Liferay.PortletURL.createRenderURL();
    portletURL.setParameter('mvcPath', '/contacts_center/edit_entry.jsp');
    portletURL.setParameter('redirect', contact.redirect);
    portletURL.setParameter('entryId', contact.entryId);
    portletURL.setPortletId('1_WAR_contactsportlet');
    portletURL.setWindowState('EXCLUSIVE');
    return portletURL.toString();
}

```

Example 2 (after refactoring):

```

<liferay-portlet:renderURL portletName="1_WAR_contactsportlet"
var="displayHelpURL" windowState="exclusive">
    <portlet:param name="mvcPath" value="/contacts_center/edit_entry.jsp" />
</liferay-portlet:renderURL>

```

```

var <portlet:namespace />createContactURL = function(contact) {
    var portletURL = '<%= displayHelpURL %>';
    }
    var namespace =
Liferay.Util.getPortletNamespace('1_WAR_contactsportlet');
    AUI().Object.each(
    {
        redirect: contact.redirect,
        entryId: contact.entryId
    },
    function(item, index, collection) {
        if (item) {
            portletURL = portletURL + '&' + namespace +
encodeURIComponent(index) + '=' + encodeURIComponent(item);
        }
    }
);
    return portletURL;
}

```

In example 2, the code uses the `<liferay-portlet:renderURL />` tag to avoid using the `Liferay.PortletURL` object. Note that in example 2, the namespace for every parameter was added manually.

3. Set the property `portlet.url.struts.action.enabled` to `false`. This ensures that everything works as in Liferay 6.2.0 M5 and previous versions of Liferay, but it presents the security risk described in LPS-34098. Avoid using this method, if possible.

Next, let's learn how to use Liferay's asset framework.

7.4. Asset Framework

Liferay's asset framework is a system that allows you to add common functionality to your application. For example, you might build an event management application that shows a list of upcoming events. It might be nice to be able to tag or categorize those events to provide users with metadata describing more about them. Or you might want to let users comment on events.

This common functionality is what Liferay's asset framework gives you. Using the power of Liferay's built-in message boards, tags, and categories, Liferay lets you infuse your application with these features in no time.

The term *asset* is a generic term that refers to any type of content, including text, an external file, a URL, an image, or a record in an online book library. Consequently, when we use the term *asset* here, we're referring to some type of Liferay content, like documents, blog entries, bookmarks, wiki pages, or anything you create in your applications.

Here are the features you can reuse thanks to the asset framework:

- Associate tags to custom content types. New tags are created automatically when the author assigns them to the content.
- Associate categories to custom content types. Authors are only allowed to select from

predefined categories within several predefined vocabularies.

- Manage tags from the Control Panel, including merging tags.
- Manage categories from the Control Panel, including creating complex hierarchies.
- Associate comments with assets.
- Rate assets using a five star rating system.
- Assign social bookmarks to assets, including via tweet, Facebook like, or +1 (Google Plus).
- Add custom fields to assets.
- Relate assets to one another.
- Flag asset content as inappropriate.
- Keep track of the number of visualizations of an asset.
- Integrate workflow with assets.
- Publish your content using the Asset Publisher portlet. Asset Publisher can publish dynamic asset lists or manually selected asset lists. It can also show an asset summary view with a link to the full view. This saves you time, since it likely won't be necessary to develop custom portlets for your custom content types.

At this point you might be saying, "Liferay's asset framework sounds great; but how do I leverage all these awesome functions?" Excellent question, young padawan, and perfect timing; we couldn't have said it better ourselves.

We'll briefly describe the first two steps here before we dive in head first:

- The first step is mandatory. You must let the framework know whenever one of your custom content entries is added, updated, or deleted.
- The second step enables the asset framework in the UI. You can use a set of taglibs to provide widgets that allow authors to enter comments, tags and categories and that show the entered tags and categories along with the content.

Let's dive head first into the first step: we need to inform the asset framework when we're adding, updating, or deleting assets.

7.4.1. Adding, Updating, and Deleting Assets

Whenever you create a new entity, you need to let the asset framework know. In this sense, it's similar to permission resources. It's a simple procedure: you invoke a method of the asset framework that adds an `AssetEntry` so that Liferay can keep track of the asset.

Specifically, you should access these methods using either the static methods of `AssetLocalServiceUtil` or an instance of the `AssetEntryLocalService` injected by Spring. To simplify this section, we'll be using the static methods of `AssetLocalServiceUtil`, since it doesn't require any special setup in your application.

The method to invoke when one of your custom content entries is added or updated is the same, and is called `updateEntry`. Here's the full signature:

```
AssetEntry updateEntry(  
    long userId, long groupId, String className, long classPK,  
    String classUuid, long classTypeId, long[] categoryIds,  
    String[] tagNames, 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, Integer priority,
        boolean sync)
    throws PortalException, SystemException
Here's an example of this method's invocation extracted from the built in blogs portlet:
assetEntryLocalService.updateEntry(
    userId, entry.getGroupId(), BlogsEntry.class.getName(),
    entry.getEntryId(), entry.getUuid(), 0, assetCategoryIds,
    assetTagNames, visible, null, null, entry.getDisplayDate(), null,
    ContentTypes.TEXT_HTML, entry.getTitle(), null, summary, null, null,
    0, 0, null, false);

```

Here's a quick summary of the most important parameters of this method:

- `userId` is the identifier of the user who created the content.
- `groupId` identifies the scope of the created content. If your content doesn't support scopes (extremely rare), just pass 0 as the value.
- `className` identifies the type of asset. The recommended convention is to use the name of the Java class that represents your content type, but you can actually use any String you want as long as you are sure that it is unique.
- `classPK` identifies the specific content being created among others of the same type. It's usually the primary key of the table where the custom content is stored. If you want, you can use the `classUuid` parameter to specify a secondary identifier; it's guaranteed to be universally unique. It's especially useful if your content will be exported and imported across separate portals.
- `assetCategoryIds` and `assetTagNames` represent the categories and tags selected by the author of the content. The asset framework stores them for you.
- `visible` specifies whether the content should be shown at all by Asset Publisher.
- `title`, `description` and `summary` are descriptive fields used by the Asset Publisher when displaying entries of your content type.
- `publishDate` and `expirationDate`, when specified, tell Asset Publisher it shouldn't show the content before a given publication date or after a given expiration date, respectively.
- All other fields are optional; it won't always make sense to include them. The `sync` parameter should always be *false* unless you're doing something very advanced (feel free to look at the code if you're really curious).

When one of your custom content entries is deleted, you should once again let the asset framework know. This way, it can clean up stored information and make sure that the Asset Publisher doesn't show any information for the content that has been deleted. The signature of method to delete an asset entry is:

```
void deleteEntry(String className, long classPK)
```

Here's an example invocation extracted again from the blogs portlet:

```
assetEntryLocalService.deleteEntry(
    BlogsEntry.class.getName(), entry.getEntryId());
```

Now that you can create, modify, and delete assets, let's learn how to categorize them.

7.4.2. Entering and Displaying Tags and Categories

In the last section, we let the asset framework know about the tags and categories that we associated with a given asset; but how does a content author specify the tags and categories?

Liferay provides a set of JSP tags you can use to make this task very easy. You can put the following Liferay UI tags in your forms to create content that can be associated with new or existing tags or predefined categories:

```
<label>Tags</label>
<liferay-ui:asset-tags-selector
    className="<%=" entry.getClassName() .getName() %>"
    classPK="<%=" entry.getPrimaryKey() %>">
/>

<label>Categories</label>
<liferay-ui:asset-categories-selector
    className="<%=" entry.getClassName() .getName() %>"
    classPK="<%=" entry.getPrimaryKey() %>">
/>
```

These two taglibs create appropriate form controls that allow the user to search for a tag or create a new one or select an existing category.



Tip: If you're using Liferay's AlloyUI Form taglibs, creating fields to enter tags and categories is even simpler. You just use `<aui:input name="tags" type="assetTags" />` and `<aui:input name="categories" type="assetCategories" />`, respectively.

Once the tags and categories have been entered, you'll want to show them along with the content of the asset. Here's how to display the tags and categories:

```
<label>Tags</label>
<liferay-ui:asset-tags-summary
    className="<%=" entry.getClassName() .getName() %>"
    classPK="<%=" entry.getPrimaryKey() %>">
/>

<label>Categories</label>
<liferay-ui:asset-categories-summary
    className="<%=" entry.getClassName() .getName() %>"
    classPK="<%=" entry.getPrimaryKey() %>">
/>
```

In both JSP tags, you can also specify a `portletURL` parameter; each tag that uses it will be a link containing the `portletURL` and `tag` or `categoryId` parameter value, respectively. This supports tags navigation and categories navigation within your portlet. You'll need to implement the look-up functionality in your portlet code; do this by reading the values of those two parameters and using the `AssetEntryService` to query the database for entries based on the specified tag or category.

Great job! You'll have no problem associating tags and categories with your assets. Before we go

further with our example, let's take a look at more JSP tags you can use to leverage the asset framework's features.

7.4.3. More JSP Tags for Assets

In addition to tags and categories, there are more features that the asset framework provides. These features allow users to do the following with your assets:

- Add comments
- Rate comments of other users
- Rate assets
- Apply social bookmarks (e.g. via tweet, Facebook like, or +1 (Google Plus))
- Relate assets to one another
- Flag content as inappropriate and notify the portal administrator

There are JSP tags, called *Liferay UI* tags, associated with each feature. You can find these tags used in the JSPs for Liferay's built-in portlets (e.g. the `edit_entry.jsp` of the Blogs portlet). Here are some examples of the JSP tags from the Blogs portlet:

- *Comments and comment ratings:*

```
<portlet:actionURL var="discussionURL">
    <portlet:param
        name="struts_action"
        value="/blogs/edit_entry_discussion"
    />
</portlet:actionURL>

<liferay-ui:discussion
    className="<%=" BlogsEntry.class.getName() %>"
    classPK="<%=" entry.getEntryId() %>"
    formAction="<%=" discussionURL %>"
    formName="fm2"
    ratingsEnabled="<%=" enableCommentRatings %>"
    redirect="<%=" currentURL %>"
    subject="<%=" entry.getTitle() %>"
    userId="<%=" entry.getUserId() %>"
/>
```

- *Rate assets:*

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

- *Social Bookmarks:*

```
<liferay-ui:social-bookmarks
    displayStyle="<%=" socialBookmarksDisplayStyle %>"
    target="_blank"
    title="<%=" entry.getTitle() %>"
    url="<%=" PortalUtil.getCanonicalURL(bookmarkURL.toString(),
        themeDisplay) %>"
/>
```

- *Related assets:*

```
<liferay-ui:input-asset-links
    className="<%= BlogsEntry.class.getName() %>"
    classPK="<%= entryId %>"
/>
```

- *Flag as inappropriate:*

```
<liferay-ui:flags
    className="<%= BlogsEntry.class.getName() %>"
    classPK="<%= entry.getEntryId() %>"
    contentTitle="<%= entry.getTitle() %>"
    reportedUserId="<%= entry.getUserId() %>"
/>
```

With Liferay's taglib tags, you can easily apply these features to your assets. No problemo, right? So let's get the assets published in your portal.

7.4.4. Publishing Assets with Asset Publisher

A huge benefit of using the asset framework is that you can leverage the Asset Publisher portlet to publish lists of your custom asset types. You can choose to have users specify lists dynamically (e.g., based on the asset tags or categories) or have administrators do it statically.

To display your assets, the Asset Publisher needs to know how to access their metadata. You also need to provide the Asset Publisher templates for the types of views (e.g. *full* view and *abstract* view) available to display your assets. You can provide all this to the Asset Publisher by implementing these two interfaces:

- **AssetRendererFactory:** A class that knows how to retrieve specific assets from persistent storage using the `classPK`. The `classPK` is typically the asset's primary key, but can be anything you specified to the `updateAsset` method, which you use to add or update the asset. Your factory implementation can grab the asset from a `groupId` (identifies a scope of data) and a `urlTitle` (a title that can be used in friendly URLs to refer uniquely to the asset within a given scope). Finally, the asset renderer factory can provide a URL for the Asset Publisher to use when a user wants to add a new asset of your custom type. This URL should point to your own portlet. There are other less important methods of the interface, but you can avoid implementing them by extending `BaseAssetRendererFactory`. By extending this base class instead of implementing the interface directly, your code will be more robust to possible interface changes in future versions of Liferay, since the base implementation will be updated to accommodate the interface changes.
- **AssetRenderer:** This is an interface that provides metadata information about one specific asset. It checks whether the current user has permission to edit or view the asset and renders the asset for the different templates (e.g. abstract and full content view) by forwarding to a specific JSP. We recommend that you extend the `BaseAssetRenderer` class rather than directly implementing the interface. The base class provides helpful defaults and contains methods that get added to the interface in the

future.

Let's look at an example of these two classes. We'll use Liferay's Blogs portlet again, and we'll start by implementing `AssetRendererFactory`:

```
public class BlogsEntryAssetRendererFactory extends BaseAssetRendererFactory {
    public static final String CLASS_NAME = BlogsEntry.class.getName();
    public static final String TYPE = "blog";
    public AssetRenderer getAssetRenderer(long classPK, int type)
        throws PortalException, SystemException {
        BlogsEntry entry = BlogsEntryLocalServiceUtil.getEntry(classPK);
        return new BlogsEntryAssetRenderer(entry);
    }
    @Override
    public AssetRenderer getAssetRenderer(long groupId, String urlTitle)
        throws PortalException, SystemException {
        BlogsEntry entry = BlogsEntryServiceUtil.getEntry(groupId, urlTitle);
        return new BlogsEntryAssetRenderer(entry);
    }
    public String getClassName() {
        return CLASS_NAME;
    }
    public String getType() {
        return TYPE;
    }
    @Override
    public PortletURL getURLAdd(
        LiferayPortletRequest liferayPortletRequest,
        LiferayPortletResponse liferayPortletResponse)
        throws PortalException, SystemException {
        HttpServletRequest request =
            liferayPortletRequest.getHttpServletRequest();
        ThemeDisplay themeDisplay = (ThemeDisplay)request.getAttribute(
            WebKeys.THEME_DISPLAY);
        if (!BlogsPermission.contains(
            themeDisplay.getPermissionChecker(),
            themeDisplay.getScopeGroupId(), ActionKeys.ADD_ENTRY)) {
            return null;
        }
    }
}
```

```

PortletURL portletURL = PortletURLFactoryUtil.create(
    request, PortletKeys.BLOGS, getControlPanelPlid(themeDisplay),
    PortletRequest.RENDER_PHASE);

portletURL.setParameter("struts_action", "/blogs/edit_entry");

return portletURL;
}

@Override
public boolean hasPermission(
    PermissionChecker permissionChecker, long classPK, String
actionId)
throws Exception {

    return BlogsEntryPermission.contains(
        permissionChecker, classPK, actionId);
}

@Override
public boolean isLinkable() {
    return _LINKABLE;
}

@Override
protected String getIconPath(ThemeDisplay themeDisplay) {
    return themeDisplay.getPathThemeImages() + "/blogs/blogs.png";
}

private static final boolean _LINKABLE = true;
}

```

Here's the AssetRenderer implementation:

```

public class BlogsEntryAssetRenderer extends BaseAssetRenderer {

    public BlogsEntryAssetRenderer(BlogsEntry entry) {
        _entry = entry;
    }

    public long getClassPK() {
        return _entry.getEntryId();
    }

    @Override
    public String getDiscussionPath() {
        if (PropsValues.BLOGS_ENTRY_COMMENTS_ENABLED) {
            return "edit_entry_discussion";
        }
        else {
            return null;
        }
    }
}

```

```

public long getGroupId() {
    return _entry.getGroupId();
}

public String getSummary(Locale locale) {
    return HtmlUtil.stripHtml(_entry.getDescription());
}

public String getTitle(Locale locale) {
    return _entry.getTitle();
}

@Override
public PortletURL getURLEdit(
    LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse)
throws Exception {

    PortletURL portletURL =
liferayPortletResponse.createLiferayPortletURL(
        getControlPanelPlid(liferayPortletRequest), PortletKeys.BLOGS,
        PortletRequest.RENDER_PHASE);

    portletURL.setParameter("struts_action", "/blogs/edit_entry");
    portletURL.setParameter("entryId",
String.valueOf(_entry.getEntryId()));

    return portletURL;
}

@Override
public String getUrlTitle() {
    return _entry.getUrlTitle();
}

@Override
public PortletURL getURLView(
    LiferayPortletResponse liferayPortletResponse,
   WindowState windowState)
throws Exception {

    PortletURL portletURL =
liferayPortletResponse.createLiferayPortletURL(
        PortletKeys.BLOGS, PortletRequest.RENDER_PHASE);

    portletURL.setWindowState(windowState);

    portletURL.setParameter("struts_action", "/blogs/view_entry");
    portletURL.setParameter("entryId",
String.valueOf(_entry.getEntryId()));

    return portletURL;
}

```

```

@Override
public String getURLViewInContext(
    LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse,
    String noSuchEntryRedirect) {

    ThemeDisplay themeDisplay =
        (ThemeDisplay)liferayPortletRequest.getAttribute(
            WebKeys.THEME_DISPLAY);

    return themeDisplay.getPortalURL() + themeDisplay.getPathMain() +
        "/blogs/find_entry?noSuchEntryRedirect=" +
        HttpUtil.encodeURL(noSuchEntryRedirect) + "&entryId=" +
        _entry.getEntryId();
}

public long getUserId() {
    return _entry.getUserId();
}

public String getUuid() {
    return _entry.getUuid();
}

@Override
public boolean hasEditPermission(PermissionChecker permissionChecker) {
    return BlogsEntryPermission.contains(
        permissionChecker, _entry, ActionKeys.UPDATE);
}

@Override
public boolean hasViewPermission(PermissionChecker permissionChecker) {
    return BlogsEntryPermission.contains(
        permissionChecker, _entry, ActionKeys.VIEW);
}

@Override
public boolean isPrintable() {
    return true;
}

public String render(
    RenderRequest renderRequest, RenderResponse renderResponse,
    String template)
throws Exception {

    if (template.equals(TEMPLATE_ABSTRACT) ||
        template.equals(TEMPLATE_FULL_CONTENT)) {

        renderRequest.setAttribute(WebKeys.BLOGS_ENTRY, _entry);

        return "/html/portlet/blogs/asset/" + template + ".jsp";
    }
    else {

```

```

        return null;
    }
}

@Override
protected String getIconPath(ThemeDisplay themeDisplay) {
    return themeDisplay.getPathThemeImages() + "/blogs/blogs.png";
}

private BlogsEntry _entry;

}

```

In the render method, there's a forward to a JSP in the case of the abstract and the full content templates. The abstract isn't mandatory and if it isn't provided, the Asset Publisher shows the title and the summary from the appropriate methods of the renderer. The full content template should always be provided. Here's how it looks for blogs entries:

```

<%@ include file="/html/portlet/blogs/init.jsp" %>

<%
BlogsEntry entry = (BlogsEntry) request.getAttribute(WebKeys.BLOGS_ENTRY);
%>

<%= entry.getContent() %>

<liferay-ui:custom-attributes-available
    className="<%= BlogsEntry.class.getName() %>">
</liferay-ui:custom-attributes-available>
<liferay-ui:custom-attribute-list
    className="<%= BlogsEntry.class.getName() %>"
    classPK="<%= (entry != null) ? entry.getEntryId() : 0 %>"
    editable="<%= false %>"
    label="<%= true %>">
</liferay-ui:custom-attribute-list>

```

</liferay-ui:custom-attributes-available>

That's about it. It wasn't that hard, right? Now it's time to get really fancy; put on your dancing shoes. If you need to extend the capabilities of the `AssetRendererFactory` for one of Liferay's core portlets, check out the article [Extending an AssetRendererFactory](#) by Juan Fernández; he talks about doing just that.

Now get out there and start enjoying the benefits of the asset framework in your custom portlets!

Next, we'll explore the Recycle Bin framework and its many benefits for your portal.

7.5. *Implementing the Recycle Bin in Your App*

How many times have you deleted something, only to realize it was a mistake and you need to restore it? Does the app you're working on have a delete function? If so, you can add the ability to move items into a recycle bin instead of deleting them immediately. Have you been looking for an easy way to implement this capability in your app? Fortunately, you can, with the help of Liferay's Recycle Bin.

By leveraging the Recycle Bin in your app, your users can have deleted content temporarily stored in the Recycle Bin until it has reached a desired expiration date or until it is explicitly deleted. And while the content is in the Recycle Bin, it can be safely restored back to its original state. The Recycling Assets with the Recycle Bin section of *Using Liferay Portal* provides complete details on the Recycle Bin's capabilities.

Interestingly, the Recycle Bin is not a single feature, but rather one of Liferay Portal's frameworks. This means that all applications in the portal can use the Recycle Bin if developers implement the feature. Also, because the Recycle Bin is completely integrated with the platform, Liferay Portal can distinguish between recycled and non-recycled assets. For example, when searching for assets in the search bar, only non-trashed assets are returned.

In this section, we'll discuss how to implement the Recycle Bin framework for a Liferay application. There are five Recycle Bin framework capabilities that you can leverage in your app.

- Moving Entries to the Recycle Bin
- Restoring Entries from the Recycle Bin
- Implementing the Undo Action
- Moving/Restoring Parent Entities
- Resolving Conflicts

While discussing these capabilities, we'll refer to code snippets taken from Liferay's Jukebox Portlet plugin. The plugin actually bundles three separate portlets: Songs, Albums, and Artists, which are integrated together.

The Jukebox portlet allows you to upload songs that you can play from your portal. You can categorize the songs by album and artist, keeping your song collection organized. The Jukebox portlet's song and album assets can be moved to and from the Recycle Bin.

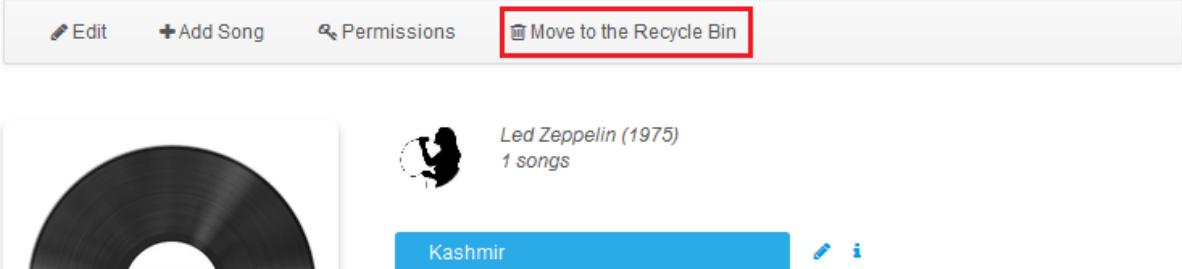
Also, the Jukebox portlet offers indexed (searchable) entities and a workflow. These are not mandatory to implement the Recycle Bin, but are present in this example portlet.

For each section, we'll walk through how the Jukebox portlet implements the given capability of the Recycle Bin. By following the steps taken to implement the Recycle Bin in the Jukebox portlet, you can similarly implement the Recycle Bin in your app. Let's begin implementing the Recycle Bin framework!

7.5.1. Moving Entries to the Recycle Bin

The first framework component to implement is the ability to move entries to the Recycle Bin. Once you implement this component, users can move entries to the Recycle Bin with the click of a button.

Physical Graffiti



The screenshot shows the Physical Graffiti portlet interface. At the top, there are buttons for 'Edit', '+ Add Song', 'Permissions', and 'Move to the Recycle Bin' (which is highlighted with a red box). Below this, there's a thumbnail of a vinyl record, the album title 'Led Zeppelin (1975)', and the song title 'Kashmir'. There are also edit and info icons.

We'll guide you through the following steps for providing the means to move entries to the Recycle Bin:

1. Enable Trash for Service Entities
2. Implement a Trash Handler for Each Trash-Enabled Entity
3. Create a Service Method to Move Entries to the Recycle Bin
4. Create a Portlet Action to Initiate Moving Entries to Recycle Bin
5. Implement a Trash Renderer for Each Trash-Enabled Entity

Let's begin implementing this component by configuring the app's service.

7.5.1.1. Moving Entries Step 1: Enable Trash for Service Entities

You must enable the trash feature for each entity in your app that you want to use with the Recycle Bin. In your app's `service.xml` file, insert the `trash-enabled="true"` attribute within the `<entity>` tag for each of those entities. Here's a code snippet from the Jukebox portlet's `service.xml`:

```
<entity name="Song" local-service="true" remote-service="true" uuid="true"  
trash-enabled="true">
```

Run Service Builder to generate back-end trash related classes for the trash-enabled entities.

Next, you'll need to implement trash handlers for those entities.

7.5.1.2. Moving Entries Step 2: Implement a Trash Handler for Each Trash-Enabled Entity

As with many other Liferay frameworks--such as those for workflow, assets, and indexing--you must implement handler classes for that framework. The Recycle Bin's handlers are required to manage basic trash operations. You must create an implementation of the `TrashHandler` interface for each trash-enabled entity. Recycle Bin trash handlers manage the entries as they are moved into the Recycle Bin, viewed in the Recycle Bin, restored, or permanently deleted. As a convenience, Liferay provides the abstract class `BaseTrashHandler` that your trash handlers can extend.

Consider the following `TrashHandler` methods as a minimal set of methods to implement/override in your trash handlers:

- `deleteTrashEntry()`
- `getClassName()`
- `getRestoreContainedModelLink()`
- `getRestoreMessage()`
- `hasTrashPermission()`
- `isInTrash()`
- `restoreTrashEntry()`

The Jukebox portlet implements a class named `JukeBoxBaseTrashHandler` that serves as a base implementation. It's leveraged by trash handlers for the app's song and album entities. As an example trash handler implementation, you can view the `SongTrashHandler` and its parent base class `JukeBoxBaseTrashHandler`.

After you've implemented trash handlers for your trash-enabled entities, you'll need to specify those handlers in your app's `liferay-portlet.xml` file. For example, the Jukebox's `SongTrashHandler` class is specified in a `<trash-handler>` element for the Songs portlet as follows:

```
<trash-handler>org.liferay.jukebox.trash.SongTrashHandler</trash-handler>
```

You can refer to the Jukebox portlet's `liferay-portlet.xml` file for examples of defining trash handlers.



Note: Notice that the album trash handler is also specified within the Songs portlet. This was done for organizational purposes. A trash handler refers to an entity, not a portlet. Thus, a trash handler can be declared in any of a plugin's portlets. For neatness, however, they are usually declared in the principal portlet of the suite.

Also, the Jukebox portlet does not send artist assets to the Recycle Bin. Therefore, there is no trash handler specified for artists.

Great! So we have trash handlers ready to manage our trash entries, but we still need a way to get the entries into the Recycle Bin. Let's create a service method to move entries there.

7.5.1.3. Moving Entries Step 3: Create a Service Method to Move Entries to the Recycle Bin

We'll implement a local service method that actually moves entries to the Recycle Bin. This service method must implement the trash service for the entity.

For example, the Jukebox portlet's `SongLocalServiceImpl` class implements the following method to move song entities to the Recycle Bin:

```
@Indexable(type = IndexableType.REINDEX)
public Song moveSongToTrash(long userId, Song song)
throws PortalException, SystemException {

    ServiceContext serviceContext = new ServiceContext();

    // Entry
```

```

User user = userPersistence.findByPrimaryKey(userId);
Date now = new Date();

int oldStatus = song.getStatus();

song.setModifiedDate(serviceContext.getModifiedDate(now));
song.setStatus(WorkflowConstants.STATUS_IN_TRASH);
song.setStatusByUserId(user.getUserId());
song.setStatusByUserName(user.getFullName());
song.setStatusDate(serviceContext.getModifiedDate(now));

// Asset

assetEntryLocalService.updateVisible(
    Song.class.getName(), song.getSongId(), false);

// Trash

UnicodeProperties typeSettingsProperties = new UnicodeProperties();
typeSettingsProperties.put("title", song.getName());

TrashEntry trashEntry = trashEntryLocalService.addTrashEntry(
    userId, song.getGroupId(), Song.class.getName(), song.getSongId(),
    song.getUuid(), null, oldStatus, null, typeSettingsProperties);

song.setName(TrashUtil.getTrashTitle(trashEntry.getEntryId()));

songPersistence.update(song);

return song;
}

```

Notice that this method is annotated as `@Indexable`. That means that every time a song entry is moved to the Recycle Bin, Liferay reindexes the song entities and their corresponding trash entries. This makes trashed entries searchable from within the Recycle Bin, but not searchable outside of it. Trashed entries are not included in search results outside of the Recycle Bin.

Another important aspect of this method is its call to

`song.setStatus(WorkflowConstants.STATUS_IN_TRASH)`, which sets the song's status so that workflows know the song is in the trash.

Next, the asset's visibility is updated so a user can no longer view it outside the Recycle Bin. Its visibility is deactivated by the following call:

```
assetEntryLocalService.updateVisible(Song.class.getName(), song.getSongId(), false);
```

On first thought, this may seem a bit odd. Why are we making the entry invisible in its original location? I thought we were moving it to the Recycle Bin? Importantly, entries that are moved to the Recycle Bin are actually left in their original location, but with their visibility turned off.



Note: If you're not using assets with your entity, you'll need to filter the elements in

your UI by status, so only approved entities are shown. Otherwise, your app will display approved entities and trashed entities together. Only assets can use the `updateVisible()` method to update their status.

Next, notice that the service method adds a new trash entry in the Recycle Bin:

```
TrashEntry trashEntry = trashEntryLocalService.addTrashEntry(
    userId, song.getGroupId(), Song.class.getName(), song.getSongId(),
    song.getUuid(), null, oldStatus, null, typeSettingsProperties);
```

Lastly, the `moveSongToTrash()` service method invokes the

`TrashUtil.getTrashTitle` method to set the entry's *trash title*, which is an alternative reference to the entry. The trash title prevents duplicate entry name conflicts, which we'll discuss in more detail in the later section on resolving conflicts.

7.5.1.4. Moving Entries Step 4: Create a Portlet Action to Initiate Moving Entries to Recycle Bin

Great! Now you must provide the means of invoking the service method from your portlet. We'll implement this using a portlet action that can be triggered from a JSP.

The `JukeboxPortlet` class, which extends `MVCPortlet`, provides the `deleteSong()` portlet action method to invoke the `SongLocalServiceImpl`'s `moveSongToTrash()` service method:

```
public void deleteSong(ActionRequest request, ActionResponse response)
    throws Exception {

    long songId = ParamUtil.getLong(request, "songId");

    boolean moveToTrash = ParamUtil.getBoolean(request, "moveToTrash");

    ServiceContext serviceContext = ServiceContextFactory.getInstance(
        Song.class.getName(), request);

    try {
        if (moveToTrash) {
            Song song = SongServiceUtil.moveSongToTrash(songId);
            ...
            SessionMessages.add(request, PortalUtil.getPortletId(request) +
                SessionMessages.KEY_SUFFIX_DELETE_SUCCESS_DATA, data);
            ...
        }
        else {
            SongServiceUtil.deleteSong(songId, serviceContext);

            SessionMessages.add(request, "songDeleted");
        }
        ...
    }
    ...
}
```

Note that the `try` block contains logic for handling whether the entry is to be moved to the

Recycle Bin or permanently deleted.

You can use a JSP to invoke your portlet action. Our Jukebox portlet's view_song.jsp file provides buttons named *Move to the Recycle Bin* and *Delete* to trigger trashing or permanently deleting a song:

```
<aui:nav>
    <portlet:actionURL name="deleteSong" var="deleteSongURL">
        <portlet:param name="songId" value="<%=
String.valueOf(song.getSongId()) %>" />
        <portlet:param name="moveToTrash" value="<%=
String.valueOf(trashEnabled) %>" />
        <portlet:param name="redirect" value="<%=
redirect %>" />
    </portlet:actionURL>

    <c:choose>
        <c:when test="<%=
trashEnabled %>">
            <aui:nav-item href="<%=
deleteSongURL %>" iconCssClass="icon-trash" label="move-to-the-recycle-bin" />
        </c:when>
        <c:otherwise>
            <aui:nav-item href="<%=
deleteSongURL %>" iconCssClass="icon-key" label="delete" useDialog="<%=
true %>" />
        </c:otherwise>
    </c:choose>
</aui:nav>
```

Notice this JSP code specifies the `JukeboxPortlet.java`'s `deleteSong` action method. Also, the button appropriate for recycling or deleting is displayed depending on the value of the `trashEnabled` boolean variable.

Let's wrap up our objective of moving entries to the Recycle Bin by providing the means to render the trashed entries in the Recycle Bin. For this, we must implement a trash renderer for the trash-enabled entities.

7.5.1.5. Moving Entries Step 5: Implement a Trash Renderer for Each Trash-Enabled Entity

Now that we have the necessary classes and methods to accomplish moving entries to the Recycle Bin, let's implement the appropriate renderer so we can see our trashed entries. Similar to the trash handler, you'll need to create a class that renders trash entries in the Recycle Bin. If you're already using an asset renderer, you can reuse it, as long as it implements the `TrashRenderer` interface too.

As an example of a combined asset renderer and trash renderer implementation, consider the Jukebox portlet's `SongAssetRenderer`.

If you don't have an asset renderer, you'll need to create a trash renderer and implement a `getTrashRenderer()` method to instantiate and return a trash renderer based on a trash entry's primary key. For an example of accessing the trash renderer from a trash handler, consider the following method from the Document Library class `DLFileShortcutTrashHandler`:

```
@Override
```

```

public TrashRenderer getTrashRenderer(long classPK)
throws PortalException, SystemException {
    DLFileShortcut fileShortcut = getDLFileShortcut(classPK);
    return new DLFileShortcutTrashRenderer(fileShortcut);
}

```

It creates a new trash renderer class, `DLFileShortcutTrashRenderer`, based on the file shortcut instance.

Congratulations! You now know how to implement moving your app's entries to the Recycle Bin.

You can test the Recycle Bin feature in the example Jukebox portlet by moving songs to the Recycle Bin.

The first screenshot shows the main 'Songs' list with a search bar and a blue 'Kashmir' entry highlighted. The second screenshot shows the details for 'Kashmir' with a red box around the 'Move to the Recycle Bin' button. The third screenshot shows the 'Actions' menu with 'Restore' and 'Delete' options.

Now that you've provided the means for putting entries into the Recycle Bin, how about providing a way to restore them? Don't worry. Restoring entries from the Recycle Bin to their original state is straightforward. Let's learn how to do this next.

7.5.2. Restoring Entries from the Recycle Bin

Now that you're able to move entries *to* the Recycle Bin, we'll make sure your app can restore entries *from* the Recycle Bin.

Besides, what's the point of having a Recycle Bin if you can't restore its entries?

Entries are restored by returning their visibility

in their original location and removing them from the Recycle Bin. As we briefly discussed in the last section, entries are never removed from their original location; their visibility is simply turned off and a reference to the original entry is placed in the Recycle Bin. We call this reference a *trash entry*. Therefore, the restoration process is very similar to the moving process: you'll just need to return the entry's visibility in its original location and delete its trash entry.

We'll guide you through the following steps for implementing the restoration capability:

1. Create a Service Method to Restore Entries from the Recycle Bin
2. Invoke the Service Method from the Trash Handler

We'll begin this process by creating a service method for the restoration process.

7.5.2.1. Restoring Entries Step 1: Create a Service Method to Restore Entries from the Recycle Bin

You'll need to create a service method that removes the trash entry from the Recycle Bin and makes the asset entry visible again in its original location.

As an example, the `restoreSongFromTrash()` service method from the Jukebox portlet's `SongLocalServiceImpl` restores songs from the Recycle Bin:

```
@Indexable(type = IndexableType.REINDEX)
@Override
public Song restoreSongFromTrash(long userId, long songId)
    throws PortalException, SystemException {

    ServiceContext serviceContext = new ServiceContext();

    // Entry

    User user = userPersistence.findByPrimaryKey(userId);
    Date now = new Date();

    TrashEntry trashEntry = trashEntryLocalService.getEntry(
        Song.class.getName(), songId);

    Song song = songPersistence.findByPrimaryKey(songId);

    song.setName(TrashUtil.getOriginalTitle(song.getName()));
    song.setModifiedDate(serviceContext.getModifiedDate(now));
    song.setStatus(trashEntry.getStatus());
    song.setStatusByUserId(user.getUserId());
    song.setStatusByUserName(user.getFullName());
    song.setStatusDate(serviceContext.getModifiedDate(now));

    songPersistence.update(song);

    assetEntryLocalService.updateVisible(
        Song.class.getName(), song.getSongId(), true);

    trashEntryLocalService.deleteEntry(Song.class.getName(), songId);

    return song;
```

```
}
```

First, we restore the item's original name with the help of `TrashUtil`'s `getOriginalTitle()` method. Then we set the entry's modified date to the current date.

```
song.setName(TrashUtil.getOriginalTitle(song.getName()));  
song.setModifiedDate(serviceContext.getModifiedDate(now));
```

Next, we update the entry's status by setting it to the original status from before the entry was trashed. For instance, if the entry was originally a draft (`STATUS_DRAFT`), it's restored back to a draft. The status is updated by the following call:

```
song.setStatus(trashEntry.getStatus());
```

There are also several other statuses we update, which include the status by user ID, by user name, and the status date. These indicate the user that restored the trash entry and set the date the status was modified.

```
song.setStatusByUserId(user.getUserId());  
song.setStatusByUserName(user.getFullName());  
song.setStatusDate(serviceContext.getModifiedDate(now));
```

Then we make the asset entry visible in its original location by calling:

```
assetEntryLocalService.updateVisible(Song.class.getName(), song.getSongId(),  
true);
```

Lastly, we delete the trash entry from the Recycle Bin by calling:

```
trashEntryLocalService.deleteEntry(Song.class.getName(), songId);
```

The entry is restored and no longer resides in the Recycle Bin.

Importantly, after writing your service method, make sure to run Service Builder to generate the corresponding service interface and utility methods.

To finish implementing the entry restoration process, let's invoke the service method from the entity's trash handler.

7.5.2.2. Restoring Entries Step 2: Invoke the Service Method from the Trash Handler

Now that our service provides a method for restoring the entry, we must invoke it from our trash handler's `restoreTrashEntry()` method. The Recycle Bin framework calls this method when a user clicks the trash entry's *Restore* button.

The following `restoreTrashEntry()` method implementation is from the Jukebox portlet's `SongTrashHandler` class.

```
@Override  
public void restoreTrashEntry(long userId, long classPK)  
throws PortalException, SystemException {  
  
    SongLocalServiceUtil.restoreSongFromTrash(userId, classPK);  
}
```

When Jukebox users want to restore a song from the Recycle Bin, they simply click the song's *Restore* button.



Note: Sometimes, conflicts can occur when restoring entries. For instance, suppose you create a file with the same name of a file that you've trashed. Although the file is

in the Recycle Bin, it's still present in its original location, but with its status changed and visibility turned off. The resolution framework avoids these two files conflicting. We'll talk more about the resolution framework in the *Resolving Conflicts* section.

You now know how to provide the means for users to restore entries from the Recycle Bin! Next, let's implement the convenient *Undo* button so users don't have to visit the Recycle Bin to restore an item they just trashed.

7.5.3. Implementing the Undo Functionality

Sometimes, you may accidentally send the wrong entry to the Recycle Bin. It seems kind of grueling to navigate away from your page to the Recycle Bin to restore the item, just to go back to where you originally started, right? For this reason, the Recycle Bin framework supports an *Undo* button so you can conveniently undo the action of sending an entry to the Recycle Bin without leaving the page. In addition, the undo functionality provides links to the trashed entry and the Recycle Bin.

The Song *Kashmir* was moved to the Recycle Bin. [Undo](#)

We'll guide you through the following steps in implementing the Undo functionality:

1. Add the Undo Tag
2. Call the Action for Restoration
3. Display the Taglib

Let's implement the Undo button and its related links!

7.5.3.1. Implementing the Undo Functionality Step 1: Add the Undo Tag

The first thing you'll need to do is add the `<liferay-ui:trash-undo>` taglib in our JSP. Then, you need to set a portlet action URL and pass it to the `<liferay-ui:trash-undo>` taglib. This maps the taglib's *Undo* button to the portlet action that we'll implement in the next step.

Here's an example of how the Jukebox portlet's view.jsp implements this:

```
<portlet:actionURL name="restoreSong" var="undoTrashURL" />  
  
<liferay-ui:trash-undo portletURL="<%=" undoTrashURL %>" />
```

Now that we've added the taglib and action URL, let's implement the portlet action to restore the entry.

7.5.3.2. Implementing the Undo Functionality Step 2: Create a Portlet Action to Initiate Restoration

You must create a portlet action method that invokes your service method to restore the entry.

The following portlet action method from the JukeboxPortlet class restores the song from the Recycle Bin:

```
public void restoreSong(ActionRequest request, ActionResponse response)
    throws Exception {

    long[] restoreEntryIds = StringUtil.split(
        ParamUtil.getString(request, "restoreEntryIds"), 0L);

    for (long restoreEntryId : restoreEntryIds) {
        SongServiceUtil.restoreSongFromTrash(restoreEntryId);
    }
}
```

This method implements the `restoreSong` action that we named in the `view.jsp`. The action URL maps the `<liferay-ui:trash-undo>` taglib to this method.

Note how it parses entry IDs from the request object. It restores all of these entries by calling the `restore*` service method we defined in the previous *Restoring Entries from the Recycle Bin* section.

Are you wondering how this portlet action gets the ID of the entries to restore? We'll show you how to pass this data to the session next.

7.5.3.3. Implementing the Undo Functionality Step 3: Providing Trash Entry Data for the Taglib

The final step for implementing the Undo button is to provide the trashed entry's information to the `<liferay-ui:trash-undo>` taglib. In order for the taglib to display properly, we must provide some information for the session messages, so that the session knows which entries were just deleted. Once the session knows which elements were just deleted, our method can use that information to restore the entries.

The following `if` statement from the `deleteSong()` method of the JukeboxPortlet class populates the session with the entries that were just deleted:

```
if (moveToTrash) {
    Song song = SongServiceUtil.moveSongToTrash(songId);

    Map<String, String[]> data = new HashMap<String, String[]>();

    data.put("deleteEntryClassName",
        new String[] {Song.class.getName()});
    data.put("deleteEntryTitle",
        new String[] {TrashUtil.getOriginalTitle(song.getName())});
    data.put("restoreEntryIds",
        new String[] {String.valueOf(songId)});
```

```

SessionMessages.add(request, PortalUtil.getPortletId(request) +
SessionMessages.KEY_SUFFIX_DELETE_SUCCESS_DATA, data);

SessionMessages.add(request, PortalUtil.getPortletId(request) +
SessionMessages.KEY_SUFFIX_HIDE_DEFAULT_SUCCESS_MESSAGE);
}

```

It gathers the elements needed to distinguish each entry instance to restore. For the Jukebox song element's we include the song's class name, title, and IDs.

```

data.put("deleteEntryClassName",
new String[] {Song.class.getName()});
data.put("deleteEntryTitle",
new String[] {TrashUtil.getOriginalTitle(song.getName())});
data.put("restoreEntryIds",
new String[] {String.valueOf(songId)});

```

Then it adds these elements to the session messages, so the session knows which elements were deleted.

```

SessionMessages.add(request, PortalUtil.getPortletId(request) +
SessionMessages.KEY_SUFFIX_DELETE_SUCCESS_DATA, data);

SessionMessages.add(request, PortalUtil.getPortletId(request) +
SessionMessages.KEY_SUFFIX_HIDE_DEFAULT_SUCCESS_MESSAGE);

```

In your portlet's delete action, you can similarly populate the session with the entry information of the entries being deleted. It's simple really. Now you know how to implement the Undo functionality for your app's trash-enabled entities.

Next, let's learn how to move and restore container entities to and from the Recycle Bin.

7.5.4. Moving/Restoring Parent Entities

To this point, we've only discussed how to move/restore single entities to/from the Recycle Bin. What happens if we need to trash a parent entity, such as a parent wiki page, or trash an entity like a web content folder that aggregates web content articles? The fundamental logic for handling them is similar to handling single entities, but there are a few different aspects that must be considered.

We'll look at the Jukebox portlet's Album entity to understand handling container entities, since each album is a collection of song entities. Let's learn how the Jukebox portlet is able to move/restore albums to/from the Recycle Bin so you can translate that knowledge to container entities in your personal app.

7.5.4.1. Moving/Restoring Parent Entities Step 1: Mark Container Model Service Entities

The first thing we'll need to do is define each container model as such in our service. To do this, open your app's service.xml file and add the `container-model="true"` attribute in the container entity's primary key `<column>` tag.

For example, the Jukebox portlet's service.xml file marks the album entity as a container model in its `albumId` column:

```
<column name="albumId" type="long" primary="true" container-model="true" />
```

On running Service Builder for the portlet's service, with its newly designated container model entities, Service Builder generates several methods that can be used to obtain the child entities of each parent, or in the case of Jukebox, songs of each album.

Service Builder generates `*Model.java`, `*ModelClp.java`, and `*ModelImpl.java` classes to implement the `ContainerModel` interface. These implementations enable the Recycle Bin framework to identify and use these models as container models.

Next, we'll handle the child entities of the parent entity.

7.5.4.2. Moving/Restoring Parent Entities Step 2: Manage Children Entities

Implementing the moving/restoring parent entities to/from the Recycle Bin is very similar to what we've already learned for single entities. However, because the parents hold child entities, the child entities must be accounted for. The service method for moving a parent entry to the Recycle Bin must also move the parent's child entries to the Recycle Bin. Likewise, the service method for restoring a parent entry from the Recycle Bin must also restore the parent's child entries.

On moving a parent entity to the Recycle Bin, the following things must be done with each of its dependent entities:

1. Update its status
2. Add a trash version for it
3. Turn off the visibility of its asset
4. Reindex it

Let's look at how Jukebox portlet's `AlbumLocalServiceImpl` class handles moving an album's songs to the Recycle Bin by calling a local convenience method

`moveDependentsToTrash()`. Here's the source code for the

`moveDependentsToTrash()` method:

```
protected void moveDependentsToTrash(List<Song> songs, long trashEntryId)
    throws PortalException, SystemException {

    for (Song song : songs) {

        // Entry

        if (song.isInTrash()) {
            continue;
        }

        int oldStatus = song.getStatus();

        song.setStatus(WorkflowConstants.STATUS_IN_TRASH);

        songPersistence.update(song);

        // Trash
    }
}
```

```

        int status = oldStatus;

        if (oldStatus == WorkflowConstants.STATUS_PENDING) {
            status = WorkflowConstants.STATUS_DRAFT;
        }

        if (oldStatus != WorkflowConstants.STATUS_APPROVED) {
            trashVersionLocalService.addTrashVersion(
                trashEntryId, Song.class.getName(), song.getSongId(),
                status, null);
        }

        // Asset

        assetEntryLocalService.updateVisible(
            Song.class.getName(), song.getSongId(), false);

        // Indexer

        Indexer indexer = IndexerRegistryUtil.nullSafeGetIndexer(
            Song.class);

        indexer.reindex(song);
    }
}

```

For each of the album's songs, let's consider how this method updates its status, adds a trash version for it, turns off the visibility of its asset, and reindexes it for search purposes.

Here's the breakdown of how the this method implements this for the song entity:

1. *Update its status:* Similar to updating the status for a single entry, the status of each album's song must reflect that the song has been moved to the Recycle Bin. The following code updates the song's status:

```

song.setStatus(WorkflowConstants.STATUS_IN_TRASH);

songPersistence.update(song);

```

2. *Add a trash version for it:* When moving content with versions to the Recycle Bin, the trash version entity stores the status of those versions, so those statuses can be set back to their original values when the entity is restored. When a parent entity with content is sent to the Recycle Bin, each element in that parent also generates a trash version. You can view trash versions by navigating inside a trash entry. Adding the trash version for the song is done with the following code:

```

trashVersionLocalService.addTrashVersion(trashEntryId,
    Song.class.getName(), song.getSongId(), status, null);

```

3. *Turn off the visibility of its asset:* The song's visibility is turned off in the following code:

```

assetEntryLocalService.updateVisible(Song.class.getName(), song.getSongId(),
false);

```

4. *Reindex it:* Since we've modified the visibility of the song entity, we must reindex the album's songs so they're searchable. Here's the code that accomplishes this:

```
Indexer indexer = IndexerRegistryUtil.nullSafeGetIndexer(Song.class);  
  
indexer.reindex(song);
```

You'll be comforted to know that restoring a parent's child entities from the Recycle Bin involves similar steps. The `restoreDependentsFromTrash()` convenience method in the Jukebox portlet's `AlbumLocalServiceImpl` class demonstrates these steps. As a summary, here's what you do for each child entity:

1. Update its status
2. Remove the trash version of it
3. Turn on the visibility of its asset
4. Reindex it

It's really very straightforward, making it easy for you to do in the service classes for your app's parent entities.

You can deploy the Jukebox in your portal and test trashing/restoring albums and songs. As a Jukebox user, you can click inside a trashed album to view its trashed songs. You also have the option of restoring individual songs or deleting them permanently. Note that the Jukebox developers didn't create this UI; it was all generated by the portal framework. As a developer, you only need to tell the portal that the parent entity has children and how to obtain them. The Recycle Bin UI automatically accounts for the parent's children for you.

Terrific! You've learned how to designate parent models as container model entities in your service definition, you've provided a means to trash/restore a parent's child entities when trashing/restoring that parent, and you've learned how the powerful Recycle Bin UI lets you work with a parent's child entities.

Have you wondered what happens when there are conflicts moving and restoring entries to and from the Recycle Bin? Let's learn how to deal with conflicts next.

7.5.5. Resolving Conflicts

The Conflict Resolution framework helps Liferay users identify and solve conflicts in the Recycle Bin. The most common conflict for the Recycle Bin is duplicate naming. For instance, you have file 1 and file 2, both with the same name. Suppose you move file 1 to the Recycle Bin from a folder. Then, in that folder, you create file 2. You would get a naming conflict because, although file 1 is in the Recycle Bin, it is also still located in the original folder, but with its status changed and visibility turned off.

Let's learn how to implement the Conflict Resolution framework so we can avoid Recycle Bin conflicts.

7.5.5.1. Resolving Conflicts Step 1: Rename Entities Sent to the Recycle Bin

When an entry is sent to the Recycle Bin, it is essentially replicated. We keep the entity in its original location and create a similar entry in the Recycle Bin. When viewing the entry from the Recycle Bin UI, the name appears the same as the original entry. But the portal needs a way to distinguish between the two entries. This requires adding logic to your entity's service and

leveraging trash utilities to generate new names for the Recycle Bin entries.

When an entry is sent to the Recycle Bin, it's still located in its original location, but with a different status. Users may create an entity with a name, move it to the Recycle Bin, and then create another entity with the same name. Since both entities are residing in the same location, a naming conflict could occur. Some applications only allow one entity to exist with a particular field; for example, names for documents, titles for songs in an album, friendly URLs for pages, etc. On moving an entity to the Recycle Bin, we need to rename its fields that could generate conflicts to something unique.

```
UnicodeProperties typeSettingsProperties = new UnicodeProperties();  
  
typeSettingsProperties.put("title", song.getName());  
  
TrashEntry trashEntry = trashEntryLocalService.addTrashEntry(  
    userId, song.getGroupId(), Song.class.getName(), song.getSongId(),  
    song.getUuid(), null, oldStatus, null, typeSettingsProperties);
```

```
song.setName(TrashUtil.getTrashTitle(trashEntry.getEntryId()));
```

We create a `UnicodeProperties` instance to hold a mapping of each song's title. The mapping is stored with the trash entry. We invoke

`TrashUtil.getTrashTitle(trashEntry.getEntryId())` from `SongLocalServiceImpl`'s `moveSongToTrash` method to set the name of the original song to a unique value that we can use to look up the trash entry associated with the song. Invoking `TrashUtil.getTrashTitle(trashEntry.getEntryId())` resets the name of the original song to a unique value--a slash followed by the ID of the song's trash entry. Since the song is now in the Recycle Bin, it is hidden from viewing in its original location; so you don't have to worry about this lookup value being visible to users. As you'll see shortly, the unique names generated by `TrashUtil.getTrashTitle(...)` are used for looking up the names of the original entities, on restoring those entities from the Recycle Bin.

Recycle Bin

Entries that have been in the Recycle Bin for more than 1 month will be automatically deleted. [Empty the Recycle Bin](#)

Keywords

Recycle Bin

Name	Type	Removed Date	Removed By
 Kashmir	Song	2 Minutes Ago	Joe Bloggs
 Kashmir	Song	6 Seconds Ago	Joe Bloggs

Next, we'll consider how to restore the original name of each trashed entity when it's restored from the Recycle Bin.

7.5.5.2. Resolving Conflicts Step 2: Restore the Entity's Original Name When Restoring From Recycle Bin

Since your entity is renamed, you'll need a way to retrieve its old name, in the case that the entity gets restored. The code snippet below, found in the `restoreSongFromTrash (long userId, long songId)` method of the Jukebox portlet's `SongLocalServiceImpl` portrays how this is done:

```
Song song = songPersistence.findByPrimaryKey(songId);
```

```
song.setName(TrashUtil.getOriginalTitle(song.getName()));
```

The original song is retrieved by its ID. Then the `TrashUtil.getOriginalTitle(...)` method is called to look up the song's original name. Remember that the song's current name is based on the trash entry's ID. `TrashUtil's getTrashTitle ()` method looks up the trash entry and returns the *title* value (the song's original name) previously mapped in the entry's type

settings properties. We set the original song name back to the song entity.

Lastly, whether an entity is in the Recycle Bin or not, it's always nice to render the entity with its original name. As an example of rendering a song's original title based on a locale, the Jukebox portlet's SongAssetRenderer provides the following implementation:

```
@Override  
public String getTitle(Locale locale) {  
    if (!_song.isInTrash()) {  
        return _song.getName();  
    }  
  
    return TrashUtil.getOriginalTitle(_song.getName());  
}
```

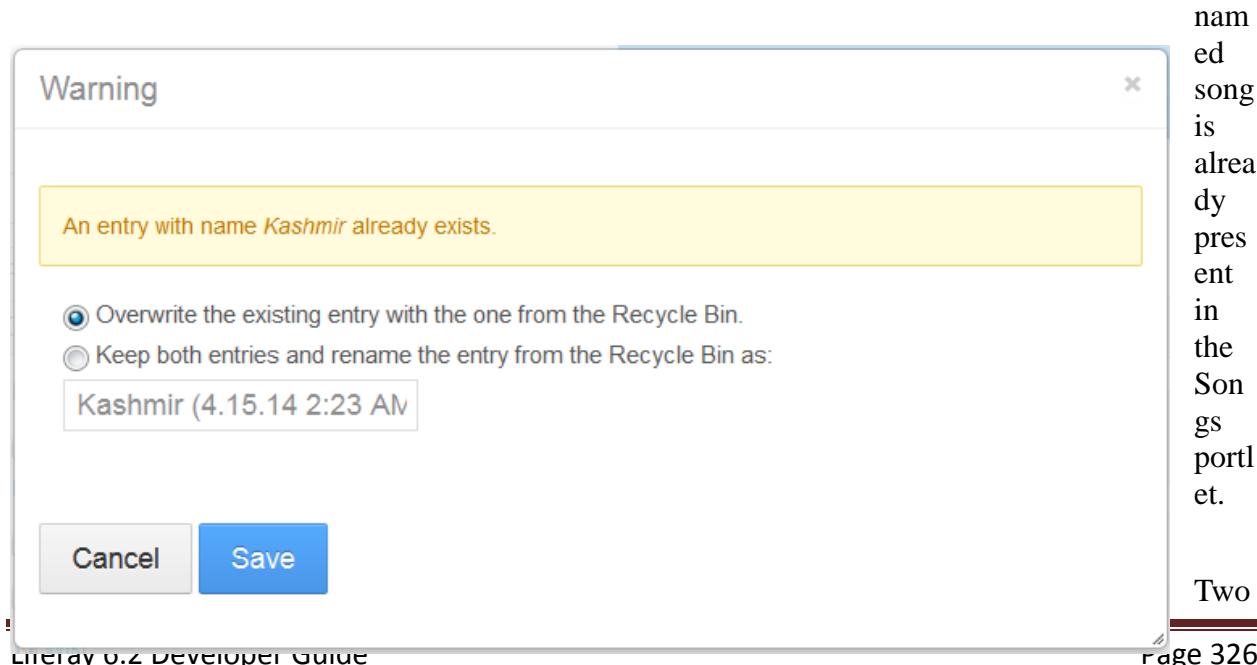
If the song isn't in the Recycle Bin, the song's current name is returned. Otherwise, the method invokes `TrashUtil.getOriginalTitle(_song.getName())` to return the song's original name.

Lastly, we need to implement some required methods to finalize the Conflicts Resolution framework.

7.5.5.3. Resolving Conflicts Step 3: Implement Conflict Resolution Trash Handler Methods

Your app can now uniquely rename entries on removal and reinstate their original names when restored. What happens when the original entry is restored to its original location where a new entry with the same already exists? This causes a naming conflict that needs to be resolved by the user.

The Recycle Bin framework provides a UI for users to decide whether to overwrite the existing entry or to keep both entries by updating the title of the entry they're restoring. The figure below exemplifies a conflict resolution pop-up on trying to restore a song for which an identically



methods need to be implemented in the trash handler to allow for the necessary checks and updates. The first method should check for duplicate trash entries. If an entry with the same name is detected in a directory, an exception must be thrown. You can reference the `checkDuplicateTrashEntry()` method from `SongTrashHandler` to see how this is done.

Lastly, implement a method that updates the entry title name. For instance, the Jukebox portlet updates a song title by calling the `updateTitle()` method, which can also be found in the `SongTrashHandler` class. This method is called when the entry you're restoring needs its title updated, so it no longer conflicts with the preexisting entry that has the same name. After implementing these methods, you should always be able to resolve naming conflicts between your trashed entries.

Fantastic! By leveraging the Conflicts Resolution framework in your app, you're able to provide a smarter Recycle Bin that handles potential conflicts with ease.

Congratulations! You've finished your journey through the Recycle Bin framework. You accomplished this feat by developing your app to move entries to the Recycle Bin, restore entries from the Recycle Bin, use the Undo action, move and restore parent entities, and use the Recycle Bin's Resolution Conflicts framework. Your *dirty* work is complete and you now know how to implement the Recycle Bin framework in your app.

Next, let's learn about Liferay's Message Bus.

7.6. Using Message Bus

The *Message Bus* is a service level API used to exchange messages within Liferay. The Message Bus is a mechanism for sending message payloads to different components in Liferay, providing loose coupling between message producers and consumers to prevent class loading issues. It's located in the global class loader, making it accessible to every deployed web application. Remote messaging isn't supported, but messages are sent across a cluster when ClusterLink is enabled.

Message Bus has several common uses, including sending search index write events, sending subscription emails, handling messages at scheduler endpoints, and running asynchronous processes.

You can leverage Message Bus to send messages between and within your plugins.

As we show you Message Bus we'll talk about things like *synchronous* and *asynchronous* messaging, *serial* vs. *in-parallel* message dispatching, and Java and JSON style messages formats.

Before we get into those topics, let's first try to understand Message Bus System's architecture.

7.6.1. The Message Bus System

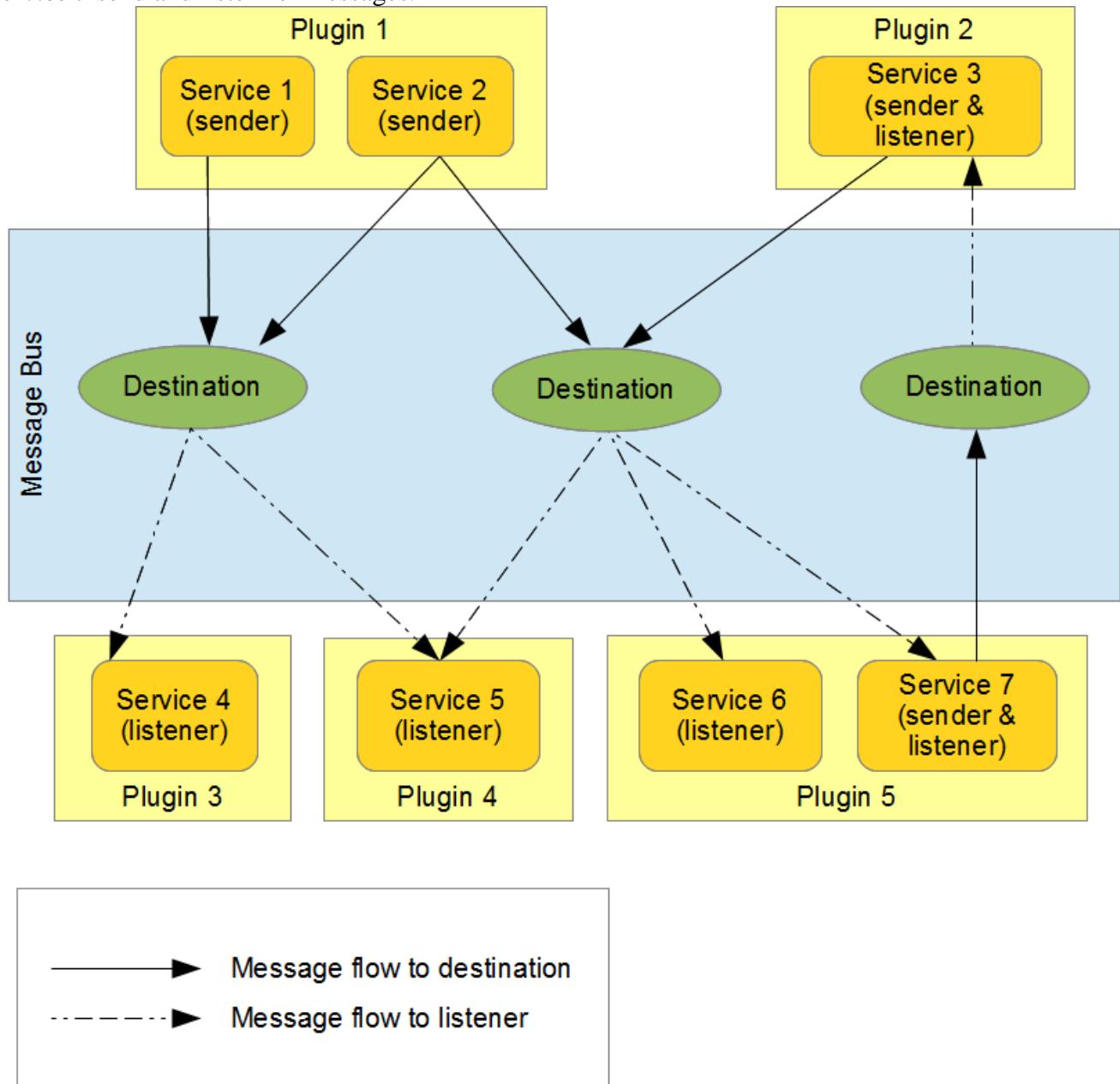
The Message Bus system contains the following components:

- **Message Bus:** Manages transfer of messages from message *senders* to message *listeners*.
- **Destinations:** Addresses or endpoints to which *listeners* register to receive messages.
- **Listeners:** Consume messages received at destinations. They receive all messages sent to

their registered destinations.

- **Senders:** Invoke the Message Bus to send messages to destinations.

Your services can send messages to one or more destinations, and can listen to one or more destinations. The figure below depicts this. An individual service can be both a message sender and a message listener. For example, in the figure below both *Plugin 2 - Service 3* and *Plugin 5 - Service 7* send and listen for messages.



The Message Bus supports *synchronous* and *asynchronous* messaging:

- **Synchronous messaging:** After it sends a message, the sender blocks waiting for a

response from a recipient.

- **Asynchronous messaging:** After it sends a message, the sender is free to continue processing. The sender can be configured to receive a call-back or can simply send and forget. We'll show you how to implement both synchronous and asynchronous messaging in this section.
 - **Call-back:** The sender can include a call-back destination key as the response destination for the message. The recipient (listener) can then send a response message back to the sender via this response destination.
 - **Send-and-Forget:** The sender includes no call-back information in the message sent and continues with processing.

Configuration of Message Bus is done using the following files:

- WEB-INF/src/META-INF/messaging-spring.xml: Specifies your destinations, listeners, and their mappings to each other.
- WEB-INF/web.xml: Holds a listing of deployment descriptors for your plugin. Make sure you add messaging-spring.xml to your list of Spring configurations in this file.



Note: The internal file META-INF/messaging-core-spring.xml of portal-impl.jar specifies the default Message Bus class, default asynchronous message sender class, and default synchronous message sender class for Liferay.

You can control your *Message Types* by using either the `Message` or `JSONObject` class. Liferay core services are typically serialized and deserialized in JSON. In our examples we'll demonstrate both types of message classes.

So far we've introduced the Message Bus System, including message types, destinations, senders, listeners, and approaches to sending messages. Next we'll show you how easy it is to create your destinations, register listeners, and send your messages. To demonstrate, we'll implement a business use case.

7.6.2. Example Use Case--Procurement Process

Our use case will consider Jungle Gyms R-Us and its distribution of playground equipment, buying the equipment from manufacturers and selling the equipment to retailers. We'll focus on the company's process for procuring new jungle gym equipment. Let's lay out this process now.

Jungle Gyms R-Us employs the following departments in their procurement process:

- *Procurement Department:* Scouts out the latest equipment deals of manufacturers.
- *Finance Department:* Determines whether the equipment can be purchased based on budget.
- *Legal Department:* Determines whether the equipment's safety ratings are acceptable.
- *Warehouse Department:* Receives the equipment, stores it, and prepares it for shipping.
- *Sales Department:* Builds relationships with prospective customers to sell them products.

The departments currently use email to exchange comments about new equipment purchases, but

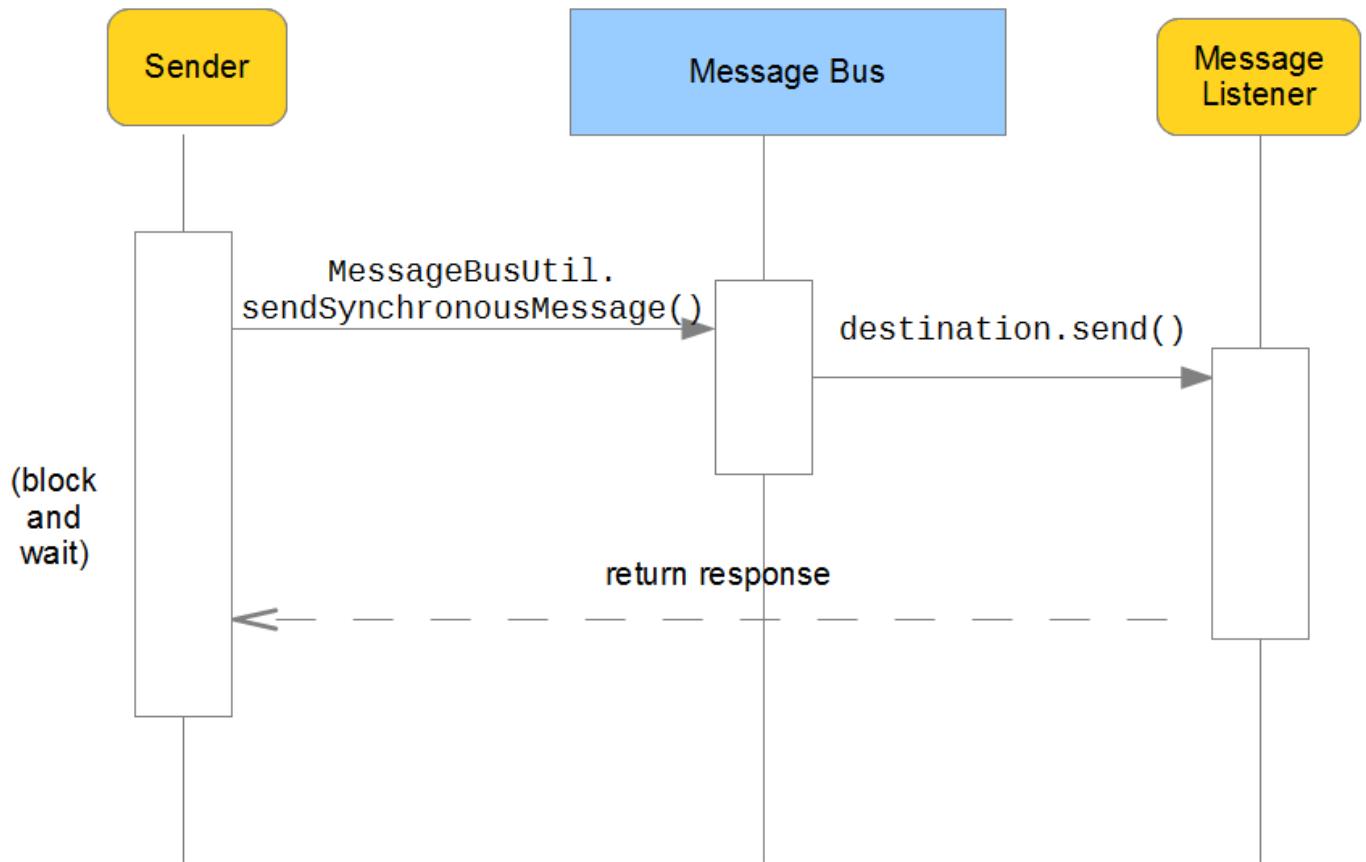
someone always seems to be left out of the loop. One time, Sales was gung-ho about getting their hands on the latest and greatest spring rider animals from Boingo-Boingo Industries, but they didn't consider the failing safety reviews discovered by the Legal department, because the Legal department forgot to copy the Sales department in their email to Procurement. Tempers flew, feelings were hurt, and everybody avoided hanging out in the company breakroom for the next couple of weeks.

Jungle Gyms R-Us could use Liferay's Workflow with Kaleo to resolve the communication breakdown, but we'll resolve the Jungle Gym's communication woes using Message Bus, to show you how it works. Here are the inter-department message exchanges we'll accommodate:

Message	Sender	Listener	Response	Response Listeners
Request permission to proceed with purchase	Procurement	Finance	required	Procurement
Request permission to proceed with purchase	Procurement	Legal	required	Procurement
Notify and solicit feedback on new purchase	Procurement	Warehouse	optional	Procurement, Sales
Notify and solicit feedback on new purchase	Procurement	Sales	optional	Procurement, Warehouse
Broadcast equipment news	Procurement	Employees	none	none
Let's implement Procurement's request to Finance first.				

7.6.3. Synchronous Messaging

In our example, equipment purchases can't proceed without approval from Finance and Legal departments. Since special offers from the manufacturers often only last for a couple hours, Procurement makes it their top priority to get approval as soon as possible. Implementing their exchange using *synchronous* messaging makes the most sense.



The following table describes how we'll set things up:

Destination Key	Type	Sender	Receivers
jungle/finance/purchase	synchronous	Procurement	Finance
jungle/finance/purchase/response	synchronous	Finance	Procurement
jungle/legal/purchase	synchronous	Procurement	Legal
jungle/legal/purchase/response	synchronous	Legal	Procurement

We've set it up so Finance sends its response messages to a destination on which Procurement will listen. That way a full-bodied response message is sent back to Procurement in addition to the response object returned from sending the message.

The Procurement Department sends a purchase approval request:

```

Message message = new Message();
message.put("department", "Procurement");
message.put("partName", part.getName(Locale.US));

message.setResponseId("1111");
message.setResponseDestinationName("jungle/finance/purchase/response");

```

```

try {
    String financeResponse = (String) MessageBusUtil.sendSynchronousMessage(
        "jungle/finance/purchase", message, 10000);

    System.out.println(
        "Procurement received Finance sync response to purchase approval for "
    +
        part.getName(Locale.US) + ":" + financeResponse);

    message.setResponseId("2222");
    message.setResponseDestinationName("jungle/legal/purchase/response");

    String legalResponse = (String) MessageBusUtil.sendSynchronousMessage(
        "jungle/legal/purchase", message, 10000);

    System.out.println(
        "Procurement received Legal sync response to purchase approval for "
    +
        part.getName(Locale.US) + ":" + legalResponse);

    if (financeResponse.contains("yes") && legalResponse.contains("yes")) {
        sendPurchaseNotification(part, userId);
    }
}
catch (MessageBusException e) {
    e.printStackTrace();
}

```

This sender takes the following steps:

1. Creates the message using Liferay's Message class.
2. Stuffs the message with key/value pairs.
3. Sets a response ID and response destination for listeners to use in replying back.
4. Sends the message to the destination with a timeout value of 10,000 milliseconds.
5. Blocks waiting for the response.

Finance Department listens for purchase approval requests and replies back:

```

public class FinanceMessagingImpl implements MessageListener {

    public void receive(Message message) {
        try {
            doReceive(message);
        }
        catch (Exception e) {
            _log.error("Unable to process message " + message, e);
        }
    }

    protected void doReceive(Message message)
        throws Exception {

        String department = (String) message.get("department");
        String partName = (String) message.get("partName");
    }
}

```

```

        System.out.println("Finance received purchase request for " +
            partName + " from " + department);

        Message responseMessage = MessageBusUtil.createResponseMessage(
            message);

        responseMessage.put("department", "Finance");
        responseMessage.put("partName", partName);
        responseMessage.setPayload("yes");

        MessageBusUtil.sendMessage(
            responseMessage.getDestinationName(), responseMessage);
    }

    private static Log _log =
        LogFactoryUtil.getLog(FinanceMessagingImpl.class);
}

```

This listener executes the following steps:

1. Implements the `receive(Message message)` method of the `com.liferay.portal.kernel.messaging.MessageListener` interface.
2. Extracts values from the `Message` parameter by getting values associated with known keys.
3. Creates a `Message` based on the message received via the `MessageBusUtil.createResponseMessage(message)` method, which accesses the response destination name from the `message` variable and sets the destination of the response message.
4. Sets the response message's payload.
5. Sends the response `Message` to the response destination.

You can implement the listener for the Legal Department similarly. Next we'll account for Legal Department-related classes in our configuration.

Message Bus Configuration for the purchase approval request process:

For Message Bus to direct messages from destinations to listeners, we must register the listeners by configuring the appropriate mappings in our plugin's `WEB-INF/src/META-INF/messaging-spring.xml` file (create this file if it's not already in your plugin). Here is the configuration:

```

<?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">

```

```

>

<!-- Listeners -->

<bean
    id="messageListener.finance_listener"
    class="com.liferay.training.parts.messaging.impl.FinanceMessagingImpl"
/>
<bean
    id="messageListener.legal_listener"
    class="com.liferay.training.parts.messaging.impl.LegalMessagingImpl"
/>
<bean
    id="messageListener.procurement_listener"
    class="com.liferay.training.parts.messaging.impl.ProcurementMessagingImpl"
/>

<!-- Destinations -->

<bean
    id="destination.finance.purchase"
    class="com.liferay.portal.kernel.messaging.SynchronousDestination"
>
    <property name="name" value="jungle/finance/purchase" />
</bean>

<bean
    id="destination.finance.purchase.response"
    class="com.liferay.portal.kernel.messaging.SynchronousDestination"
>
    <property name="name" value="jungle/finance/purchase/response" />
</bean>

<bean
    id="destination.legal.purchase"
    class="com.liferay.portal.kernel.messaging.SynchronousDestination"
>
    <property name="name" value="jungle/legal/purchase" />
</bean>

<bean
    id="destination.legal.purchase.response"
    class="com.liferay.portal.kernel.messaging.SynchronousDestination"
>
    <property name="name" value="jungle/legal/purchase/response" />
</bean>

<!-- Configurator -->

<bean
    id="messagingConfigurator"
    class=

```

```

"com.liferay.portal.kernel.messaging.config.PluginMessagingConfigurator"
    >
        <property name="messageListeners">
            <map key-type="java.lang.String" value-type="java.util.List">
                <entry key="jungle/finance/purchase">
                    <list
                        value-type=
                
```

"com.liferay.portal.kernel.messaging.MessageListener"

```

        >
            <ref bean="messageListener.finance_listener" />
        </list>
    </entry>
    <entry key="jungle/finance/purchase/response">
        <list
            value-type=
    
```

"com.liferay.portal.kernel.messaging.MessageListener"

```

        >
            <ref bean="messageListener.procurement_listener" />
        </list>
    </entry>
    <entry key="jungle/legal/purchase">
        <list
            value-type=
    
```

"com.liferay.portal.kernel.messaging.MessageListener"

```

        >
            <ref bean="messageListener.legal_listener" />
        </list>
    </entry>
    <entry key="jungle/legal/purchase/response">
        <list
            value-type=
    
```

"com.liferay.portal.kernel.messaging.MessageListener"

```

        >
            <ref bean="messageListener.procurement_listener" />
        </list>
    </entry>
</map>
</property>
<property name="destinations">
    <list>
        <ref bean="destination.finance.purchase"/>
        <ref bean="destination.finance.purchase.response"/>
        <ref bean="destination.legal.purchase"/>
        <ref bean="destination.legal.purchase.response"/>
    </list>
</property>
</bean>
</beans>
```

The configuration above specifies the following beans:

- *Listener beans*: Specify classes to handle messages.
- *Destination beans*: Specify the class *type* and *key* names of the destinations.
- *Configurator bean*: Maps listeners to their destinations.

When Finance sends its purchase approval request message for a new three-story spiral slide, the console reports Finance's receipt of the message, Procurement's receipt of the callback response from Finance, and Procurement's receipt of the synchronous response returned from sending the message. Here's what the console message looks like:

```
Finance received purchase request for three-story spiral slide from  
Procurement  
Procurement received Finance callback response to purchase approval for  
three-  
story spiral slide: yes  
Procurement received Finance sync response to purchase approval for three-  
story  
spiral slide: yes  
Legal received purchase request for three-story spiral slide from Procurement  
Procurement received Legal callback response to purchase approval for three-  
story spiral slide: yes  
Procurement received Legal sync response to purchase approval for three-story  
spiral slide: yes
```

Do you know what all those *yes* messages mean? Success! Jungle Gyms R-Us has the cash to purchase this cool new slide, and the Legal Department has no gripes about the slide's safety ratings!

Next let's have Procurement notify the Sales and Warehouse departments and solicit their feedback.

7.6.4. Asynchronous Messaging with Callbacks

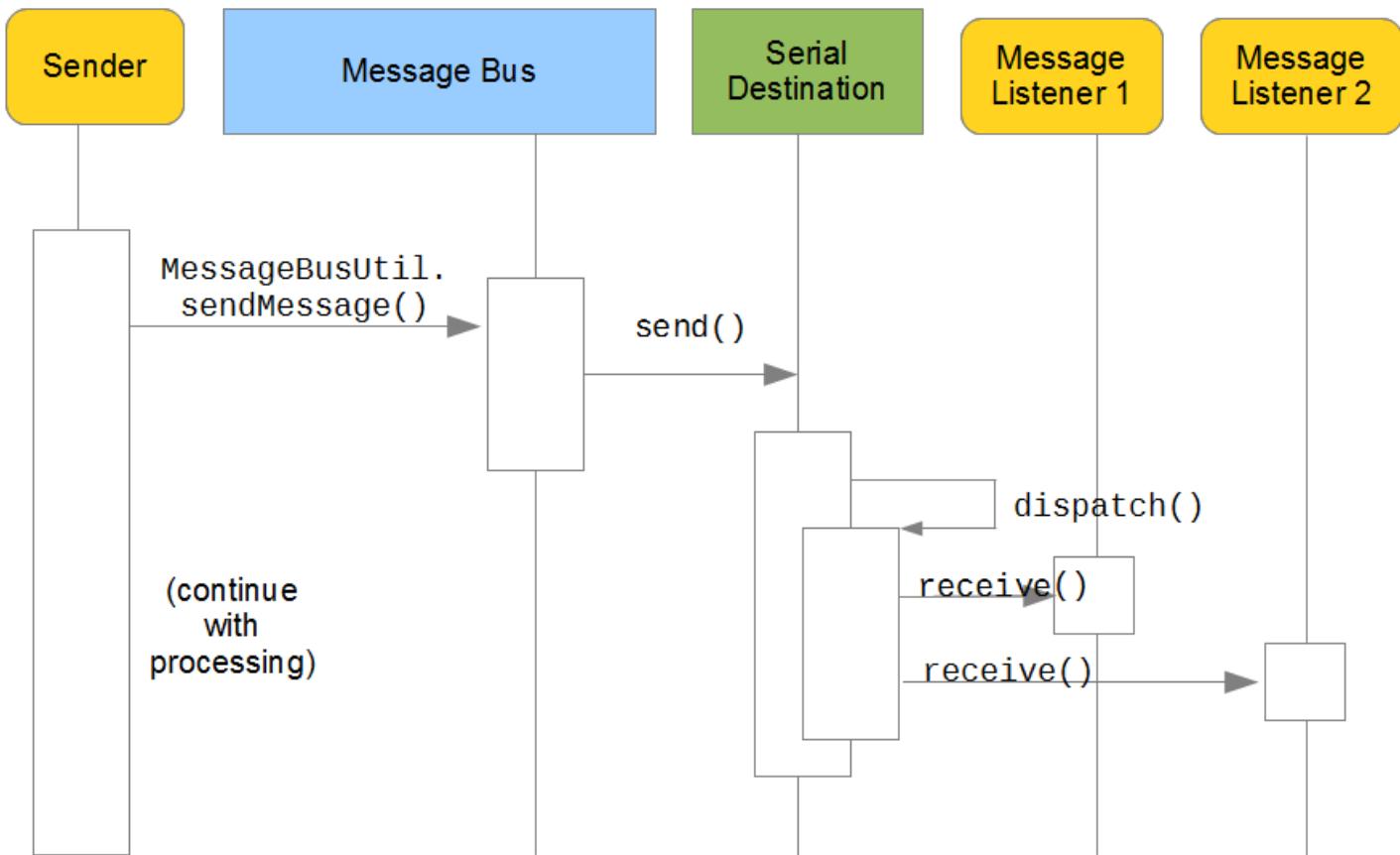
Asynchronous messaging consists of sending a message and then continuing with processing without blocking waiting for an immediate response. This allows the sender to continue with other tasks. It's often necessary, however, that the listener can respond to the sender. This can be done using a call-back.

Jungle Gyms R-Us's Procurement Department must notify the Sales and Warehouse departments of incoming equipment while simultaneously soliciting their feedback. To assure all three departments are up to speed, any responses from the Sales or Warehouse departments are posted to a shared destination.

The following table describes how we'll set things up:

Destination Key	Type	Sender	Receivers
jungle/purchase	async serial	Procurement	Sales, Warehouse
jungle/purchase/response	synchronous	Sales, Warehouse	Procurement

The following image shows asynchronous messaging, with serial dispatching of messages:



Let's package the message as a `JSONObject` and send it to the destination:

```

JSONObject jsonObject = JSONFactoryUtil.createJSONObject();

jsonObject.put("department", "Procurement");
jsonObject.put("partName", part.getName(Locale.US));
jsonObject.put("responseDestinationName", "jungle/purchase/response");

```

`MessageBusUtil.sendMessage("jungle/purchase", jsonObject.toString());`

Here's how the Warehouse Department listens for and handles messages:

```

public void receive(Message message) {

    try {
        doReceive(message);
    }
    catch (Exception e)
    {
        _log.error("Unable to process message " + message, e);
    }
}

```

```

protected void doReceive(Message message)
    throws Exception {

    String payload = (String)message.getPayload();

    JSONObject jsonObject = JSONFactoryUtil.createJSONObject(payload);

    String department = jsonObject.getString("department");
    String partName = jsonObject.getString("partName");
    String responseDestinationName = jsonObject.getString(
        "responseDestinationName");

    System.out.println("Warehouse received purchase notification for " +
        partName + " from " + department);

    jsonObject = JSONFactoryUtil.createJSONObject();

    jsonObject.put("department", "Warehouse");
    jsonObject.put("partName", partName);
    jsonObject.put("comment", "Ugh! We're running out of space!!");

    MessageBusUtil.sendMessage(
        responseDestinationName, jsonObject.toString());
}

```

Here's how this listener deserializes the `JSONObject` from the message:

1. Gets the message payload and casts it to a `String`.
2. Creates a `JSONObject` from the payload string.
3. Gets values from the `JSONObject` using its getter methods.

The class also demonstrates how the Warehouse Department packages a response message and sends it back to the Procurement Department, using these steps:

1. Create a `JSONObject`.
2. Stuff it with name-value pairs.
3. Send the response message to the original message's response destination.

The Sales department listener can be implemented the same way, substituting *Sales* as the department value; the comment would likely be different, too.

You just used the `JSONObject` message type to send an asynchronous response message using a call-back.

Remember, we want the Procurement, Sales, and Warehouse departments to be aware of any message regarding the new playground equipment purchasing process. Let's leverage our destination keys and department names in handling shared responses.

Here's how the Warehouse might handle messages it receives:

```

public void receive(Message message) {

    try {
        if (message.getDestinationName().equals("jungle/purchase"))
        {
            doReceive(message);

```

```

        }
        else if (
            message.getDestinationName().equals("jungle/purchase/response"))
        {
            doReceiveResponse(message);
        }
    }
    catch (Exception e)
    {
        _log.error("Unable to process message " + message, e);
    }
}

protected void doReceiveResponse(Message message)
throws JSONException {

    String payload = (String)message.getPayload();

    JSONObject jsonObject = JSONFactoryUtil.createJSONObject(payload);

    String department = jsonObject.getString("department");

    if (!department.equals("Warehouse")) {
        System.out.println(
            "Warehouse is in the loop on response from " + department);
    }
}

```

Let's look at `receive (Message)` for a minute. We've set it up to handle messages differently depending on their destinations: messages to `jungle/purchase` are handled as Procurement's purchase notifications, while messages to `jungle/purchase/response` are treated as departmental responses to Procurement's purchase notifications. The `doReceiveResponse (Message)` method performs an important task, checking that the response comes from a department other than itself, and printing an error if it doesn't.

Here are the configuration elements we added to the `messaging-spring.xml` from the previous section:

Listener beans:

```

<bean
    id="messageListener.warehouse_listener"
    class="com.liferay.training.parts.messaging.impl.WarehouseMessagingImpl"
/>
<bean
    id="messageListener.sales_listener"
    class="com.liferay.training.parts.messaging.impl.SalesMessagingImpl"
/>

```

Destination beans: The purchase notifications will be sent to a *serial* destination and the responses will be sent to a *synchronous* destination.

```

<bean
    id="destination.purchase"
    class="com.liferay.portal.kernel.messaging.SerialDestination"
>
```

```

<property name="name" value="jungle/purchase" />
</bean>

<bean
    id="destination.purchase.response"
    class="com.liferay.portal.kernel.messaging.SynchronousDestination"
>
    <property name="name" value="jungle/purchase/response" />
</bean>

```

Configuration bean listener map entry: Warehouse and Sales are registered to listen for the notifications from Procurement. All three departments are registered to listen for inter-departmental responses.

```

<entry key="jungle/purchase">
    <list value-type="com.liferay.portal.kernel.messaging.MessageListener">
        <ref bean="messageListener.warehouse_listener" />
        <ref bean="messageListener.sales_listener" />
    </list>
</entry>
<entry key="jungle/purchase/response">
    <list value-type="com.liferay.portal.kernel.messaging.MessageListener">
        <ref bean="messageListener.procurement_listener" />
        <ref bean="messageListener.warehouse_listener" />
        <ref bean="messageListener.sales_listener" />
    </list>
</entry>

```

Configuration bean destination list references:

```

<ref bean="destination.purchase"/>
<ref bean="destination.purchase.response"/>

```

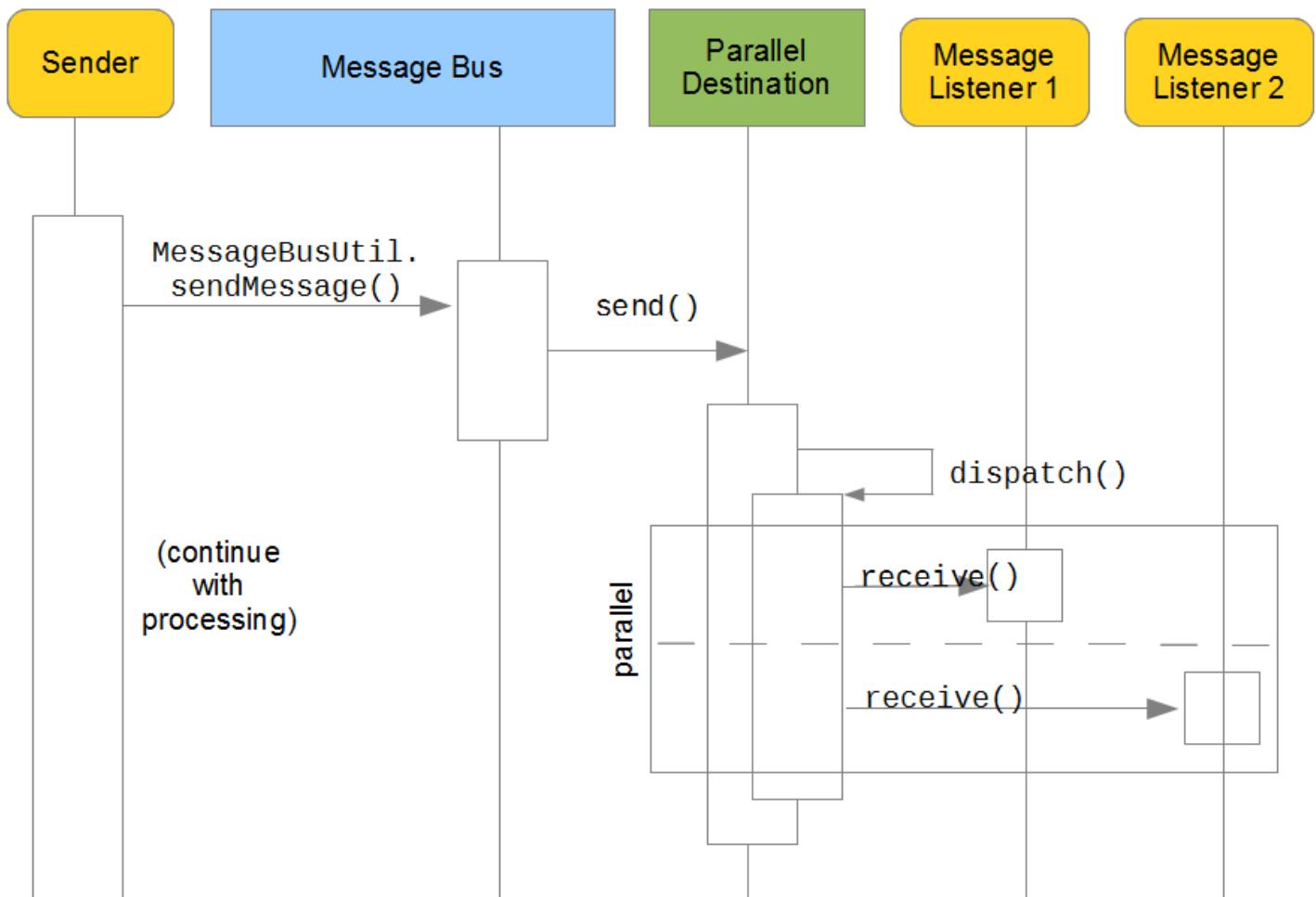
Don't forget to send news of these new products to *all* Jungle Gyms R-Us employees.

Next, let's explore the asynchronous *send and forget* model.

7.6.5. Asynchronous "Send and Forget"

In the *send and forget* model, the sender sends messages and continues processing. We'll apply this behavior to Jungle Gym's company-wide new product notification.

Procurement isn't expecting response messages from individual employees, so there's no need for the company-wide listener to package up responses. We do, however, want everyone to get product news at the *same time*, so instead of dispatching news to employees *serially* we'll dispatch *in parallel*.



We'll specify a parallel destination type in our `messaging-spring.xml`:

Destination bean:

```
<bean
    id="destination.employee.news"
    class="com.liferay.portal.kernel.messaging.ParallelDestination"
>
    <property name="name" value="jungle/employee/news" />
</bean>
```

Listener bean:

```
<bean
    id="messageListener.employee_listener"
    class="com.liferay.training.parts.messaging.impl.EmployeeMessagingImpl"
/>
```

Configuration bean listener map entry:

```
<entry key="jungle/employee/news">
```

```
<list value-type="com.liferay.portal.kernel.messaging.MessageListener">
    <ref bean="messageListener.employee_listener" />
</list>
</entry>
```

Configuration bean destination list reference:

```
<ref bean="destination.employee.news"/>
```

Congratulations! You implemented inter-departmental communications for the procurement process at Jungle Gyms R-Us.

Along the way you used Message Bus to implement the following:

- Sender, listener, and destination components.
- Synchronous and Asynchronous messaging schemes.
- *Serial* and *parallel* message dispatching.
- Java and JSON message types.

Next we'll show you the Device Detection API and its capabilities.

7.7. **Device Detection**

As you know, Internet traffic has risen exponentially over the past decade and shows no sign of stopping. With the latest and greatest devices, mobile Internet access has become the norm and is predicted to pass PC based Internet access soon. Because of the mobile boom, new obstacles and challenges are presented for content management. How will content adapt to all devices with different capabilities? How can your grandma's gnarly tablet and cousin's awesome new mobile phone request the same information from your portal?

The Device Detection API detects the capabilities of any device making a request to your portal. It can also determine what mobile device or operating system was used to make a request, and then follows rules to make Liferay render pages based on the device. To use this feature, you first need to install the *Device Recognition Provider* app from Liferay Marketplace. Find more information on Device Recognition Provider CE app and Device Recognition Provider EE at Liferay Marketplace.

The Device Recognition plugin comes bundled inside the Device Recognition Provider app; it uses a device database called *WURFL* to determine the capabilities of your device. Visit the WURFL website for more information at <http://wurfl.sourceforge.net>.

You can create your own plugin to use your device's database. Let's look at some simple uses of the Device Detection API and talk about its capabilities.

7.7.1. **Using the Device API**

Let's look at a couple of code snippets to get you started. You can obtain the object `Device` from the `themeDisplay` object like this:

```
Device device = themeDisplay.getDevice();
```

You can view the Device API at <http://docs.liferay.com/portal/6.2/javadocs>. Using some of the methods from the Javadocs, here's an example that obtains a device's dimensions:

```
Dimensions dimensions = device.getScreenSize();
float height = dimensions.getHeight();
float width = dimensions.getWidth();
```

Now your code can get the `Device` object and the dimensions of a device. Of course this is just a simple example; you can acquire many other device attributes that help you take care of the pesky problems that arise when sending content to different devices. You can refer to the `Device` Javadocs mentioned above for assistance. Let's look at some device capabilities next.

7.7.2. Device Capabilities

Most of the capabilities of a device can be detected, depending on the device detection implementation you're using. The Device Recognition plugin's device database (WURFL) has a list of capabilities, described at <http://www.scientiamobile.com/wurflCapability/tree>. For example, you can obtain the capability of a brand name with this code:

```
String brand = device.getCapability("brand_name");
```

You can grab many other device capabilities, including `model_name`, `marketing_name`, and `release_date`. You can also get boolean values like `is_wireless_device`, `is_tablet`, etc.

With the Device Detection API, you can detect the capabilities of a device making request to your portal and render content accordingly; so your grandma's gnarly tablet and your cousin's awesome new mobile phone can make requests to your portal and receive identical content. This will make everyone happy!

7.8. Summary

You've learned how to leverage `ServiceContext` objects in your use of Liferay services and how Liferay's permissions and JSR portal security work.

With Liferay's frameworks, implementing complex functionality in your custom portlets becomes easy. We discussed how to use Liferay's `ServiceContext` object, permissions framework, asset framework, message bus API, and device detection API. Check back regularly to find more detailed descriptions of current frameworks. You might also discover brand new frameworks that'll knock your socks off, or at least simplify your custom portlet development.

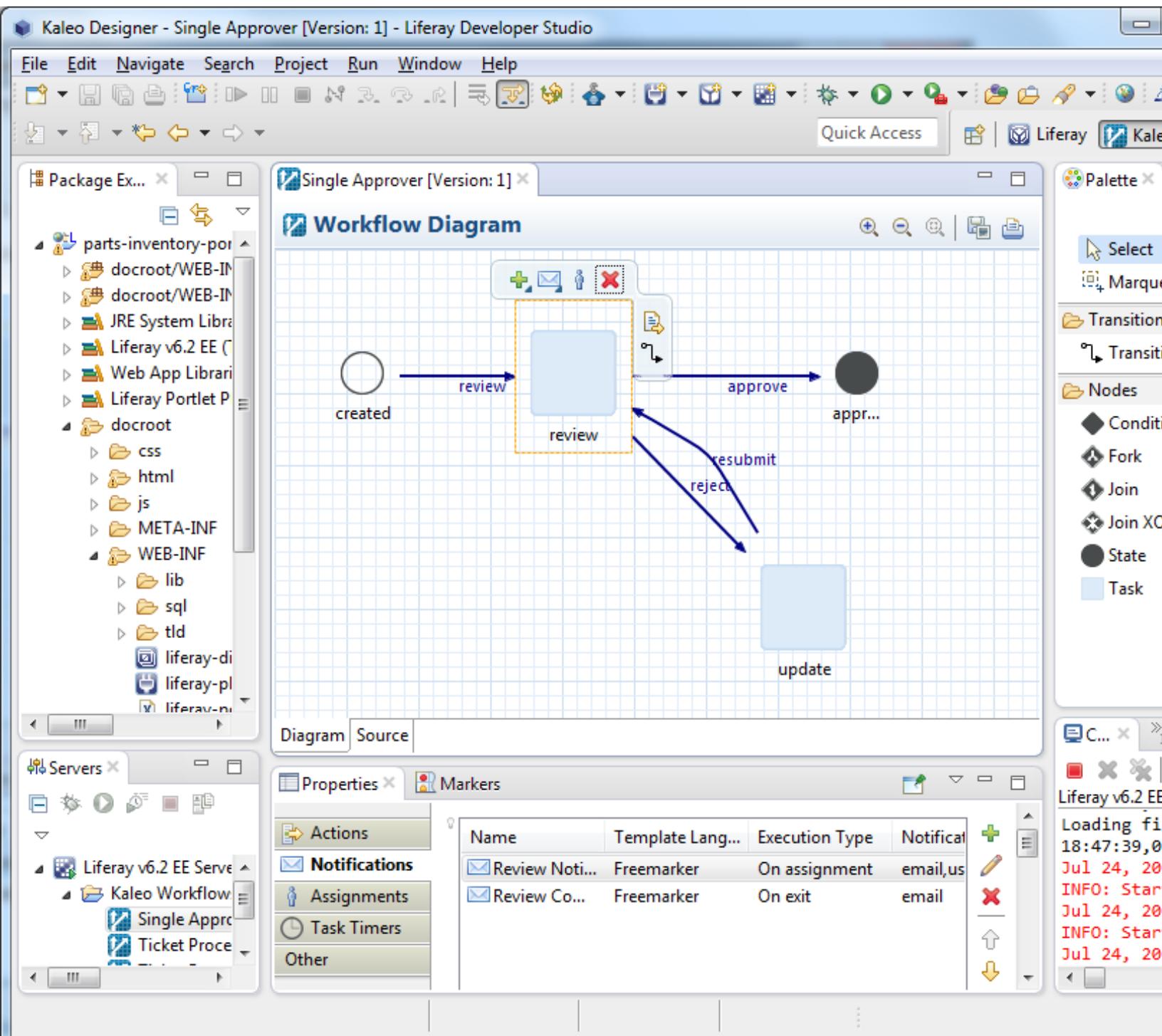
Not only does Liferay Portal boast of fabulous frameworks but also of its unwavering support of workflow development. Naturally, privileged portal users can create workflows right in portal. But as workflows incorporate API calls and complex user interaction, you the developer, are called on to deliver robust workflow solutions. As you implement your advanced workflows, it only makes sense that we give you an optimal development environment. Enter Liferay Developer Studio and the Kaleo Designer for Java plugin! With this plugin, you not only get the drag-and-drop functionality you've come to know and love, but you also get access to an optimal development environment to work with Java APIs and FreeMarker templates. You're going to love building your workflows in Kaleo Designer for Java. Get ready to drop in and make waves with your workflows. Cowabunga!

8. Designing Workflow with Kaleo

Liferay Portal includes a workflow engine called *Kaleo*. Kaleo allows portal administrators to set up workflows for their organization's needs; the workflow calls users to participate in processes

designed for them. Kaleo workflows, called *process definitions*, are essentially XML documents. Kaleo supports a host of XML element types to trigger decisive actions in your business process instances. You can fine-tune your process definition's logic by incorporating scripts and templates.

The *Kaleo Forms EE* app from Marketplace includes Liferay's *Kaleo Workflow Designer* that lets you create and modify portal workflows in your browser. With Kaleo Designer for Java, you can design and publish Kaleo workflows from Liferay Developer Studio!



Kaleo Designer for Java lets you incorporate back-end Java development and scripting in your workflows. Its graphical interface lets you drag and drop nodes into your workflow. A shortcut on each node gives you easy access to the node's XML, letting you edit its current

implementation to make subtle modifications or inject new business logic. In addition, Liferay Developer Studio comes bundled with a Java/Groovy editor (made available by Spring Source), giving you the same rich editing experience you're accustomed in editing Kaleo Groovy scripts. From the editor you can delegate workflow decisions to your custom business logic APIs or access any of the Liferay Portal APIs. In Developer Studio, you can leverage editors for Beanshell, Drl, JavaScript, Python, and Ruby scripting languages. You can also leverage the editor for the FreeMarker template language. Kaleo Designer for Java gives you a rich tool set for creating/editing workflows, manipulating workflow nodes, and implementing business logic. But there's more!

With Kaleo Designer for Java, you can remotely add and update workflow definitions directly to and from your Liferay server. You can publish your workflow drafts to your portal by simply dragging the workflow file onto your portal server in Developer Studio's *Servers* tab. The *Servers* tab shows workflows you've published from studio and gives you access to workflows already published on the portal server. You can edit existing workflows and create custom business logic in Developer Studio locally, then republish them on your portal; you don't have to navigate back and forth from your portal and Developer Studio to complete these tasks. As you can see, Kaleo Designer for Java is a powerful application for creating, modifying, and publishing workflows in Liferay Developer Studio.

Although Kaleo Designer for Java is the tool of choice for EE workflow designers, CE workflow designers can write Kaleo workflows too. But they are limited to writing them in their favorite XML editor. All Kaleo process definitions must follow the schema http://www.liferay.com/dtd/liferay-workflow-definition_6_2_0.xsd. As we show you how to design workflows, we'll include their resulting XML code for your convenience.

We'll cover the following topics as we design workflows:

- Installing Kaleo Designer for Java
- Creating a Workflow
- Using Workflow Scripts
- Leveraging Template Editors for Notifications
- Viewing Workflow Definition XML Source
- Publishing Workflows to the Server
- Using Workflows in Liferay Portal
- Using Dynamic Data Lists (DDLs) with Workflows

8.1. *Installing Kaleo Designer for Java*



To use Kaleo Designer for Java, install the *Kaleo Forms EE* app from Liferay Marketplace. The app includes three plugins--*kaleo-forms-portlet*, *kaleo-designer-portlet*, and *kaleo-web* plugins.

Here's how to download and install *Kaleo Forms EE*:

1. Go to Liferay Marketplace.
2. Click on EE Marketplace in the left navigation area.

3. Download and install the *Kaleo Forms EE* app.

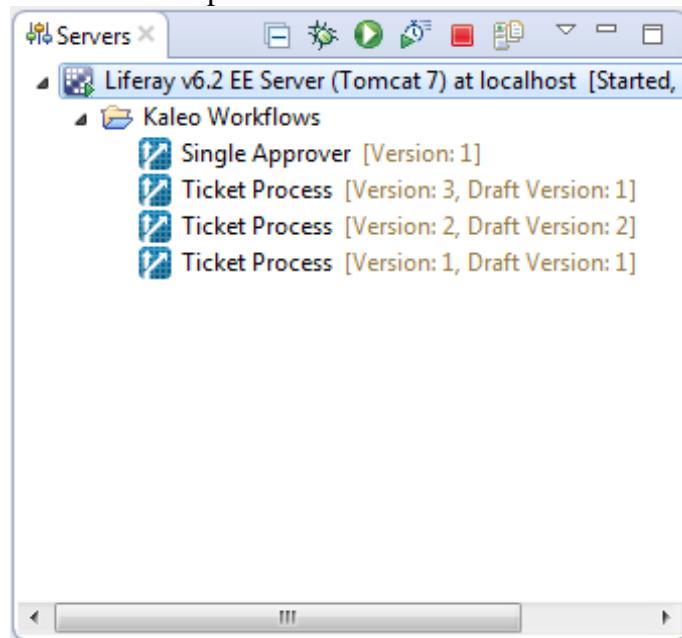


Note: The *Kaleo Forms EE* app comes with an existing workflow designer that's used *within* Liferay Portal. It's used to design workflow configuration and is described in the Kaleo Forms: Defining Business Processes chapter of *Using Liferay Portal*. Refer to the Using Workflow chapter of *Using Liferay Portal* if you're unfamiliar with basic Kaleo workflow concepts or want to know how to design your workflow within Liferay Portal.

After downloading and installing the *Kaleo Forms EE* application, you must restart the Liferay Server. For Studio 2.0 to connect to the *Kaleo* APIs to open workflow definitions in Portal 6.2, you must make sure to use a portal admin username and password in your server's settings.

1. Stop your 6.2 server
2. Open the the server's configuration editor by double-clicking the server from within the Servers view in Studio.
3. In the configuration editor under Liferay Settings add your portal admin username and password. Save the configuration changes.
4. Start the server

A *Kaleo Workflows* folder automatically appears underneath the server instance in the Servers view of Developer Studio.



Developer Studio retrieves all workflow definitions published on your portal server. As mentioned previously, the *Kaleo Designer for Java* lets you remotely add and update *Kaleo* workflow definitions directly to and from the portal server. The *Single Approver* workflow comes already installed with the *Kaleo* app.

To open a workflow, double click it in the *Kaleo Workflows* folder. This retrieves the workflow definition from the Liferay server so you can edit it in Developer Studio.



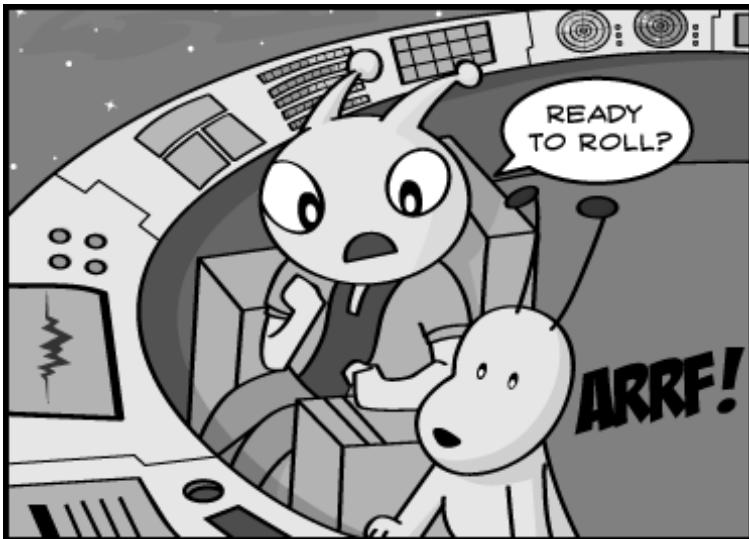
Note: When you open a workflow, you'll be prompted to choose whether to switch to the *Kaleo* designer perspective. Clicking *Yes* lets you use the perspective's helpful features, including the palette toolbar, properties view, and outline view.

Next, let's get into the flow (pun intended) by creating our own workflow using the Kaleo Workflow Designer for Java.

8.2. Creating a Workflow

Developer Studio makes it easy for you to write custom business logic that enhances Kaleo workflows. Let's create our own workflow in Developer Studio and highlight features from Kaleo Designer for Java (Designer) along the way.

To demonstrate Designer's features, let's create a workflow definition for a software ticketing process. Are you ready to roll? Let's get started!



1. Create your new workflow definition by going to *File* → *New* → *Liferay Kaleo Workflow*. Alternatively, you can select *Liferay Kaleo Workflow* from the toolbar button shown in the figure below.
2. The *Create New Kaleo Workflow File* wizard will guide you through the steps necessary to complete the initial setup of your new workflow definition.

The first window you'll see is the *Create Kaleo Workflow* form. There are several fields here:

Project: Specify an existing Liferay project to house your workflow definition.

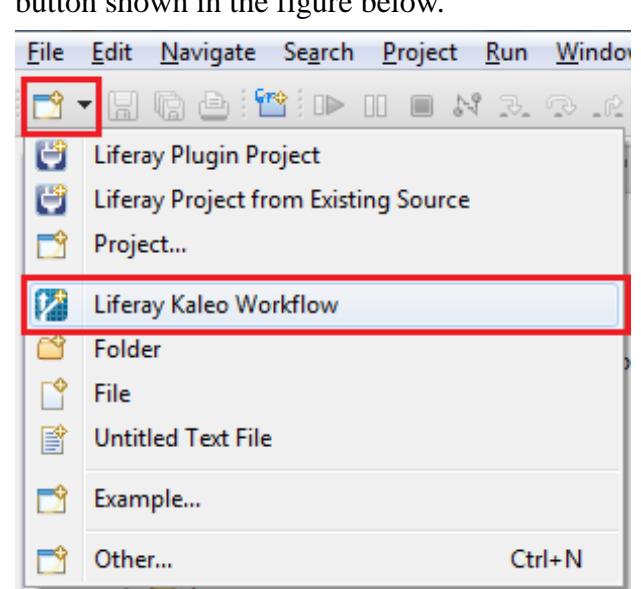
Folder: Specify where in the project the workflow definition XML file will be stored.

Name: Give your workflow definition a descriptive name. We've chosen *Ticket Process* for our example.

Initial state name and **Final state name**: Name your workflow's initial and final state nodes.

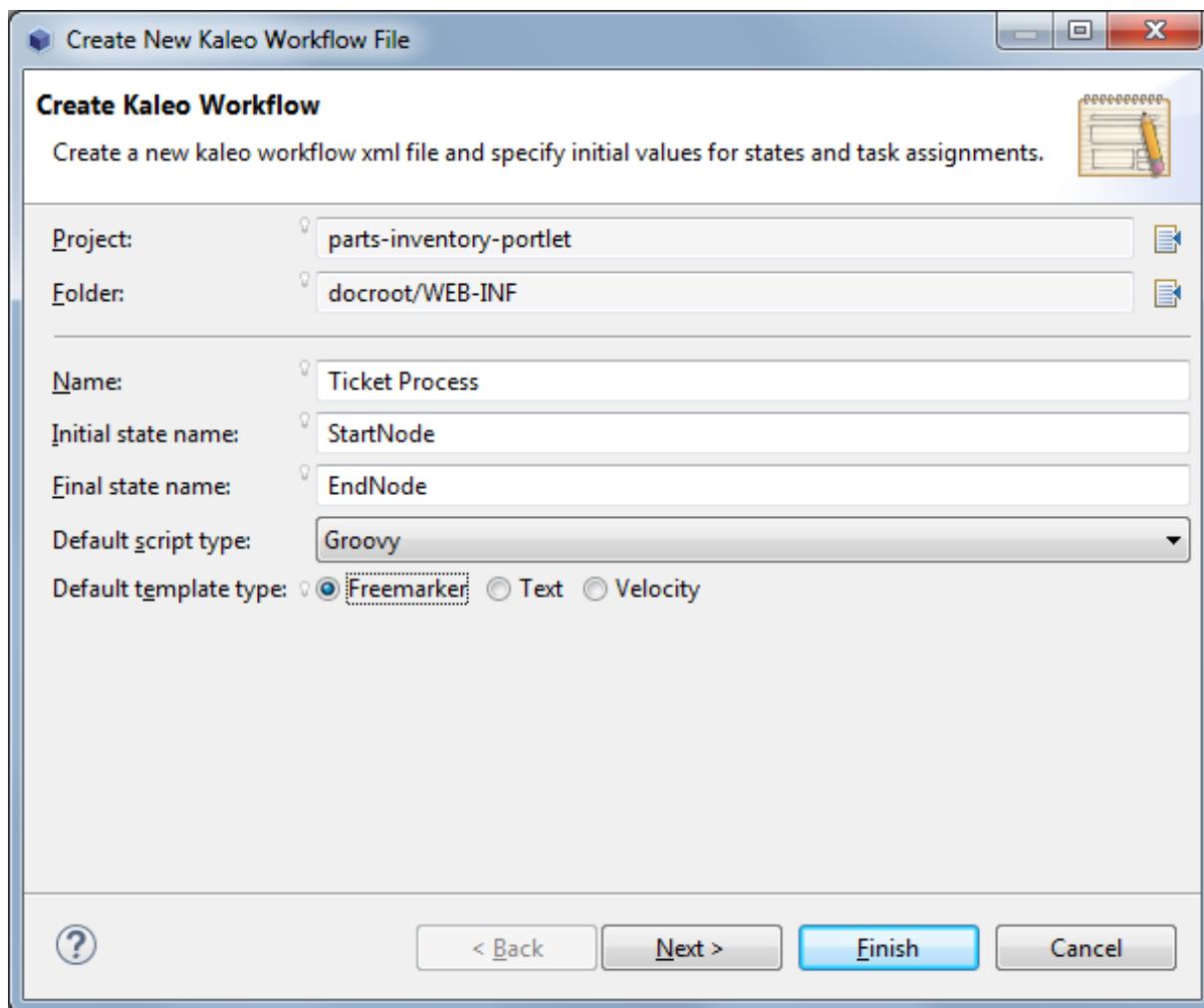
Default script type: Choose a default script type; Designer will bring its editor up when you're done creating the workflow.

Default template type: Choose a default template editor; Designer will open it when you're done creating the workflow.



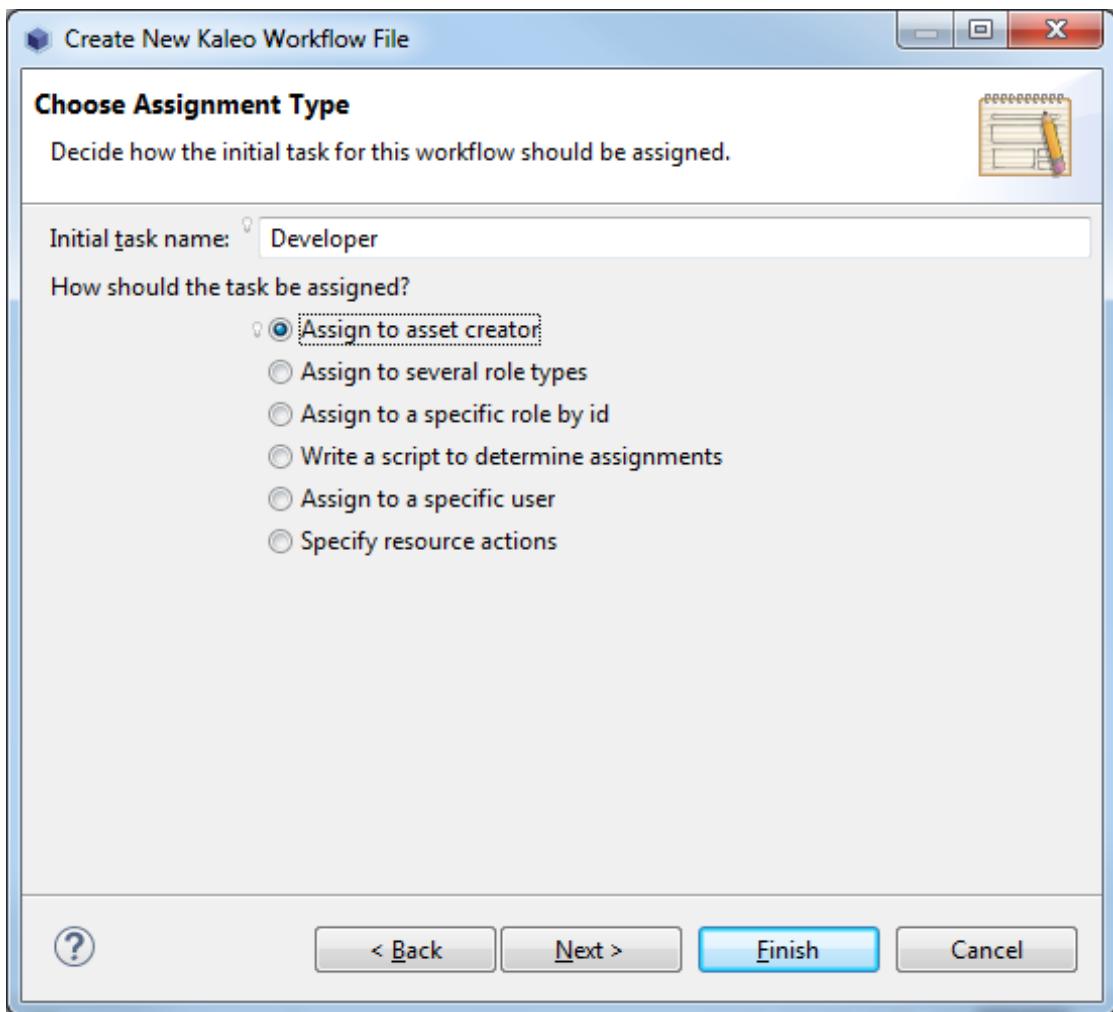
The snapshot below displays the setup menu for our ticket process workflow definition.

When you're finished in this window click *Next*.



You'll be directed to the *Choose Assignment Type* window next. Here you'll provide an *Initial task name*, then choose an assignment type from the list of options.

For our workflow example, select *Assign to asset creator* and name the task *Developer*. When our workflow's *Developer* task is invoked, the creator of the workflow's asset is assigned to it. In Liferay Portal, each Kaleo workflow is associated with an asset type. Later in our exercise, we'll associate our workflow with a Dynamic Data List (DDL).

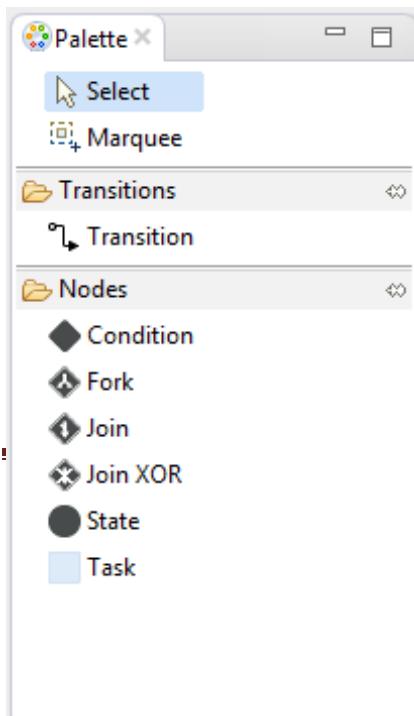


Note:
You must specify a Liferay project as a home for the workflow definitions you create.

3. Click *Finish* to complete

the initial setup of our workflow definition.

Your workflow appears as a workflow diagram, letting you interact with your workflow graphically. Graphical features and toolbars allow you to customize your workflow definition. The *Palette* view is one of Designer's most commonly used tools. Let's explore it next.



8.2.1. Palette and Floating Palette

The *Palette* lets you graphically customize your workflow with nodes and transitions. In addition, you can choose different behaviors for your mouse pointer.

Here are your pointer options:

- *Select*: The default pointer setting used for selecting options on the workflow diagram by clicking the icon.
- *Marquee*: Used for drawing an invisible selection box

around multiple icons. This is useful when you want to manipulate multiple nodes and/or transitions on the workflow diagram.

Transitions connect one node to another. On exiting the first node, processing follows the transition to the node it points to. Selecting a *Transition* turns your pointer into a connector; you connect the starting end of a transition to one node and the other end to the next node in your process.

In addition to *Start* and *End* node types, there are five node types you can use in your workflow:

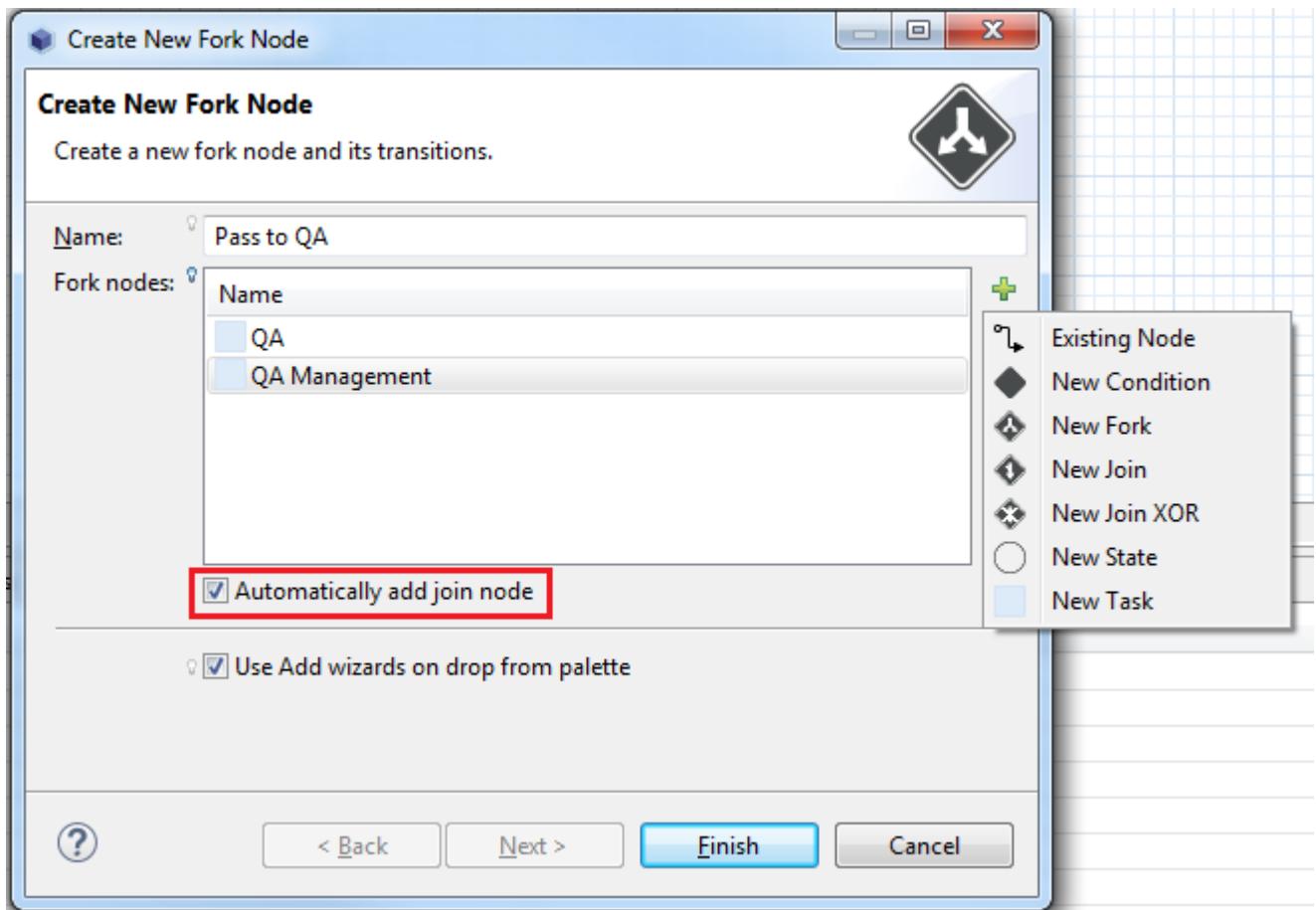
- *Condition*: Directs workflow execution to an appropriate transition based on conditional logic of the node's script.
- *Fork*: Forks the workflow execution into two parallel threads.
- *Join*: Joins parallel workflow threads.
- *Join XOR*: Joins parallel workflow threads, but only needs to receive workflow execution from one of them.
- *State*: Represents a workflow state.
- *Task*: Represents a task that can be assigned.

Drag and drop any nodes you need onto your workflow diagram. Each node type supports execution of scripted actions and sending notifications that can use templates. For additional information on the node types, refer to the Designing Workflows with Kaleo Designer for Java chapter of *Using Liferay Portal*.

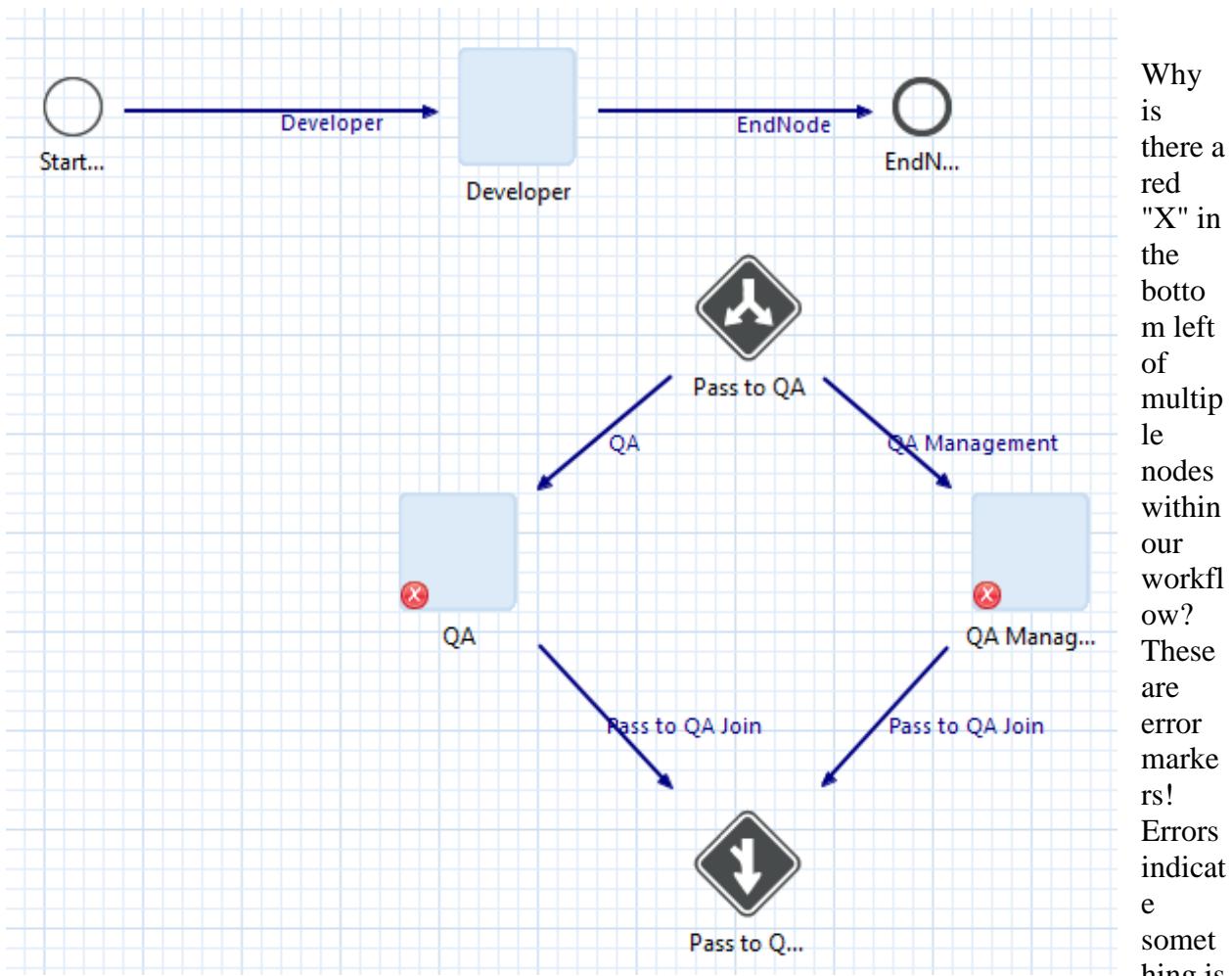
For our ticket-process-definition workflow diagram, we have a simple `StartNode State` node, followed by the `Developer Task` node, followed by the `EndNode State` node. There are two transitions, from `StartNode → Developer` and from `Developer → EndNode`.

We want a developer to approve his fix and send it for quality assurance to *QA*, where it must pass testing by a QA engineer. Then it'll go to *QA Management*, where it must be approved by a QA manager. Let's use a *Fork* node to accurately depict these parallel approval tasks.

Drag and drop a *Fork* node onto your workflow diagram. A wizard helps you create your node. Click the green plus symbol to select new or existing nodes to process in parallel threads. A drop-down menu gives you options to select tasks to be done in your fork threads. In addition, a checkbox lets you indicate whether to automatically add a corresponding join node to your workflow.



On finishing your fork node in the wizard, Kaleo Designer places your new nodes onto the workflow diagram's canvas. If you're not happy with the location of your new nodes, drag them to place them where you want on your canvas. Our ticket process workflow now looks something like this:



specified incorrectly or is missing from your nodes. If you click on an error marker, Developer Studio displays hints on resolving the problem. Don't worry, we'll address these error markers momentarily.

Clicking on a node brings up a floating palette; use it to make quick, convenient customizations to a node.

The floating palette has several features you can use:

- *Add* (green cross): Add an action or notification on the node. To edit the added action or notification, bring up Designer's *Properties* view.
- *Edit Actions* (paper with arrow): Edit the node's existing actions.
- *Edit Notifications* (envelope): Edit existing notifications on the node.
- *Edit Script* (pencil): Edit the script of the condition node. This feature only applies to the condition node.
- *Change Assignments* (person): Assign or reassign a task. This feature only applies to the Task node type.
- *Delete* (red "X"): Delete the node.
- *Show in Source* (paper with folded corner and arrow): Show the node specified as a

model element in the workflow definition's XML source and switches the main editor to *Source* mode.

- *Start Transition* (black arrow): Change the pointer to transition mode letting you create a workflow transition from the current node to another node.

Obviously, there is still work to be done in our workflow definition. We have multiple error markings and the fork and join nodes aren't connected to anything. Let's change the assignments for our two new task nodes, QA and QA Management, by clicking the *Change Assignments* icon from the floating palette for each node. The *Choose Assignment Type* menu appears for each node, letting you choose their assignment type. After we assign the QA and QA Management task nodes, the error markers disappear.

For our `ticket` process workflow's QA tasks, let's assign someone other than the asset creator. Realistically, each of these tasks would be assigned to different site roles. For simplicity, let's assign both the QA and QA Management tasks to the same user. If you have a user in mind, specify that user. Otherwise, create a user named "Joe Bloggs" with screen name "joe". For the user to receive emails, he must be registered within Liferay Portal. If you haven't registered Joe Bloggs ("joe") already, see the Adding and Editing Users section of *Using Liferay Portal* for instructions. To configure the user's email, login to the user's account and visit *Control Panel* → *Server Administration* → *Mail* for setup options.

Select the *Change Assignments* icon from the floating palette for each QA task node. Then, select *Assign to a specific user* from the *Choose Assignment Type* menu and click *Next*. You have options to enter the user's *User-id*, *Screen name*, or *Email address*. Enter the user's screen name and click *Finish*.

Assigning the QA and QA Management task nodes resolved their error markings (no more red "X"!). The join node's error marking

Page 354

won't disappear until you connect the join node to another task.

Let's take a moment to consider the XML code of the ticket process workflow definition in its current state.

It specifies its XML version, encoding, and its document root element called `workflow-definition`. Nested within the `workflow-definition` element are its name, description (optional), version, and its nodes: 1 fork, 1 join, a start and end state. Here's the general overview of our workflow definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<workflow-definition xmlns="urn:liferay.com:liferay-workflow_6.2.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:liferay.com:liferay-workflow_6.2.0
http://www.liferay.com/dtd/liferay-workflow-definition_6_2_0.xsd">
    <name>Ticket Process</name>
    <version>1</version>

    <!-- fork node -->

    <!-- join node -->

    <!-- start and end state nodes -->

    <!-- task nodes -->

</workflow-definition>
```

We'll describe each of our workflow definition's nodes, starting from top to bottom, left to right.

Our start state node, named `StartNode`, simply transitions the flow of execution to our `Developer` task. Here is this state node (note, we left off the optional `<metadata>...</metadata>` elements to shorten the code snippets):

```
<state>
    <name>StartNode</name>
    <initial>true</initial>
    <transitions>
        <transition>
            <name>Developer</name>
            <target>Developer</target>
            <default>true</default>
        </transition>
    </transitions>
</state>
```

The `Developer` task is assigned to the creator and transitions to the `EndNode` state node upon approval. Each workflow instance is associated with an asset. By default, a task is associated with the asset creator.

```
<task>
    <name>Developer</name>
    <assignments>
        <user></user>
    </assignments>
    <transitions>
```

```

<transition>
    <name>approved</name>
    <target>approved</target>
</transition>
<transition>
    <name>EndNode</name>
    <target>EndNode</target>
</transition>
</transitions>
</task>

```

The ending state node is named *EndNode*. It specifies that an *Approve* action is to be executed on entering this node.

```

<state>
    <name>EndNode</name>
    <actions>
        <action>
            <name>Approve</name>
            <description>Approve</description>
            <script>
<![CDATA[ Packages.com.liferay.portal.kernel.workflow.WorkflowStatusManagerUtil.updateStatus(Packages.com.liferay.portal.kernel.workflow.WorkflowConstants.toStatus("approved"), workflowContext); ]]> </script>
            <script-language>javascript</script-language>
            <execution-type>onEntry</execution-type>
        </action>
    </actions>
</state>

```

Our workflow definition has one fork node named *Pass to QA*. It forks the process to the *QA* and *QA Manager* task nodes:

```

<fork>
    <name>Pass to QA</name>
    <transitions>
        <transition>
            <name>QA</name>
            <target>QA</target>
        </transition>
        <transition>
            <name>QA Management</name>
            <target>QA Management</target>
        </transition>
    </transitions>
</fork>

```

Both the the QA-related task nodes are assigned to the user with screen name *joe*.

```

<task>
    <name>QA</name>
    <assignments>
        <user>
            <screen-name>joe</screen-name>
        </user>
    </assignments>
    <transitions>
        <transition>
            <name>Pass to QA Join</name>

```

```

        <target>Pass to QA Join</target>
    </transition>
</transitions>
</task>

<task>
    <name>QA Management</name>
    <assignments>
        <user>
            <screen-name>joe</screen-name>
        </user>
    </assignments>
    <transitions>
        <transition>
            <name>Pass to QA Join</name>
            <target>Pass to QA Join</target>
        </transition>
    </transitions>
</task>
```

Lastly, the *QA* and *QA Manager* task nodes transition into the *Pass to QA Join* join node:

```
<join>
    <name>Pass to QA Join</name>
</join>
```

Now you know what the resulting XML is like for your workflow definition. You can check your definition's source code anytime from within Liferay Studio or your favorite XML editor. To learn more on the different workflow nodes available to use in Liferay workflow definitions, see Using Liferay Portal 6.2.

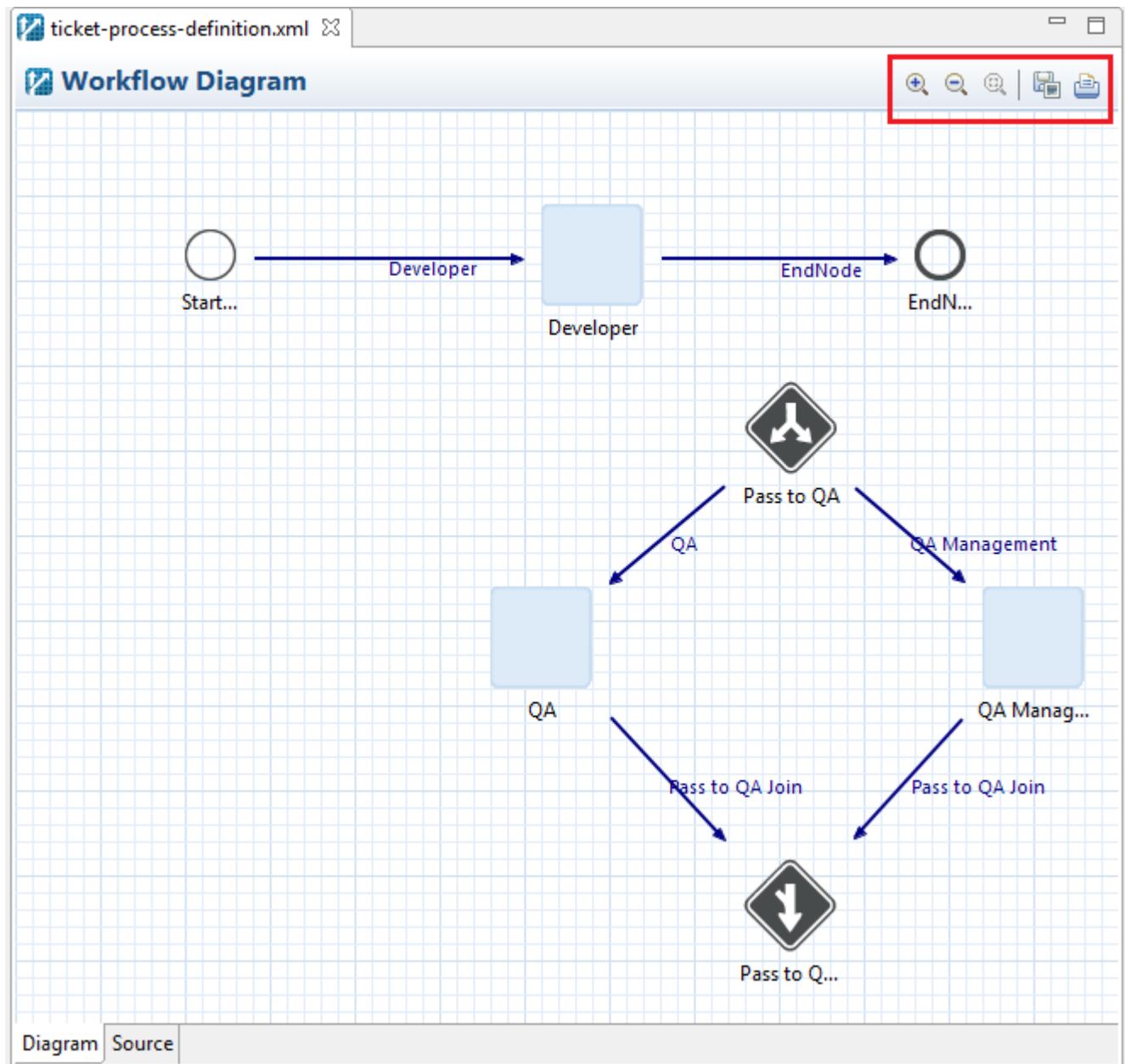
Since we've been using Designer's Workflow Diagram mode, let's go over some of its features.

8.2.2. Workflow Diagram Features

Developer Studio provides you with additional features within the workflow diagram. Below we list some of these features; they can greatly enhance your workflow designing experience.

- *Workflow Diagram Actions* are available via the toolbar in the upper right corner of the Workflow Diagram:
 - *Zoom In*
 - *Zoom Out*
 - *Zoom Actual*
 - *Save as Image*
 - *Print*

These toolbar icons are shown in the figure below.



- *More Workflow Diagram Actions* are accessible by right clicking on the Workflow Diagram's canvas:
 - The *Layout* feature lets you arrange nodes vertically or horizontally, so it's easy to organize your workflow quickly without having to touch a node.
 - The *Rename* feature lets you rename a node or transition; select by double clicking its current name and typing in a new name.
 - The *Surveyor's Level* feature helps you center a node, vertically or horizontally, with respect to another node.

You've probably noticed the *Properties* and *Outline* views below your workflow diagram. Let's

explore them next.

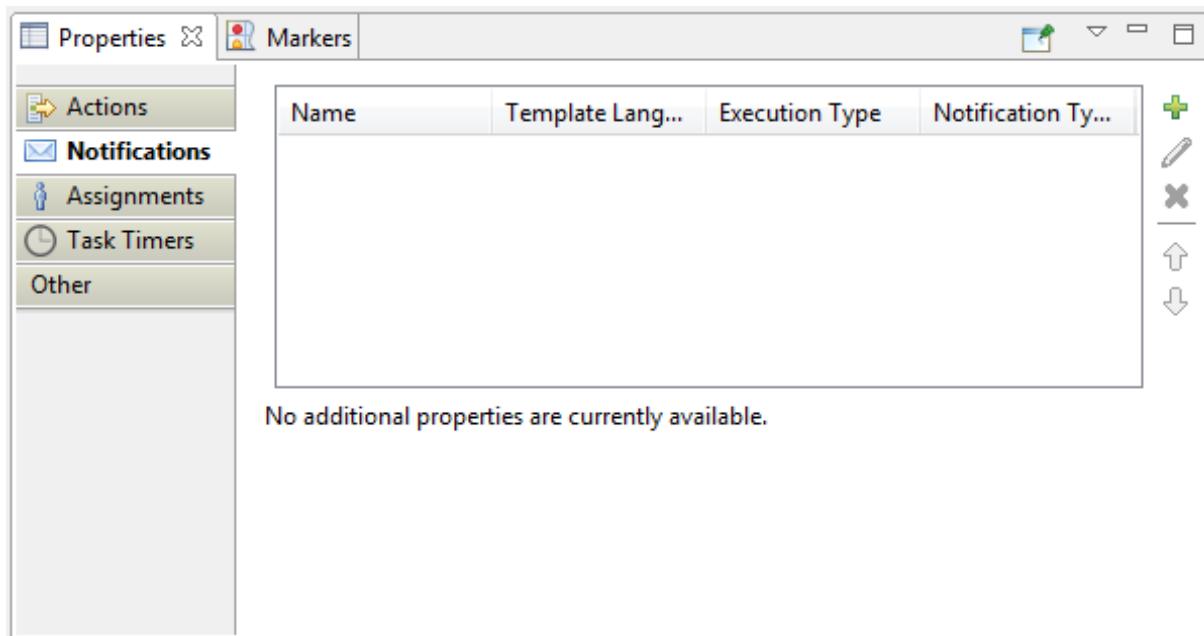
8.2.3. Properties View and Outline View

The *Properties* and *Outline* views contain more cool features you can use to customize your workflow; they're located on the bottom and bottom right of Developer Studio, respectively.

The *Properties* view lets you edit the current node's properties. If no node is selected or you select the workflow canvas, the Properties view displays your workflow's general properties; you can edit these, too. Once you click on an individual node, its properties appear.

Node properties are grouped as follows:

- *Actions*: Execute scripts with respect to your node. Each action has a *Name*, *Script Language*, *Execution Type*, and a *Priority* (optional). Clicking *Edit Script* brings up the script in the default editor for the *Script Language* you specified.
- *Notifications*: Notify users with respect to your node. Each notification has a *Name*, *Template Language*, *Execution Type*, and one or more *Notification Types*.
- *Assignments*: Assign tasks to users or roles. Click *Change Task Assignments* to specify or change the assignment.
- *Task Timers*: Name timers for a task and whether the timers are blocking. Each task timer has a *Name* and *Blocking* indicator.
- *Script*: Edit a script for your condition node. Clicking *Edit Script* brings up the script in the default editor for the *Script Language* you specified.
- *Other*: Edit miscellaneous properties like *Name* and *Description*, applicable to your node's type.



Here's what the Properties view looks like in Developer Studio:

Workflows frequently become too large to view in entirety on the workflow diagram screen; the *Outline* view is a huge asset when this happens. It's a top level view that displays your entire workflow definition, no matter how large it becomes. In addition, it highlights what you're currently viewing on your Workflow diagram, giving you a picture of where you're located in the broader picture of your workflow. You can use the *Outline* view to change your position in the Workflow Diagram by dragging the highlighted box where you'd like to focus.

Developer Studio's *Properties* and *Outline* views make customizing your workflow easier than ever! Developer Studio also offers a convenient way to edit your workflow scripts, which is our next topic.

8.3. Using Workflow Scripts

You can use Developer Studio to edit workflow scripts; it recognizes multiple script languages, so you can choose one you're comfortable with. Developer Studio provides you many script editing features so you can quickly implement business logic in your workflows.

Developer Studio supports several script languages:

- Beanshell
- Drl
- Groovy
- JavaScript
- Python
- Ruby

Let's dive back into our ticket-process workflow definition and create a script. It's not guaranteed that every ticket submitted has a resolution. If the issue was due to a silly user error, there's no reason to change the product. In such cases the developer will resolve the ticket and indicate there is no resolution in the product (i.e., no modifications were made). Regardless, we'll have the developer fill out an online Dynamic Data List (DDL) form to initiate a workflow for each of her tickets. Once the workflow is invoked, its associated DDL record is accessible from our workflow's context. Let's use a condition node to handle the ticket based on the DDL record.

To set up the workflow process we described above, we'll need to add a *Condition* node and two transitions.

1. Drag and drop a *Condition* node onto your workflow diagram. A *Create New Condition Node* menu should appear.
2. Name the node *Resolution*.
3. Choose a script language for the condition node. Select *Groovy* and you'll see how easy it is to embed Java code. In our Groovy script, we'll access the DDL record to determine whether the ticket warrants a modification to the product. If it does, we'll assign it to a developer via the Developer task node. Otherwise we'll end the workflow by transitioning to the workflow's *EndNode*.
4. From the *Create New Condition Node* menu, add two transitions—one to the *Developer* node and the other to the *EndNode* state. We'll add the transition to the *Developer* node

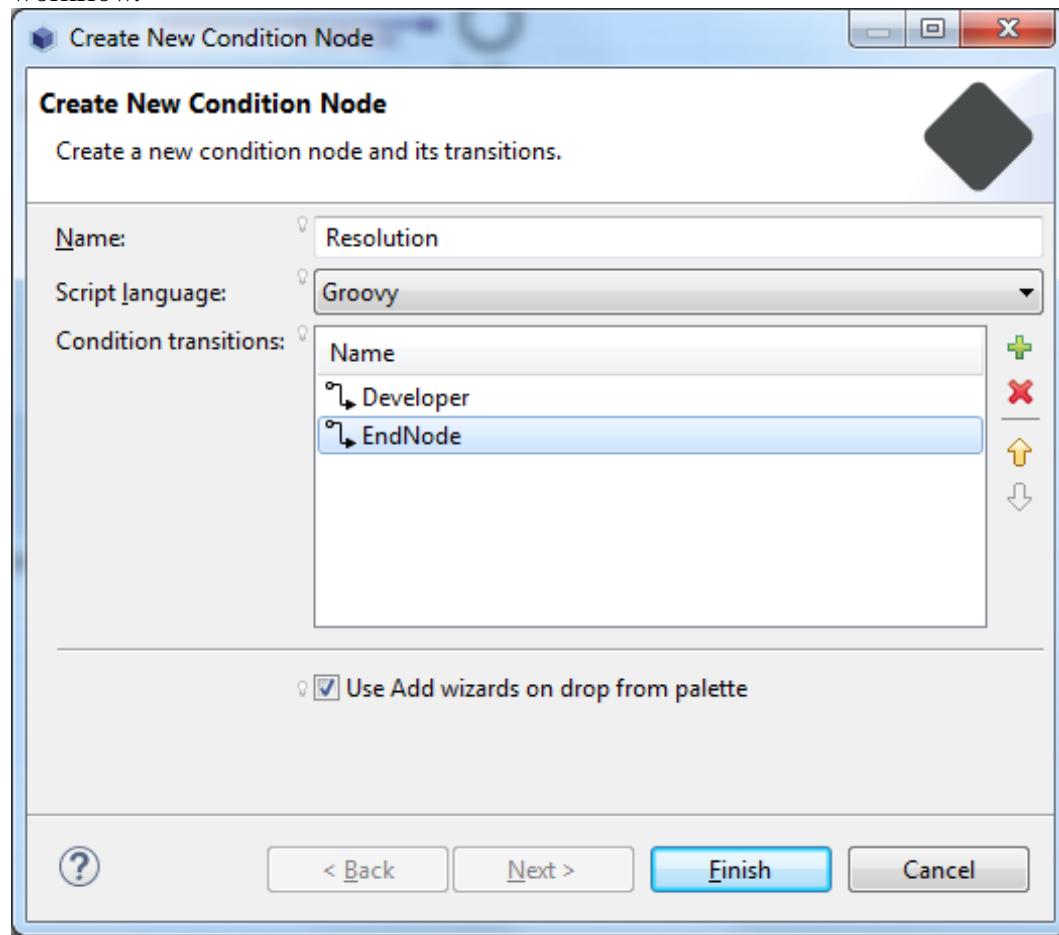
first.

Click the green plus sign and select *Existing Node* from the menu. An entry for the transition appears in the named list of *Condition transitions*.

Click the browse icon in the entry and select the *Developer* node.

5. Add a transition to the *EndNode* state in the same manner that added the transition to the *Developer* node in the previous step.
6. Click *Finish*.

Here's a snapshot of the *Create New Condition Node* menu configured for the ticket process workflow.



Before adding a script to our condition node, let's make some changes to our workflow transitions:

- Add a transition from the *Developer* task node to the *Pass To QA* fork node.
- Add a transition from the *StartNode* state node to the

Resolution condition node.

- Delete the transition that currently connects the *StartNode* state node to the *Developer* task node.
- Delete the transition that currently connects the *Developer* task node to the *EndNode*.

To add a transition from one node to another, do the following:

1. Click the transition icon from the palette. Your pointer's icon shows as a plug indicating you are in *connector* mode.

2. Select a node on your workflow diagram from which the transition will start. A dotted line appears with one end connected to the selected node and the other end following your pointer.
3. Select a node to which the transition will end. The dotted line changes into a fixed ray with the arrow pointing to the transition's end node.
4. To exit connector mode, hit *Escape* on your keyboard and click your pointer at empty space in your workflow diagram.

You may notice the error marking on the condition node. When you hover over the marking, a hint indicates a script must be specified for the node.

Open the script editor for your *Resolution* condition node by doing one of the following:

- Select the node and click *Edit Script* from the *Script* tab of the Properties view.
- Click the *Edit Script* tool from the node's floating palette.
- Right-click the node and select *Edit Script*.

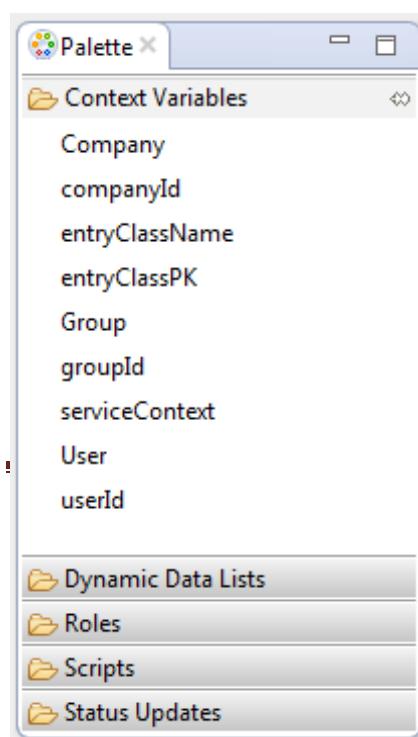
We set our default script language to Groovy, so the Java/Groovy editor appears. To learn more about the Groovy editor, see the Groovy User Guide. If you set the script language to another language, the editor for that specific language appears. The editor runs in the context of editing the specific node you selected. Anything you type in the script editor for this condition node is written inside the `<script></script>` tags for the `<condition/>` element that represents our node in our workflow definition's XML file (in our case, `ticket-process-definition.xml`).



Note: Developer Studio lets you use multiple script editors even while modifying the same workflow definition XML file.

The *Palette* view is much different from when you were working in the workflow diagram; it's associated with your Java/Groovy script editor now and includes folders containing the following entities for your script:

- Context Variables
- Dynamic Data Lists
- Roles
- Scripts
- Status Updates



You can expand and collapse a folder by clicking its name bar. Here's a snapshot of the palette with the *Context Variables* folder open:

Drag and drop an entity from your palette onto your Java/Groovy editor and code representing that entity appears in the editor. The inserted code is free of compile errors and warnings because the editor is running in the context of

Liferay Portal. All of the Liferay Portal APIs are available to you. In the editor you can invoke code-assist and access built in Kaleo workflow variables.

Let's get the DDL record that's being worked on in our workflow process. We'll need the `serviceContext` entity, under `Context Variables` in the palette. To learn more about Service Context and its parameters, see Chapter 6.

Let's use Designer's palette features in conjunction with our Java/Groovy editor to implement our condition:

1. Drag and drop the `serviceContext` entity from the `Context Variables` folder in your palette onto the script editor. This grabs the Service Context.
2. Drag and drop the `ddlRecord` entity from the `Dynamic Data Lists` folder in your palette onto the script editor. We get the `ddlRecordId` from the Service Context and use that ID to look up the DDL record via Liferay service utility
`DDLRecordLocalServiceUtil`.

Append the following Java code to the `DDLRecordLocalServiceUtil` script:

```
Field field = ddlRecord.getField("status");

String status = GetterUtil.getString(field.getValue());
if (status.contains("not")) {
    returnValue = "No"
}
else {
    returnValue = "Yes"
}
```

We're pulling out the status from the DDL record and returning a value indicating "Yes" to continue fixing the ticket issue or "No" to transition to the workflow's end state.

Add the following to the script's imports to finish things up:

```
import com.liferay.portlet.dynamicdatamapping.storage.Field;
```

Now the script accurately implements the condition logic we want. As a reminder, all of the code was injected into our workflow's XML file within the `<condition>` element that represents our condition node. Here's what this block of XML looks like, including the Java in our Goovy script:

```
<condition>
    <name>Resolution</name>
    <script><![CDATA[import com.liferay.portal.kernel.util.GetterUtil;
        import com.liferay.portal.kernel.workflow.WorkflowConstants;
        import com.liferay.portal.service.ServiceContext;
        import com.liferay.portlet.dynamicdatamapping.storage.Field;
        import com.liferay.portlet.dynamicdatalists.model.DDLRecord;
        import
com.liferay.portlet.dynamicdatalists.service.DDLRecordLocalServiceUtil;

        long companyId = GetterUtil.getLong((String)
workflowContext.get(WorkflowConstants.CONTEXT_COMPANY_ID));
        ServiceContext serviceContext = (ServiceContext)
workflowContext.get(WorkflowConstants.CONTEXT_SERVICE_CONTEXT);
```

```

        long classPK =
GetterUtil.getLong((String)workflowContext.get(WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));
        DDLRecord ddlRecord = DDLRecordLocalServiceUtil.getRecord(classPK);

        Field field = ddlRecord.getField("status");

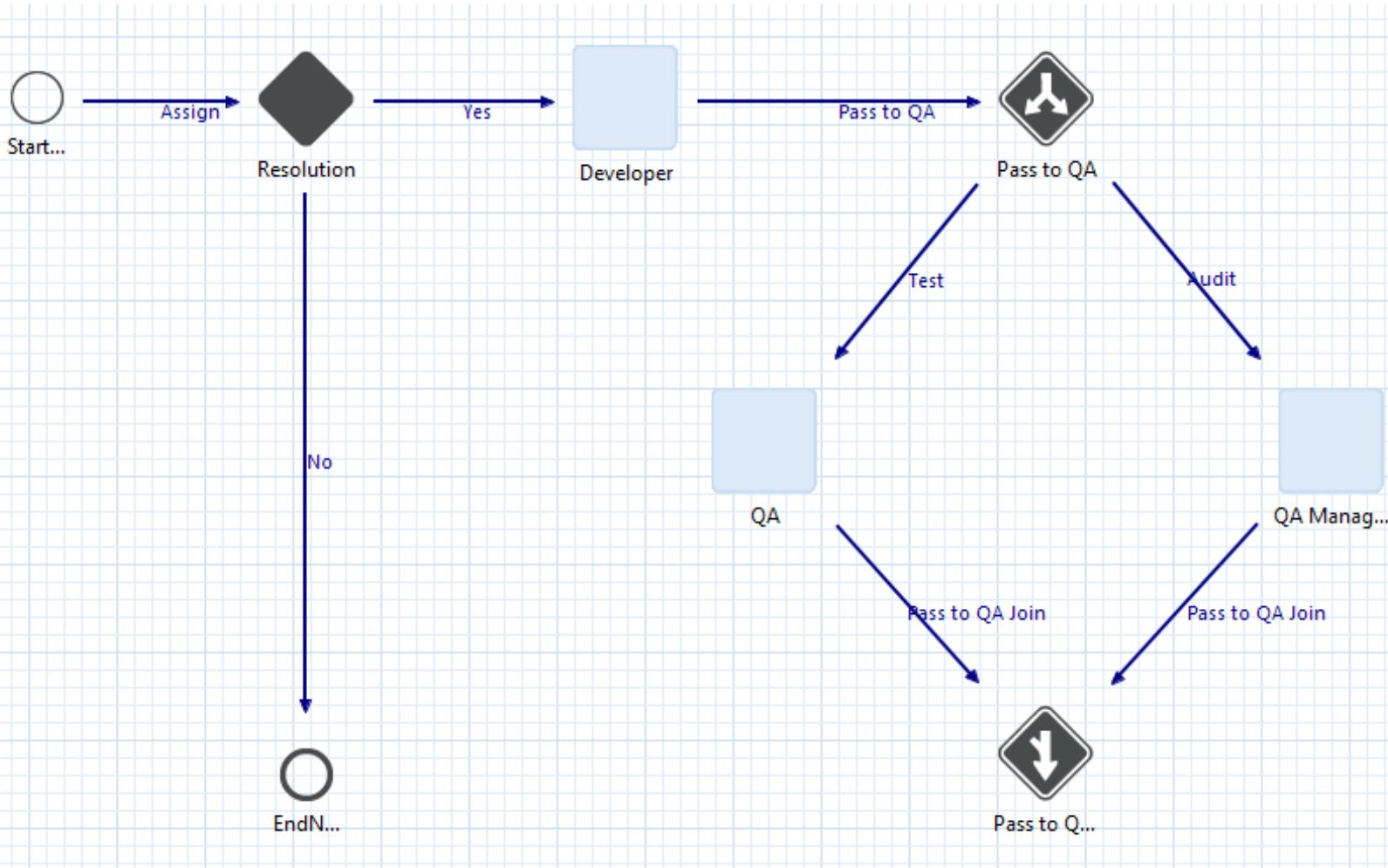
        String status = GetterUtil.getString(field.getValue());
        if (status.contains("not")) {
            returnValue = "No"
        }
        else {
            returnValue = "Yes"
        }]]></script>
<script-language>groovy</script-language>
<transitions>
    <transition>
        <name>Yes</name>
        <target>Developer</target>
    </transition>
    <transition>
        <name>No</name>
        <target>EndNode</target>
    </transition>
</transitions>
</condition>

```



Note: Make sure you correctly name the transitions stemming from the condition node. The "No" transition should point to the EndNode, while the "Yes" transition should point to the Developer. If the condition script's return values don't match the transition names, the workflow engine won't know which transition to use.

Here's a snapshot of our current ticket process workflow after inserting the condition node. If your transition names don't match the ones in this screenshot, you can change them by simply double clicking the transition names and editing them.



We need to create a valid DDL record to invoke this workflow properly. If you're thinking "How do we set up a DDL record?" or "How does this DDL record thingy work?", you're on the right track. If you're jumping up and down screaming "Liferay is da bomb!" We welcome your reaction, too. Regardless, we'll address DDLs soon in the *Configuring a DDL record* section of this chapter.

Next let's create a custom notification and write a template for it using a template editor.

8.4. Leveraging Template Editors for Notifications

Designer lets you leverage the FreeMarker editor to customize templates for your workflow notifications. A FreeMarker editor comes bundled with Developer Studio.

8.4.1. Creating Notifications

To access the template editor, click on the node of your choice and select the *Notifications* sub-tab in the *Properties* view. Create a new notification by clicking the green "plus" symbol.

The screenshot shows the Liferay 6.2 developer interface with the 'Properties' tab selected. On the left, a sidebar lists categories: Actions, Notifications (which is selected), Assignments, Task Timers, and Other. The main area displays a table for a notification named 'ticket process email'. A red box highlights the 'Template Language' column, which contains a dropdown menu with four options: 'Freemarker' (selected), 'Freemarker', 'Text', and 'Velocity'. To the right of the table are several icons: a green plus sign, a pencil, a red X, and arrows for sorting. Below the table, there are sections for 'Name' (set to 'ticket process email'), 'Template language' (set to 'Freemarker'), 'Execution type' (set to 'On entry'), 'Notification transports' (set to 'Notification Transport' with 'email' listed), and 'Addresses' (with an empty 'Address' field). On the far right, there are additional sorting and configuration icons.

There are several fields to fill in for your notification:

- **Name**
- **Template Language**
- **Execution Type**
- **Notification Type**
- **Notification Transports**
- **Addresses**

Click the pencil icon to open the editor associated with your notification's template language. Like the script editor, the template editor's *Palette* view lists entities that you can drag and drop

onto your workflow diagram.

Because Developer Studio lets you leverage a full featured FreeMarker template editor, content-assist is available for you to use right away. For example, content-assist suggests FreeMarker functions when you are editing your notification template. In addition, when you're doing a FreeMarker variable insertion, the editor gives you all the available variables that are a part of the Kaleo workflow. You can visit the documentation pages for FreeMarker for more information on the variables and functions available.

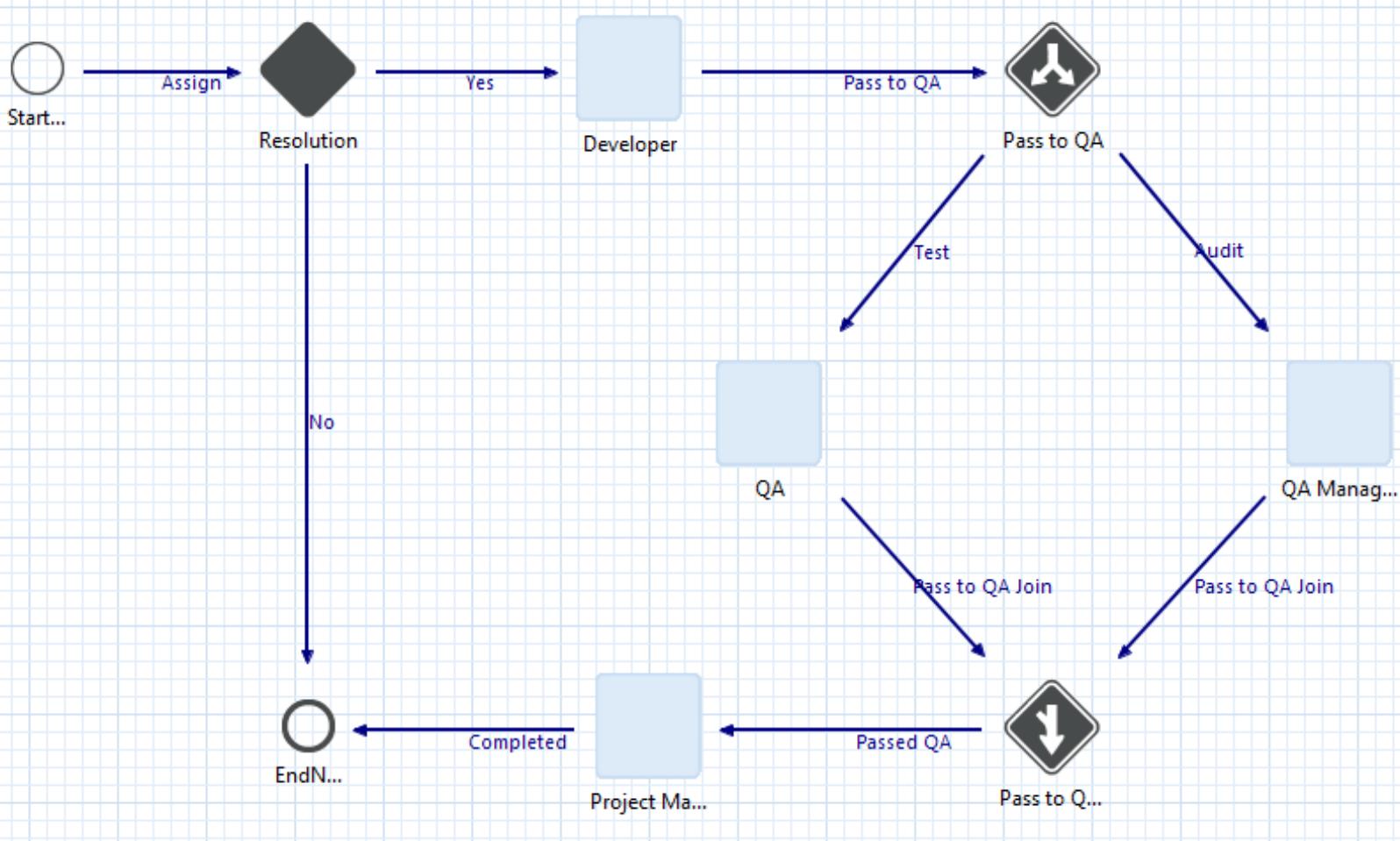


Note: Similar to the bundled script editors, Developer Studio lets you use the FreeMarker template editor to customize notifications in your workflow definition.

Let's continue editing our ticket process workflow. After completion of the QA task assignments, our Project Management team should be notified. To set up a notification email, we'll add a new task node that transitions from our join node. This new task node will hold our email notification. Typically, we'd assign this task to a project management role and email it to the project management team's email alias. For demonstration purposes, we'll use "Joe Blogs" for both purposes. As mentioned previously, you can specify an existing user that has an email or create a user with screen name "joe" having your email address. This process is similar to how we assigned our previous task nodes.

1. Drag a new *Task* node onto your workflow diagram.
2. Name the new node *Project Management* and select *Assign to a specific user*.
3. Click *Next*.
4. Enter *Screen name* "joe".
5. Click *Finish*.

Now we just need to incorporate the Project Management node into our workflow process. Add a transition named *Passed QA* from our join node to the Project Management node. Lastly, add a transition named *Completed* from our Project Management node to our *EndNode*. Here's an updated screenshot of what your workflow diagram should look like:



Let's create our email notification for our Project Management task node next. Click on the Project Management node and select *Notifications* in the Properties window.

To create the email notification, follow these steps:

1. Click the green "plus" symbol to create a new notification.
2. In the **Name** text field, enter "ticket process email".
3. Select *FreeMarker* from the **Template Language** drop-down menu.
4. Select *On entry* from the **Execution type** drop-down menu.
5. Select *email* under **Notification transports**.

Now open the FreeMarker template editor by clicking the pencil icon beneath the green "plus" symbol.

Insert FreeMarker code into the FreeMarker editor to specify your email notification template. We've provided an example email notification code snippet written in FreeMarker.

```

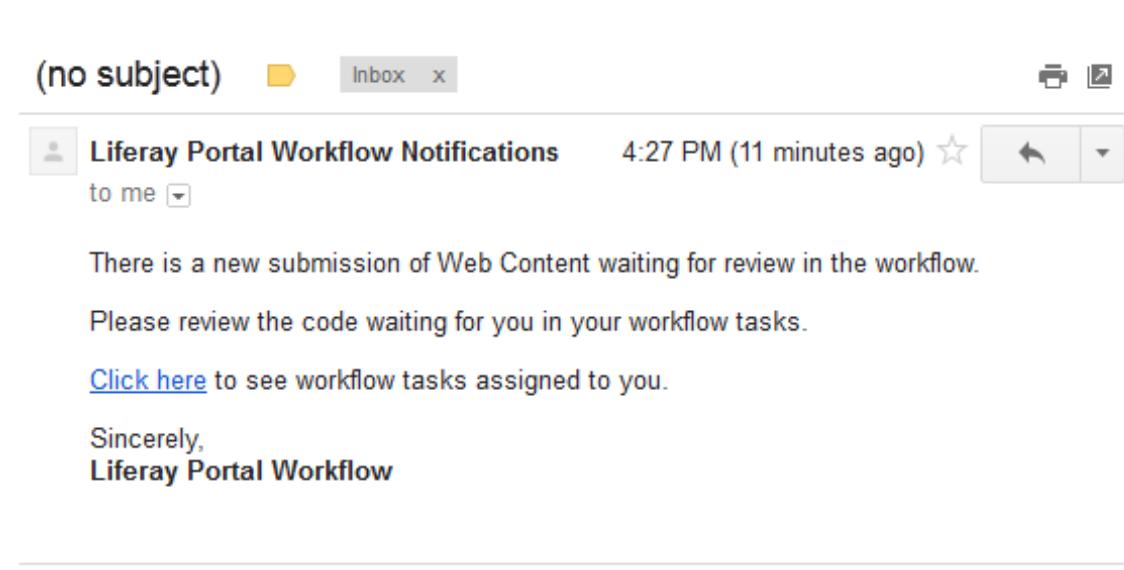
<#assign comments = taskComments!"">
<#assign portalURL = serviceContext.portalURL!"">
<#assign wTasksURL =
portalURL+{/group/control_panel/manage?p_p_id=153&p_p_lifecycle=0&p_p_state=m
aximized&p_p_mode=view&doAsGroupId="+groupId+"&refererPlid="}>

<!-- email body -->
<p> There is a new submission of ${entryType} waiting for review in the
workflow. </p>

<!-- personal message to assignee -->
<p> Please review the code waiting for you in your workflow tasks.
<if comments != "" > <br/> Assignment comment says:
<strong>${comments}</strong> </if>
</p>
<p> <a href="${wTasksURL}">Click here</a> to see workflow tasks assigned to
you. </p>

<!-- signature -->
<p>Sincerely,<br /><strong>Liferay Portal Workflow</strong></p>

```



The snapshot below illustrates what the snippet sends to the configured email recipient.

Your email notification is set up! Now, when the Project Management task node is activated in the workflow, the specified user (i.e. *joe*) will receive the notification email, all dressed up with your FreeMarker template (you might say it's all dressed up with somewhere to go).

With template editors, customizing your notification templates is easier than ever. FreeMarker comes bundled with Developer Studio so it's obviously the simplest solution.

Here's what the XML source looks like (with the embedded FreeMarker template) for the Project Management task we created:

```
<task>
  <name>project management</name>
```

```

<actions>
    <notification>
        <name>ticket process email</name>
        <template>/*
            specify task notification template */
            &lt;#assign comments = taskcomments!quot;&quot;&gt;;
            &lt;#assign portalurl =
servicecontext.portalurl!quot;&quot;&gt;;
            &lt;#assign wtasksurl =
portalurl!quot;/group/control_panel/manage?p_p_id=153&amp;p_p_lifecycle=0&am
p;p_p_state=maximized&amp;p_p_mode=view&amp;doasgroupid="+groupid+"+
&amp;refererplid="&gt;;
            &lt;!-- email body --&gt;;
            &lt;p&gt; there is a new submission of ${entrytype} waiting
for review in the workflow. &lt;/p&gt;

            &lt;!-- personal message to assignee --&gt;;
            &lt;p&gt; please review the code waiting for you in your
workflow tasks.
            &lt;if comments != " &gt; &lt;br/&gt; assignment
comment says: &lt;strong>${comments}&lt;/strong> &lt;#if&gt;
            &lt;/p&gt;
            &lt;p&gt; &lt;a href="${wtasksurl}"&gt;click
here&lt;/a&gt; to see workflow tasks assigned to you. &lt;/p&gt;

            &lt;!-- signature --&gt;
            &lt;p&gt;sincerely,&lt;br /&gt;&lt;strong&gt;liferay portal
workflow&lt;/strong&gt;&lt;/p&gt;
        </template>
        <template-language>freemarker</template-language>
        <notification-type>email</notification-type>
        <execution-type>onEntry</execution-type>
    </notification>
</actions>
<assignments>
    <user>
        <screen-name>joe</screen-name>
    </user>
</assignments>
<transitions>
    <transition>
        <name>Completed</name>
        <target>EndNode</target>
    </transition>
</transitions>
</task>

```

In the next section you'll see a list of workflow and service context content you can use when creating a customized script or template.

8.4.2. Workflow Context and Service Context Variables

A context variable provides a uniform variable to insert into your templates and scripts. When executed, a context variable is automatically deleted and replaced with the value pertaining to

that key. When you create notifications for a workflow, assign liferay portal context variables for a cleaner and more efficient process. With context variables, your notifications become more customizable, rather than following the same format for every recipient. The context variables you declare in your notifications refer to your liferay instance and the values it holds for your declarations.

Below you'll see tables listing numerous context variables and service context content. The context variables are the first table, followed by the service context content for web content, blog entries, and message board messages. We've separated service context content from the workflow context variables because service context keys depend on asset type, while context variables don't. Also, note the asterisks (*); they're used to flag context variables that depend on workflow activity.

Workflow Context Variables

key type description ---- ---- -----	companyid java.lang.string primary key of the company
entryclassname java.lang.string class name for entry used by the task (e.g. com.liferay.portlet.journal.model.journalarticle)	entryclasspk java.lang.string primary key of the entry class
entrytype java.lang.string type of entry used by the task (e.g. web content, blog entry, mb message)	groupid java.lang.string primary key of the assigned group
taskcomments* java.lang.string workflow comments assigned to the task	taskname* java.lang.string workflow task that activates the notification (e.g. review)
transitionname* java.lang.string name of transition pointing to the task (e.g. approve)	userid java.lang.string primary key of the assigned user ---

Web Content Service Context Variables - obtain via key servicecontext

key type description ---- ---- -----	articleid java.lang.string primary key of the web content
articleurl java.lang.string link to the web content in maximized mode	assetlinkentryids java.lang.string primary keys of the asset entries linked to the web content
assetlinkssearchcontainerprimarykeys java.lang.string primary keys of the asset link search container	assettagnames java.lang.string tag names applied the asset
autoarticleid java.lang.string boolean variable indicating whether an article id is generated (e.g. false)	classnameid java.lang.string primary key of the class name used by the task
classspk java.lang.string primary key of the model entity	content java.lang.string content of the web content
defaultLanguageId java.lang.String Primary key of the default language (e.g. en_US)	description_en_US java.lang.String Description of the web content (in English)
displayDateDay java.lang.String Calendar day the web content is set to display (e.g. 12)	displayDateHour java.lang.String Hour the web content is set to display (e.g. 4)
displayDateMinute java.lang.String Minute the web content is set to display (e.g. 26)	displayDateMonth java.lang.String Month the web content is set to display (e.g. 5)
displayDateYear java.lang.String Year the web content is set to display (e.g. 2012)	doAsGroupId java.lang.String Primary key of the organization associated with the web content
folderId java.lang.String Primary key of	

the web content's folder | `indexable` | `java.lang.String` | Boolean variable indicating whether the web content is searchable (e.g. true) | `indexableCheckbox` | `java.lang.String` | Boolean variable indicating whether the *Searchable* checkbox is checked (e.g. false) | `inputPermissionsShowOptions` | `java.lang.String` | Boolean variable indicating whether permission options are viewable (e.g. true) | `inputPermissionsViewRole` | `java.lang.String` | Role type that has permission to view web content (e.g. Site Member) | `languageId` | `java.lang.String` | Primary key of the selected language (e.g. en_US) | `localized` | `java.lang.String` | Boolean variable indicating whether the *Localizable* checkbox is selected (e.g. false) | `neverExpire` | `java.lang.String` | Boolean variable indicating whether the web content is set to expire (e.g. true) | `neverExpireCheckbox` | `java.lang.String` | Boolean variable indicating whether the *Never Auto Expire* checkbox is checked (e.g. false) | `neverReview` | `java.lang.String` | Boolean variable indicating whether the web content is set to review (e.g. true) | `neverReviewCheckbox` | `java.lang.String` | Boolean variable indicating whether the *Never Review* checkbox is checked (e.g. false) | `refererPlid` | `java.lang.String` | Primary key of the page hosting the web content | `smallImage` | `java.lang.String` | Indicates whether a small image is being used (e.g. on) | `smallImageURL` | `java.lang.String` | URL for the web content's attached image | `structureDescription` | `java.lang.String` | Description of the configured structure | `structureId` | `java.lang.String` | Primary key of the configured structure | `structureName` | `java.lang.String` | Name of the configured structure | `structureXSD` | `java.lang.String` | The XML schema definition used for the configured structure | `templateId` | `java.lang.String` | Primary key of the configured template | `title_en_US` | `java.lang.String` | Title of the web content (in English) | `type` | `java.lang.String` | Categorization type associated with the web content (e.g. Press Release) | `variableName` | `java.lang.String` | Custom variable name set for the web content's configured structure (e.g. Content) | `version` | `java.lang.String` | Current version of the web content (e.g. 1.0) | `workflowAction` | `java.lang.String` | Numerical value for the workflow action in progress (e.g. 1) | --- |

Blog Entry Service Context Variables - obtain via key serviceContext

Key | Type | Description | ---- | ---- | ----- | `assetLinkEntryIds` | `java.lang.String` | Primary keys of the asset entries linked to the blog entry | `assetLinksSearchContainerPrimaryKeys` | `java.lang.String` | Primary keys of the asset entries linked to the blog entry | `assetTagNames` | `java.lang.String` | Tag names applied to the asset (e.g. history, news, programming) | `attachments` | `java.lang.String` | Boolean variable indicating if blog entry has any attachments (e.g. true) | `content` | `java.lang.String` | Content of the blog entry | `description` | `java.lang.String` | Description of the blog entry (e.g. The comparison between two Fortune 500 companies) | `displayDateAmPm` | `java.lang.String` | Time "period" (based on the 12-hour clock) the blog entry is set to display (AM=0, PM=1) | `displayDateDay` | `java.lang.String` | Calendar day the blog entry is set to display (e.g. 3) | `displayDateHour` | `java.lang.String` | Hour the blog entry is set to display (e.g. 26) | `displayDateMinute` | `java.lang.String` | Minute the blog entry is set to display (e.g. 32) | `displayDateMonth` | `java.lang.String` | Month the blog entry is set to display (e.g.

8) | displayDateYear | java.lang.String | Year the blog entry is set to display (e.g 2012) | doAsGroupId | java.lang.String | Primary key of the organization associated with the blog entry | editor | java.lang.String | Content of the blog entry (equivalent to the content value) | entryId | java.lang.String | Primary key of the blog entry | refererPlid | java.lang.String | Primary key of the page hosting the blog entry | smallImage | java.lang.String | Indicates whether a small image is being used (e.g. on) | smallImageURL | java.lang.String | URL for the blog entry's attached image | title | java.lang.String | Title of the blog entry (e.g. My Blog Entry) | workflowAction | java.lang.String | Numerical value for the workflow action in progress (e.g. 2) | ---

Message Board Message Service Context Variables - obtain via key serviceContext

Key | Type | Description | ---- | ---- | ----- | anonymous | java.lang.String | Boolean variable indicating if message is anonymous (e.g. true) | assetLinkEntryIds | java.lang.String | Primary keys of the asset entries linked to the message | assetLinksSearchContainerPrimaryKeys | java.lang.String | Primary keys of the asset link search container | assetTagNames | java.lang.String | Tag names applied the asset (e.g. sea, sailing, swimming) | attachments | java.lang.String | Boolean variable indicating whether the message has any attachments (e.g. false) | body | java.lang.String | Content of the message | editor | java.lang.String | Content of the message (equivalent to the body value) | mbCategoryId | java.lang.String | Primary key of the message's category | messageId | java.lang.String | Primary key of the message | parentMessageId | java.lang.String | Primary key of the message's parent message | preview | java.lang.Boolean | Boolean variable indicating if message is available to preview (e.g. true) | question | java.lang.String | Boolean variable indicating if message is marked as a question (e.g. false) | subject | java.lang.String | Subject line of the message (e.g. My MB Subject) | threadId | java.lang.String | Primary key of the message board thread hosting the message | workflowAction | java.lang.String | Numerical value for the workflow action in progress (e.g. 3) | ---

Next you'll learn how to view your workflow definition XML file in Developer Studio.

8.5. Viewing Workflow Definition XML Source

The workflow diagram view of your workflow definition is convenient; sometimes you'll also want to edit and review your workflow definition's XML source code. Selecting the *Source* tab next to the *Diagram* tab in the main editor view takes you to the XML, and you can easily switch contexts as you need.

```

<?xml version="1.0" encoding="UTF-8"?>
<workflow-definition xmlns="urn:liferay.com:liferay-workflow_6.2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema"
    <name>Ticket Process</name>
    <version>1</version>
    <state>
        <name>StartNode</name>
        <metadata><![CDATA[{"transitions": {"Resolution": {"bendpoints": []}, "Assign": {"bendpoints": []}, "Developer": {"bendpoints": []}}, "initial": true}></metadata>
        <initial>true</initial>
        <transitions>
            <transition>
                <name>Assign</name>
                <target>Resolution</target>
            </transition>
        </transitions>
    </state>
    <task>
        <name>Developer</name>
        <metadata><![CDATA[{"transitions": {"approved": {"bendpoints": []}, "EndNode": {"bendpoints": []}, "Pass to QA": {"bendpoints": []}}, "user": null}]}></metadata>
        <assignments>
            <user/>
        </assignments>
        <transitions>
            <transition>
                <name>Pass to QA</name>
                <target>Pass to QA</target>
            </transition>
        </transitions>
    </task>

```

Source mode offers you its own cool features:

- *Section Highlighting*: Shows XML source code for the node or transition currently selected in the workflow diagram. Click the document icon in an entity's floating palette or right click the entity and select *Show in source*.
- *Editor Validation*: Displays an error marking in the editor's gutter if your code is invalid. Click the error marking to see hint for resolving the error.
- *Content Assist*: Suggests language specific functions and variables as you edit code. When you're inserting variables, it also lists all available variables that are a part of the Kaleo workflow.



Note: Using Liferay Portal section Creating New Workflow Definitions explains how to define workflows via XML.

With the *Source* view, you can keep track of your edits while using Developer Studio's powerful graphical features.

Let's save your workflow definition and publish it to your Liferay server.

8.6. Publishing Workflows to the Server

After you create a new workflow or modify an existing one, you'll have to publish it onto your Liferay server before your site's members can use it. Let's publish the `ticket` process workflow definition onto your Liferay server.

To publish your `ticket` process workflow definition:

1. Right-click the *Kaleo Workflows* folder listed under your Liferay server in the *Servers* view.
2. Select *Upload new workflow...* to bring up the *workspace files* browser.
3. Browse for your workflow definition file and select it for publishing.

Alternatively, you can publish your new workflow XML file by dragging it from your *Package Explorer* view onto your Liferay server in your *Servers* view.

If you are not using the Kaleo Designer for Java plugin and you'd like to publish workflow definitions you've written in XML, you can do so by uploading them from within Liferay Portal's workflow configuration screens. For details, see the chapter on using workflow in *Using Liferay Portal 6.2*.



Note: To update your Kaleo Workflows folder with the latest workflow versions created or modified in Liferay Portal using Kaleo Workflow Designer from the *Kaleo Forms* portlet, right click *Kaleo Workflows* under your server and select *Refresh*.

You probably understand why it's necessary to publish new workflow definitions onto the Liferay server; it might be less clear why you need to republish existing workflow definitions that you've modified. When you save changes to a workflow, they're not immediately available in your portal; it's still using the previous version of the workflow. Developer Studio saves the workflow as a draft, so you can work on multiple iterations of the same version until you're ready to publish your changes. Once you publish, you now have a new version to make changes on top of. For example, you might be working on *Version 1* of your workflow definition; as you make changes, you save them in multiple drafts. When you are finished with all of your changes, you publish the workflow triggering creation of a new version (*Version 2*) of the workflow. The new version is made available on the server immediately for your workflow administrators to associate with asset publications, DDLs, and with Kaleo Forms.

Unlike other Java editors, Developer Studio lets you test your workflow definition as a draft. You can also publish your workflow definition straight to Liferay Portal for quick and easy configuration.

Are you ready for our *Kaleo Designer for Java* finale? You just have to activate the workflow in your Liferay Portal, then we'll set up the DDL record and try out our new workflow. Let's activate!

8.7. Using Workflows in Liferay Portal

Let's put some finishing touches on your workflow and test drive it in Liferay Portal. Before you can use a workflow definition, it must be activated in your Liferay Portal. Navigate to the *Control Panel* and, under the *Configuration* heading, select *Workflow*. Then, in the *Definitions* tab, click on the *Actions* button and select *Activate*.

A workflow definition can be associated with publication of an asset or DDL record. Let's associate our ticket process workflow definition with a DDL record that lets a developer indicate whether she'll fix a ticket's issue. You can find detailed instructions for creating a DDL by visiting the section Defining data types in *Using Liferay Portal*. We'll demonstrate how easy it is.

8.8. Using Dynamic Data Lists (DDLs) with Workflows

Let's associate our workflow with a Dynamic Data List (DDL) record. To learn more about DDLs, visit Using Web Forms and Dynamic Data Lists in *Using Liferay Portal*.

First we'll create a data definition that lets the user select a status value.

1. In Liferay Portal, go to *Site Administration* → *Content* → *Dynamic Data Lists*.
2. Click the *Manage Data Definitions* link, then *Add* a new data definition.
3. Name the data definition *Status*. Then, in the *Fields* tab, drag and drop the *Select* field onto the canvas.

The screenshot shows the Liferay Dynamic Data List (DDL) configuration interface. At the top, there is a 'Default Language' dropdown set to 'English (United States)' with a 'Change' button and an 'Add Translation' button. Below this, there are two tabs: 'Fields' (which is selected and highlighted with a red box) and 'Settings'. On the left, there is a list of field types: Boolean, Date, Decimal, Document, HTML, Integer, Link to P..., Number, Radio, Text, and Text Box. Each field type has a corresponding icon. The 'Select' field type icon is also present and is highlighted with a red box. On the right, a configuration dialog for the 'Select' field is open, showing a dropdown menu with 'option 1' selected. There are also icons for edit, add, and delete operations.

4. In the *Settings* tab, double click the *Name* property to open the property editor--enter *status*, in lowercase, as the value. Then click *Save*.
5. Edit the *Options* setting; give your *status* field option values of *fix* with label "Fix" and *not* with label "Do not fix".
6. Click *Save*.

Recall the code we inserted for our condition node:

```
Field field = ddlRecord.getField("status");

String status = GetterUtil.getString(field.getValue());
if (status.contains("not")) {
    returnValue = "No"
}
else {
    returnValue = "Yes"
}
```

In our code, the `getField()` method ingests the value of our DDL field named "status". When the script is invoked, if the value for the status field is *not*, the value *No* is returned and our workflow transitions to our *EndNode* state. Otherwise, the workflow transitions to our *Developer* task node.

After you create the data definition, create a DDL. Make sure you select the ticket process workflow and the Status data definition when creating this DDL. *Save* the DDL.

Now our DDL is set for use inside our ticket process workflow! Let's use the Kaleo Forms portlet to test our new workflow definition!

8.8.1. Using Kaleo Forms to Run Workflows

Let's use the Kaleo Forms portlet to invoke our workflow from Liferay Portal. Deploy the Kaleo Forms portlet to your portal and add it to a page on your site. You can learn how to use Kaleo Forms in the Kaleo Forms chapter of *Using Liferay Portal*; we'll demonstrate its use here by using it with our ticket process workflow.

1. Create a new *Process* in Kaleo Forms; name it *Ticket Process*.
2. Select the *status* entry definition we created earlier.
3. Add an *Initial Form* based on our *status* data definition.
4. Select our ticket process workflow.
5. Leave *Workflow Task Forms* unassigned for our demonstration.

Kaleo Forms

New Process

Name (Required)
Ticket Process

Description

Entry Definition (Required)
Status Select

Initial Form (Required)
Status Select

Workflow
Ticket Process (Version 1) Select

Workflow Task Forms
Assign

Save Cancel

After saving, select the *Summary* tab in Kaleo Forms, click the *Submit New* button, and select *Ticket Process*.

Now you can interact with the DDL and progress throughout the ticket process using Kaleo Forms! Joe Bloggs is assigned the task once the ticket reaches the QA and Pass to QA tasks. Remember to sign in as Joe Bloggs to access the tasks assigned to him. Mr. Bloggs should also receive an email when the Project Management node is activated.

You successfully created a workflow definition and created a workflow process within Liferay Portal! You're officially a workflow master (you can get your framed certification at the front desk on your way out--tell them we sent you)!

Have you noticed that there's a lot of depth to Liferay IDE? You can easily come across difficult

questions and run into very specific problems, but someone else might have already solved your issue or answered your question. So where would you go to find out? Don't reinvent the wheel, visit the Liferay IDE Community page! On the *Forums* page, you can look up resolutions to specific errors and ask questions. Be sure to fully describe any problems you have to ensure you get a working answer. You can even track known issues from the *Issue Tracker* page.

8.9. Summary

In this chapter we looked at managing workflows using Kaleo Designer for Java in Liferay Developer Studio. With Developer Studio's dynamic workflow diagram, you can visualize your workflow and create steps to ensure a successful business process. The script and template editors give you a convenient workspace to further enhance your workflow development environment.

Because Developer Studio has access to your custom business logic APIs and Liferay Portal APIs, it's simple to customize your workflow definition. To get your workflows running in

Liferay, you can publish directly to your existing Liferay server; the configuration process is quick and easy. With all of its features, the Kaleo Designer for Java in Developer Studio is a powerful yet simple tool.

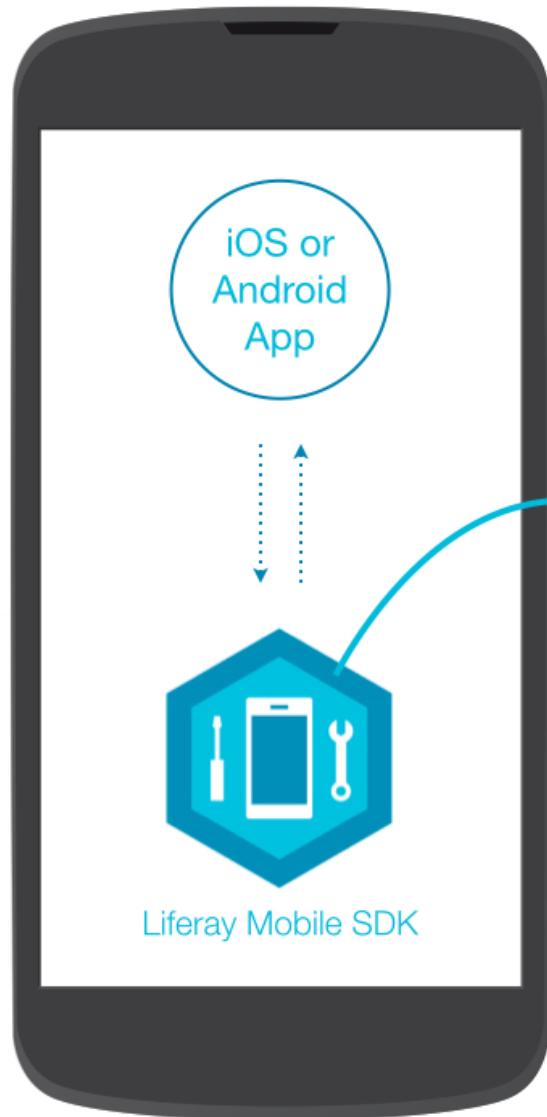
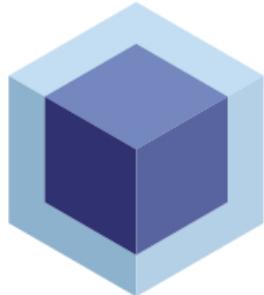
Have you been wondering how your mobile apps can tap into the capabilities of Liferay's built-in apps, and your own custom apps? If this has been on your mind, the next chapter on using Liferay's Mobile SDK is for you. You'll learn how to leverage Liferay utilities, Liferay's services, and your custom app services from your Android and iOS apps. So, buckle up and grab your mobile devices as you integrate Liferay with your mobile apps.

9. Creating Mobile Apps that Use Liferay

The Liferay Mobile SDK is a way to streamline consuming Liferay core web services, Liferay utilities, and custom portlet web services. It wraps Liferay JSON web services, making them easy to call in native mobile apps. It takes care of authentication, makes HTTP requests (synchronously or asynchronously), parses JSON results, and handles server-side exceptions so you can concentrate on *using* the services in your app.

The Liferay Mobile SDK is compatible with Liferay Portal 6.2 and later. The Liferay Android SDK and Liferay iOS SDK are ready for you to download and use. The Mobile SDK's official project page gives you access to the SDK releases, provides the latest SDK news, and has forums for you to engage in mobile app development discussions.

Liferay Developer Studio



Liferay Instance

The Liferay Mobile SDK plugin for Eclipse simplifies developing Android apps that use Liferay. You can configure the Mobile SDKs manually to use with Android apps and/or iOS apps. Once configured, you can invoke Liferay services from your app. The Liferay Mobile SDK bridges your native app with Liferay services.

In this chapter, we'll demonstrate developing Android and iOS apps that communicate with Liferay via the Mobile SDK. Along the way, we'll discuss these topics:

- Setting Up the Mobile SDK
- Creating the Liferay Android Sample Project

- Calling Liferay Services in your Android App
- Using Custom Services in your Android App
- Using the Android SDK
- Using the iOS SDK

Let's get started by accessing Liferay from an Android app in Eclipse.

9.1. Setting Up the Mobile SDK

Liferay provides the *Liferay Mobile SDK Eclipse plugin* for you to use in developing your mobile apps. Its powerful *Mobile SDK Builder* generates libraries that enable your app to communicate with Liferay Portal and with the custom portlet services deployed on your Liferay Portal instance. The plugin also comes with a *Liferay Android Sample Project* that you can use as a reference for building more Android apps of your own. Since Eclipse only supports Android apps, you can't use the plugin to customize iOS apps.



Note: If you want to use the Mobile SDK within an existing Android or iOS project or if you're using a different IDE, like *Android Studio*, you can download the latest version of the Mobile SDK and add it to your project library. If you want to use the Mobile SDK in a Maven or CocoaPods project, you can configure it as a dependency.

For more information about these topics, read the Android or iOS instructions on manually setting up the Mobile SDK for an existing project.

The Liferay Mobile SDK plugin depends on the *Android SDK Tools* and *Android Development Tools (ADT)* Eclipse plugins. To satisfy these dependencies, you can install these plugins onto an existing Eclipse instance, or you can install Google's *Android Developer Tools SDK* bundle which includes both of the plugins. Here are the instructions for each of these options:

- **To install the *Android SDK Tools* and *ADT* plugins manually:** Follow the Setting Up an Existing IDE section on the Android developer site to download and install the ADT Eclipse plugin to your pre-existing Eclipse or Liferay Developer Studio/IDE installation.
- **To install the *Android Developer Tools SDK* bundle:** Follow the Setting Up the ADT Bundle section on the Android developer site to download and install the ADT bundle, which is built on Eclipse and which includes the required ADT plugins.

Once you've installed the required Android plugins, you can install the Liferay Mobile SDK to your Eclipse instance by following these steps:

1. Go to the *Help → Install New Software* menu.
2. Copy the following URL into the *Work with* field:

<http://releases.liferay.com/tools/ide/latest/stable/>

In the *Liferay* drop-down menu that appears, click the drop-down arrow, select *Liferay Mobile SDK*, and click *Next*.

Install

Available Software

Check the items that you wish to install.

Work with: <http://releases.liferay.com/tools/ide/latest/stable/> Add...

Find more software by working with the ["Available Software Sites"](#) preferences

type filter text

Name	Version
Liferay	
Liferay Mobile SDK (Requires Android Development Tools, ADT)	2.1.0.201403281241-ga1

Select All Deselect All

Details

Show only the latest versions of available software Hide items that are already installed

Group items by category [What is already installed?](#)

Show only software applicable to target environment

Contact all update sites during install to find required software

?

< Back Next > Finish Cancel

3. Click *Finish* to complete installing the Liferay Mobile SDK plugin.

Terrific! You're ready to start developing Android apps that use Liferay!

Next, let's create a *Liferay Android Sample Project* to learn how easy it is to create a mobile app that uses Liferay.

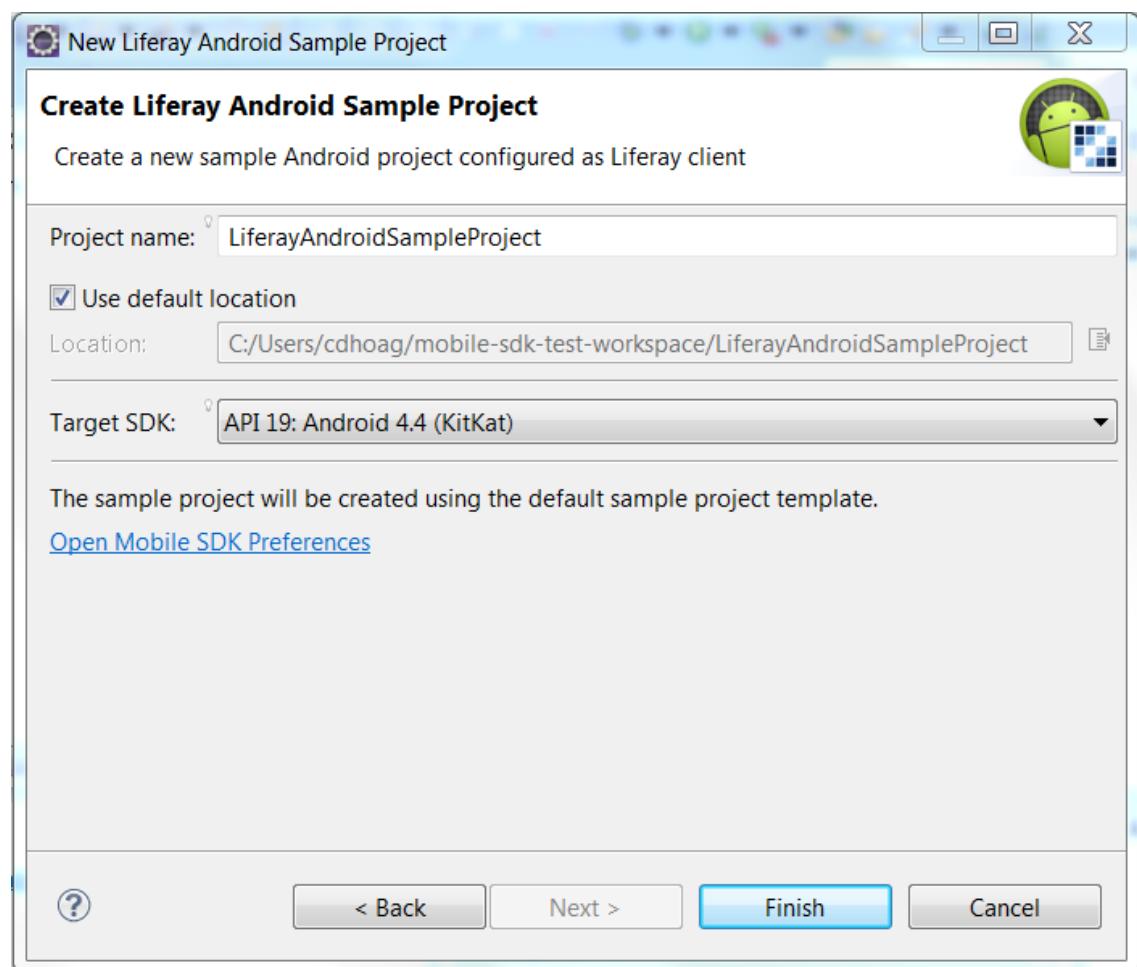
9.2. Creating the Liferay Android Sample Project

The Mobile SDK Eclipse plugin offers the ability to create a new sample Android project from scratch. The sample project includes Liferay Mobile SDK libraries and sample app code. We call this project the *Liferay Android Sample Project*. You can refer to it to see how it connects with Liferay and invokes Liferay services.

Alternatively, you can also browse and download the *Liferay Android Sample Project* source code in GitHub.

Let's create the sample project now!

1. Go to *File* → *Project...* → *Liferay* → *Liferay Android Sample Project*.
2. Click *Next*.
3. Specify the *Project name*, *Location*, and *Target SDK*. Notice that you also have the option to choose a sample project template from the Mobile SDK preferences menu.
4. Click *Finish* to create the sample Android app.



The sample Android app has the standard Liferay Mobile SDK JARs as well as sample classes that characterize how the app integrates with

your portal instance.

The Liferay Android Sample Project lists the portal's users as contacts, so that you can view their

detailed information, including their names, email addresses, phone numbers, and birth dates. Just as evidence that this sample project is using the Liferay Mobile SDK libraries, open the sample app's `MainActivity` class from the `src/com/liferay/mobile/sample/activity` directory. In Eclipse, scroll your mouse over the `Session` object instance within the `onListItemClick()` method.

The screenshot shows the `MainActivity.java` code in Eclipse and the `Liferay Contacts` application running on an Android emulator. The code highlights the `Session` interface reference. The application interface displays a list of users: Cody, Jesse, Russ, Rich, Steve, and Jim.

```
return true;
}

public void onListItemClick(
    ListView listView, View view, int position, long id) {
    User user = (User) getListAdapter().getItem(position);

    Session session = SettingsUtil.getSession();
    com.liferay.mobile.android.service.Session contactService(session);
    contactService(contactId());
}
```

The `Session` interface is referenced from the `com.liferay.mobile.android.service` package.

This app uses the Liferay Mobile SDK from the library JARs in this project.

Let's run the sample app in Android's emulator.

1. Start a local portal instance on port 8080, if one is not already running.

2. In Eclipse, go to *Run* → *Run As* → *Android Application*.

3. Go to the *Liferay Contacts* app.

The *Liferay Contacts* app lists your site's users. You can select users' names to see their information.

Congratulations on creating a

mobile app that uses Liferay!

Now let's consider the Android apps you've been developing. Is there data or functionality that you'd like to leverage from your Liferay Portal instances? The Mobile SDK enables you to use Liferay core services to get what you need from your portal. This is easy to do with Liferay's Mobile SDK Eclipse plugin. We'll show you how to use it next.

9.3. Calling Liferay Services in your Android App

The Mobile SDK Eclipse plugin makes it easier than ever for you to call Liferay core services and utilities from your Android app. All you need to do is add the Liferay Android Mobile SDK libraries to your project. The Liferay Mobile SDK Builder that comes with our Eclipse plugin enables you to generate these libraries.



Note: This section assumes that you created your Android project using the ADT plugin for Eclipse. If you didn't create the project using the ADT plugin or if it uses Gradle or Maven, see the manual Liferay Mobile SDK setup instructions for Android.

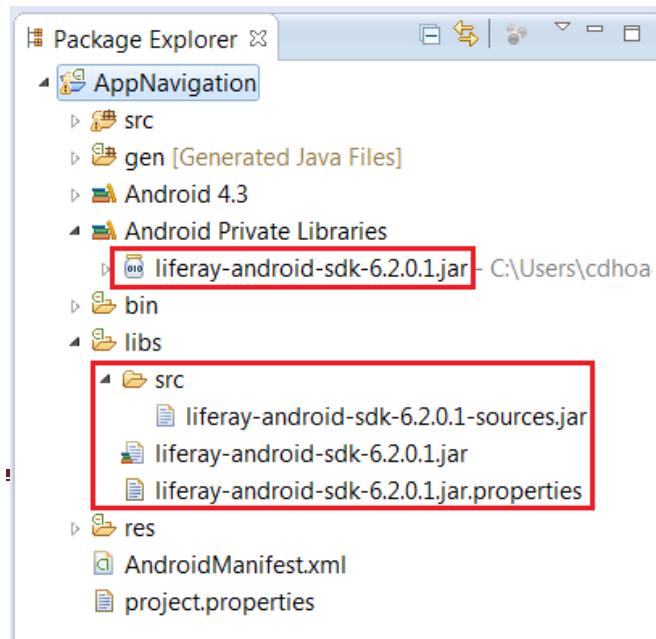
Let's see how easy it is to set up the Liferay Android SDK in your Android app project.

1. In Eclipse, import your Android app into an Android application project.
2. Right-click your app and select *Configure → Add Liferay Android SDK libraries*.

In the Eclipse console, the plugin prints a success message indicating that the Liferay Android SDK libraries were added to the project.

Wasn't that easy? Your Android app now has the Liferay Android SDK libraries, enabling it to call any of Liferay's core web services! Let's take a look behind the scenes and discover what happened.

In your project's `libs` folder, the Liferay Android SDK's `.jar` file and `.properties` file were added. Furthermore, a JAR file containing the Liferay Android SDK's source code was added to the project's `src` folder. Lastly, the Liferay Android SDK library JAR was added as one of your project's *Android Private Libraries*, making it available in your project's classpath.



Great! Now you know how to set up your Android apps to use Liferay core web services and Liferay utilities. But how do you access custom portlet services? Does the Liferay Mobile SDK Eclipse plugin make that possible too? The answer is an emphatic *YES*. We'll show you how to configure your app to use custom portlet services next.

9.4. Using Custom

Portlet Services in your Android App

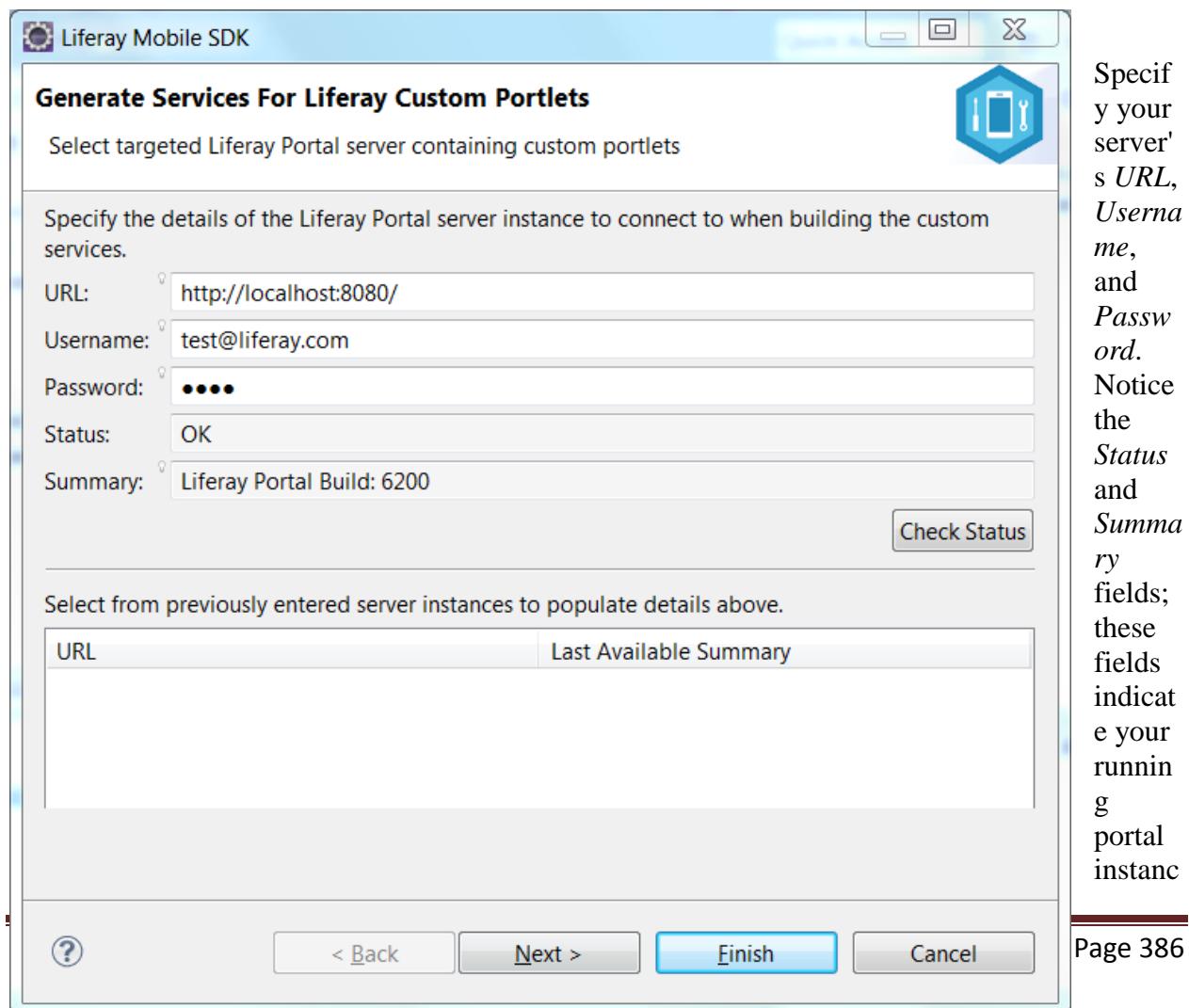
The Liferay Mobile SDK makes it easier to call your custom portlet services. The Liferay Mobile SDK Builder that comes with our Eclipse plugin generates libraries that enable you to access your custom portlet's remote JSON web services.

You can also use the Liferay Mobile SDK Builder via command line. Read these instructions for more details.

You can specify a running Liferay Portal instance in Eclipse, and Eclipse queries the server for all the remote APIs that are available to generate .jar files to access your custom portlet services. The Liferay Mobile SDK Eclipse plugin finds your custom services for you to select the ones you want to use in your app. The Mobile SDK Builder generates a .jar file and resource files that let you access the services in from your Android app.

Now that you know the basic gist of how the Mobile SDK Eclipse plugin generates custom portlet service access libraries, let's actually generate them for an existing Android app.

Right-click your Android app project in the Package Explorer and select *Configure → Generate services for Liferay custom portlets*. This opens a wizard where you'll specify your running portal server.

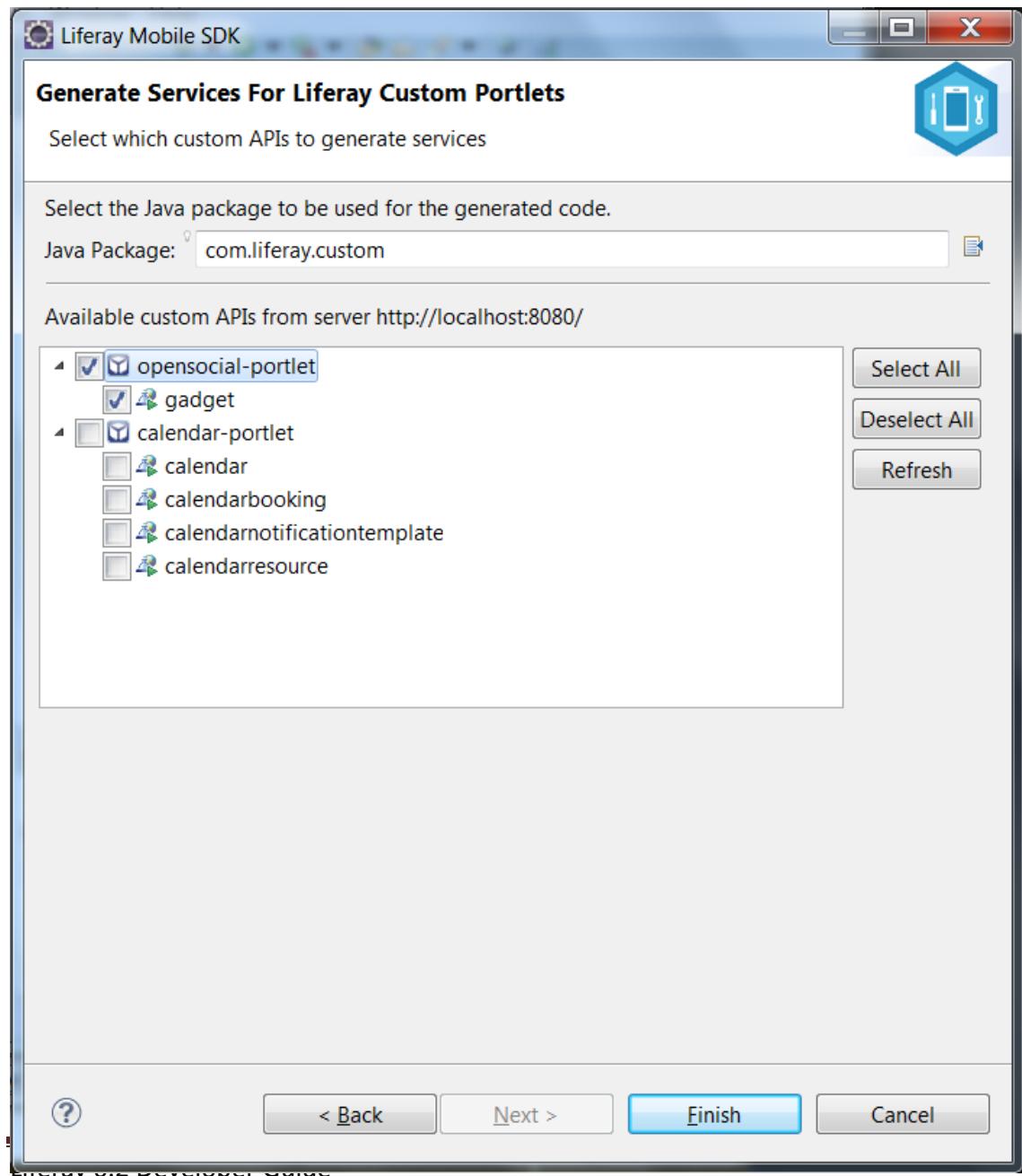


e's status after clicking the *Check Status* button.

The plugin saves previously configured server instance connection information in a listing at the bottom so that you can configure your Android app to use any previously configured portal server. Each server's URL and Last Available Summary is displayed.

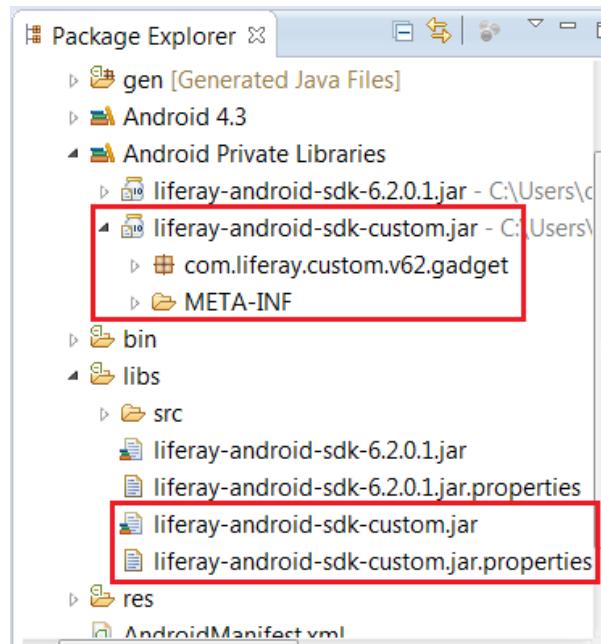
Once you've checked the status of your server, click *Next*. The plugin queries the server you specified for all the remote APIs that are available, and generates for your custom portlet services.

In the next menu, specify the Java package for the SDK source code that the builder generates. Then select the custom portlet APIs that you want to access.



For demonstration purposes, we'll specify the Java package as com.liferay.custom and select an arbitrary custom API (such as the *opensocial-portlet* → *gadget* custom API).

m API shown in the figure below).



Notice that you have the standard and custom .jar and .properties files. When generating custom services, Eclipse adds both the standard JAR file used for accessing Liferay core services and utilities and a custom JAR file for accessing your custom portlet services.

You've generated custom services for your Liferay portlets! Now your app can access Liferay core services, Liferay utilities, *and* your custom portlet services.

Now that your Android app has the access it needs to use Liferay and your custom portlets, let's take an in-depth look at using the Android SDK to invoke their services.

9.5. Using the Android SDK

Suppose you're creating an Android app, and you'd like to access some of Liferay's core services. All you need to do is download the SDK and put it in your classpath, and you can access the services immediately. If you want to invoke custom portlet web services, you'll need to generate client libraries for them. You can learn more about generating these libraries by reading the Building an SDK for Custom Portlet Services section.

If you're developing in an Eclipse environment and are using the Liferay Mobile SDK Eclipse plugin, you can skip the next section. Otherwise, continue onto the next section where we'll get started by downloading the Android SDK and configuring it in your Android environment!

9.5.1. Manually Setting Up the Android SDK

As we stated earlier, you'll need to download the latest version of `liferay-android-sdk.jar`. In addition, if you want to debug the SDK source code, you can download `liferay-android-sdk-sources.jar` and attach the source code to your IDE project. You can download these JAR files from the Liferay Mobile SDK project page.

Once you've downloaded your JAR file, copy it into the `/libs` directory of your Android project. Android Developer Tools should add this JAR automatically to your classpath. If you're using a different IDE, make sure this JAR is added to the project classpath. Now, you'll be able to import the SDK classes for your app to use.

Great! Now let's start accessing Liferay services from your app.

9.5.2. Invoking Liferay Services in Your Android App

Now that you've downloaded the Liferay Mobile SDK for Android and placed it in your Android

project's classpath, let's consider how to access and invoke Liferay services from within an Android application. Here are the steps to follow:

1. Create a session.
2. Import the Liferay services for your app to use.
3. Create a service object and call its service methods.

We'll show how the Liferay Mobile SDK Sample Android App demonstrates these steps. In particular, we'll outline the steps that its `UsersAsyncTask` class takes in accessing and calling Liferay services. In your app, you can follow these steps too.

9.5.2.1. Step 1: Create a session

The session is a conversion state between the client and server, which consists of multiple requests and responses between the two. We need a session to pass requests between your app and the Mobile SDK.

The sample app establishes a session by means of user authentication. It creates the session in the `UsersAsyncTask` class as follows:

```
Session session = new SettingsUtil.getSession();
```

The `getSession()` method returns a `Session` instantiated like this:

```
SessionImpl("http://10.0.2.2:8080", "test@liferay.com", "test");
```

Here's an explanation of each of the session parameters:

Server: The URL of the Liferay instance you're connecting to. If you're running your app on an Android Emulator, the URL should point to your local Liferay instance. In this particular case, `http://10.0.2.2:8080` is equivalent to `http://localhost:8080`, which means the emulator and Liferay are running on the same machine.

Username: Can either be the user's email address, screen name, or user ID. Your session login user name must be consistent with the authentication method your Liferay instance is using. Liferay's default authentication method requires the user's email address.

Password: The user's password.



Warning: Take care when using administrator credentials on a production Liferay instance, as you'll have permission to call any service. Make sure not to modify data accidentally. Of course, the default administrator credentials should be disabled on a production Liferay instance.

9.5.2.2. Step 2: Import the Liferay services for your app to use

Being a contacts app, the sample app imports the Mobile SDK's `UserService` class to connect to Liferay Portal's `UserService`:

```
import com.liferay.mobile.android.v62.user.UserService;
```

Note, the Liferay version (`.v62`) is used in the package namespace. Since the SDK is built for a specific Liferay version, service classes are separated by their package name. In this example, our Mobile SDK classes use the `.v62` package, which means this SDK is compatible with

Liferay 6.2. But, you can use several SDKs in your classpath simultaneously to support different Liferay versions.

Your portal's JSON web services page (e.g., <http://localhost:8080/api/jsonws>) lists all available portal services and portlet services.

9.5.2.3. Step 3: Create a service object and call its service methods

The sample app creates a `UserService` object and calls its `getGroupUsers(...)` method to fetch all of the Guest site's users:

```
UserService userService = new UserService(session);

...

long groupId = getGuestGroupId(session);

JSONArray jsonArray = userService.getGroupUsers(groupId);

for (int i = 0; i < jsonArray.length(); i++) {
    JSONObject jsonObj = jsonArray.getJSONObject(i);

    users.add(new User(jsonObj));
}
```

Since the `userService.getGroupUsers(...)` method requires a site group ID, we invoke the method `getGuestGroupId(session)` of the sample app's `UsersAsyncTask` class to get the Guest site's group ID, and then we pass that group ID as the `groupId` parameter in the call `userService.getGroupUsers(groupId)`.



Note: Many service methods require a group ID as a parameter. The SDK's `GroupService` class, which uses Liferay Portal's `GroupService`, provides methods for getting a site's Group and other scope groups.

The call `JSONArray jsonArray = userService.getGroupUsers(groupId)` demonstrates making a basic synchronous service call; the method only returns after the request has finished.

Service method return types can be `void`, `String`, `JSONArray`, and `JSONObject`. Primitive type wrappers can be `Boolean`, `Integer`, `Long`, and `Double`.

So far, we've explained the basic process of accessing Liferay services through the Mobile SDK. Next, we'll explore making asynchronous HTTP requests to your portal's services.

9.5.3. Invoking Services Asynchronously from Your Android App

Android doesn't allow making synchronous HTTP requests from the main UI thread.

Synchronous HTTP requests must be made from threads other than the main UI thread. For example, they can be made from an `AsyncTask` instance.

The SDK can help you make asynchronous HTTP requests if you don't want to create an `AsyncTask` yourself. Implement a callback class, instantiate it, and set the callback instance on the session. When the SDK makes your service calls for that session, it makes them asynchronously. To make synchronous calls again, set `null` as the session's callback.

The following steps outline how to implement asynchronous requests in your app:

1. Implement and instantiate your callback class.
2. Set the callback on the session.
3. Call Liferay services.

The following code is an implementation of these steps:

```
import com.liferay.mobile.android.task.callback.AsyncTaskCallback;
import
com.liferay.mobile.android.task.callback.typed.JSONArrayAsyncTaskCallback;

...
JSONArrayAsyncTaskCallback callback = new JSONArrayAsyncTaskCallback() {

    public void onFailure(Exception exception) {
        // Implement exception handling code
    }

    public void onSuccess(JSONArray result) {
        // Called after request has finished successfully
    }
};

session.setCallback(callback);
userService.getGroupUsers(groupId);
```

Let's consider how this code asynchronously invokes the Liferay service.

It imports the `AsyncTaskCallback` callback class and the callback class `JSONArrayAsyncTaskCallback`, which is related to the service's `JSONArray` return type. Then it implements and instantiates the callback class. Lastly, it sets the callback on the session and calls the Liferay service.

The `onFailure()` method is called if an exception occurs during the request. This could be triggered by a connection exception (e.g., a request timeout) or a `ServerException`. If a `ServerException` occurs, it's because something went wrong on the server side. For instance, if you pass a `groupId` that doesn't exist, the portal complains about it, and the SDK wraps the error message with a `ServerException`.

There are multiple `AsyncTaskCallback` implementations, one for each method return type: `JSONObjectAsyncTaskCallback`, `JSONArrayAsyncTaskCallback`, `StringAsyncTaskCallback`, `BooleanAsyncTaskCallback`, `IntegerAsyncTaskCallback`, `LongAsyncTaskCallback`, and

`DoubleAsyncTaskCallback`. All you'll need to do is pick the appropriate implementation for your service method return type. In the example code snippet above, we used a `JSONArrayAsyncTaskCallback` instance since `getGroupUsers` returns a `JSONArray`.

It's also possible to use a generic `AsyncTaskCallback` implementation called `GenericAsyncTaskCallback`. For this implementation, you must implement a transform method and handle JSON parsing yourself.



Note: If you still don't want to use any of these callbacks, you can implement `AsyncTaskCallback` directly, but be careful, you should always get the first element of the `JSONArray` passed as a parameter to the `onPostExecute (JSONArray jsonArray)` method (i.e., `jsonArray.get(0)`).

After the request has finished, the `onSuccess ()` method is called on the main UI thread. Since the request is asynchronous, the service call immediately returns a `null` object. The service delivers the service's real return value to the callback's `onSuccess ()` method, instead.

Besides using synchronous and asynchronous requests, you can also send requests using batch processing. Let's learn about this next.

9.5.4. Sending Your Android App's Requests Using Batch Processing

The Mobile SDK also allows sending requests using batch processing, which can be much more efficient than sending separate requests. For example, suppose you want to delete ten blog entries at the same time; instead of making one request for each deletion, you can create a batch of calls and send them all together.

Here is a code snippet from an app that deletes blog entries synchronously as a batch:

```
import com.liferay.mobile.android.service.BatchSessionImpl;

BatchSessionImpl batch = new BatchSessionImpl(session);
BlogsEntryService service = new BlogsEntryService(batch);

service.deleteEntry(1);
service.deleteEntry(2);
service.deleteEntry(3);

JSONArray jsonArray = batch.invoke();
```

First, the `BatchSessionImpl` session is created. You can either pass credentials or pass another session to the constructor. Passing another session to the constructor is useful when you already have a `Session` object and want to reuse the same credentials. Then, it makes service calls as usual. With asynchronous calls, these methods return a `null` object immediately.

Finally, it calls the `invoke()` method from the batch session object. It returns a `JSONArray` containing the results for each service call. Since there are three `deleteEntry` calls, the `JSONArray` contains three objects. The order of the result matches the order of the service calls.

If you want to make batch calls asynchronously, set the callback as a `BatchAsyncTaskCallback` instance:

```
import com.liferay.mobile.android.task.callback.BatchAsyncTaskCallback;

batch.setCallback(new BatchAsyncTaskCallback() {

    public void onFailure(Exception exception) {
    }

    public void onSuccess(JSONArray results) {
        // The result is always a JSONArray
    }
});
```

It's just that easy to make efficient service calls in batch!

Next, let's dive into using the iOS SDK to access Liferay services.

9.6. Using the iOS SDK

You've just created a custom iOS app and now want your app to access Liferay services. How do you access Liferay services from an iOS mobile app? Use Liferay's iOS SDK, of course. If you'd like to invoke remote web services, you'll need to generate the client libraries. You can learn more about the SDK builder and how to generate client libraries by reading the Custom Portlet Building an SDK for Custom Portlet Services section.

Let's get started by downloading the iOS SDK and configuring it in your iOS environment!

9.6.1. Setting Up the iOS SDK

To install the iOS SDK to your machine, you'll need to download the latest version of `liferay-ios-sdk.zip`. You can download this file from the Liferay Mobile SDK project page. These installation instructions assume you're using the XCode developer tool provided by Apple, which can be downloaded from the Mac App Store.

After you've downloaded the Zip file, unzip it into your XCode project. Within XCode, right-click on your project and click *Add Files to*. Then, add both `core` and `v62` folders. The `v62` folder name can change for each Liferay version. In this example, the Mobile SDK is built for Liferay 6.2.

Awesome! Let's learn how to configure the SDK next.

9.6.2. Invoking Liferay Services in Your iOS App

For your mobile app to access the Mobile SDK, you'll need to complete several steps:

1. Create a session.
2. Import the Liferay services for your app to use.
3. Create a service object and call its services.

We'll demonstrate these steps by providing access to a sample Blogs app. Note that the following code snippets are written in the *Objective C* programming language. Let's begin!

9.6.2.1. Step 1: Create a session

```
#import "LRSession.h"

LRSession *session = [[LRSession alloc] init:@"http://localhost:8080"
username:@"test@liferay.com" password:@"test"];
```

To learn more about the session and its three parameters, refer to Setting Up the Android SDK.

9.6.2.2. Step 2: Import the Liferay services for your app to use

For this sample app, we'll import the `BlogsEntryService`.

```
#import "LRBlogsEntryService_v62.h"
```

Note the Liferay version (.v62) is used in the package namespace. Since the SDK is built for a specific Liferay version, service classes are separated by their package name. Our Mobile SDK classes use the .v62 package, which means this SDK is compatible with Liferay 6.2. However, you can use several SDKs simultaneously to support different Liferay versions.

Your portal's JSON web services page (e.g., <http://localhost:8080/api/jsonws>) lists all available portal services and plugin services.

9.6.2.3. Step 3: Create a service object and call its services

For this sample app, we'll create an `LRBlogsEntryService_v62` object and make a service call that fetches all blog entries from the *Guest* site. In this example, the `groupId` is 10184.

```
LRBlogsEntryService_v62 *service = [[LRBlogsEntryService_v62 alloc]
init:session];
```

```
NSError *error;
NSArray *entries = [service getGroupEntriesWithGroupId:1084 status:0 start:-1
end:-1 error:&error];
```

This is a basic example of a synchronous service call, which means the method returns only after the request is finished.

Service method return types can be `void`, `NSString`, `NSArray`, `NSDictionary`, `NSNumber`, and `BOOL`.



Note: Many service methods require a group ID as a parameter. You can call `[LRGroupService_v62 getUserSitesGroups:&error]`, which uses Liferay Portal's `GroupService.getUserSitesGroups`, to get a site group.

That's it! You've given the blogs app access to the `BlogsEntryService`. Next, let's discuss

making asynchronous HTTP requests.

9.6.3. Invoking Services Asynchronously from Your iOS App

The SDK allows asynchronous HTTP requests; all you need to do is set a callback to the session object. You can set the callback to `nil` if you want to make synchronous requests again.

Let's continue our example with the blogs app. To configure asynchronous requests, the first thing we need to do is create a class that conforms to the `LRCallback` protocol.

```
#import "LRCallback.h"

@interface BlogsEntriesCallback : NSObject <LRCallback>

@end

#import "BlogsEntriesCallback.h"

@implementation BlogsEntriesCallback

- (void)onFailure:(NSError *)error {
    // Implement error handling code
}

- (void)onSuccess:(id)result {
    // Called after request has finished successfully
}

@end
```

Then set this callback to the session and call your service as usual:

```
BlogsEntriesCallback *callback = [[BlogsEntriesCallback alloc] init];

[session setCallback:callback];
[service getGroupEntriesWithGroupId:1084 status:0 start:-1 end:-1
error:&error];
```

If a server side exception or a connection error occurs during the request, the `onFailure` method is called with an `NSError` that contains information about the error.

Since the request is asynchronous, the `getGroupEntriesWithGroupId` method returns immediately with `nil`, and the `onSuccess` method of your callback is invoked with the results once the request has finished successfully.

The `onSuccess` result parameter doesn't have a specific type. Therefore, you need to check the service method signature in order to figure out which type you can cast it to safely. In this example, the `getGroupEntriesWithGroupId` method returns an `NSArray`; so you can cast to this type.

```
- (void)onSuccess:(id)result {
    NSArray *entries = (NSArray *)result;
}
```

The `onSuccess` method is called on the main UI thread after the request has finished.

Let's talk about another way to send your app's requests: batch processing.

9.6.4. Sending Your iOS App's Requests Using Batch Processing

Another popular method of sending requests to the Mobile SDK is through batch processing, which can be more efficient than sending requests separately.

For example, suppose you want to delete 10 blog entries at the same time; instead of making one request for each delete call, you can create a batch of delete calls and send them together. Here's an example of how to do this:

```
#import "LRBatchSession.h"

LRBatchSession *batch = [[LRBatchSession alloc] init:@"http://localhost:8080"
username:@"test@liferay.com" password:@"test"];
LRBlogsEntryService_v62 *service = [[LRBlogsEntryService_v62 alloc]
init:batch];
NSError *error;

[service deleteEntryWithEntryId:1 error:&error];
[service deleteEntryWithEntryId:2 error:&error];
[service deleteEntryWithEntryId:3 error:&error];

NSArray *entries = [batch invoke:&error];
```

First, create an `LRBatchSession` session. You can either pass credentials or pass another session to the constructor. This is useful when you already have a `Session` object and want to reuse the same credentials. Then make the service calls as usual. With asynchronous calls, these methods return `nil` right away.

Finally, call `[batch invoke:&error]`, which returns an `NSArray` containing the results for each service call. Since there are three `deleteEntryWithEntryId` calls, the `entries` array contains three objects. The order of the results matches the order of the service calls.

If you want to make batch calls asynchronously, set the callback to the session.

```
[batch setCallback:callback];
```

The return type for batch calls is always an `NSArray`.

As you can see, it's very simple to make efficient service calls in batch using the iOS SDK!

9.7. Summary

In this chapter, we showed you how easy it is to download and configure Liferay's Mobile SDK. First, we explored Eclipse's integration with the Mobile SDK. Eclipse offers ways to build standard and custom JARs for an Android app to reference. Eclipse also gives you the option to create a sample Liferay Android project to familiarize yourself with a Liferay Android app using the Mobile SDK. Then, we explained the Android and iOS mobile platforms separately, giving you step-by-step examples for accessing Liferay services.

Did you know that there is a breed of social applications called OpenSocial gadgets that lend themselves well to humans sharing information and application functionality within defined

networks? They are light-weight and easy to write and distribute. We'll get into OpenSocial gadgets next.

10. Creating and Integrating with OpenSocial Gadgets

OpenSocial is a public specification for creating web applications using standard technologies like HTML, CSS, and JavaScript. It was originally developed by Google, Myspace, and others to standardize common social networking API's but has evolved into a general platform for building web applications. Whereas "standard" applications work with data on a per-user basis, "social" applications share data within well defined networks, facilitating communication of information between groups of users. OpenSocial applications, called *gadgets*, are similar to portlets because they can be added to your portal's pages and used for all kinds of tasks. Gadgets are characterized as being simple, widely available, and easy to deploy.

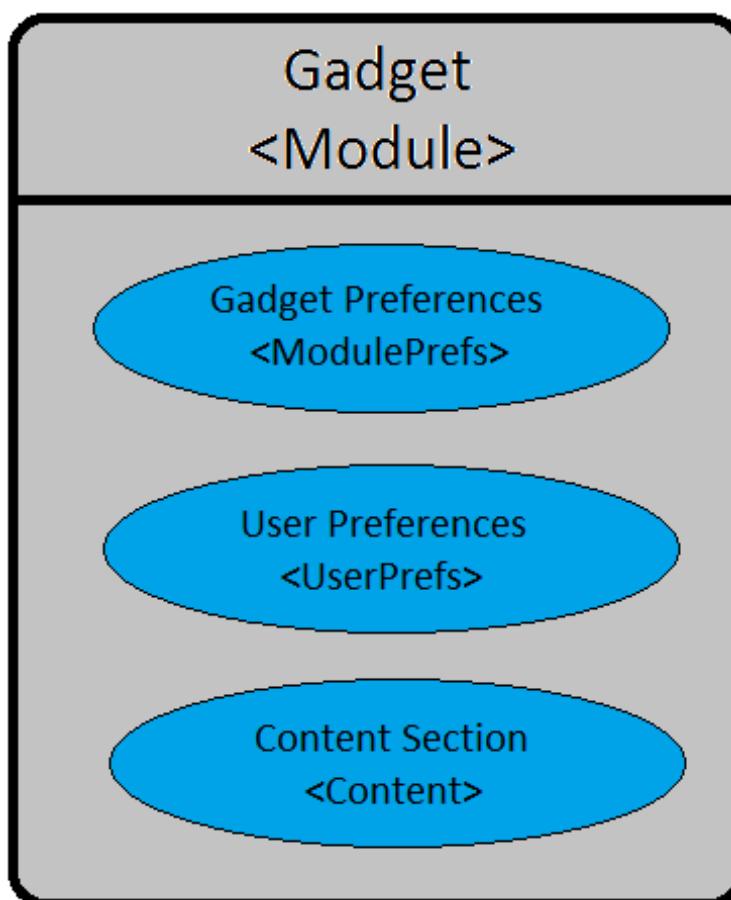
In this chapter, we'll cover the following topics:

- OpenSocial Gadget Basics
- Accessing Third-Party Applications from Your Gadget
- Gadget/Portlet Communication with PubSub
- Liferay's Gadget Editing Environment

Let's first look at the structure of an OpenSocial gadget and consider the concepts involved with OpenSocial gadgets.

10.1. OpenSocial Gadget Basics

An OpenSocial Gadget is specified in an XML document that has two parts. The first part of the document specifies *meta-data* declaring gadget dependencies, defining characteristics about the gadget, and specifying user preferences. The meta-data can be broken up into *module preferences* and *user preferences*. The second part of the document contains the gadget *content*, defining the gadget's user interface and business logic. We'll spend time going over all these facets of gadget XML.



Below is an example OpenSocial gadget XML file for a map gadget:

```
<?xml version="1.0"  
encoding="UTF-8" ?>  
<Module>  
    <ModulePrefs  
        title="Location Map"  
        height="300"  
        author="Cody Hoag"  
        author_email="cody.hoag@liferay.com" />  
        <UserPref name="lat"  
        display_name="Latitude"  
        required="true" />
```

```

<UserPref name="lng" display_name="Longitude" required="true" />
<Content type="html">
<![CDATA[
<script
src="https://maps.googleapis.com/maps/api/js?v=3.exp&sensor=false"
type="text/javascript">
</script>
<div id="map" style="width: 100%; height: 100%;"></div>
<script type="text/javascript">

var includeMarker = false;
var latlng;
var userPrefs = new gadgets.Prefs();
var latLocation = userPrefs.getFloat('lat');
var lngLocation = userPrefs.getFloat('lng');

if (latLocation == "" && lngLocation == "") {
    latlng = new google.maps.LatLng(33.997547,-117.814305);
}
else {
    latlng = new google.maps.LatLng(latLocation,lngLocation);
    includeMarker=true;
}

function initialize() {
    var mapOptions = {
        center: latlng,
        mapTypeId: google.maps.MapTypeId.ROADMAP,
        zoom: 7
    }
    map = new google.maps.Map(document.getElementById('map'),
mapOptions);

    if (includeMarker == true) {
        var marker = new google.maps.Marker({
            map: map,
            position: latlng
        });
    }
}

google.maps.event.addDomListener(window, 'load', initialize);

</script>
]]>
</Content>
</Module>

```

For the official documentation on Gadget anatomy, see Google's Anatomy of a Gadget. For convenience, however, let's consider gadget *meta-data* and *content* here to highlight their important aspects.

10.1.1. Gadget Metadata

Gadget meta-data specifies characteristics of the overall gadget and specifies controls for the user to customize the gadget. Here is a simple breakdown of these two types of meta-data:

Gadget Preferences (required): Define characteristics of the gadget. They are specified as attributes of and nested elements in the gadget's `<ModulePrefs>` element.

User Preferences (optional): Store user configuration values and allow users to give input to the application. They are specified in `<UserPref>` elements.

Now, let's dive deeper into gadget preferences.

10.1.1.1. Gadget Preferences (`ModulePrefs`)

The gadget preferences (`ModulePrefs`) define characteristics of your gadget, including features that it uses, how it authenticates content, how it is displayed to the user, and how it is displayed in a gadget directory. The preferences are wrapped in

`<ModulePrefs>...</ModulePrefs>` tags.

Within the `ModulePrefs`, you specify the gadget's dependencies with `<Require>` tags and optional features with `<Optional>` tags. If you specify a feature in a `<Require>` tag, the gadget only renders itself if the feature is available. If you specify a feature in an `<Optional>` tag, the gadget renders despite feature availability.

`ModulePrefs` can also be used to configure important features such as PubSub and OAuth. The PubSub feature allows for communication between gadgets on the same page: they can publish and subscribe on message channels. Liferay also enables gadgets to communicate with portlets using the same mechanism. The OAuth feature provides secure connections between your portal and third-party sites. These are just a couple of useful features that can be defined in the `ModulePrefs`. You'll learn more about them later in this chapter.

Overall, this `<ModulePrefs>` element allows you to register your gadget on a gadget directory and establish settings and tools to use in your gadget's business logic. For complete details on `ModulePrefs`, see Google's `Moduleprefs Elements and Attributes` reference.

Next, let's consider gadget user preferences.

10.1.1.2. User Preferences (`UserPrefs`)

When a gadget is configured to fit a specific page, it can really enhance the user's experience. OpenSocial gadgets are capable of storing user preferences, allowing gadgets to present information and options tailored for individual pages. The title `UserPrefs` can also be referred to as application preferences, since the preferences being changed only apply to the single gadget and not to each user that uses it. A gadget's user preferences are specified in

`<UserPref>...</UserPref>` tags in the gadget's XML. The owner of a page can adjust permissions to allow certain users to change the user preferences. However, every time the `UserPrefs` are modified, every user that views the gadget has a modified gadget. As a user enters information and adjusts controls via a gadget's user interface, those changes are persisted in the database for that gadget instance. Each time the gadget is reloaded, the `UserPrefs` are extracted

from the database, fed back to the gadget, and rendered in the user interface.

Map gadgets, for example, are prime candidates for UserPrefs. When users first bring up map gadgets, they are typically interested in their current location or a location of interest--not some random distant land. Therefore, it makes sense for a map gadget to take the user's location of interest as input. UserPrefs facilitate taking in this information, storing it, and processing it to present gadget user interfaces customized to the user. Here are the UserPrefs for our current map gadget:

```
<UserPref name="lat" display_name="Latitude" required="true" />
<UserPref name="lng" display_name="Longitude" required="true" />
```

Notice the `lat` and `lng` UserPrefs of the map gadget. These user preferences take in the user's latitude and longitude preferences, stores them, and displays them on the mapping interface.

Lastly, we'll look at what a user sees in the gadget's user interface when setting the *Latitude* and *Longitude* user preferences.

Here is a snapshot of what this window looks like on Liferay Portal:

The screenshot shows a configuration window titled "Map of Location - Configuration". It has three tabs: "Setup" (selected), "Permissions", and "Sharing". On the right side, there is a "Archive/Restore Setup" button. The main area contains two text input fields: "Latitude" and "Longitude", each with a corresponding empty input box below it. At the bottom left is a blue "Save" button.

UserPrefs can be displayed in many different ways and help your gadget

become more adaptable to your users' ever changing expectations. And, there is an extensive variety of UserPref data types available to properly personalize your gadgets. For complete details on UserPrefs see Google's User preferences reference.

10.1.2. Gadget Content

Within the `<Content>...</Content>` tags of your gadget, you define your gadget's user interface and business logic. The content can be implemented in HTML or linked via a URL. HTML allows you to use JavaScript and gives you plenty of flexibility. But if you simply want to leverage content found at a particular URL, use the URL content type.

To help you decide which content type is best for you, see Google's Choosing a Content Type documentation. But for convenience, let's consider basic aspects of these content types here.

10.1.2.1. HTML Content Type

HTML is the default content type for OpenSocial gadgets. Because HTML is so flexible and commonly used for gadgets, you may find yourself using it often.

Note that all HTML content must be specified in a CDATA section within your gadget's content.

The following simple content example demonstrates proper CDATA specification:

```
<Content type="html">
  <![CDATA[ "Your HTML goes here" ]]>
</Content>
```

Also, it is forbidden to use `<body>`, `<head>`, or `<html>` tags in your HTML content, as these tags conflict with the ones automatically generated by the container for your gadget.

10.1.2.2. URL Content Type

This content type is convenient when you just want to reference content within an existing URL, and nothing more. When using this content type, specify the URL as your `href` attribute value. The gadget assumes all programming and user interface logic resides in your specified URL. Therefore, when using the `url` content type, you do not need any HTML or JavaScript. Note that the URL content type has better consistency in the specification than the HTML content type regarding proxied content.

Here is a sample of what the URL content type looks like for an example gadget:

```
<Content type="url" href="http://www/cgi-bin/example/gadgets/mystats.cgi" />
```

Both HTML and URL content types offer beneficial traits and can be used effectively. The content type you use for your gadgets depends on your needs and preferences.

Now that you're fundamentally sound on the gadget basics, let's explore what gadgets offer you in Liferay Portal.

10.2. Accessing Third-Party Applications from Your Gadget

Is there data on sites like Evernote, Facebook, Google, Netflix, Photobucket, Twitter, or Yahoo you'd like to access in your OpenSocial gadgets? Perhaps you'd like to provide a gadget for portal users to add movies to their Netflix queue or for users to display their Photobucket pictures within gadgets in Liferay Portal. You may be concerned that users would have to share their third-party application credentials with Liferay portal in order to use the applications. Good news! *OAuth* technology resolves the issue.

You can think of OAuth as a "handshake mechanism" where, instead of requiring the exchange of personal information, your site redirects portal users directly to the service provider (e.g. Netflix, Photobucket, etc). Users approve the gadget's access to their resources on the external web applications. It's just that easy!

Read the Gadget personalization section of the *Social Networking* chapter in *Using Liferay Portal 6.2* for instructions on configuring and using OAuth enabled gadgets. To learn how to write OAuth enabled gadgets, see Google's Writing OAuth Gadgets reference.

Let's now shift our focus to gadget/portlet communication with PubSub.

10.3. Gadget/Portlet Communication with PubSub

Have you ever wanted your gadgets to communicate with each other or with portlets? You can do so with *PubSub*. It is a messaging pattern in which publishers send messages to topics and subscribers receive the messages on those topics. Hence, the term PubSub is short for "Publish and Subscribe." PubSub is implemented in Liferay to facilitate interaction between gadgets and interaction between gadgets and portlets. PubSub is a diverse messaging system that allows messages to be sent in the following manner:

- gadget to gadget
- portlet to gadget or vice versa
- portlet to portlet

With the use of PubSub, the worlds of gadgets and portlets blend together, facilitating their ability to communicate and interact with each other.

Publishers don't send messages to subscribers directly; instead, they publish messages that are characterized into classes and sent across message channels (topics). Therefore, the publishers have no knowledge of *whom* they are sending their messages to; they simply broadcast the messages over a channel. So you may ask: "How do subscribers receive messages if the messages are not sent directly to them?" Subscribers express interest by subscribing to certain channels. Once subscribed, they receive messages sent to those channels. Like publishers, they have no knowledge of *who* sends the messages, they only know channels and receive messages coming in on those channels. This process makes it much easier for multiple gadgets and portlets to communicate with each other, without specifically stating with whom they are communicating.

PubSub can be compared to a TV station and your TV. The TV station does not directly send their material to your TV, but instead broadcasts the material over a channel. You subscribe to that channel by tuning your TV to that channel to watch what's on it. This process of broadcasting and tuning into TV programs is similar to publishing and subscribing to messages sent via PubSub.

PubSub also offers the ability for portlets and gadgets to publish to and subscribe to multiple channels. Although this may seem like a complex process, PubSub's use of message channels creates a network for gadgets to communicate across, and the network is easy to maintain and understand.

Let's try out PubSub for ourselves to explore how PubSub works. In this fundamental example, we'll import two simple gadgets and send messages from one to the other. One acts as a publisher and the other as a subscriber. Follow the steps below:

1. Navigate to the Control Panel and select *OpenSocial Gadget Publisher* from under the *Apps* heading.
2. Select *Publish Gadget* and, for each gadget, enter the URL and click *Save*.

Sample PubSub Publisher URL:

`http://svn.apache.org/repos/asf/shindig/trunk/content/gadgets/sample-pubsub-2-publisher.xml`

Sample PubSub Subscriber URL:

<http://svn.apache.org/repos/asf/shindig/trunk/content/gadgets/sample-pubsub-2-subscriber.xml>

3. Go back to a page on your site, navigate to *Add → Applications*, and add your new gadgets to the page
4. Click *Publish a random number* on the PubSub Publisher gadget. Notice that it publishes a number; but the PubSub Subscriber gadget does not receive the number.

The figure consists of four screenshots of Liferay portlets arranged in a 2x2 grid. The top row shows the 'Sample PubSub Publisher' portlet, which contains a button labeled 'Publish a random number' and a text area showing the value '0.26539419658342445'. The bottom row shows the 'Sample PubSub Subscriber' portlet, which has two buttons: 'Subscribe' (highlighted in red) and 'Unsubscribe'. In the first screenshot of the subscriber, the 'Subscribe' button is highlighted. In the second screenshot, the 'Unsubscribe' button is highlighted. In the third screenshot, the 'Subscribe' button is highlighted again. The fourth screenshot shows the subscriber portlet with a message box containing 'message : 0.06380942232851727' and 'received at: Tue Oct 15 2013 19:24:50 GMT-0400 (Eastern Standard Time)'.

5. Select *Subscribe* on the subscriber gadget.

6. Click *Publish a random number* again from the publisher gadget. You now see the random number received by the subscriber.

7. Select *Unsubscribe* on the subscriber gadget.

8. Click *Publish a random number*.

As you would expect, the subscriber portlet no longer receives the random number.

This simple example illustrates what PubSub does. It opens up a message channel through which publishers and subscribers can interact. You probably also noticed that while publishing and subscribing with your gadgets, there was no need to refresh the page. This is because the gadgets use *AJAX (Asynchronous JavaScript and XML)*. This technology refreshes your applications automatically, allowing PubSub to work efficiently and effectively throughout your Liferay Portal.

I bet you are wondering how to implement PubSub messaging in your gadgets and portlets. We'll dive into the code next. In fact, we'll complete exercises demonstrating gadget to gadget interaction and portlet to gadget interaction.

10.3.1. Gadget to Gadget

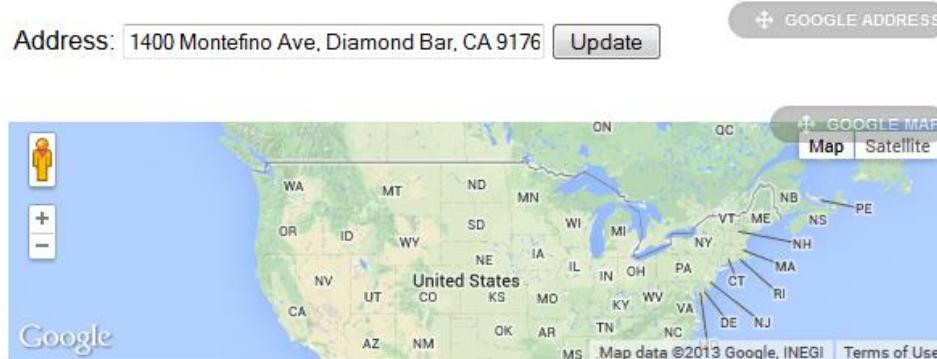
For gadget to gadget communication, two independent gadgets are placed on a page and configured with PubSub. These two gadgets are able to communicate with one another and provide tools that the user could not otherwise produce. We will complete a simple example for gadget to gadget communication where two gadgets work together to display an address on Google Maps. The first gadget represents a publisher that enables the user to input a specific address and publish the address. The second gadget represents the subscriber who receives the address, displays the address, and locates the address on Google Maps. Follow the steps below:

1. Publish the *Google Address* and *Google Map* gadgets, as we had done previously with other gadgets. The URLs are given below:

Google Address URL: <https://raw.github.com/dejuknow/opensocial-gadgets/master/GoogleMaps/GoogleMapsPublisher.xml>

Google Map URL: <https://raw.github.com/dejuknow/opensocial-gadgets/master/GoogleMaps/GoogleMapsViewer.xml>

2. Add both gadgets to a page.



Note: The address bar is already filled with an address. This default address is specified in the *Google Address* gadget's source code. We will edit this setting using Liferay's gadget editor later in the chapter.

3. Input an address into the *Address* bar and then click *Update*. You should now be able to see that address location displayed in the *Google Map* gadget.



Congratulations! Your gadgets are communicating well with each other.

This simple example demonstrates two gadgets communicating with each other using PubSub.



Note: Gadget-to-gadget communication using the publish-subscribe framework has been deprecated for Google gadgets:
<https://developers.google.com/gadgets/docs/pubsub>. However, PubSub is still the primary communication for gadgets and will be supported with gadgets on Liferay.

We will now dive into the source code and analyze how this interaction is accomplished. First, we'll look at the contents of the *Google Maps* XML file *GoogleMapsPublisher.xml*:

```
<?xml version="1.0" encoding="UTF-8" ?>

<Module>
    <ModulePrefs title="Google Address">
        <Require feature="pubsub-2">
            <Param name="topics">
                <![CDATA[
                    <Topic
                        title="Google Maps"
                        name="com.liferay.opensocial.gmapsdemo" publish="true"
                    />
                ]]>
            </Param>
        </Require>
        <Require feature="dynamic-height" />
    </ModulePrefs>

    <Content type="html">
        <![CDATA[
            <table>
                <tr>
                    <td>Address:</td>
                    <td>
                        <input
                            type="text"
                            id="address"
                            name="address"
                            size="40"
                            value="1400 Montefino Ave, Diamond Bar, CA 91765"
                        >
                    </td>
                    <td>
                        <input
                            type="button"
                            value="Update"
                            onclick="updateLoc()"
                        >
                    </td>
                </tr>
            </table>
        </![CDATA]>
    </Content>
</Module>
```

```

<script type="text/javascript">
    function updateLoc() {
        var address = document.getElementById("address").value;
        gadgets.Hub.publish(
            "com.liferay.opensocial.gmapsdemo", address);
    }

    gadgets.window.adjustHeight();
</script>
]]>
</Content>
</Module>

```

The following excerpt from *Google Maps* XML file enables the gadget to use PubSub and specifies the channel (topic) to which the gadget publishes:

```

<Require feature="pubsub-2">
    <Param name="topics">
        <![CDATA[
            <Topic
                title="Google Maps"
                name="com.liferay.opensocial.gmapsdemo"
                publish="true"
            />
        ]]>
    </Param>
</Require>

```

Notice the opening `<Require feature="pubsub-2">`, mandating the pubsub-2 feature for the gadget. The `<Param name="topics">` section establishes the topic `com.liferay.opensocial.gmapsdemo` that the gadget publishes to. Within the *topics* parameter, you define all the topics your gadget uses for communication. Furthermore, the `publish="true"` attribute specifies the gadget's role as a *publisher* to the topic.

In the JavaScript of the gadget's *content*, the gadget publishes to the topic with the following invocation:

```
gadgets.Hub.publish("com.liferay.opensocial.gmapsdemo", address);
```

When the user clicks the gadget's *Publish* button, the message is sent by the publishing gadget to the topic named `com.liferay.opensocial.gmapsdemo`. PubSub broadcasts the message received on that topic to all subscribers, such as your *Google Address* gadget. Each subscriber receives and processes the message. In the case of your *Google Maps* gadget, the message, containing an address, is processed by the gadget to show the address location on its map.

Next, we'll analyze the example's *subscribing* gadget's source code specified in `GoogleMapsViewer.xml`:

```

<?xml version="1.0" encoding="UTF-8" ?>

<Module>
    <ModulePrefs title="Google Map">
        <Require feature="pubsub-2">
            <Param name="topics">
                <![CDATA[

```

```

<Topic
    title="Google Maps"
    name="com.liferay.opensocial.gmapsdemo"
    subscribe="true"
/>
[]]
</Param>
</Require>
<Require feature="dynamic-height" />
</ModulePrefs>

<Content type="html">
<![CDATA[
<script
    src="https://maps.googleapis.com/maps/api/js?sensor=false"
    type="text/javascript">
</script>

<div id="map" style="width:100%;height:100%"></div>

<script type="text/javascript">
    gadgets.HubSettings.onConnect = function(hub, suc, err) {
        gadgets.Hub.subscribe("com.liferay.opensocial.gmapsdemo",
callback);
    }

    function callback(topic, data, subscriberData) {
        geocoder.geocode( { 'address': data }, showAddress);
    }

    function showAddress(results, status) {
        if (status == google.maps.GeocoderStatus.OK) {
            while(overlays[0]) {
                overlays.pop().setMap(null);
            }
            map.setCenter(results[0].geometry.location);

            var marker = new google.maps.Marker(
            {
                map: map,
                position: results[0].geometry.location
            }
            );

            overlays.push(marker);
        }
        else {
            alert('Failed to locate address. Reason: ' + status);
        }
        map.setZoom(12);
    }
}

```

```

var overlays = [];

var geocoder = new google.maps.Geocoder();

var mapOptions = {
    center: new google.maps.LatLng(43, -100),
    zoom: 3,
    mapTypeId: google.maps.MapTypeId.ROADMAP
};

var map = new google.maps.Map(document.getElementById("map"),
mapOptions);

        gadgets.window.adjustHeight();
</script>
]]>
</Content>
</Module>
```

The subscriber source code is similar to that of the publisher. It mandates the pubsub-2 feature for the gadget and specifies `com.liferay.opensocial.gmapsdemo` as one of its topics, as you would expect. Of course, the only difference is the fact that this gadget subscribes to the topic, hence the attribute setting `subscribe="true"`.

The following JavaScript from the gadget's *content* registers a callback on the `com.liferay.opensocial.gmapsdemo` topic:

```

gadgets.HubSettings.onConnect = function(hub, suc, err) {
    gadgets.Hub.subscribe("com.liferay.opensocial.gmapsdemo", callback);
}
```

The `gadgets.HubSettings.onConnect` function is called by the OpenSocial container once the gadget connects to the PubSub messaging hub. In our example, the gadget subscribes to the previously mentioned topic. All subscribers to this topic receive messages sent to it.

When a message is received, the gadget's `callback()` function is executed. In this example, the `callback` method sends the received message (the address sent by the publisher) and calls `google.maps.Geocoder.geocode()` to get the locations. And finally, the locations are processed and displayed on the map.

In summary, subscriber gadgets need to specify a topic and register a callback function on that topic to handle the messages they receive.

As you can see, PubSub allows your site to run efficiently and enables otherwise unconnected gadgets to communicate and flourish within Liferay Portal.

Gadgets are not limited to only communicating with other gadgets. In the next section, we will demonstrate the capabilities of communication between portlets and gadgets.

10.3.2. Communicating Between Portlets and Gadgets

For this section, we will continue implementing the *Google Map* gadget on your site. If you have portlet applications that can take advantage of the functionality your Maps gadgets have to offer, it would certainly be convenient for a user to allow communications between those gadgets and

your portlets.



Note: The publish-subscribe framework has been deprecated for Google gadgets: <https://developers.google.com/gadgets/docs/publish>. However, PubSub is still a primary communication mechanism between portlets and gadgets and will be supported on Liferay.

As a demonstration, we will send messages from a custom *Directory* portlet to the *Google Map* gadget. For each user listed in the *Directory* portlet, we will create a *Show in Google Maps* link that, when selected, displays the location of the user's address in the *Google Map* gadget.

This portlet is much like Liferay's *Portal Directory* portlet. The first thing we need to do is edit the portlet's `address.jsp` file to configure a Google Maps link.

As you know from explanations given earlier, the *Google Map* gadget is distinguished as a subscriber. Therefore, the *Directory* portlet needs to take on a publisher role to enable communication. To enable the *Directory* portlet to publish to the topic on which the *Google Map* gadget is subscribed, we insert the following JavaScript into the `address.jsp` file:

```
<script type="text/javascript">
    function publishAddress(address) {
        Liferay.fire('gadget:com.liferay.opensocial.gmapsdemo', address);
    }
</script>
```

This code involves a `publishAddress()` function that's called whenever you click on the *Show in Google Maps* link. The function invokes the `Liferay.fire()` function, passing in the name of the channel receiving the message and the user's address as the message. One thing to note is that when a portlet sends data to a gadget, there must be a `gadget:` prefix before the channel declaration. This distinguishes who the messages are intended for when they are broadcast across a channel. Notice that you don't need to change anything for your *Google Map* gadget, since it's already subscribed to that channel. You only needed to define the *Directory* portlet as a publisher to that channel.



Note: If you would like to broadcast messages to portlets, follow the same guidelines, but don't use the `gadget:` prefix in your topic parameter value for the call to the `Liferay.fire()` function.

After editing the JSP, you can add the *Directory* Portlet and *Google Map* gadget to a Liferay page and test it out. Here is a snapshot of what the interaction would look like:

The screenshot shows a Liferay Portal page with a sidebar titled "Portal Directory". In the main content area, there is a "Test Test" portlet containing a user profile picture and personal information: Email Address (test@liferay.com), Birthday (1/1/70), and Gender (Male). To the right, there is a "Address" portlet under "Personal Address" with a checked "Business" option, address (1400 Montefino Ave, 91765, Diamond Bar), and a "Show in Google Maps" button. Below these is a "Google Map" portlet displaying a map of Diamond Bar, California, with the address marked. The map includes labels for Avocado Heights, La Puente, Walnut, Chino, and Chino Hills.

Letting your portlets communicate with gadgets enhances your portlet applications and gives you a plethora of different ways you can enhance your users' experience.

nce.

We will now switch gears and dive into Liferay's gadget editor!

10.4. Using the Gadget Editor

As part of Liferay's OpenSocial integration, the *OpenSocial Gadget Editor* is included with Liferay Portal. The gadget editor is a complete development environment for gadgets, providing syntax highlighting, preview functionality, undo/redo options, and built in tabs for working on multiple gadgets at the same time. You can also organize and manage gadgets easily using the editor's file manager. All of this gives you the convenience of creating and improving your gadgets right from within Liferay Portal.

Within the editor, each gadget's XML file has a drop-down menu allowing you to close, rename, delete, publish, or unpublish the gadget, or to simply show the gadget's URL. The *Publish* button directs you to a screen, similar to the *OpenSocial Gadget Publisher*, allowing you to publish your gadget. Gadgets published through the editor are stored in the site's Document and Media Library. The *Show URL* button gives you the URL so that the gadget may be shared with other sites. These options offer a user-friendly and easy to use testing station for enhancing the gadgets on your sites.

The screenshot shows the Liferay Gadget Editor interface. On the left, there is a file tree with "OpenSocial Gadgets" expanded, showing "GoogleMapsPublisher.xml". A red box highlights the dropdown menu next to "GoogleMapsPublisher.xml". The menu options are: Close, Rename, Delete, Publish, Unpublish, Show URL, and a "Module" option. The "GoogleMapsPublisher.xml" file is open in the main editor area, showing XML code:

```

<Module>
  <ModulePrefs title="Google Ac
    ure feature="pubsub-2"
    param name="topics">
    <![CDATA[
      <Topic title="Google Ma
      ]>
    <Param
      require>
    <ModulePrefs>
      <Require feature="dynamic-t
    </ModulePrefs>
  </Module>

```

Note: When you publish a new gadget, remember that your Liferay Portal installation is the new host when specifying the gadget's URL.

For a brief exercise, we will improve the *Google Address* gadget using the gadget editor. As we referenced earlier, the *Google Address* gadget automatically displays the address *1400 Montefino Avenue, Diamond Bar, CA 91765* in its text window. By using the OpenSocial Gadget Editor, you can edit the XML file and specify a customized address or remove the default address entirely from the gadget's UI. For our example, we will remove the default address entirely from our gadget's text window.

1. Copy the gadget XML contents into the gadget editor. Navigate to *OpenSocial Gadget Publisher* from under the *Apps* heading and select the URL for *Google Address*.
2. Copy the XML content onto your clipboard.
3. Navigate to the *OpenSocial Gadget Editor* from the *Site Administration → Content* tab and paste your clipboard contents into the gadget editor.
4. Click the floppy disk button to save your new gadget XML, naming your gadget *GoogleMapsPublisher.xml*. Press the check button to save the file.

OpenSocial Gadget Editor

The screenshot shows the Liferay OpenSocial Gadget Editor interface. At the top, there is a toolbar with various icons, one of which is highlighted with a red box. Below the toolbar is a navigation bar showing a folder icon and the path "OpenSocial Gadgets / GoogleMapsPublisher.xml". The main area contains an XML code editor with several lines of XML. One line, specifically the address element, is also highlighted with a red box. To the right of the XML editor is a "Preview" window titled "Preview" which displays the text "Iress: 1400 Montefino Ave, Diamond Bar, CA 9176!".

Note: Liferay will not allow you to publish your new gadget without attaching .xml to the end of your file name.

5. Select the *Preview* tab from the toolbar and a preview of your gadget appears.

6. Close the Preview window and locate the following element in the gadget XML:

```
<input type="text" id="address"  
name="address" size="40" value="1400  
Montefino Avenue, Diamond Bar, CA 91765">
```

Then, remove the `value="1400 Montefino Avenue, Diamond Bar, CA 91765"` setting and save the file.

7. Select the editor's *Preview* mode again. As you would expect, the gadget's default address is now blank.
8. Publish your gadget for portal-wide use by selecting the drop-down menu next to the *GoogleMapsPublisher.xml* file and clicking *Publish*. A *Publish Gadget* screen opens up, showing your gadget's URL and categories to select for your gadget.

Publish Gadget

New Gadget

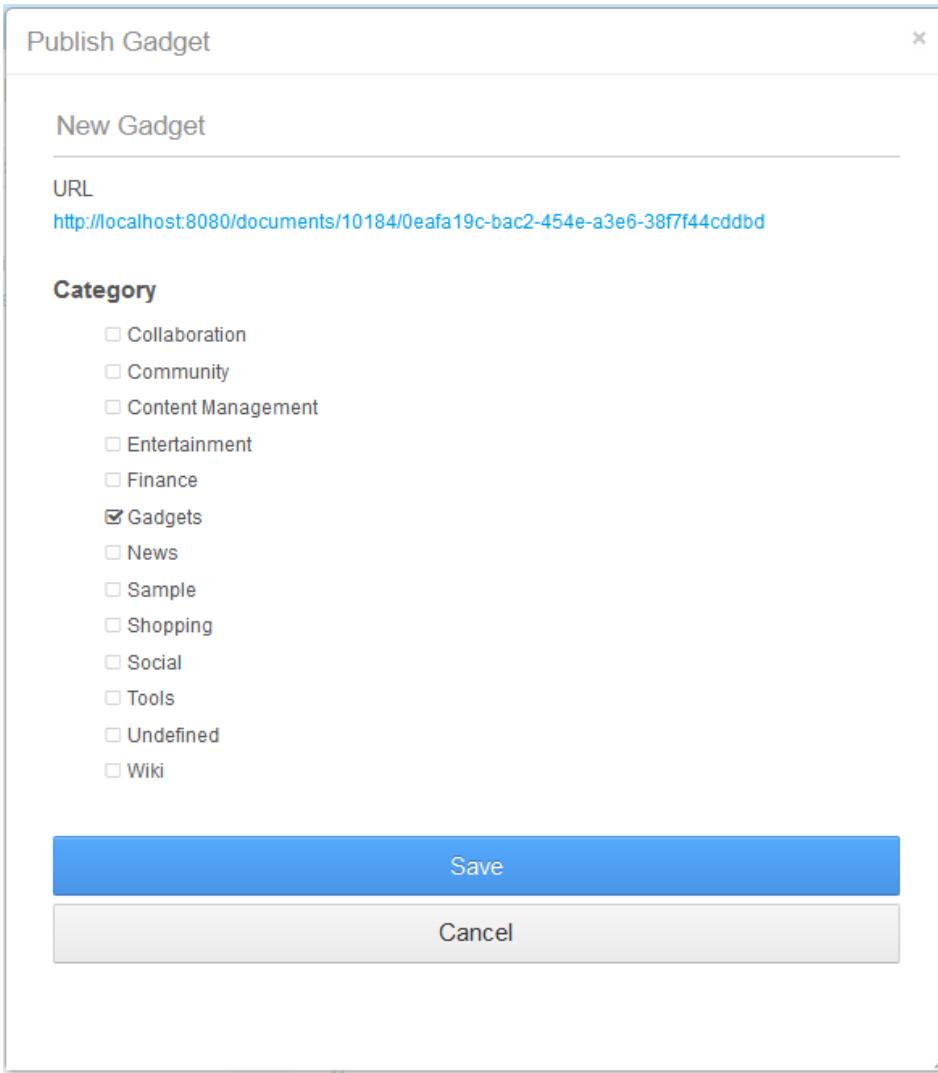
URL
<http://localhost:8080/documents/10184/0eafa19c-bac2-454e-a3e6-38f7f44cddbd>

Category

- Collaboration
- Community
- Content Management
- Entertainment
- Finance
- Gadgets
- News
- Sample
- Shopping
- Social
- Tools
- Undefined
- Wiki

Save

Cancel



9. Choose a category and click **Save**.

Your new gadget is now available for portal-wide use!

The OpenSocial Gadget Editor allows you to create and improve gadgets within the comfort of your own Liferay Portal instance.

You can facilitate the social interactions on your sites and increase your portal content's popularity across your social network by leveraging the power of OpenSocial gadgets.

10.5. Summary

OpenSocial gadgets

offer a plethora of new features to Liferay that present new opportunities for your portal customization. You've learned the anatomy of a gadget, how to access third-party applications from a gadget, and gadget communication with PubSub. Also, you learned that Liferay's gadget editor makes editing and customizing your gadgets easier than ever. Take a deep breath and give yourself a pat on the back; you're now a trained gadget guru!

What better way to keep things rolling than to focus on your portal's overall look and feel using Liferay Themes! While we're at it, let's learn how to develop Layout Templates too. We'll cover both of these topics next.

11. Creating Liferay Themes and Layout Templates

Do you want to transform the look and feel of your Liferay Portal? Create your own Liferay Theme! Do you want to arrange your pages' portlets differently than what Liferay's templates support out-of-the-box? Create your own Layout Template! In this chapter, we'll show you how

to do both.

We'll go over the following topics in this chapter:

- Creating Liferay Themes
- Using Developer Mode with Themes
- Creating a Theme Thumbnail
- Designing a Look and Feel
- Understanding Your Theme's JavaScript Callbacks in `main.js`
- Importing Resources with Your Themes
- Creating Liferay Layout Templates
- Embedding Portlets in a Layout Template
- Variables Available to a Layout Template

Let's get started creating Liferay themes.

11.1. ***Creating Liferay Themes***

Themes are hot deployable plugins unique to a site served by the portal. With themes, you can alter the user interface so completely that it's difficult or impossible to tell that the site is running on Liferay.

Liferay provides a well organized, modular structure to its themes. Themes follow the same philosophy as Liferay configuration: they are modifications, or differences from the default. Because of this, every line of markup and every style has a default value that your theme can fall back on if you have chosen not to customize it. In other words, your theme inherits the styling, images, and templates from any of the built-in themes. This saves you time and keeps your themes smaller and less cluttered, because your theme contains only its own resources, using defaults for the rest, like emoticon graphics for the message boards portlet.

Liferay themes are easy to create. You can start by making changes only in the CSS files. When you need to customize themes more extensively, you can change the HTML.

If you hope to become a theme customization guru, there are several technologies you should know:

- *CSS*: Create a new theme simply by modifying a CSS file.
- *Velocity*: Customize the markup generated by the theme.
- *JavaScript*: Add special behaviors to your theme.
- *XML*: Some theme settings are specified in XML.

To follow the examples in this guide, you should be familiar with the command line or the Liferay IDE.

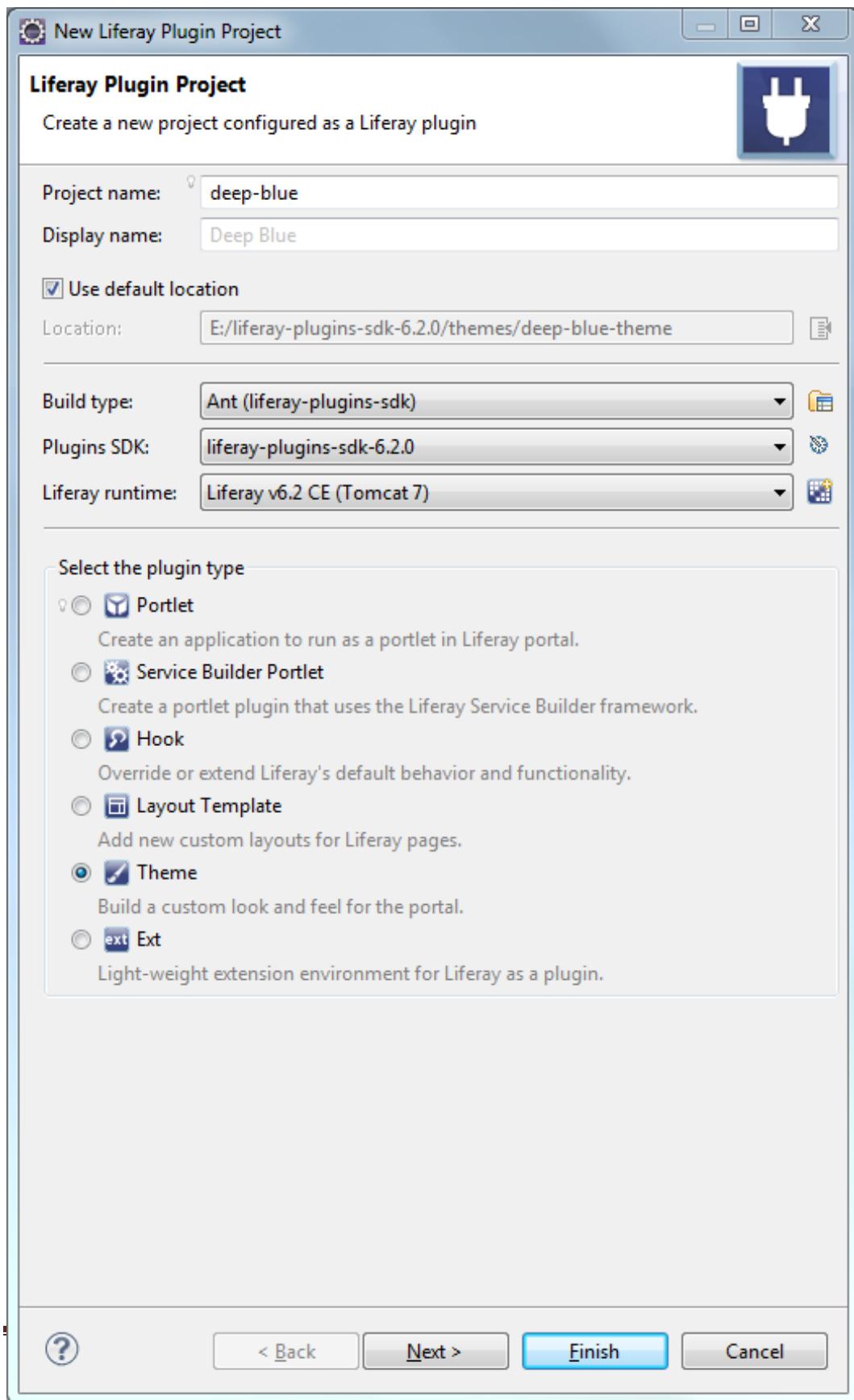
Let's create a theme!

11.1.1. ***Creating a Theme Project***

The theme creation process is nearly identical to the portlet creation process that we covered in Developing Apps with Liferay IDE. Our theme will be named *Deep Blue*, so the project name (without spaces) is *deep-blue*, and the display name (which can have spaces) is *Deep Blue*. Let's create the theme using Liferay Developer Studio first, and then with the terminal.

Using Developer Studio:

1. Go to *File* → *New* → *Liferay Project*.
2. Fill in *deep-blue* for the Project name and *Deep Blue* for the Display name.
3. Select the Liferay Plugins SDK and Portal Runtime you've configured.
4. Select *Theme* for your Plugin type.
5. Click *Next*.
6. Select a theme parent. Your theme inherits the parent theme's styling as a base from which to build your theme. In addition to the *_styled* theme, you can choose to inherit from the *_unstyled* theme, which contains no styling. There's also the *classic* theme that has a smooth look and feel and works well. For now, select *_styled* as the theme parent.
7. Select your theme's framework. You can select the *Freemarker* or *Velocity* template frameworks for your theme. Or you can select *JSP* as your theme's framework.
8. Click *Finish*.



With Developer Studio, you can create a new plugin project, or if you already have a project, create a new plugin in an existing project. A single Liferay project can contain multiple plugins.

Using the terminal:
Navigate to the themes directory in the Plugins SDK and enter the appropriate command for your operating system:

1. In Linux and Mac OS X, enter

```
./create.sh deep-blue "Deep Blue"
```

2. In Windows,

enter

```
create.bat deep-blue "Deep Blue"
```

Now there's a blank theme in your themes folder, which the Plugins SDK automatically named by appending "-theme" to your project name. Right now your theme is empty. Your next step is to set a base theme that serves as the default for your theme.

11.1.2. Setting a Base Theme

All themes in Liferay are built on top of two base themes, named *_unstyled* and *_styled*. *Your newly created theme is based on these by default, but they contain very limited styling. You can take advantage of an existing* theme's styling by setting the theme of your choice as the base for your theme.*

Base themes are added in layers. First *_unstyled* is added, giving you the core of the theme, then *styled, providing the most basic elements. When you set a different base theme, it's added on top of styled* and overrides the default styling wherever there are differences. After the base themes are added, your own custom styling is added on top.

By default, themes are based on the *_styled* theme, which provides only basic styling of portlets. If you open the build.xml file in your theme's directory, you see the following code:

```
<?xml version="1.0"?>
<!DOCTYPE project>

<project name="deep-blue-theme" basedir"." default="deploy">
    <import file="../build-common-theme.xml" />

    <property name="theme.parent" value="_styled" />
</project>
```

The theme.parent property determines the theme your theme inherits its styling from. In addition to the *_styled* theme, you can choose to inherit from the *_unstyled* theme, which contains no styling. This makes more work for you, but offers full flexibility to design your CSS files from scratch.

You can also use the default Liferay theme **Classic** as a parent theme. You'll start with a look and feel that's already smooth and works well. But since so much is already done for you, there's less flexibility when building your design. It's a compromise between creating a theme as quickly as possible versus having full control of the result. It's your choice, and another example of the flexibility Liferay offers.

To specify a base theme, edit the build.xml file for your theme and change *_styled* in <property name="theme.parent" value="_styled"> to the name of any existing theme that's installed or in your Plugins SDK.

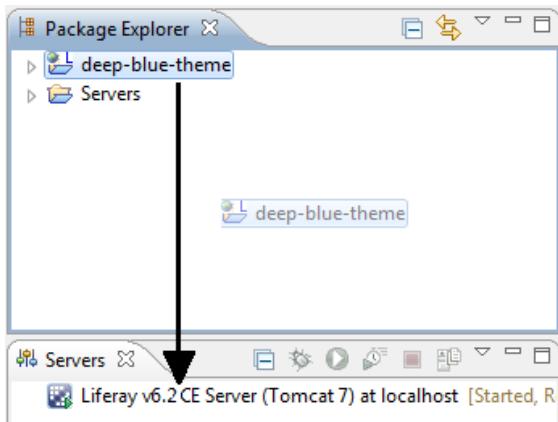
Now that your base theme is set, let's deploy the theme to your portal instance.

11.1.3. Deploying the Theme

If you're already familiar with portlet deployment from reading Developing Apps with Liferay IDE, theme deployment will be a piece of cake! You can deploy your theme in Developer Studio

or the terminal.

Deploying in Developer Studio: Click and drag your theme project onto your server.



Upon deploying, your server outputs messages indicating your plugin is read, registered, and available for use.

```
Reading plugin package for deep-blue-theme  
Registering themes for deep-blue-theme  
1 theme for deep-blue-theme is available  
for use
```

Deploying in the terminal: Open a terminal window in your themes/deep-blue-theme directory and enter

```
ant deploy
```

A BUILD SUCCESSFUL message indicates your theme is now being deployed. If you switch to the terminal window running Liferay, within a few seconds you will see the message 1 theme for deep-blue-theme is available for use.

Let's apply your theme to a page:

1. Go to your web browser and log in to the portal.
2. Click the *Edit* button from the left side menu.
3. Click the *Look and Feel* tab, select *Deep Blue* beneath the *Available Themes* heading, and click *Save*.

Now that you've built and deployed a theme, let's study theme anatomy.

11.1.4. Anatomy of a Theme Project

Custom themes are created by layering your customizations on top of one of Liferay's built-in themes.

The structure of a theme separates different types of resources into easily accessible folders. Here's the full structure of our Deep Blue theme:

- deep-blue-theme/
 - docroot/
 - _diffs/ (subfolders not created by default)
 - css/
 - images/
 - js/
 - templates/
 - css/
 - aui
 - (many directories)
 - _aui_custom.scss

- `_aui_variables.scss`
- `_liferay_custom.scss`
- `application.css`
- `aui.css`
- `base.css`
- `custom.css`
- `dockbar.css`
- `extras.css`
- `layout.css`
- `main.css`
- `navigation.css`
- `portlet.css`
- `images/`
 - (many directories)
- `js/`
 - `main.js`
- `templates/`
 - `init_custom.vm`
 - `navigation.vm`
 - `portal_normal.vm`
 - `portal_pop_up.vm`
 - `portlet.vm`
- `WEB-INF/`
 - `liferay-look-and-feel.xml`
 - `liferay-plugin-package.properties`

The `_diffs` folder that's created inside the `docroot` directory of your theme is important; this is where you place your theme's code. The `_diffs` folder must mirror the parent theme's directory structure. Since you'll only customize the parts of your theme that differ from the parent theme, place only the folders and files you'll customize there.

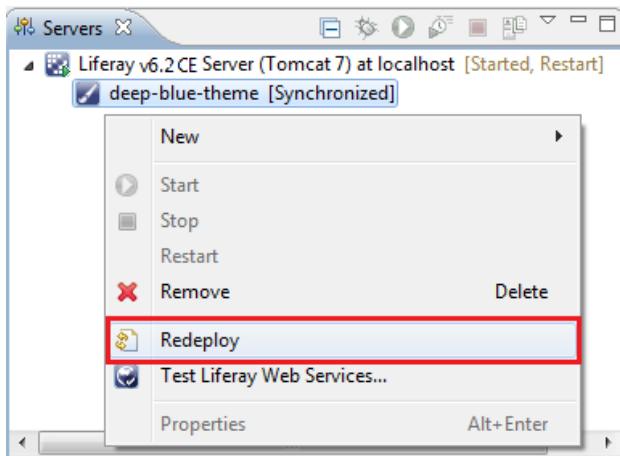
The other folders inside `docroot` were copied over from the parent theme in your Liferay bundle when you deployed your theme. Use these files as the basis for your modifications. For example, to customize the navigation, copy `navigation.vm` from `deep-blue-theme/docroot/templates/navigation.vm` into the `deep-blue-theme/docroot/_diffs/templates` folder (you may have to create this folder first). You can then open this file and customize it to your liking.

For custom styles, create a folder named `css` inside your `_diffs` folder and place a single file there called `custom.css`. This is where you'll define all your new styles. Because `custom.css` is loaded last, styles defined here override any styles in the parent theme.

It's a best practice to add your styles only to the `custom.css` file. This keeps all of your changes in one place and makes future upgrades easier, because you won't have to manually modify your templates to add support for new Liferay features.

Whenever you modify your theme in Developer Studio, redeploy it by right-clicking your theme

(located underneath your server), then selecting *Redeploy* from the menu.



Alternatively, redeploy your theme by opening a terminal, navigating to themes/deep-blue-theme and entering the command

```
ant deploy
```

Wait a few seconds until the theme deploys, then refresh your browser to see your changes.

Would you rather see your changes immediately, rather than having to redeploy to make your changes visible? Let's talk about Liferay Developer Mode to learn how.

with Themes

Do you want to develop Liferay resources without having to redeploy to see your portal modifications? Use Liferay Developer Mode! In Developer Mode, all caches are removed, so any changes you make are visible right away. Also, you don't have to reboot the server as often in Developer Mode.

How does Developer Mode let you see your changes more quickly? In Developer Mode, there are several changes to the normal order of operations. Here is a list of Developer Mode's key behavior changes and the portal property override settings that trigger them:

- 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 (`freemarker.engine.modification.check.interval=0`).
- 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.

You can use Developer Mode whether you're developing your theme plugin from in Liferay IDE/Developer Studio or not.

Using Developer Studio/IDE

For Liferay Portal servers of version 6.2 or greater, the *Liferay settings* section of the server runtime environment creation wizard lets you select either *Standard* or *Development*(`portal-developer.properties`) for the runtime's *Server Mode*. The Standard server mode is selected by default. To enable Development Mode, select `Development`(`portal-developer.properties`) and

save the runtime environment. The next time you start Liferay server that are based on this runtime environment, they start in Development Mode.

Liferay settings

Configure Liferay portal server settings.

Memory args:	<input type="text" value="-Xmx718m -XX:MaxPermSize=384m"/>
User timezone:	<input type="text" value="GMT"/>
External properties:	<input type="text"/> Browse...
Server Mode:	<input checked="" type="radio"/> Standard <input checked="" type="radio"/> Development (portal-developer.properties)
Username	<input type="text" value="test@liferay.com"/>
Password	<input type="password"/>

[Restore defaults.](#)



Wa
rnin
g:
Onl

y change the Server Mode from within the runtime environment's Liferay settings section. On

server startup, if Standard mode is set in Liferay Developer Studio/IDE the `portal-developer.properties` file is not included for overriding portal properties.

For Liferay Portal servers less than version 6.2 (e.g., Liferay v6.1 CE Server, Liferay v6.0 CE Server), Developer Studio/IDE enables Developer Mode by default. On starting your Liferay server for the first time, Studio/IDE creates a `portal-ide.properties` file in your Liferay Portal directory. This properties file has the property setting: `include-and-override=portal-developer.properties`, which enables Developer Mode.

Without Using Developer Studio/IDE

If you're not using Liferay Developer Studio/IDE, you must add the `portal-developer.properties` file to your application server's configuration file in order to enable Developer Mode. Since each application server has a different configuration file or UI to specify system properties, you must follow your application server's specific method for adding the `portal-developer.properties` file's properties to the system properties.

For example, to deploy Liferay in Developer Mode on a Tomcat application server, you'd add `-Dexternal-properties=portal-developer.properties` to the list of options for your `CATALINA_OPTS` variable, in your `setenv.sh` file (`setenv.bat` in Windows).



Tip: If you're already using the system property `external-properties` to load other properties files, add `portal-developer.properties` to the list and use a comma to separate it from other entries.

Great! You've set up your Liferay server for Developer Mode. Now, when you modify your

theme's `custom.css` file directly in your Liferay bundle, you can see your changes applied immediately on redeploying your theme! Make sure you copy any changes you make back into your `_diffs` folder, or they'll be overwritten when you redeploy your theme.

Let's add a thumbnail image for our theme now.

11.3. Creating a Theme Thumbnail

Now that your theme is available in Liferay, it's time to dress it up for a stylistic appeal. Currently in the *Look and Feel* settings, your theme's thumbnail is nonexistent. To remedy this, create a 150 pixels wide by 120 pixels high image to use as your theme's thumbnail. You may want to take a snapshot of your theme and re-size it to these dimensions. It is very important to abide by these *exact* dimensions or your image will not display properly as a thumbnail. Save the image as a `.png` file named `thumbnail.png` and place it in your theme's `docroot/_diffs/images` directory (create this directory if it doesn't already exist). On redeployment, your `thumbnail.png` file automatically displays as your theme's thumbnail.

Now go to the *Look and Feel* settings. Your theme's thumbnail should appear there, along with the *Classic* theme's thumbnail.

Let's learn how to design a theme's look and feel next.

11.4. Designing a Look and Feel

You define a theme's look and feel via a file named `liferay-look-and-feel.xml` in the `WEB-INF` directory.

Let's consider how to make your theme configurable to administrative users.

11.4.1. Making Themes Configurable with Settings

You can define settings to make your theme configurable. Create a file named `liferay-look-and-feel.xml` in the `WEB-INF` directory (if necessary), with the following content:

```
<?xml version="1.0"?>
<!DOCTYPE look-and-feel PUBLIC "-//Liferay//DTD Look and Feel 6.2.0//EN"
"http://www.liferay.com/dtd/liferay-look-and-feel_6_2_0.dtd">

<look-and-feel>
    <compatibility>
        <version>6.2.0+</version>
    </compatibility>
    <theme id="deep-blue" name="Deep Blue">
        <settings>
            <setting key="my-setting" value="my-value" />
        </settings>
    </theme>
</look-and-feel>
```

To define additional settings, add more `<setting>` elements to the file. Access the settings from the theme templates using the following code:

```
$theme.getSetting("my-setting")
```

Let's say you want to be able to choose from two different page headers (perhaps one includes more details, while the other is smaller). Instead of creating two themes that are identical except for some changes in the header, you can create one and define a setting that lets you choose which header is displayed.

Make sure you have a docroot/_diffs/templates folder created and copy the docroot/templates/portal_normal.vm file into that directory. It's a good rule of thumb to modify files for your new theme in the _diffs folder. Now, open your _diffs/templates/portal_normal.vm template and insert the following:

```
#if ($theme.getSetting("header-type") == "detailed")
    #parse ("$full_templates_path/header_detailed.vm")
#else
    #parse ("$full_templates_path/header_brief.vm")
#end
```

If you're following along with this example, you'll need to create the header_detailed.vm and header_brief.vm files and place them in the _diffs/templates folder. For this simple tutorial, you can keep these VM templates blank.

Then, add two different entries in the liferay-look-and-feel.xml file that refer to the same theme, but have different values for the header-type setting:

```
<theme id="deep-blue" name="Deep Blue">
    <settings>
        <setting key="header-type" value="detailed" />
    </settings>
</theme>
<theme id="deep-blue-mini" name="Deep Blue Mini">
    <settings>
        <setting key="header-type" value="brief" />
    </settings>
</theme>
```

Alternatively, you can make your settings configurable from within Liferay portal. Use *configurable* settings to let users turn certain theme features on or off or to allow users to provide input to a theme setting.

As an example, you can create an option to display a slogan next to your company's name in the footer of your site's pages:

1. Insert logic into your portal_normal.vm template to display a slogan along with your company's name (e.g. Nosester) in the footer of your site pages:

```
<footer id="footer" role="contentinfo">
    <p>
        #if ($theme.getSetting("display-slogan-footer") == true)
            Nosester $theme.getSetting("slogan")
        #else
            Nosester
        #end
    </p>
</footer>
```



Note: Let's look more closely at two theme setting variables appearing in the above logic. The `display-slogan-footer` variable holds a boolean value indicating whether to display the version of the footer that contains your slogan. The `slogan` variable holds your slogan text.

2. Declare the two theme setting variables for the Deep Blue theme in your `liferay-look-and-feel.xml`, located in your theme's `WEB-INF` folder:

```
<settings>
    <setting configurable="true"
        key="slogan"
        type="textarea"
        value="" />

    <setting configurable="true"
        key="display-slogan-footer"
        type="checkbox"
        value="true" />
</settings>
```



Warning: Make sure you have an up-to-date DTD version specified for your `liferay-look-and-feel.xml` file. For example, http://www.liferay.com/dtd/liferay-look-and-feel_6_2_0.dtd. When referencing older DTD files (e.g., 6.0.0), the slogan settings are unavailable.

The portal administrator can enter a slogan and activate it for the portal via the *Look and Feel* section of the *Site Administration → Site Pages* panel (see the *Creating and Managing Pages* section of Using Liferay Portal).

Look and Feel

Current Theme



(○)Deep Blue

Author

Liferay, Inc.

Settings

slogan

- Your nose knows best!



display-slogan-footer

When the portal administrator saves the settings, your site's pages show the new footer, including the slogan.

Sign In

You are signed in as [Joe Bloggs](#).

Nosester - Your nose knows best!



Note:
Use a
language

properties hook to display configurable theme settings properly, like the slogan text area and footer checkbox from the previous example. For details, see the

Overriding a *Language.properties* File section found in the *Hooks* chapter of this guide.

Next, let's customize your theme's color scheme.

11.4.2. Specifying Color Schemes

Specify color schemes with a CSS class name, which of course also lets you choose different background images, different border colors, and more.

Here's how you can define your color schemes in `liferay-look-and-feel.xml`:

```
<theme id="deep-blue" name="Deep Blue">
    <settings>
        <setting key="my-setting" value="my-value" />
    </settings>
    <color-scheme id="01" name="Day">
        <css-class>day</css-class>
        <color-scheme-images-path>
            ${images-path}/color_schemes/${css-class}
        </color-scheme-images-path>
    </color-scheme>
    <color-scheme id="02" name="Night">
        <css-class>night</css-class>
    </color-scheme>
</theme>
```

In your `_diffs/css` folder, create a `color_schemes` folder and place a `.css` file in it for each color scheme. In the case above, we can have either one file called `night.css`, letting the default styling handle the first color scheme, or we can use both `day.css` and `night.css` to specify each scheme. Let's use the latter option here, creating both files to define our color schemes.

Place the following lines at the bottom of your `docroot/css/custom.css` file:

```
@import url(color_schemes/day.css);
@import url(color_schemes/night.css);
```

The color scheme CSS class is placed on the `<body>` element, so you can use it to identify your styling. In `day.css`, prefix all your CSS styles like this:

```
body.day { background-color: #ddf; }
.day a { color: #66a; }
```

In `night.css`, prefix all your CSS styles like this:

```
body.night { background-color: #447; color: #777; }
.night a { color: #bbd; }
```

You can create separate thumbnail images for your color schemes. The `<color-scheme-images-path>` element tells Liferay where to look for these images (you only have to place this element in one color scheme for it to affect both). For our example, create the folders `_diffs/images/color_schemes/day` and `_diffs/images/color_schemes/night`. In each folder place a `thumbnail.png` and `screenshot.png` file, according to the specifications defined in the Thumbnails section above.

Let's review the predefined settings available for your theme.

11.4.3. Leveraging Portal Predefined Settings

The portal defines some settings that allow the theme to determine certain behaviors. As of this writing, predefined settings are only available for portlet borders, bullet styles, and the site name, but more settings may be added in the future. Modify these settings from the `liferay-look-and-feel.xml` file. Remember, your `liferay-look-and-feel.xml` file should have the 6.2.0 doctype for the following predefined settings to work correctly.



Note: To override default behavior for individual portlets, you can modify the a portlet's `liferay-portlet.xml` file.

Let's get on with learning about predefining settings using themes. First, let's take a look at settings for portlet borders.

11.4.3.1. Portlet Borders

The theme turns on portlet borders, by default. But you can turn them off by setting `portlet-setup-show-borders-default` to `false` in your theme's `liferay-look-and-feel.xml` file. For example, the following setting, makes border display configurable for the portal administrator, and disables showing the borders as the default:

```
<settings>
  ...
  <setting
    configurable="true"
    key="portlet-setup-show-borders-default"
    type="checkbox"
    value="false"
  />
  ...
</settings>
```

Now that you've configured portlet borders, let's configure bullet styles used in your sites.

11.4.3.2. Bullet Styles

Liferay's Navigation portlet can be configured to use any bullet styles inherited by your theme or implemented in your theme. For example, if your theme uses Liferay's *classic* theme as its base parent, you can leverage its *arrows* bullet style. Here is the arrow bullet style's class from the *classic* theme's `_diffs/css/custom.css` file:

```
.nav-menu-style-arrows ul {
  list-style-image: url(@theme_image_path@/navigation/bullet_selected.png);
}
```

You can make this bullet style, along with any bullet styles you implement, available for site administrators to use in their site's Navigation portlet. Just follow the naming convention as demonstrated below, substituting `[bullet style name]`, with your bullet style's name:

```
.nav-menu-style-[bullet style name] ul {  
    ... CSS selectors ...  
}
```

Then, make the `bullet-style` setting configurable in your `liferay-look-and-feel.xml` file. From this setting, list optional bullet styles you want available to site administrators, and set a default bullet style as well:

```
<settings>  
    ...  
    <setting  
        configurable="true"  
        key="bullet-style"  
        options="arrows,dots,classic,modern"  
        value="dots"  
    />  
    ...  
</settings>
```

Your site administrators can now choose the bullet style to apply to the Navigation portlet. They select it from the site's *Look and Feel* control page.

Using CSS, and maybe some unobtrusive JavaScript, you can create a navigation menu that looks just the way you want it. Next, let's take a look at how to configure display your site's name.

11.4.3.3. Site Names

The site name settings let site administrators decide whether to display a site's name (i.e., title). But, if you are using a logo that mentions your company or site on each site page, you may find

the default site name display distracting.



Since the themes you create in the Plugins SDK use Liferay's *_unstyled* theme as a

base theme, you have the following settings available for configuring site name display:

- `show-site-name-default` configures site name display and lets you turn it on/off by default.
- `show-site-name-supported` configures support for site name display and lets you turn it on/off by default.

Here is how you might specify them in your `liferay-look-and-feel.xml` file:

```
<settings>  
    ...  
    <setting
```

```

    configurable="true"
    key="show-site-name-default"
    type="checkbox"
    value="true"
/>
<setting
    configurable="true"
    key="show-site-name-supported"
    type="checkbox"
    value="true"
/>
...
</settings>

```

With these settings configurable, site administrators can control site name display from the each site's *Look and Feel* tab, which can be found by clicking the *Edit* button from the left side menu of any page.

Let's talk about Liferay's JavaScript library next.

11.5. Understanding Your Theme's JavaScript Callbacks in main.js

Liferay has its own JavaScript library called AlloyUI, an extension to Yahoo's YUI3 framework. You can take advantage of AlloyUI or YUI3 in your themes. Inside your theme's `main.js` file, you'll find definitions for three JavaScript callbacks:

- **AUI().ready(fn):** Executed after the HTML in the page has finished loading (minus any portlets loaded via AJAX).
- **Liferay.Portlet.ready(fn):** Executed after each portlet on the page has loaded. The callback receives two parameters: `portletId` and `node`. `portletId` is the ID of the portlet that was just loaded. `node` is the Alloy Node object of the same portlet.
- **Liferay.on('allPortletsReady', fn):** Executed after everything else (including AJAX portlets) has finished loading.

The contents of the `main.js` file are listed below:

```

AUI().ready(
    /*
    This function gets loaded when all the HTML, not including the portlets,
    is
    loaded.
    */

    function() {
    }
);

Liferay.Portlet.ready(
    /*
    This function gets loaded after each and every portlet on the page.

```

```

portletId: the current portlet's id
node: the Alloy Node object of the current portlet
*/
function(portletId, node) {
}
);

Liferay.on(
  'allPortletsReady',
  /*
  This function gets loaded when everything, including the portlets, is on
  the page.
  */
  function() {
  }
);

```

Want to learn how to import resources with your theme? We'll discuss how you can do this in the next section.

11.6. Importing Resources with Your Themes

A theme without content is like an empty house. If you're trying to sell an empty house, it may be difficult for prospective buyers to see its full beauty. However, staging the house with some furniture and decorations helps prospective buyers imagine what the house might look like with their belongings. Liferay's resources importer application is a tool that allows a theme developer to have files and web content automatically imported into the portal when a theme is deployed. Usually, the resources are imported into a site template but they can also be imported directly into a site. Portal administrators can use the site or site template created by the resources importer to showcase the theme. This is a great way for theme developers to provide a sample context that optimizes the design of their theme. In fact, all standalone themes that are uploaded to Liferay Marketplace must use the resources importer. This ensures a uniform experience for Marketplace users: a user can download a theme from Marketplace, install it on their portal, go to Sites or Site Templates in the Control Panel and immediately see their new theme in action. In this section, we discuss how to include resources with your theme.



Note: The resources importer can be used in any type of plugin project to import resources. Importing resources within a theme plugin is just one of the more common use cases.

Liferay's welcome theme includes resources that the resources importer automatically deploys to the default site. (Note: The welcome theme is only applied out-of-the-box in Liferay CE.) The welcome theme and the pages and content that it imports to the default site provide a good example of the resources importer's functionality.

Welcome To Liferay Portal



Start

Review our [Quick Start Guide](#) for an overview of Liferay's features.



Learn

Read the official [Liferay User Guide](#) for detailed information about setting up and configuring Liferay.



Engage

Visit the [Liferay Community](#) to post questions, find answers, and contribute.



Develop

Explore our [Developer Resources](#) to develop apps and more for Liferay Portal.



Evaluate

Learn more about partners, support, training and other [enterprise level options](#) available for Liferay.

[Download this page as a PDF](#)

If it's not already installed, you can download the resources importer application from Liferay Marketplace. Search for either *Resources Importer CE* or *Resources Importer EE*, depending on your Liferay Portal platform, and download the latest version. Install and deploy the resources importer to your Liferay instance the same way you would deploy any other Liferay plugin or Marketplace app.



Tip: If you deploy a theme to your Liferay Portal instance and don't have the resources importer already deployed, you might see a message like this:

```
19:21:12,224 INFO [pool-2-thread-2] [HotDeployImpl:233] Queuing test-theme for deploy because it is missing resources-importer-web
```

Such a message appears if the resources importer is declared as a dependency in your theme's `liferay-plugin-package.properties` file but is not deployed. You can deploy the resources importer application to satisfy the dependency or you can remove or comment out the dependency declaration if you're not going to use the resources importer with your theme (see below).

When you create a new theme project using the Liferay Plugins SDK, check your theme's `docroot/WEB-INF/liferay-plugin-package.properties` file for two entries related to the resources importer. One or both of these might be commented out or missing, depending on the version of your Plugins SDK:

```
required-deployment-contexts=\nresources-importer-web
```

```
resources-importer-developer-mode-enabled=true
```

The first entry, `required-deployment-contexts=resources-importer-web`, declares your theme's dependency on the resources importer plugin. If you're not going to use the resources importer with your theme and don't want to deploy the resources importer, you can remove or comment out this entry.

The second entry, `resources-importer-developer-mode-enabled=true`, is a convenience feature for theme developers. With this setting enabled, importing resources to a site or site template that already exists, recreates the site or site template. Importing resources into a site template reapplies the site template and its resources to the sites that are based on the site template. Without `resources-importer-developer-mode-enabled=true`, you have to manually delete the sites or site templates built by the resources importer, each time you want to apply changes from your theme's `docroot/WEB-INF/src/resources-importer` folder.



Warning: the `resources-importer-developer-mode-enabled=true` setting can be dangerous since it involves *deleting* (and re-creating) the affected site or site template. It's only intended to be used during development. Never use it in production.

If you'd like to import your theme's resources directly into a site, instead of into a site template, you can specify the following in your `liferay-plugin-package.properties` file:

```
resources-importer-target-class-name=com.liferay.portal.model.Group
```

```
resources-importer-target-value=[site-name]
```

If you're using the `resources-importer-target-value=[site-name]` property, double check the site name that you're specifying. If you specify the wrong value, you could end up deleting (and re-creating) the wrong site!



Warning: It's safer to import theme resources into a site template than into an actual site. The `resources-importer-target-class-name=com.liferay.portal.model.Group` setting can be handy for development and testing but should be used cautiously. Don't use this setting in a theme that will be deployed to a production Liferay instance or a theme that will be submitted to Liferay Marketplace. To prepare a theme for deployment to a production Liferay instance, use the default setting so that the resources are imported into a site template. You can do this explicitly by setting `resources-importer-target-class-name=com.liferay.portal.model.LayoutSetPrototype` or implicitly by commenting out or removing the `resources-importer-target-class-name` property.

All of the resources a theme uses with the resources importer go in the `<theme-name>/docroot/WEB-INF/src/resources-importer` folder. The assets to be imported by your theme should be placed in the following directory structure:

- `<theme-name>/docroot/WEB-INF/src/resources-importer/`
 - `sitemap.json` - defines the pages, layout templates, and portlets
 - `assets.json` - (optional) specifies details on the assets
 - `document_library/`
 - `documents/` - contains documents and media files
 - `journal/`
 - `articles/` - contains web content (HTML) and folders grouping web content articles (XML) by template. Each folder name must match the file name of the corresponding template. For example, create folder `Template 1/` to hold an article based on template file `Template 1.vm`.
 - `structures/` - contains structures (XML) 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.xml`.
 - `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.xml`.

When you create a new theme using the Liferay Plugins SDK (`liferay-plugins-sdk-6.1.1-ce-ga2-20121004092655026` or later), this folder structure is created automatically. Also, a default `sitemap.json` file is created and a default `liferay-plugin-package.properties` file is created in the `WEB-INF` folder.

You have two options for specifying resources to be imported with your theme. The recommended approach is to add resource files to the folders outlined above and to specify the contents of the site or site template in a `sitemap.json` file (described below). Alternatively, you can use an `archive.lar` file to package the resources you'd like your theme to deploy. To create such an `archive.lar`, just export the contents of a site from Liferay Portal using the site scope. Then place the `archive.lar` file in your theme's `<theme-name>/docroot/WEB-INF/src/resources-importer` folder. If you choose to use an archive file to package all of your resources, you won't need a `sitemap.json` file or any other files in your `<theme-name>/docroot/WEB-INF/src/resources-importer` folder. Note, however, a LAR file is version-specific; it won't work on any version of Liferay other than the one from which it was exported. For this reason, using a `sitemap.json` file to specify resources is the most flexible approach. If you're developing themes for Liferay Marketplace, you should use the `sitemap.json` to specify resources to be imported with your theme.

The `sitemap.json` in the `<theme-name>/docroot/WEB-INF/src/resources-importer` folder specifies the site pages, layout templates, web content, assets, and portlet configurations provided with the theme. This file describes the contents and hierarchy of the site

for Liferay to import as a site or site template. Even if you're not familiar with JSON, the `sitemap.json` file is easy to understand. Let's examine a sample `sitemap.json` file:

```
{  
    "layoutTemplateId": "2_columns_ii",  
    "privatePages": [  
        {  
            "friendlyURL": "/private-page",  
            "name": "Private Page",  
            "title": "Private Page"  
        }  
    ],  
    "publicPages": [  
        {  
            "columns": [  
                [  
                    {  
                        "portletId": "58"  
                    },  
                    {  
                        "portletId": "71"  
                    },  
                    {  
                        "portletId": "56",  
                        "portletPreferences": {  
                            "articleId": "Without Border.html",  
                            "groupId": "${groupId}",  
                            "portletSetupShowBorders": "false"  
                        }  
                    },  
                    {  
                        "portletId": "56",  
                        "portletPreferences": {  
                            "articleId": "Custom Title.html",  
                            "groupId": "${groupId}",  
                            "portletSetupShowBorders": "true",  
                            "portletSetupTitle_en_US":  
                                "Web Content Display with Custom Title",  
                            "portletSetupUseCustomTitle": "true"  
                        }  
                    }  
                ],  
                [  
                    {  
                        "portletId": "47"  
                    },  
                    {  
                        "portletId": "71_INSTANCE_${groupId}",  
                        "portletPreferences": {  
                            "displayStyle": "[custom]",  
                            "headerType": "root-layout",  
                            "includedLayouts": "all",  
                            "nestedChildren": "1",  
                            "rootLayoutLevel": "3",  
                        }  
                    }  
                ]  
            ]  
        }  
    ]  
}
```

```

        "rootLayoutType": "relative"
    }
},
"Web Content with Image.html",
{
    "portletId": "118",
    "portletPreferences": {
        "columns": [
            [
                {
                    "portletId": "56",
                    "portletPreferences": {
                        "articleId":
                            "Child Web Content 1.xml",
                        "groupId": "${groupId}",
                        "portletSetupShowBorders":
                            "true",
                        "portletSetupTitle_en_US":
                            "Web Content Display with
                            Child Structure 1",
                        "portletSetupUseCustomTitle":
                            "true"
                    }
                }
            ],
            [
                {
                    "portletId": "56",
                    "portletPreferences": {
                        "articleId":
                            "Child Web Content 2.xml",
                        "groupId": "${groupId}",
                        "portletSetupShowBorders":
                            "true",
                        "portletSetupTitle_en_US":
                            "Web Content Display with
                            Child Structure 2",
                        "portletSetupUseCustomTitle":
                            "true"
                    }
                }
            ]
        ],
        "layoutTemplateId": "2_columns_i"
    }
}
],
"friendlyURL": "/home",
"nameMap": {
    "en_US": "Welcome",
    "fr_CA": "Bienvenue"
},
"title": "Welcome"

```

```

},
{
  "columns": [
    [
      {
        "portletId": "58"
      }
    ],
    [
      {
        "portletId": "47"
      }
    ]
  ],
  "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": "/hidden-page",
  "name": "Hidden Page",
  "title": "Hidden Page",
  "hidden": "true"
}
]
}

```

The first thing you should declare in your `sitemap.json` file is a layout template ID so the target site or site template can reference the layout template to use for its pages. You can also specify different layout templates to use for individual pages. You can find layout templates in your Liferay installation's `/layouttpl` folder. Next, you have to declare the layouts, or pages, that your site template should use. Note that pages are called *layouts* in Liferay's code. You can specify a name, title, and friendly URL for a page, and you can set a page to be hidden. To declare that web content should be displayed on a page, simply specify an HTML file. You can declare portlets by specifying their portlet IDs which can be found in Liferay's `WEB-INF/portlet-custom.xml` file. You can also specify portlet preferences for each portlet. Optionally, you can create an `assets.json` file in your `<theme-name>/docroot/WEB-INF/src/resources-importer` folder. While the `sitemap.json` file defines the pages of the site or site template to be imported, along with the layout templates, portlets, and portlet

preferences of these pages, the `assets.json` file specifies details about the assets to be imported. Tags can be applied to any asset. Abstract summaries and small images can be applied to web content articles. For example, the following `assets.json` file specifies two tags for the `company_logo.png` image, one tag for the `Custom Title.html` web content article, and an abstract summary and small image for the `Child Web Content 1.xml` article structure:

```
{
  "assets": [
    {
      "name": "company_logo.png",
      "tags": [
        "logo",
        "company"
      ]
    },
    {
      "name": "Custom Title.html",
      "tags": [
        "web content"
      ]
    },
    {
      "abstractSummary": "This is an abstract summary.",
      "name": "Child Web Content 1.xml",
      "smallImage": "company_logo.png"
    }
  ]
}
```

Now that you've learned about the directory structure for your resources, the `sitemap.json` file for referencing your resources, and the `assets.json` file for describing the assets of your resources, it's time to put resources into your theme. You can create resources from scratch and/or bring in resources that you've already created in Liferay. Let's go over how to leverage your HTML (basic web content), XML (structures), or VM or FTL (templates) files from Liferay:

- **web content (basic):** Edit the article, click *Source*, and copy its contents into an HTML file in the `resources-importer/journal/articles/` folder.
- **web content (based on structure and template):** Edit the article, click *Download* to download it as a file `article.xml`. Create a folder for the template under `resources-importer/journal/articles/`, rename the downloaded `article.xml` file as desired, and copy it into the folder for the template. The web content article's XML fills in the data required by the structure.
- **structure:** Edit the structure by clicking *Source*, and copy and paste its contents into a new XML file for the structure in the `resources-importer/journal/structures/` folder. The structure XML sets a wireframe, or blueprint, for an article's data.
- **template:** Edit the template by clicking *Source*, and copy and paste its contents into a new XML file for the template in the `resources-`

`importer/journal/templates/` folder. The template defines how the data should be displayed.

Here is an outline of steps you can use in developing your theme and its resources:

1. Create your theme.
2. Add your resources under the `<theme-name>/docroot/WEB-INF/src/resources-importer` folder and its subfolders.
3. Create a `sitemap.json` file in your `resources-importer/` folder. In this file, define the pages of the site or site template to be imported, along with the layout templates, portlets, and portlet preferences of these pages.
4. Create an `assets.json` file in your `resources-importer/` folder. In this file, specify details of your resource assets.
5. In your `liferay-plugin-package.properties` file, include `resources-importer-web` in your `required-deployment-contexts` property's list and set `resources-importer-developer-mode-enabled=true`. For the `resources-importer-target-value` property, specify the name of the site or site template into which you are importing or comment it out to use the theme's name. For the `resources-importer-target-class-name` property, comment it out to import to a site template or set it to `com.liferay.portal.model.Group` to import directly into a site.
6. Deploy your plugin into your Liferay instance.
7. View your theme, and its resources, from within Liferay. Log in to your portal as an administrator and check the Sites or Site Templates section of the Control Panel to make sure that your resources were deployed correctly. From the Control Panel you can easily view your theme and its resources:
 - › If you imported into a site template, select its *Actions* → *View Pages* to see it.
 - › If you imported directly into a site, select its *Actions* → *Go to Public Pages* to see it.

You can go back to any of the beginning steps in this outline to make refinements. It's just that easy to develop a theme with resources intact!

To see a simple working example of the resources importer in action, visit <https://github.com/liferay/liferay-docs/tree/master/devGuide/code/test-resources-importer-theme-6.1.1.1.war>. This is just the classic Liferay theme with some sample resources added. If you're interested in extending the functionality of the resources-importer application, you can use the test-resources-importer-portlet to check that you aren't breaking existing functionality. The test-resources-importer-portlet is available on Github here: <https://github.com/liferay/liferay-plugins/tree/master/portlets/test-resources-importer-portlet>. The sample resources included in the test-resources-importer-theme are the same ones included in the test-resources-importer-portlet. If you'd like to examine another example, check out the code for Liferay's welcome theme: <https://github.com/liferay/liferay-plugins/tree/master/themes/welcome-theme>. Note that this theme imports resources directly into the default site. Typically, this won't be something you'll need to do; instead, you'll usually have your theme's resources imported into a site template. For

further examples, please examine the Zoe themes which you can find on Github here <https://github.com/liferay/liferay-plugins/tree/master/themes> and which you can download from Liferay Marketplace.

As promised, we'll show you how to create Layout Templates next.

11.7. ***Creating Liferay Layout Templates***

By now, you've likely added portlets to a page by dragging them from the *Add* menu and dropping them into place. Are there times, though, when you find yourself limited by Liferay's page layout options? Maybe your Feng Shui (pronounced *fung SHWAY*) senses are picking up on some negative energy? Or perhaps you find yourself adding the same portlets over and over again onto the same types of pages? Don't despair! Break the monotony by creating your own custom layout templates. Layout template plugins let you design layouts that flow nicely, embed commonly used portlets, and apply CSS, Velocity, and HTML to make your pages visually pop.

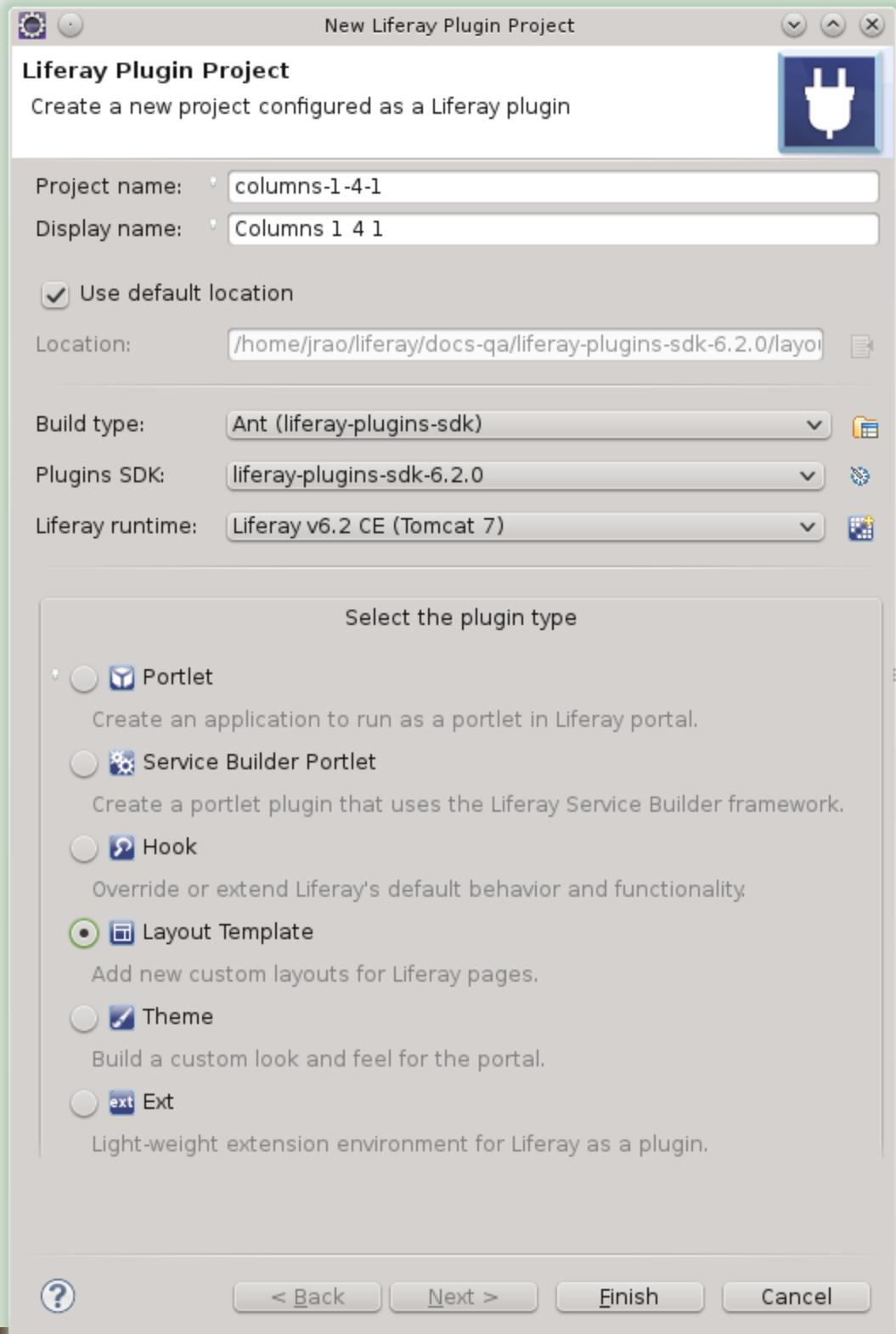
Let's create a custom layout template!

11.7.1. ***Creating a Layout Template Project***

With the Plugins SDK, you can deploy layout templates as plugins, and creating layout templates with Liferay Developer Studio is easier than ever. Let's create a layout template called *Columns 1 4 1*.

Using Developer Studio:

1. Go to *File* → *New* → *Liferay Plugin Project*.
2. Enter *columns-1-4-1* for the Project name and *Columns 1 4 1* for the Display name.
3. Choose whichever build type you prefer (Ant or Maven) and select the appropriate *Plugins SDK* and *Liferay runtime*.
4. Select *Layout Template* as your plugin type.
5. Click *Finish*.



Using the terminal: Navigate to your Plugins SDK's `layouttpl` folder, and execute the `create` script in your terminal. Here's the generic version of the `create` script, followed by operating system-specific commands:

```
./create.[sh|bat] <project-name> "<layout template title>"
```

1. Example in Linux and Mac OS X:

```
./create.sh columns-1-4-1 "Columns 1 4 1"
```

2. Example in Windows:

```
create.bat columns-1-4-1 "Columns 1 4 1"
```

Developer Studio's *New Project* wizard and the `create` scripts generate layout template projects in your Plugin SDK's `layouttpl` folder. Layout template project names must end with `-layouttpl` so when you enter `columns-1-4-1` for the project name, `-layouttpl` is automatically appended to the project name.

11.7.2. Anatomy of a Layout Template Project

Let's look at the directory structure of a layout template project and learn about its various files:

- `columns-1-4-1-layouttpl/`
 - `docroot/`
 - `META-INF/`
 - `WEB-INF/`
 - `liferay-layout-templates.xml`
 - `liferay-plugin-package.properties`
 - `columns_1_4_1.png`
 - `columns_1_4_1.tpl`
 - `columns_1_4_1.wap.tpl`
 - `build.xml`

Navigate to your Plugins SDK's `layouttpl/` folder and you'll see that the Plugins SDK automatically appended `-layouttpl` to your project's name. A layout template project can contain multiple layout templates. The directory structure is the same, but you'll have a `.png`, `.tpl`, and `.wap.tpl` file for each layout template in the `docroot/` folder. The `liferay-*` files describe the layout templates for packaging and deployment.

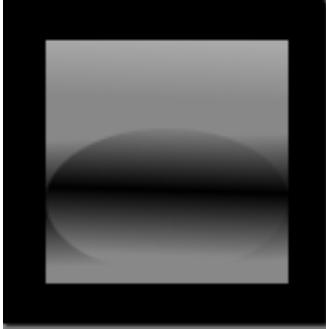
Now that you're well-versed on the anatomy of a layout template, let's explore the layout template files.

11.7.3. Layout Template Files

One or more layout template plugins can reside in a layout template project. Let's see what each template file does:

- `[project-name].tpl`: Generates the HTML structure of the template.
- `[project-name].wap.tpl`: Variant template for mobile devices. WAP stands for wireless application protocol.
- `[project-name].png`: Thumbnail representation of the template that you see in Liferay Portal from the Page Layout screen. You'll have to create the thumbnail image,

but you can use the default PNG for layout templates as a starting point.



Let's move on to Liferay configuration files.

11.7.4. Liferay Configuration Files

In addition to the three template-specific files, a layout template project has two Liferay configuration files:

- `liferay-layout-templates.xml`: Specifies the name of the layout templates and the location of their TPL and PNG files.
- `liferay-plugin-package.properties`: Describes the plugin project to Liferay's hot deployer.

Now that you're familiar with the layout template's files and directory structure, let's deploy a layout template on the server.

11.7.5. Deploying Layout Templates

If you've ever deployed a theme or portlet, you already know how to deploy layout templates! Use Developer Studio or the terminal to deploy your layout templates:

- **Deploying in Developer Studio:** Drag your layout template project onto your server.
- **Deploying in the terminal:** If you're using Ant, execute the following command From your layout template project directory:

```
ant deploy
```

If you're using Maven, please refer to this guide's section on deploying Liferay plugins with Maven.

When deploying your plugin, the server displays messages indicating that your plugin was read, registered, and is now available for use.

Example server output:

```
Reading plugin package for columns-1-4-1-layouttpl
Registering layout templates for columns-1-4-1-layouttpl
1 layout template for columns-1-4-1-layouttpl is available for use
```

Wait a minute! We can deploy the template, but we still haven't designed it. We'll need to add content to the TPL files that were generated when we created our layout template.

11.7.6. Designing a Layout Template

Initially, the layout template's generated TPL files are empty, a fresh canvas on which you can design layout templates. If this seems overwhelming, don't worry. We'll build a new layout template and explain how it works. If you want to see more examples, check out the Page Layouts section of Liferay Marketplace, download some CE layout templates provided by Liferay, and examine the source. You also can examine Liferay's core layout templates. These can be found in Liferay's source in the `liferay-portal/portal-web/docroot/layouttpl/custom/` folder.

Let's describe the layout template that we're about to create. We named it *Columns 1 4 1* because we want the first row to have just one column, the second row to have 4 (equal width) columns,

and the third row to have just one column. Liferay provides a similar layout template called *1-2-1 Columns Layout CE* on Liferay Marketplace. Here's the source of the 1-2-1 Columns Layout template:

```
<div class="columns-1-2-1" id="main-content" role="main">
    <div class="portlet-layout row-fluid">
        <div class="portlet-column portlet-column-only span12"
id="column-1">
            $processor.processColumn("column-1", "portlet-column-
content portlet-column-content-only")
        </div>
    </div>

    <div class="portlet-layout row-fluid">
        <div class="portlet-column portlet-column-first span8"
id="column-2">
            $processor.processColumn("column-2", "portlet-column-
content portlet-column-content-first")
        </div>

        <div class="portlet-column portlet-column-last span4"
id="column-3">
            $processor.processColumn("column-3", "portlet-column-
content portlet-column-content-last")
        </div>
    </div>

    <div class="portlet-layout row-fluid">
        <div class="portlet-column portlet-column-only span12"
id="column-4">
            $processor.processColumn("column-4", "portlet-column-
content portlet-column-content-only")
        </div>
    </div>
</div>
```

A CSS class named after the layout template project must be applied to the root `<div>`: `class="columns-1-2-1"`. An ID of *main-content* and a role of *main* must also be applied to the root `<div>`.

Inside of the root `<div>`, you need to create `<div>`s for each row of your layout template. You must apply the *portlet-layout* and *row-fluid* CSS classes to these `<div>`s: `<div class="portlet-layout row-fluid">`.

Inside each row `<div>`, you must specify one or more column `<div>`s. For each column `<div>`, make sure to specify the *portlet-column* CSS class. If a column is the first, last, or only column in a row, you must specify another CSS class: *portlet-column-first*, *portlet-column-last*, or *portlet-column-only*.

Liferay 6.2 themes use a fork of Twitter Bootstrap called Alloy Bootstrap:
<https://github.com/liferay/alloy-bootstrap> Alloy Bootstrap affects Liferay's layout templates as well as its themes. Liferay 6.2 layout templates use Bootstrap's 12 column grid system:
<http://getbootstrap.com/css/#grid>. For each column `<div>`, you must specify another CSS class

called *span[width]* where *width* is the numerator of a fraction over 12 representing the width of the column. For example, if you apply a *span8* CSS class to a column `<div>`, the column will take up $8/12 = 2/3$ of the page width. Similarly, a *span3* CSS class means that the column will take up $1/4$ of the page width and a *span12* CSS means that the column will take up the entire page width.

Next, for each column `<div>`, you need to specify a unique CSS ID. E.g., `id="column-1"`, `id="column-2"`, etc.

Finally, inside each column `<div>`, you need to include a Velocity template directive. This directive is responsible for rendering the portlets that have been added to each column:

```
$processor.processColumn("column-1", "portlet-column-content portlet-column-content-only")
```

The `processor.processColumn` function takes two arguments. The first is the CSS column ID and the second is a list of CSS classes. You always need to pass `"portlet-column-content"` in the second argument. If the column is the first, last, or only column in a row, you also have to pass both `"portlet-column-content"` and `portlet-column-content-[first|last|only]` in the second argument, separated by a space.

Now that we've discussed how layout template TPL files are designed, let's convert the 1 2 1 column template that we presented above into our 1 4 1 column template.

columns_1_4_1.tpl

```
1<div class="columns-1-4-1" id="main-content" role="main">
2  <div class="portlet-layout row-fluid">
3    <div class="portlet-column portlet-column-only span12" id="column-1">
4      $processor.processColumn("column-1", "portlet-column-content portlet-column-content")
5    </div>
6  </div>
7
8  <div class="portlet-layout row-fluid">
9    <div class="portlet-column portlet-column-first span3" id="column-2">
10      $processor.processColumn("column-2", "portlet-column-content portlet-column-content")
11    </div>
12
13    <div class="portlet-column portlet-column span3" id="column-3">
14      $processor.processColumn("column-3", "portlet-column-content")
15    </div>
16
17    <div class="portlet-column portlet-column span3" id="column-4">
18      $processor.processColumn("column-4", "portlet-column-content")
19    </div>
20
21    <div class="portlet-column portlet-column-last span3" id="column-5">
22      $processor.processColumn("column-5", "portlet-column-content portlet-column-content")
23    </div>
24  </div>
25
26  <div class="portlet-layout row-fluid">
27    <div class="portlet-column portlet-column-only span12" id="column-6">
28      $processor.processColumn("column-6", "portlet-column-content portlet-column-content")
29    </div>
30  </div>
31 </div>
```

1. Change the first CSS class of the root `<div>` from `columns-1-2-1` to `columns-1-4-1`.
2. We don't need to change the first row `<div>` since its already set up with a single column. We do need to change the second row `<div>` since we need to set up four equal width columns. Replace the second row `<div>` with the following:

```
<div class="portlet-layout row-fluid">
  <div class="portlet-column portlet-column-first span3" id="column-2">
    $processor.processColumn("column-2", "portlet-column-content
portlet-column-content-first")
  </div>

  <div class="portlet-column portlet-column span3" id="column-3">
    $processor.processColumn("column-3", "portlet-column-
content")
  </div>

  <div class="portlet-column portlet-column span3" id="column-4">
    $processor.processColumn("column-4", "portlet-column-
```

```

content")
    </div>

    <div class="portlet-column portlet-column-last span3" id="column-5">
        $processor.processColumn("column-5", "portlet-column-content
portlet-column-content-last")
    </div>
</div>

```

3. The find row `<div>` is set up with a single column so the only thing we need to change is its ID. Replace it with the following:

```

<div class="portlet-layout row-fluid">
    <div class="portlet-column portlet-column-only span12" id="column-6">
        $processor.processColumn("column-6", "portlet-column-content
portlet-column-content-only")
    </div>
</div>

```

Just like that, the rows and columns of the *Columns 1 4 1* layout template are arranged and sized to fit your needs.

Now that we've generated some positive Feng Shui through the design of our layout, let's increase our control over the layout by embedding portlets.

11.8. Embedding Portlets in a Layout Template

Are there portlets you need displayed in the same location on all pages using a particular layout template? Perhaps you want to prevent others from disrupting the Feng Shui you've generated with your design? You can embed portlets in layout templates, ensuring that specified portlets always display in consistent locations on your pages. Users can minimize embedded portlets but can't move or remove them. Whether instanceable or non-instanceable, core portlets and custom portlets you created with the Plugins SDK can be embedded in layout templates.

Let's embed some portlets in our *Columns 1 4 1* layout template. We'll place the *navigation portlet* and *search portlet* in the first and last columns of our layout template's middle row. Additionally, we'll embed a custom portlet in the template's upper and lower rows.

First, specify some attributes of the embedded portlet:

- **Portlet ID:** The portlet's name, `<portlet-name>`, found in `docroot/WEB-INF/portlet.xml`. For core portlets, find the name in `liferay-portal/portal-web/docroot/WEB-INF/liferay-portlet.xml`.
- **Core vs. Custom:** Specify whether the portlet is a core portlet or custom.
- **Instanceable:** Specify whether multiple instances of the portlet can exist in the portal.
- **Web Application Context** - Required for *custom* portlets only. Log into your Liferay Portal. Go to the portlet's *Look and Feel → Advanced Styling* to find the context in the Fully Qualified Portlet ID (FQPI). The context is the portion of the Portlet ID string that follows `WAR_`. The *Web Application Context* in the following figure is `myhelloworldportlet`.

Look and Feel

Portlet Configuration

Enter your custom CSS class names.

Your current portlet information is as follows:
Portlet ID: #portlet_myhelloworldportlet_WAR_myhelloworldportlet
Portlet Classes: .portlet

Here's
a
descri-
ption
of the
portlet
s
we're
embe

dding in the layout:

Portlet	ID	Row	Column	Type	Instanceable	Context	-----	-----
---	---	---	---	Navigation	71 2 1	core	yes	N/A
core	no	N/A	Hello World	my-hello-world-portlet	1 1	custom	no	
myhelloworldportlet	Goodbye World	my-goodbye-world-portlet	3 1	custom	yes			
myhelloworldportlet	---							

Here's the TPL that implements embedding of these portlets:

```
<div class="columns_1_4_1" id="main-content" role="main">
    <div class="portlet-layout row-fluid">
        <div class="portlet-column portlet-column-only" id="column-1">
            $processor.processPortlet(
                "my-hello-world-portlet_WAR_myelloworldportlet")
            $processor.processColumn("column-1",
                "portlet-column-content portlet-column-content-only")
        </div>
    </div>
    <div class="portlet-layout row-fluid">
        <div class="portlet-column portlet-column-first span3" id="column-2">
            $processor.processPortlet("71_INSTANCE_xyz1")
            $processor.processColumn(
                "column-2",
                "portlet-column-content portlet-column-content-first")
        </div>
        <div class="portlet-column span3" id="column-3">
            $processor.processColumn("column-3", "portlet-column-content")
        </div>
        <div class="portlet-column span3" id="column-4">
            $processor.processColumn("column-4", "portlet-column-content")
        </div>
        <div class="portlet-column portlet-column-last span3" id="column-5">
            $processor.processPortlet("3")
            $processor.processColumn(
                "column-5",
                "portlet-column-content portlet-column-content-last")
        </div>
    </div>
    <div class="portlet-layout row-fluid">
        <div class="portlet-column portlet-column-only" id="column-6">
```

```

$processor.processPortlet(
    "my-goodbye-world-portlet_WAR_myelloworldportlet_INSTANCE_jkl1")
$processor.processColumn(
    "column-6",
    "portlet-column-content portlet-column-content-only")
</div>
</div>
</div>

```

What would a page using our *Columns 1 4 1* layout template look like? Check out the following

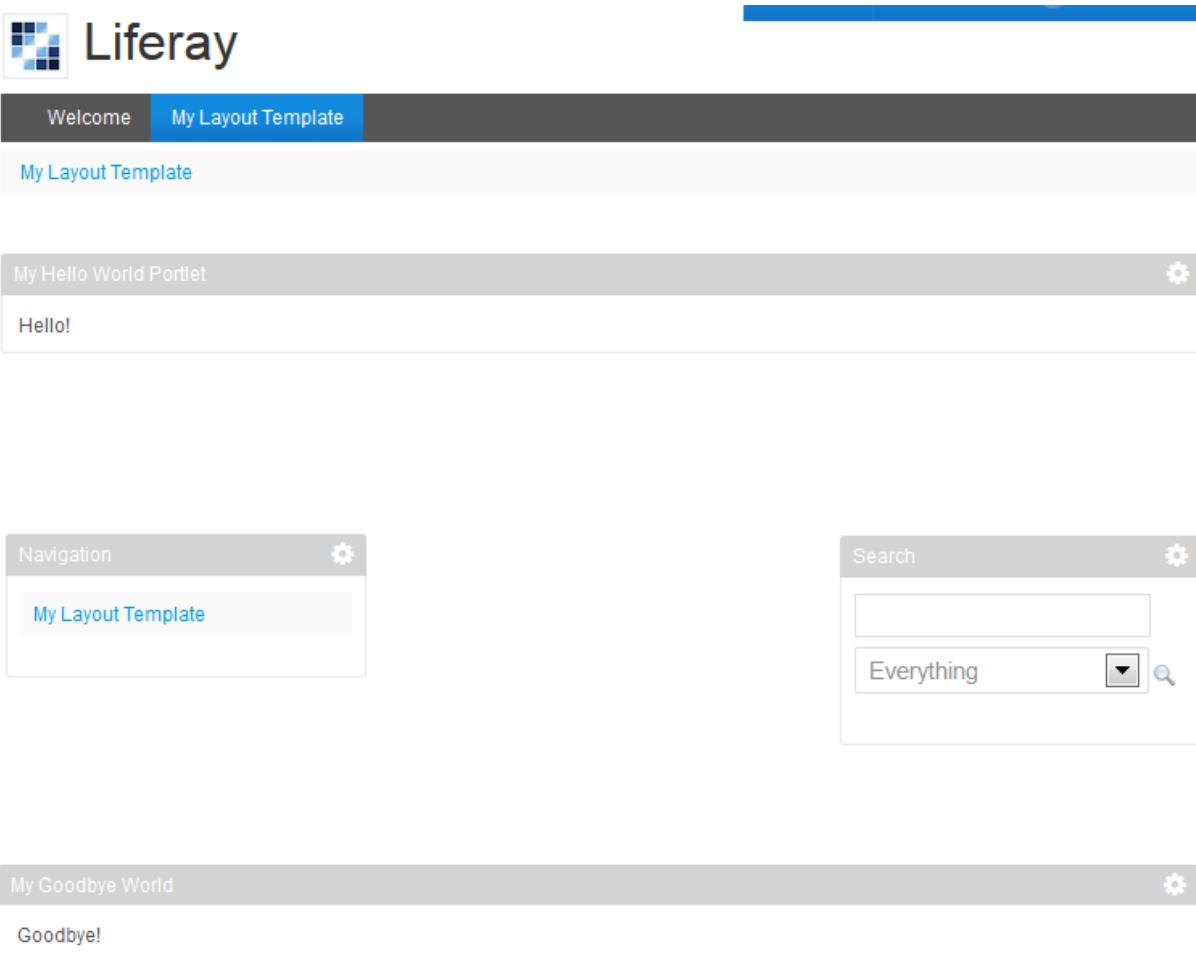


figure for a screenshot of our layout template with its embedded portlets.

See how simple it is to embed portlets in your pages?

Wouldn't it be nice to have an organized reference of available layout template variables? You're in luck! We'll dive into available variables next!

11.9. Variables Available to Layout a Template

A number of variables are available for you to use in your custom TPL files. For your

convenience, we've listed all of them in the following table.

Variable	Type	Description
<i>processor</i>		<i>com.liferay.portal.layoutconfiguration.util.velocity.TemplateProcessor</i> [Javadoc](http://docs.liferay.com/portal/6.2/javadocs-all/com/liferay/portal/layoutconfiguration/util/velocity/TemplateProcessor.html) request javax.servlet.http.HttpServletRequest
<i>themeDisplay</i>	<i>com.liferay.portal.theme.ThemeDisplay</i> [Javadoc](http://docs.liferay.com/portal/6.2/javadocs/com/liferay/portal/theme/ThemeDisplay.html) company com.liferay.portal.model.Company Javadoc user	
<i>com.liferay.portal.model.User</i>	[Javadoc](http://docs.liferay.com/portal/6.2/javadocs/com/liferay/portal/model/User.html) realUser	
<i>com.liferay.portal.model.User</i>	Javadoc layout <i>com.liferay.portal.model.Layout</i> [Javadoc](http://docs.liferay.com/portal/6.2/javadocs/com/liferay/portal/model/Layout.html) layouts java.util.List <i>plid</i> java.lang.Long layoutTypePortlet	
<i>com.liferay.portal.model.LayoutTypePortlet</i>	Javadoc portletGroupId java.lang.Long locale java.util.Locale timeZone java.util.TimeZone theme	
<i>com.liferay.taglib.util.VelocityTaglib</i>	Javadoc	
<i>colorScheme</i>	<i>com.liferay.portal.model.ColorScheme</i> [Javadoc](http://docs.liferay.com/portal/6.2/javadocs/com/liferay/portal/model/ColorScheme.html) portletDisplay com.liferay.portal.theme.PortletDisplay Javadoc ---	

Now your layout template toolbox is complete.

11.10. Summary

In this chapter, you learned how to customize the look and feel of your Liferay Portal by creating custom themes. During this process, you created your own theme, learned about its directory structure, and discovered the value of style inheritance from a parent theme. You also learned about Liferay's JavaScript library, AlloyUI, and how to make your theme configurable by adding settings that portal administrators can manage within Liferay. Your CSS options, including color schemes, and predefined settings for your theme, were discussed to round out your understanding of theme development.

We also created layout templates, arranged their rows and columns, and embedded portlets in them. Congratulations on mastering the fundamentals of Liferay's layout templates, but be careful. If your Feng Shui skills become widely known, your friends may ask you to re-arrange their living room furniture!

If you're up for it, let's learn how to customize core Liferay portlets using hooks--sounds "catchy", right??

12. Customizing and Extending Functionality with Hooks

Liferay Hooks are the best plugin for customizing Liferay's core features. If possible, use hooks whenever you need to override Liferay's core functionality. It's possible to use Ext plugins for

many of the same tasks, but hooks are hot-deployable and more forward compatible, so we urge you to use them preferentially.

In this chapter, we'll learn how to create hooks and we'll explore their most common uses.

We'll cover the following topics:

- Creating a Hook
- Overriding Web Resources
- Customizing JSPs by Extending the Original
- Customizing Sites and Site Templates with Application Adapters
- Performing a Custom Action
- Overriding and Adding Struts Actions
- Extending and Overriding *portal.properties*
- Overriding a Portal Service
- Overriding a *Language.properties* file
- Extending the Indexer Post Processor
- Supporting Right-to-Left Languages in Plugins
- Other Hooks

As with portlets, layout templates, and themes, the easiest way to create and manage hooks is via Liferay IDE or Developer Studio. However, if you don't want to use an IDE, you can use the terminal. We'll demonstrate how to create and deploy a hook using both Liferay IDE and the terminal.

12.1. ***Creating a Hook***

Regardless of whether you use Liferay IDE or your terminal to create hooks, hooks projects are stored in the Plugins SDK's `hooks` directory.

Using Liferay IDE:

1. Go to *File* → *New* → *Liferay Project*.
2. Assign a project name and display name. To demonstrate, we'll use *example-hook* and *Example* for the project name and display name, respectively. Notice that upon entering *example-hook* as the project name, the wizard conveniently inserts *Example* in grayed-out text as the plugin's default display name. The wizard derives the default display name from the project name, starts it in upper-case, and leaves off the plugin type suffix *Hook* because the plugin type is automatically appended to the display name in Liferay Portal. The IDE saves the you from repetitively appending the plugin type to the display name; in fact, the IDE ignores any plugin type suffix if you happen to append it to the display name.

Enter the following values for the project name and display name:

- **Project name:** *example-hook*
- **Display name:** *Example*

3. Select the build type, Plugins SDK, and Liferay runtime.

If you select the Maven build type, you'll be prompted to enter an artifact version, group ID, and

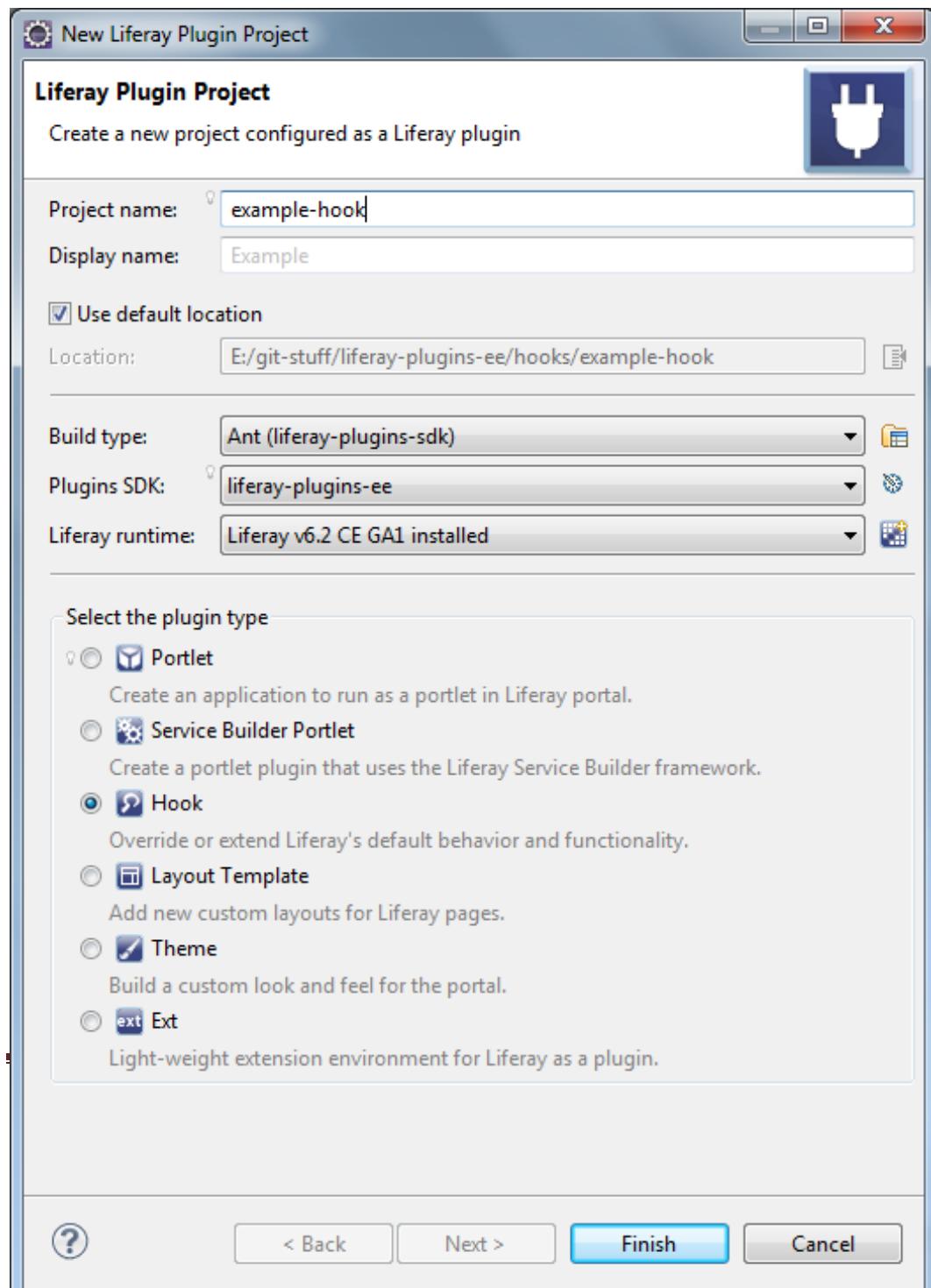
active profile for your project. See *Developing Plugins Using Maven* for more information. Otherwise, select the Ant build type, a Plugins SDK and a Liferay runtime.

For this demonstration, make the following selections:

- › **Build type:** *Ant*
- › **Plugins SDK:** [a configured Plugins SDK]
- › **Liferay runtime:** [a configured Liferay runtime]

For more information, see sections *Setting Up the Liferay Plugins SDK* and *Liferay Portal Runtime and Server Setup* from *Developing Apps with Liferay IDE*

4. Select the *Hook* Plugin Type.



5. Click *Finish*.

Figure 12.1 shows the values you specified for the hook plugin.

The Plugins SDK automatically names the hook by appending "-hook" to the project name. With Liferay IDE, you can create a hook in a completely new plugin project or create a hook in an existing plugin project. Use *File → New*

→ *Liferay Project* to create a new plugin project and *File → New → Liferay Hook* to create a hook in an existing plugin project.

Using the terminal: Navigate to your Plugins SDK directory in a terminal and enter the appropriate command for your operating system:

1. In Linux and Mac OS X, enter

```
./create.sh example "Example"
```

2. In Windows, enter

```
create.bat example "Example"
```

A BUILD SUCCESSFUL message from Ant tells you there's a new folder named `example-hook` inside the Plugins SDK's `hooks` folder. The Plugins SDK automatically named the hook by appending "-hook" to the project name.

Now that you've created a hook, let's go ahead and deploy it.

12.1.1. Deploying the Hook

Using Liferay IDE: Click and drag your hook project onto your server. Upon deployment, your server displays messages indicating that your hook was read, registered and is now available for use.

```
Reading plugin package for example-hook  
Registering hook for example-hook  
Hook for example-hook is available for use  
Voila! Your hook deployed.
```



Note: If the Liferay server prints the following message to your console, the *Marketplace Portlet* and *Portal Compatibility Hook* must not already be deployed on your server.

Plugin example-hook requires marketplace-portlet, portal-compat-hook
For Liferay 6.2.0 CE GA1, you can fork and clone Liferay's *liferay-plugins* project from GitHub, checkout the respective branch and/or tag, and deploy each plugin. You can install the Plugins SDK in Liferay IDE and import each plugin and deploy them. Here is information on each of the plugins:

- *Marketplace Portlet* (`marketplace-portlet`) - is available at `liferay-plugins/portlets/marketplace-portlet`.
- *Portal Compatibility Hook* (`portal-compat-hook`) - is available at `liferay-plugins/hooks/portal-compat-hook`.

If you ever need to redeploy your plugin while in Liferay IDE, right-click your plugin's icon located underneath your server and select *Redeploy*.

Using the terminal: Open a terminal window in your `hooks/example-hook` directory and enter

```
ant deploy
```

A BUILD SUCCESSFUL message indicates your hook is now being deployed. If you switch to

the terminal window running Liferay, in a few seconds you should see the message "Hook for example-hook is available for use".

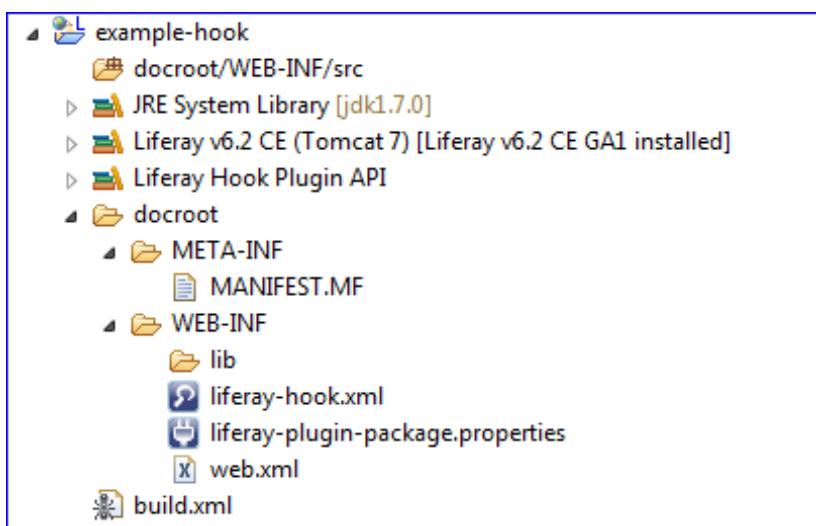


Note: When we created portlets and themes, they were fully functional upon deployment. Hooks aren't like that because they're Liferay customizations. The default customization is the original implementation!

12.1.2. Anatomy of the Hook

To make your hook useful, you need to customize something in Liferay. You begin by mirroring the structure of Liferay's code that you plan to customize. A hook plugin is built to contain this:

- example-hook/
 - docroot/WEB-INF/src/
 - docroot/
 - META-INF/
 - MANIFEST.MF
 - WEB-INF/
 - lib/
 - liferay-hook.xml
 - liferay-plugin-package.properties
 - web.xml
 - build.xml



In Liferay IDE's *Package Explorer*, here's what the hook structure looks like:

The particular files you'll work on depend on the Liferay features you're overriding with your hook. We'll start by making one of the most common hook plugin customizations: a customization of Liferay's web resources.

12.2. Overriding

Web Resources

Hooks are commonly used to override web resources, found in `portal-web` in Liferay's source. You can use a hook to override JSP files, JSPF files, JavaScript files, CSS files, or images.

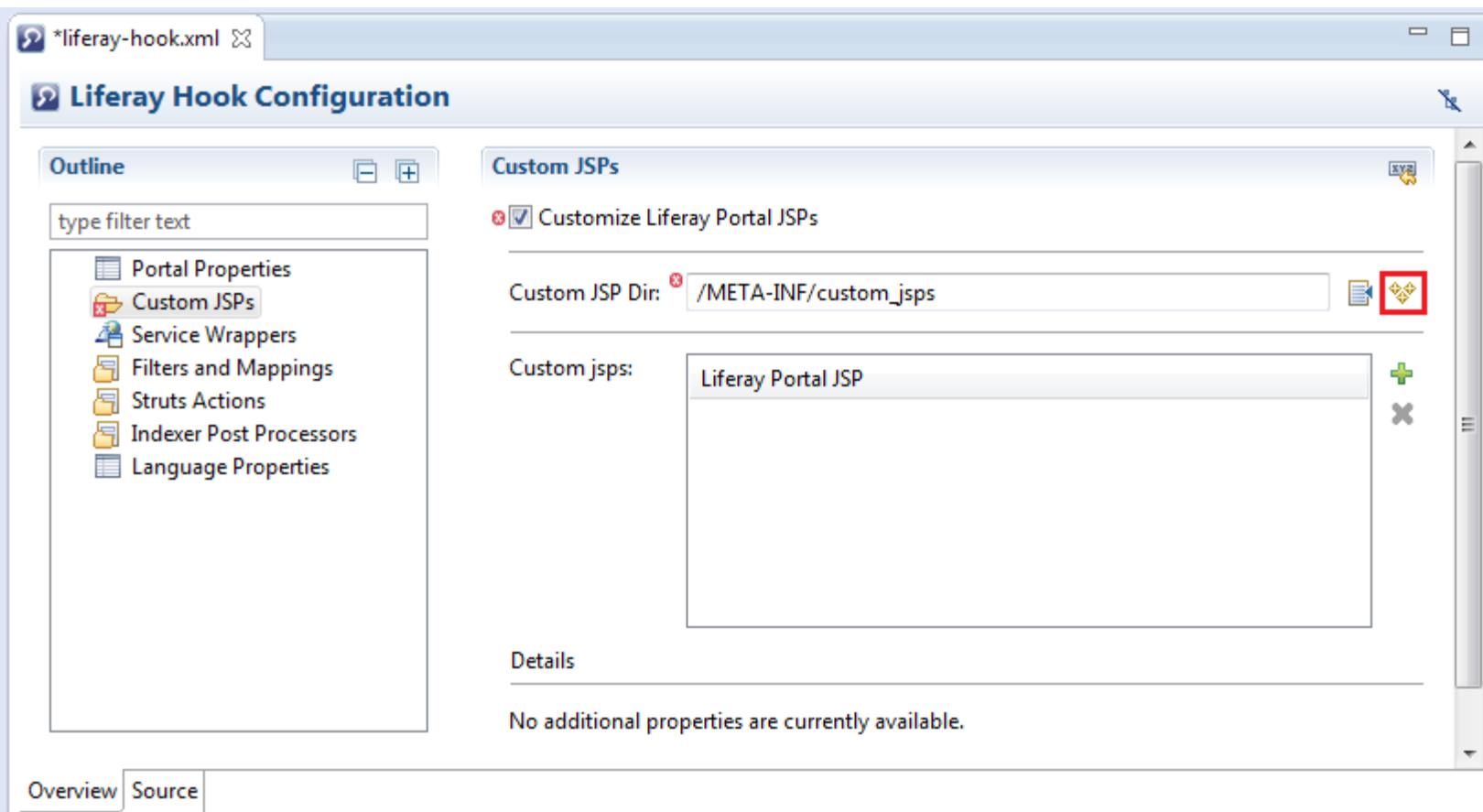


Important: Some resources have additional requisites:

- **JSPF:** Changes won't take effect unless you modify the JSP that includes it.
- **CSS:** When modifying a CSS file imported by another CSS file, the changes won't take effect unless you modify the parent CSS file (usually main.css).

Replacing a portal JSP is a simple task with hooks. Let's create and deploy a hook to modify your portal's *Terms of Use* page.

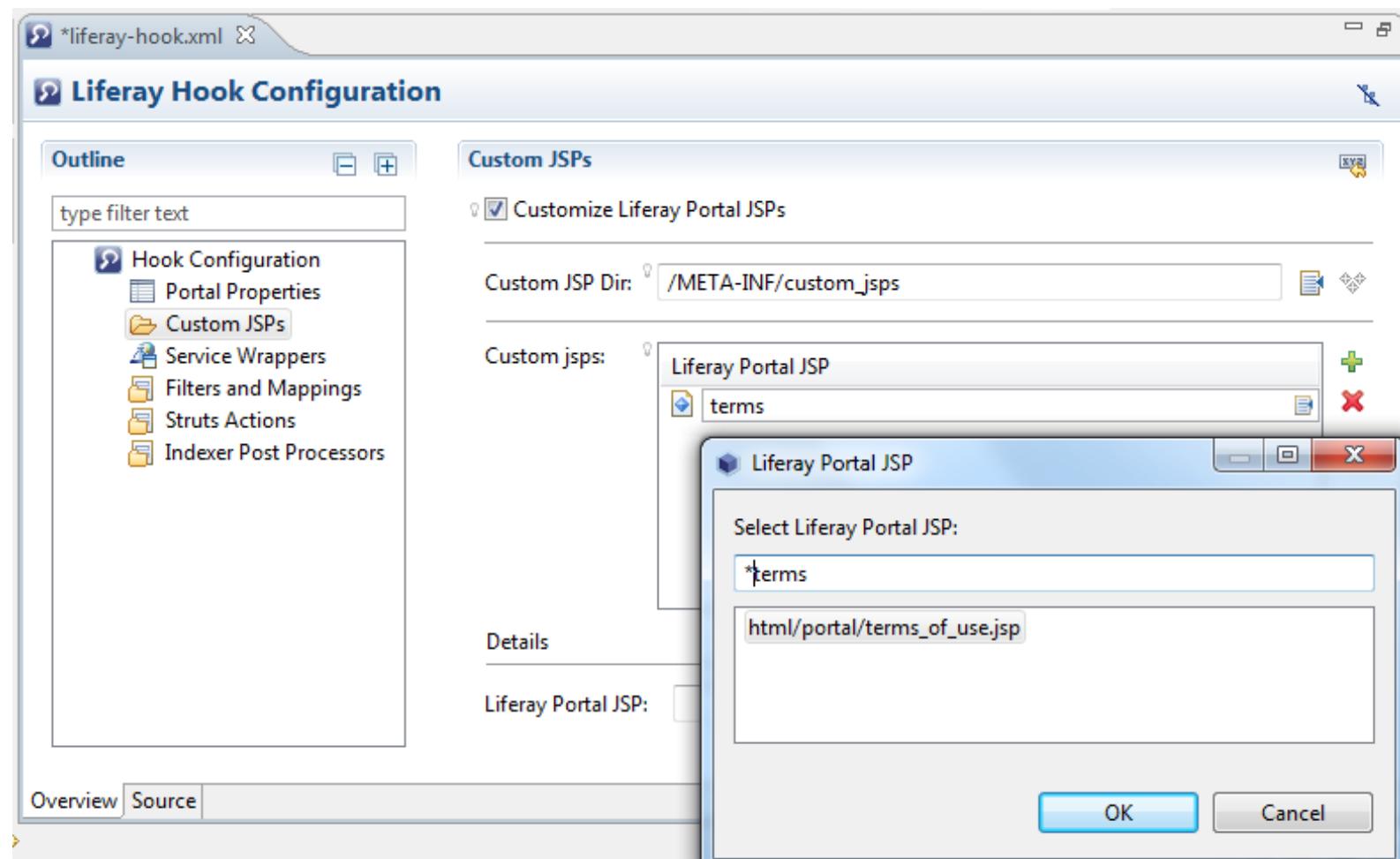
1. Use the hook project we created earlier or create a new hook project.
2. Open the `liferay-hook.xml` file from your project's `docroot/WEB-INF` folder in Liferay IDE. By default, it opens in *Overview* mode. This mode, gives you a graphical user interface for developing your hook. You can toggle between overview mode and source mode via their respective tabs for the `liferay-hook.xml` file's editor.
3. Select the *Custom JSPs* folder from the outline to bring up the custom JSP options. Select the checkbox *Customize Liferay Portal JSPs* and create the default custom JSP folder



/META-INF/custom_jsp, by clicking the icon that has the three yellow diamonds.

4. Add to the listing of custom JSPs by clicking the plus icon and specifying Portal's

`html/portal/terms_of_use.jsp` file. Hint, the browse icon on the right-hand side within the custom JSP text field simplifies finding the JSP you want to customize. It lets you scroll through the JSPs that are accessible and lets you specify key words to



narrow your search.

5. Open your hook's `docroot/META-INF/custom_jsp/html/portal/terms_of_use.jsp` file and modify it as necessary.

Note, lots of errors will show in the editor because the resources used in the JSP (e.g., `PortalUtil`) are not available in the project; but they'll be available from the portal once the hook plugin is deployed to the portal server.

6. Deploy your hook and wait until it is deployed successfully.
7. Create a new user and log in. The *Terms of Use* page should include the changes you made above.

Now there are two *Terms of Use* JSP files in the `liferay-portal-[version]/tomcat-[tomcat-version]/webapps/ROOT/html/portal` directory. One is called `terms_of_use.jsp` and another `terms_of_use.portal.jsp`. `terms_of_use.jsp` is your hook's version, while `terms_of_use.portal.jsp` is the original. To revert back to the original, undeploy your hook. Your replacement JSP is removed, and the `.portal.jsp` file is automatically renamed, taking its place. You can override any JSP in the Liferay core, while retaining the ability to easily revert your changes. However, it's not possible to override the same JSP from multiple hooks; Liferay won't know which version to use.



Note: We don't recommend changing Liferay's *Terms of Use* with a hook. You can replace the *Terms of Use* with a piece of web content simply by specifying values for these two properties in `portal-ext.properties`:

```
terms.of.use.journal.article.group.id=
terms.of.use.journal.article.id=
```

Although our example hook doesn't provide any new functionality, it demonstrates how to override Liferay's JSP files.

Next, we'll look at a different way to customize a JSP.

12.3. ***Customizing JSPs by Extending the Original***

If we can replace a JSP with a hook plugin, why learn another way to accomplish the same thing? Good question. Let's say you want to preserve the original JSP's content and functionality, but you want to add more to the JSP. And when you upgrade Liferay, you want to benefit from any changes made to that upgraded JSP. Well, you can; simply include the original JSP and then add more stuff to it.

Here's an example that customizes the search page of the Blogs portlet. Specifically, it adds helpful text to aid the user in searching for content. Since this technique involves string manipulation, it's mainly useful for making a small number of changes to a JSP.

1. Use the hook project we created earlier or create a new hook project.
2. Open the `liferay-hook.xml` file from your project's `docroot/WEB-INF` folder in Liferay IDE and select the file's *Overview* mode tab.
3. Select the *Custom JSPs* folder from the outline to bring up the custom JSP options. Select the checkbox *Customize Liferay Portal JSPs* and create the default custom JSP folder `/META-INF/custom_jsp`s by clicking the icon that has the three yellow diamonds.

Add to the listing of custom JSPs by clicking the plus icon and specifying Portal's `html/portlet/blogs/search.jsp` file. Hint, the browse icon on the right-hand side within the custom JSP text field simplifies finding the JSP you want to customize.

Click OK and save the `liferay-hook.xml` file. Liferay IDE pulls a copy of the Liferay Portal JSP into your project so you can modify it.

4. Open the JSP file `docroot/META-`

INF/custom_jsp/html/blogs/search.jsp that Liferay IDE pulled into your project.

5. Replace the JSPs code with the following:

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

<%@ page import="com.liferay.portal.kernel.util.StringUtil" %>

<liferay-util:buffer var="html">
    <liferay-util:include page="/html/portlet/blogs/search.portal.jsp" />
</liferay-util:buffer>

<%
html = StringUtil.add(
    html,
    "Didn't find what you were looking for? Refine your search and " +
        "try again!",
    "\n");
%>

<%= html %>
```

Notice how this code assigns the original JSP's HTML content to the variable `html`. We proceed to add some more content of our own to that HTML and then display it.

6. Deploy the hook plugin and add the Blogs portlet to a page.

7. Add a blog and then use the Blog portlet's search.

Your custom message now shows below the search results.

Next, we'll explore application adapters and what they can do for your sites and site templates.

12.4. Customizing Sites and Site Templates with Application Adapters

The JSP hooks that we've demonstrated so far are scoped to the portal. What if you need to customize specific sites without propagating the customizations throughout the entire portal? You can! *Application Adapters* are special hooks that let you make changes at the site level. They are used for overriding JSPs.

There's a Sample Application Adapter in the Liferay Plugins Repository. How do we build an Application Adapter of our own?

To create an Application Adapter, you need a hook with custom JSPs, and you need to turn the hook's global custom JSP setting off. You can do this by configuring your `liferay-hook.xml` with the following directives:

```
<custom-jsp-dir>/META-INF/custom_jsp/</custom-jsp-dir>
<custom-jsp-global>false</custom-jsp-global>
```

When you deploy your hook, Liferay installs the Application Adapter under the name of the hook. An Application Adapter hook named *Foo* becomes available to sites and site templates under the name *Foo Hook*.

Now, let's discuss the perks of including the original JSP when overriding it.

12.4.1. Including an Original JSP

If you override a JSP from the portal, we recommend you include the original JSP (when possible).

As we already demonstrated, including the original JSP file for global hooks is accomplished by referencing the original JSP file from a `<liferay-util:include>` tag and appending the suffix `.portal.jsp` to the original file's name. Here's what including the original Navigation portlet's view JSP in a global hook looks like:

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

<liferay-util:include page="/html/portlet/navigation/view.portal.jsp" />
```

For Application Adapter hooks, we include the original JSP by setting the `<liferay-util:include>` tag's `useCustomPage` attribute to `false`, as below:

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

<liferay-util:include
    page="/html/portlet/navigation/view.jsp"
    useCustomPage="false"
/>
```

The view JSP is specified as `view.jsp`, *not* `view.portal.jsp` as for global hooks. In the next section, we'll create and test an application adapter.

12.4.2. Creating an Application Adapter

Let's create an Application Adapter hook named `example-hook`. It will override the Navigation portlet's `view.jsp`, while including the original Navigation portlet's JSP with some custom text after its contents.

Here's how we do it:

1. Modify your hook's `liferay-hook.xml` to specify the location of your custom JSP and set the global custom JSP setting to `false`:

```
<hook>
    <custom-jsp-dir>/META-INF/custom_jsps</custom-jsp-dir>
    <custom-jsp-global>false</custom-jsp-global>
</hook>
```

2. Create a `docroot/META-INF/custom_jsps/html/portlet/navigation` directory in your hook project, create a new `view.jsp` file in this directory, and insert code to include the original JSP:

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

<liferay-util:include
    page="/html/portlet/navigation/view.jsp"
    useCustomPage="false"
/>
```

```
<p>  
This was modified by the Example Application Adapter.  
</p>
```

3. Deploy your Application Adapter hook plugin.
4. In your web browser, navigate to the Liferay site where you'll use the Application Adapter.

Site Settings

Description

Site ID

Active

Membership Options

Membership Type

 ▾

Allow Manual Membership Management

Pages

Public Pages

Private Pages

Configuration

Application Adapter

 ▾

5. Select *Admin → Configuration* to access the Site Settings section of the Site Administration interface. From the *Application Adapter* field's drop-down selector menu, select *Example*. Then click *Save*.

6. Navigate to your site's pages, add the Navigation portlet to a page, and make sure that the modification message from your Application Adapter hook plugin's *view.jsp* file is displayed there.

7. Navigate to a different site's Navigation portlet to verify that only the content of the portlet's *original view.jsp* file displays.

Using Application Adapter hook plugins to override Liferay's core functionality at the site scope is easy!

You can also apply Application Adapters to Site Templates.

Suppose you want to make an Enterprise Resource Planning (ERP) solution for a company's departments. Your ERP solution requires an extension of Liferay's Wiki portlet so you implement that extension as an Application Adapter. Then you incorporate the Application Adapter in a Site Template (named *ERP site*) for the company's ERP sites. The company's administrative user creates the sites by going to *Control Panel → Sites* and adding sites based on the "ERP site" template. The added sites include your Application Adapter

automatically.

That's it for Application Adapters. Let's learn about performing custom actions through hooks.

12.5. Performing a Custom Action

Hooks are useful for triggering custom actions on common portal events, like user login or system startup. The actions for each of these events are defined in `portal.properties` so we need to extend this file to create a custom action. Hooks make this a simple task.

1. In your hook project, create the directory `docroot/WEB-INF/src/com/liferay/sample/hook` and create a file called `LoginAction.java` inside it with this content:

```
package com.liferay.sample.hook;

import com.liferay.portal.kernel.events.Action;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginAction extends Action {
    public void run(HttpServletRequest req, HttpServletResponse res) {
        System.out.println("## My custom login action");
    }
}
```

2. Create a `portal.properties` file inside your hook project's `docroot/WEB-INF/src` folder with this content:

```
login.events.pre=com.liferay.sample.hook.LoginAction
```

3. Edit your `docroot/WEB-INF/liferay-hook.xml` file, adding the following line above `<custom-jsp-dir>`:

```
<portal-properties>portal.properties</portal-properties>
```

4. Redeploy your hook. Once deployment is complete, log out and back in, and you should see your custom message, `## My custom login action`, displayed in the terminal window running Liferay.

Custom action hook plugins aren't limited to the login event. You can define custom actions for other events, too. For actions that require access to the `HttpServletRequest` object, extend `com.liferay.portal.kernel.events.Action`; for others, extend `com.liferay.portal.struts.SimpleAction`.



Important: For better forward compatibility, use hooks to customize Struts actions rather than Ext plugins.

Extending and overriding portal properties is just as easy, so let's do that next.

12.6. Extending and Overriding portal.properties

In our hook that created a custom login action, we modified the `login.events.pre` portal property. This property accepts *multiple* values, so our value was appended to the existing `login.events.pre` values. We can repeatedly modify the property from additional hooks because it accepts multiple values. Some portal properties only accept a *single* value, such as the `terms.of.use.required` property, which is either `true` or `false`. Only modify single value properties from a single hook plugin; otherwise Liferay won't know which value to use.



Note: Hooks support customizing a specific list of predefined properties. For a list of portal properties that can be overridden via hook, see the `liferay-hook_6_2_0.dtd`. In addition to defining custom actions, hooks can override portal properties to define model listeners, validators, generators, and content sanitizers. If you want to customize a property that's not found in this list, you must use an Ext plugin (see Advanced Customization with Ext Plugins). For more information about the properties themselves, you can view an online version of Liferay's portal properties file.

Next, let's learn how to override and add Struts actions from a hook plugin.

12.7. Overriding and Adding Struts Actions

Do you want to add a new Struts action to Liferay or override existing Struts actions? *Struts action hooks* let you do just that.

Let's consider the interfaces used for Struts actions. There are two:

- `com.liferay.portal.kernel.struts.StrutsAction`
- `com.liferay.portal.kernel.struts.StrutsPortletAction`

The `StrutsAction` interface is for regular Struts actions from the portal, like `/c/portal/update_email_address`. The `StrutsPortletAction` interface is used for similar Struts actions, but from portlets.

Struts actions are defined as classes, and they're all connected in a `struts-config.xml` file. The `struts-config.xml` for a Liferay Portal instance running on Apache Tomcat can be found in the `liferay-portal-[version]/tomcat-[version]/webapps/ROOT/WEB-INF` directory. The `struts-config.xml` file links actions to specific JSP pages. Each action performs a specific task and then returns a *forward*, an object containing a name and path. The forward defines the page the portal sends the user to after the action completes. When a user submits a form that maps to one of these actions, the action class is loaded, executed, and returns a forward.

A Struts action hook can wrap or override existing Struts actions or create a new Struts path; we'll demonstrate both here. We'll override the Struts actions in the `struts-config.xml` file by using a Struts action hook to point to a custom class. Then we'll create a new Struts path: `/c/portal/sample` and navigate to it. Let's get started!

First, let's override the Sign In portlet's Struts action using the example-hook project that we've been using in this chapter.

Here's the current action in your portal's `struts-config.xml` file:

```
<action path="/login/login"
       type="com.liferay.portlet.login.action.LoginAction">

    <forward
        name="portlet.login.login"
        path="portlet.login.login"
    />
    <forward
        name="portlet.login.login_redirect"
        path="portlet.login.login_redirect"
    />
</action>
```

1. Navigate to your `example-hook/docroot/WEB-INF` folder and open `liferay-hook.xml`.
2. Insert the following code between the `<hook>...</hook>` tags:

```
<portal-properties>portal.properties</portal-properties>
<custom-jsp-dir>META-INF/custom_jsps</custom-jsp-dir>
<struts-action>
    <struts-action-path>/portal/sample</struts-action-path>
    <struts-action-impl>
        com.liferay.sample.hook.action.ExampleStrutsAction
    </struts-action-impl>
</struts-action>
<struts-action>
    <struts-action-path>/login/login</struts-action-path>
    <struts-action-impl>
        com.liferay.sample.hook.action.ExampleStrutsPortletAction
    </struts-action-impl>
</struts-action>
```

3. Create a new package `com.liferay.sample.hook.action` in your `example-hook/docroot/WEB-INF/src` folder.
4. In your new package, create a class named `ExampleStrutsPortletAction`, which will wrap the login portlet Struts action. Insert the following code:

```
package com.liferay.sample.hook.action;

import com.liferay.portal.kernel.struts.BaseStrutsPortletAction;
import com.liferay.portal.kernel.struts.StrutsPortletAction;
import com.liferay.portal.theme.ThemeDisplay;
import com.liferay.portal.kernel.util.WebKeys;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.PortletConfig;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import javax.portlet.ResourceRequest;
```

```

import javax.portlet.ResourceResponse;

public class ExampleStrutsPortletAction extends BaseStrutsPortletAction {

    public void processAction(
        StrutsPortletAction originalStrutsPortletAction,
        PortletConfig portletConfig, ActionRequest actionRequest,
        ActionResponse actionResponse)
    throws Exception {
        ThemeDisplay themeDisplay =
            (ThemeDisplay)actionRequest.getAttribute(WebKeys.THEME_DISPLAY);

        Long currentuser = themeDisplay.getUserId();

        if (currentuser != null) {
            System.out.println("Wrapped /login/ action2");

        }
        originalStrutsPortletAction.processAction(
            originalStrutsPortletAction, portletConfig, actionRequest,
            actionResponse);
    }

    public String render(
        StrutsPortletAction originalStrutsPortletAction,
        PortletConfig portletConfig, RenderRequest renderRequest,
        RenderResponse renderResponse)
    throws Exception {

        System.out.println("Wrapped /login/ action");

        return originalStrutsPortletAction.render(
            null, portletConfig, renderRequest, renderResponse);
    }

    public void serveResource(
        StrutsPortletAction originalStrutsPortletAction,
        PortletConfig portletConfig, ResourceRequest resourceRequest,
        ResourceResponse resourceResponse)
    throws Exception {

        originalStrutsPortletAction.serveResource(
            originalStrutsPortletAction, portletConfig, resourceRequest,
            resourceResponse);
    }
}

```

5. Create a new class named `ExampleStrutsAction` in the `com.liferay.sample.hook.action` package. It will implement your new portal Struts action. Insert the following code:

```

package com.liferay.sample.hook.action;

import com.liferay.portal.kernel.struts.BaseStrutsAction;
import com.liferay.portal.kernel.util.ParamUtil;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ExampleStrutsAction extends BaseStrutsAction {

    public String execute(
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        String name = ParamUtil.get(request, "name", "World");

        request.setAttribute("name", name);

        return "/portal/sample.jsp";
    }

}

```

We've overridden the `execute(HttpServletRequest, HttpServletResponse)` method of `BaseStrutsAction`, but not the `execute(StrutsAction, HttpServletRequest, HttpServletResponse)` method. The original Struts action's `execute()` method is ignored. That's fine for our example.



Best practice: When overriding an existing Struts action, it's usually best to override the method that takes the original Struts action handle as a parameter and execute that original Struts action. Think of the original action as a servlet filter or aspect. If you override the method that *takes* the original action handle as a parameter and don't explicitly execute it, the original action won't be executed. If you override the `execute` method that *does not take* the original action as a parameter, you are ignoring the original action and it won't be executed.

That's it for overriding the Struts actions! Now let's get our new Struts path working.

1. Create a `sample.jsp` file in the `example-hook/docroot/META-INF/custom_jsp/html/portal` directory. Insert the following code:

```

<%
String name = (String)request.getAttribute("name");
%>
Hello <%= name %>

```

2. Add `/portal/sample` to your portal's list of paths that don't require authentication by copying your existing `auth.public.paths` property assignment from your portal's `portal.properties` into your `portal-ext.properties` file and adding `/portal/sample` to the end of the value list. It looks similar to the assignment below:

```
auth.public.paths=\
/asset/get_categories,\n\
...\n\
/wiki/rss,\n\
/portal/sample
```

3. Restart your portal server.

Congratulations! Your Struts action hook plugin is complete! Now when you access the *Sign In* portlet, this message prints to your console:

Wrapped /login/ action

When you actually log in, this message prints to your console:

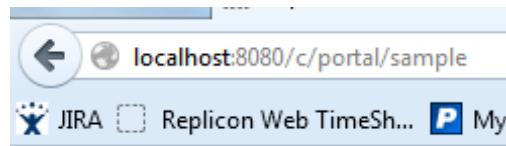
Wrapped /login/ action2

Wrapped /login/ action

Both custom Struts actions are executed via your Struts action hook!

Try your new Struts path by accessing it from your browser (e.g.,

<http://localhost:8080/c/portal/sample>)



Let's continue our hooks expedition by overriding a portal service.

Hello World!

12.8. Overriding a Portal Service

All the functionality provided by Liferay is enclosed in a layer of services that are accessed by the controller layer in portlets. This is a standard architecture, and it lets you change how a core portlet of Liferay behaves without changing the portlet itself; you're customizing the backend services that the portlet uses. You can leverage this architecture to customize portal service behavior, and hook plugins are your tool for doing so.

When extending Liferay Portal with hooks, you should try to avoid implementing the portal's interfaces directly. In some cases, patches are added to the interfaces in fix packs to fix an issue (e.g., adding a new method to a service). If you implement the API directly, a patch can break your customization. However, if you extend the basic implementation, a patch won't break your customization. Therefore, the best practice is to extend the Liferay Portal's base implementations. For example, if you'd like to modify the implementation of the `UserLocalService` interface, then extend `UserLocalServiceWrapper`. If you'd like to modify the `SanitizerUtil` class, then extend `BaseSanitizer`.



Tip: Your `portal.properties` file also provides options to extend portal services. For example, you can extend `BaseSanitizer` to use a custom sanitizer by setting the `sanitizer.impl` property. By setting this property to your custom sanitizer class, you're extending the `BaseSanitizer` already included in Liferay Portal.

Liferay generates dummy wrapper classes for all its services. For example,

`UserLocalServiceWrapper` is created as a wrapper for `UserLocalService`, a service for adding, removing, and retrieving user accounts. To modify the functionality of `UserLocalService` from our hook, create a class that extends `UserLocalServiceWrapper`, override the methods you want to modify, and instruct Liferay to use your service class instead of the original.

1. Inside your example-hook project's `/docroot/WEB-INF/src/com/liferay/sample/hook` folder, create a new file called `MyUserLocalServiceImpl.java` with the following content:

```
package com.liferay.sample.hook;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.model.User;
import com.liferay.portal.service.UserLocalService;
import com.liferay.portal.service.UserLocalServiceWrapper;

public class MyUserLocalServiceImpl extends UserLocalServiceWrapper {

    public MyUserLocalServiceImpl(UserLocalService userLocalService) {
        super(userLocalService);
    }

    public User getUserById(long userId)
        throws PortalException, SystemException {

        System.out.println(
            "## MyUserLocalServiceImpl.getUserById(" + userId + ")");
    }

    return super.getUserById(userId);
}
}
```



Note: The wrapper class (`MyUserLocalServiceImpl` in this example) will be loaded in the hook's class loader. That means it will have access to any other class included in the same WAR file, but *not* the *internal* classes of Liferay.

2. Edit `liferay-hook.xml`, located in the `example-hook/docroot/WEB-INF` directory, by adding the following after `</custom-jsp-dir>`:

```
<service>
    <service-type>
        com.liferay.portal.service.UserLocalService
    </service-type>
    <service-impl>
        com.liferay.sample.hook.MyUserLocalServiceImpl
    </service-impl>
</service>
```

Redeploy your hook and refresh your browser. In the terminal window running Liferay, you

should see `## MyUserLocalServiceImpl.getUserById(...)` messages displayed by your hook.

There are other Liferay services that you may need to extend to meet advanced requirements:

- **OrganizationLocalService**: Adds, deletes and retrieves organizations. Also assigns users to organizations and retrieves the list of organizations of a given user.
- **GroupLocalService**: Adds, deletes and retrieves sites.
- **LayoutLocalService**: Adds, deletes, retrieves and manages pages of sites, organizations and users.

For a complete list of available services and their methods, check the Liferay Portal 6.2 Javadocs or access the Javadocs for your version of Liferay at <http://docs.liferay.com/portal> and click on the *javadocs* link.

Now that you know how to override a portal service, let's learn how to override a `Language.properties` file.

12.9. Overriding a `Language.properties` File

Hooks let you change any of the messages displayed by Liferay to suit your needs. To do so, create a `Language` file for the locale of the messages you want to customize, then refer to your file from your `liferay-hook.xml` file. For example, to override the Spanish and French message translations of portal's `Language.properties` file, create `Language` files for them in the same directory and refer to these language files in your `liferay-hook.xml` file, like this:

```
<hook>
  ...
  <language-properties>content/Language_es.properties</language-properties>
  <language-properties>content/Language_fr.properties</language-properties>
  ...
</hook>
```



Tip: Check the DTD of each Liferay XML file, you modify for the elements and attributes that can be included in the XML and the specified order for those elements. You can find the Liferay DTDs online here:
<http://docs.liferay.com/portal/6.2/definitions>.

Great! You now know how to customize language keys. Next, let's discuss extending your Indexer Post Processor.

12.10. Extending the Indexer Post Processor

Would you like to modify the search summaries, indexes, and queries available in your portal instance? Developing an Indexer Post Processor hook lets you do just that. The indexer hook implements a post processing system on top of the existing indexer to allow plugin hook developers to modify their search, index, and query capabilities. Let's run through a simple example to preview what you can accomplish with an indexer hook. For our example, we're going to add *Job Title* into the User Indexer so we can search for users by their Job Title.

1. In your existing example-hook project, open the `liferay-hook.xml` file and insert the following lines before the closing `</hook>` tag:

```
<indexer-post-processor>
    <indexer-class-name>com.liferay.portal.model.User</indexer-class-name>
        <indexer-post-processor-
impl>com.liferay.hook.indexer.SampleIndexerPostProcessor</indexer-post-
processor-impl>
</indexer-post-processor>
```

The `<indexer-class-name>` tag clarifies the model entity for the indexer. Furthermore, the `<indexer-post-processor-impl>` tag clarifies the implementation of the interface.

2. Create a new class in the docroot/WEB-

INF/src/com/liferay/hook/indexer directory of your example-hook named `SampleIndexerPostProcessor`. Then replace the Java source file's contents with the following lines:

```
package com.liferay.hook.indexer;

import java.util.Locale;
import javax.portlet.PortletURL;
import com.liferay.portal.kernel.log.Log;
import com.liferay.portal.kernel.log.LogFactoryUtil;
import com.liferay.portal.kernel.search.BaseIndexerPostProcessor;
import com.liferay.portal.kernel.search.BooleanQuery;
import com.liferay.portal.kernel.search.Document;
import com.liferay.portal.kernel.search.Field;
import com.liferay.portal.kernel.search.SearchContext;
import com.liferay.portal.kernel.search.Summary;
import com.liferay.portal.model.User;

public class SampleIndexerPostProcessor extends BaseIndexerPostProcessor {
    public void postProcessContextQuery(BooleanQuery booleanQuery,
SearchContext searchcontext)
        throws Exception {
        if(_log.isDebugEnabled())
            _log.debug(" postProcessContextQuery()");
    }

    public void postProcessDocument(Document document, Object object)
        throws Exception {
        User userEntity = (User) object;
        String indexerUserTitle = "";
        try {
            indexerUserTitle = userEntity.getJobTitle();
        } catch (Exception e) {}
        if(indexerUserTitle.length() > 0)
            document.addText(Field.TITLE, indexerUserTitle);
    }

    public void postProcessFullQuery(BooleanQuery fullQuery,
```

```

    SearchContext searchcontext)
        throws Exception {
    if(_log.isDebugEnabled())
        _log.debug(" postProcessFullQuery()");
}

    public void postProcessSearchQuery(BooleanQuery searchquery,
SearchContext searchcontext)
        throws Exception {
    if(_log.isDebugEnabled())
        _log.debug(" postProcessSearchQuery()");
}

    public void postProcessSummary(Summary summary, Document document,
Locale locale,
        String snippet, PortletURL portletURL) {
    if(_log.isDebugEnabled())
        _log.debug("postProcessSummary()");
}
    private static Log _log =
LogFactoryUtil.getLog(SampleIndexerPostProcessor.class);
}

```

Notice the `SampleIndexerPostProcessor` class extends Liferay's `BaseIndexerPostProcessor` base implementation. Then we add our own logic to enable users to search for *Job Title* amongst all the portal's users. Thus, we've added a new feature for the User Indexer. Let's give it a try!

Navigate to the *Control Panel → Users and Organizations* and make sure a user has a job title, which can be added in any user's *My Account* interface. Then test out the indexer hook by searching for that job title.

	First Name	Last Name	Screen Name	Job Title	Organizations	User Groups	Action
<input type="checkbox"/>	Joe	Bloggs	joebloggs	Blogger			<input type="button" value="Action"/>
<input type="checkbox"/>	John	Doe	john.doe	Blogger			<input type="button" value="Action"/>

As you can see, hooks serve to enhance the functionality of your portal and applications. Next, we will take a look at Liferay's Right to Left Language Support and how it can enhance both your portal and plugins.

12.11. Supporting Right-to-Left Languages in Plugins

Middle Eastern languages, such as Hebrew and Arabic, are written right-to-left (RTL). However, many sites are multilingual, requiring both RTL and left-to-right (LTR) content. Conveniently, browsers use language and direction HTML attributes to adapt and align page content automatically. Sites, however, may consist of elements that are absolutely positioned on the page by a style sheet; these elements aren't automatically aligned by the browser. Instead, you must adapt your style sheets to handle such elements. Since style sheets are usually designed for LTR languages, the typical challenge is creating alternative versions of the CSS for RTL languages. This can be an arduous task.

Thankfully, Liferay's *Right to Left Language Support* app automatically adapts Liferay Portal styles for RTL languages. When you deploy it, it mirrors your site's content for RTL languages. The app is available on the Liferay Marketplace. You can purchase, install, and deploy the app as described in the Leveraging the Liferay Marketplace chapter of Using Liferay Portal.

As a before-and-after example, the figure below shows a page displayed in English, an LTR language.



Pages

Site Pages

Content

Users

Configuration

Site Pages

Public Pages

Private Pages

Public Pages

Home

Products

Services

Partners

Documentation

Community

Downloads

About Us

View Pa

Look and F

Current The



Classic

Description
Portlets, theme
Liferay Portal.

Author
Liferay, Inc.

Compare it to the following figure of a page displayed in Hebrew, an RTL language.

[לעיל](#)[למטה](#)[הוסף דף](#)[צפה בדףים](#)

ראה והרגש

[לוגו](#)[JavaScript](#)[Mobile Device Rules](#)[Rule Group Instance](#) [לבטל](#)[שמור](#)

ראה והרגש

ערכת נושא נוכחת



Classic

תיאור

Portlets, themes, and layout templates included with
.Liferay Portal

מחבר

.Liferay, Inc

You get the point, right? The Right to Left Language Support (RTL Support) app does the heavy lifting of rendering the RTL content appropriately!

Now that you've seen Liferay Portal and its apps rendered using RTL Support, have you wondered how you might leverage RTL Support in your custom plugins? You'll learn how to use RTL Support with your plugins next.

12.11.1. Applying the RTL Support to Custom Plugins

You can use RTL Support with any plugin type, though theme plugins are the most common. The following steps focus on using RTL Support in a custom theme, but they also mention what's needed to make similar changes to support using RTL Support in the other plugin types.

1. Make sure to deploy the Right to Left Language Support app to your application server.

If upon initial deployment you don't notice any changes when switching to an RTL language, reload the page to force a clean cache.

Note, since dynamic generation of CSS from SASS is not yet supported in the context of the RTL Support app, the hook loads the current theme's merged CSS files, disregarding whether theme CSS fast loading is disabled. Disabling CSS fast load (i.e., setting `theme.css.fast.load=false`) is commonly done while debugging for development.

2. Restart your application server.
3. Extract the contents of the Liferay RTL Hook web application.
4. Copy the `rtl-hook.jar` and its dependencies, `ant.jar`, `jodd.jar`, `jruby.jar` and `rhino.jar`, from your RTL hook web app's `WEB-INF/lib` folder to the `docroot/WEB-INF/lib` folder of your plugin project.
5. Add the following filtering elements before the closing `</web-app>` tag in your project's `docroot/WEB-INF/web.xml` file.

```
<filter>
    <filter-name>Dynamic CSS Filter</filter-name>
    <filter-
class>com.liferay.rtl.hook.filter.dynamiccss.DynamicCSSFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>Dynamic CSS Filter</filter-name>
    <url-pattern>*.css</url-pattern>
</filter-mapping>
```

6. Add targets for building and cleaning your plugin's RTL CSS files.

For example, here are the targets to add to the `build.xml` file of your theme plugin project:

```
<target name="build-css" depends="clean-rtl-css, build-common-theme.build-
css, build-rtl-css" />

<target name="build-rtl-css">
    <java
        classname="com.liferay.rtl.tools.RtlCssBuilder"
        classpathref="plugin.classpath"
```

```

        fork="true"
        newenvironment="true"
    >
        <jvmarg value="-Dliferay.lib.portal.dir=${app.server.lib.portal.dir}" />
    />
        <arg value="sass.dir=/ " />
        <arg value="sass.docroot.dir=${basedir}/docroot" />
    </java>
</target>

<target name="clean-rtl-css">
    <delete failonerror="false" includeemptydirs="true">
        <fileset dir="${basedir}/docroot" includes="**/.sass-cache/*_rtl.*" />
    </delete>
</target>

```

If you're using the RTL hook with another type of plugin project, rename the `build-common-theme.build-css` dependency target reference appropriately for your plugin type:

Portlet: `build-common-portlet.build-css`

Hook: `build-common-hook.build-css`

Ext: `build-common-ext.build-css`

Web App: `build-common-web.build-css`

7. Deploy your plugin to the portal.

Ant echoes *Generated RTL cache for ...* messages that mention the cache that the RTL hook generates in your plugin. In your plugin's `css/.sass-cache` folder, the hook creates `*_rtl.css` versions of each of your `*.css` files.

You now know how to use the Right to Left Language Support app in your custom plugins. Now it's time to learn how you can extend the Right to Left Language Support app's style with your own custom CSS for RTL languages.

12.11.2. Defining Custom CSS for RTL Languages

As you learned in the previous section, the Right to Left Language Support (RTL Support) app automatically generates RTL versions of your CSS files by applying rules, such as changing `margin-left` to `margin-right`. You may, however, want to extend the generated CSS by defining your own custom styles for RTL languages. You can achieve this by following these steps:

1. Create a CSS file with the suffix `_rtl` in the same location as a CSS file that you want to extend. For example, create a file `main_rtl.css` to extend a file named `main.css`.
2. Edit the `_rtl` file, adding *only* the lines that define your custom styles for RTL languages.
3. Deploy your plugin.

Check your plugin's `css/.sass-cache` folder to see that the generated `_rtl.css` file in this folder not only contains the automatically generated CSS from the original file, but also contains your custom CSS code at the end.

4. In your site, add the Language portlet to a page and change the current language to an RTL language (e.g., Hebrew).

Notice your custom RTL styles applied to your site.

Now your plugin styles are automatically adapted for RTL languages. In case you have defined any custom styles, they are also applied.

Whether you want to adapt the portal and/or your custom plugins to RTL languages, the RTL Support app makes it easy to offer your users the ideal viewing experience.

In the next section, you'll explore more hooks that allow for customizing Liferay's core features.

12.12. Other Hooks

Since hooks are the preferred plugin type for customizing Liferay's core features, the Liferay team is happy to keep providing you new hooks. This section is a placeholder for hooks that are available in Liferay Portal 6.2, but aren't yet fully documented.

Servlet filter hook: Servlet filters allow you to pre-process requests going *to* a servlet and post-process responses coming *from* a servlet. As server requests are received that match URL patterns or match servlet names specified in your servlet filter mappings, your specified servlet filters are applied. Hook elements `servlet-filter` and `servlet-filter-mapping` have been added to `liferay-hook.xml` so you can configure your servlet filters. For a working example, see the `sample-servlet-filter-hook` in the Plugins SDK.

12.13. Summary

In this chapter, we discussed some of the many uses of the versatile hook plugin, the preferred tool for customizing Liferay. You learned how to perform custom portal actions, override and extend custom portal JSPs, modify portal properties, and replace portal services and language properties.

Next, we'll introduce you to Liferay's powerful UI framework: AlloyUI.

13. Designing User Interfaces with AlloyUI

Liferay's User Interface (UI) Team continually strives to provide stylish, lightning-fast components that are extensible and built to last. I liken the team to an elite band of metallurgists: cutting-edge scientists that extract metals from craggy rocks deep beneath the earth's crust. They purify these metals and mix them together to make some of the most awesome things known to mankind--seemingly weightless bike frames, bridges with enough tensile strength to withstand hurricanes, and swords that can cut through almost anything. The Liferay UI Team is like these scientists. They've put forth equal sweat and ingenuity to bring you a mixture, or *alloy*, of the best UI technologies and have wrapped it up in one ultimate framework--AlloyUI!

AlloyUI gives you skinnable, scalable UI components, so you can provide a consistent look and

feel for your application. It's a framework containing JavaScript extensions to Yahoo UI (YUI) that leverages all of YUI's modules and adds its own components to help you build terrific UIs. AlloyUI also incorporates Twitter Bootstrap to make styling components a snap.

AlloyUI was built on YUI for several reasons. First, YUI facilitates building high quality production-level widgets quickly. YUI has a flexible, elegant architecture that is easy to extend. It is useful in both small and large scale projects. YUI is also documented well at <http://yuilibrary.com/yui/docs/>.

By using AlloyUI, and therefore leveraging YUI, you can reap significant performance benefits. YUI helps reduce the size of your up-front JavaScript request download, and lazily loads other modules as needed. YUI manages dependencies for you, by making sure modules are only downloaded once for a page and by specifying modules your page needs in a single request. Also, AlloyUI provides special tags that let you designate JavaScript for parsing only *after* your page's HTML and CSS have been loaded. This often speeds up the availability of your UI to your user. As you use YUI through AlloyUI, you'll realize these benefits.

We know that many developers in the Liferay community like to use jQuery. You can use jQuery in Liferay Portal, but we strongly recommend you use AlloyUI. Note, AlloyUI is always loaded and available to you in Liferay. If you use something else, your page must load your library *and* AlloyUI, which slows down performance.

By reading this chapter and following along with its exercises, you'll learn what AlloyUI is and how to use it in Liferay Portal. For further details on the AlloyUI project, tutorials, examples, and API documentation, make sure to visit <http://alloyui.com>. We'll show you around AlloyUI in this chapter by exploring the following topics:

- A simple AlloyUI example
- Using an AlloyUI Carousel in Your Portlet
- Working with the AlloyUI project

To start things off right, let's go over a simple example using AlloyUI.

13.1. A simple AlloyUI example

AlloyUI is easy to use. Let's demonstrate by using AlloyUI in an HTML file.

1. Create an HTML file and insert the following lines:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Example</title>

  <script src="http://cdn.alloyui.com/2.0.0/aui/aui-min.js"></script>
  <link href="http://cdn.alloyui.com/2.0.0/aui-css/css/bootstrap.min.css"
    rel="stylesheet">
</head>
<body>
  <input type="text" id="some-input" />
  <span id="counter"></span> character(s) remaining
```

```

<script>
YUI().use(
  'aui-char-counter',
  function(Y) {
    new Y.CharCounter(
      {
        counter: '#counter',
        input: '#some-input',
        maxLength: 10
      }
    );
  }
);
</script>
</body>
</html>

```

2. Open the HTML file in your browser.
3. Enter some characters into the text field.

AlloyUI's character counter reports the number of characters you can enter in the text field before reaching the 10 character limit.

123

7 character(s) remaining

Let's look at how we did this with AlloyUI. First we added HTML that displays an HTML `<input/>` element with `id=some-input`. Then we referenced an element called `counter` and added some text describing that counter.

Then we used a script element to reference Alloy's seed file, `aui-min.js` from a content delivery network (CDN). The seed file includes the bare minimum core code required for AlloyUI. Any additional code is loaded dynamically by YUI.

```
<script src="http://cdn.alloyui.com/2.0.0/aui/aui-min.js"></script>
```



Note: For performance reasons, it is almost always best to reference the seed file from the CDN rather than from a designated server. On receiving the request for the seed file, the CDN returns it from the nearest server on the CDN, minimizing latency time.

Lastly, the code in the `script` element reports the number of characters remaining in the text field. As you enter or delete characters from the field, the script recalculates the number of remaining characters and displays that number via the `counter` element on your page. How's that for dynamic content!

This script uses YUI and AlloyUI in what is commonly referred to as a "sandbox." Code is sandboxed when elements of the code are set off in their own namespaces. Why do this? Because JavaScript, like many programming languages, has both a local and a global scope. Code placed in a JavaScript function is locally scoped, which means that nothing inside that function can be seen outside that function. Another way to describe this code is that it has been sandboxed.

This reminds me of the countless hours I spent as a child using my die-cast metal toy tractor to plant imaginary crops in my toy sandbox. It was a wonderful place to let my imagination go wild and grow acres and acres of fictitious corn fields.

The UI sandbox is similar to a toy sandbox--but safer and perhaps more fun. Unlike my childhood sandbox that was inevitably raided by friends, siblings, and my dog, your UI sandbox avoids namespace clashes with code in other sandboxes on your page.

In the example above, the sandbox is the callback where you run your code. It follows this format:

```
YUI().use([package 1, ... package n],function(Y) { // Your code goes here } );
```

`YUI().use()` is a function call that instantiates modules for you to use. As parameters, you pass in packages and a function containing your code. The example code required Alloy's `aui-char-counter` package. The final argument is the YUI object as parameter `Y`. Alloy's classes are stored in this `Y` object. In this function, you place presentation logic, leveraging AlloyUI's API via the mighty `Y` object. We'll get into more details on the API shortly.

Now that we've dissected the example, let's get it working in a portlet. Instead of referencing AlloyUI's seed file, simply reference the `aui` taglib in your JSP:

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

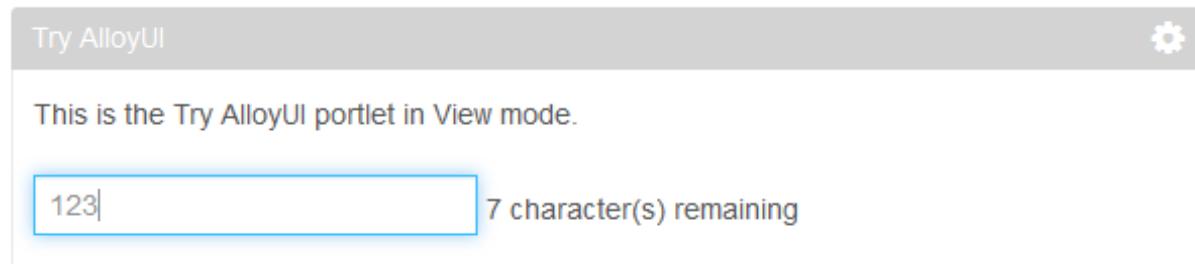
Then add the code for the input element and the Alloy script to your portlet's JSP. This time, however, replace the `<script>` tag with the `<aui:script>` tag. The `<aui:script>` tag combines the contents of all `<aui:script>` tags used on a page into one script block at the bottom of the page, and it wraps the functions in a `YUI.use()` call to bring in necessary module dependencies.

With all these simplifications, the AlloyUI code in your JSP looks like this:

```
<input type="text" id="some-input" />
<span id="counter"></span> character(s) remaining

<aui:script>
YUI().use(
  'aui-char-counter',
  function(Y) {
    new Y.CharCounter(
    {
      counter: '#counter',
      input: '#some-input',
      maxLength: 10
    }
  );
}
);
</aui:script>
```

Voila! You're using AlloyUI in Liferay!

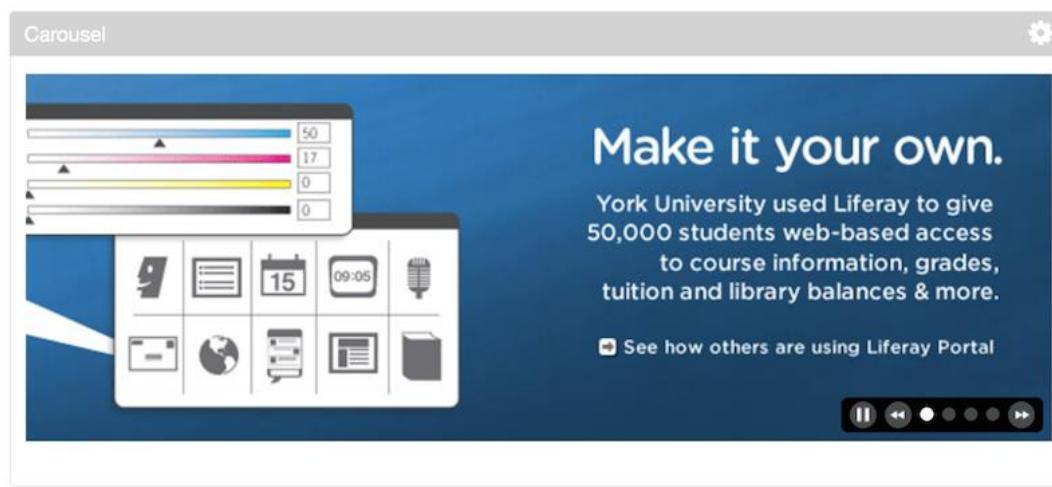


Now that we've gone over using a simple `aui-char-counter` in your portlet, let's move on to something a little more challenging.

13.2. Using an AlloyUI Carousel in Your Portlet

We went over a simple example of using the `aui-char-counter` in an HTML file and then showed you how to place it into a portlet. Next, let's kick things up a notch and use an AlloyUI component that has more versatility: an image carousel!

Image carousels are often the first thing people see when they visit sites. They provide an interactive way of cycling through visual elements and are an effective means of communicating information to users. AlloyUI's `aui-carousel` module makes it easy to get an image carousel up and running in no time flat. First, you'll learn how to set up a basic portlet with the carousel and see what's happening behind the scenes. Then you'll have some real fun by customizing the carousel to suit your individual needs.



To give you an idea of what's next, here is what the basic carousel looks like:

All right, enough discussing the future. Time to do the work!

13.2.1. Adding a Carousel to a Portlet

To add a carousel to a portlet, follow these steps:

1. Insert this code in your portlet's view JSP:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@ taglib prefix="aui" uri="http://liferay.com/tld/aui" %>

<div id="myCarousel">
    <div class="carousel-item" style="background:
url(http://alloyui.com/carousel/img/1.jpg);"></div>
    <div class="carousel-item" style="background:
url(http://alloyui.com/carousel/img/2.jpg);"></div>
    <div class="carousel-item" style="background:
url(http://alloyui.com/carousel/img/3.jpg);"></div>
    <div class="carousel-item" style="background:
url(http://alloyui.com/carousel/img/4.jpg);"></div>
</div>

<aui:script>
AUI().use(
    'aui-carousel',
    function(Y) {
        new Y.Carousel(
            {
                contentBox: '#myCarousel',
                height: 250,
                width: 700
            }
        ).render();
    }
);
</aui:script>
```

If you try to deploy the portlet now, you'll notice that no images are displayed; this is because we need to write some CSS to tell the portlet how to display the carousel.

2. Create a `main.css` file in your portlet's `docroot/css/` directory and add this code to it:

```
div.carousel-item
{
width: 700px;
height: 250px;
}
```

3. Deploy your portlet to your portal.

Your images display correctly. Give yourself a pat on the back; you've just successfully used the `aui-carousel` in a portlet! Next, it's time to understand the inner-workings of the carousel and see what makes it tick.

The JSP code you inserted in the portlet specified your carousel. You included directives for using the `java` and `aui` taglibs. Below them, you specified a `<div>` named `myCarousel`, to identify the carousel's images. The default images are provided by AlloyUI. This set the foundation for using the AUI script.

The script uses the `aui-carousel` module. You gave it some basic attributes to specify where

to display the carousel and the size it should be. You told the `aui-carousel` to display in the `myCarousel <div>` by placing the `<div>`'s ID as the value of the `contentBox` attribute. You also set the `width` and `height` attributes at the resolution of 700px X 250px.

Finally, you used the `main.css` file to style the `carousel-item` divs, giving them `width` and `height` property values to match the carousel's `width` and `height` attributes, specified in the JSP. That wasn't so hard, was it?

Next you'll look at how you can customize the carousel to give it your own flare.

13.2.2. Customizing the AUI-Carousel

Now comes the really fun part: making the carousel your own!

1. Open the same view JSP file and replace its code with the following code:

```
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>
<%@ taglib prefix="aui" uri="http://liferay.com/tld/aui" %>

<div id="myCarousel">
    <div id="image1"></div>
    <div id="image2"></div>
    <div id="image3"></div>
    <div id="image4"></div>
</div>

<aui:script>
AUI().use(
    'aui-carousel',
    function(Y) {
        new Y.Carousel(
            {
                contentBox: '#myCarousel',
                height: 250,
                width: 700,
                intervalTime: 2,
                animationTime: 1,
                activeIndex: 0,
                boundingBox: '#myCarousel'
            }
        ).render();
    }
);
</aui:script>
```

The code above has some attributes that can be styled to customize the carousel widget. The bounding box of the widget, used for positioning the carousel, is set to the `#myCarousel` div. The transition between images is set to last for one second. The carousel is set to display each image for two seconds. Setting the `activeIndex` to 0 displays the first image listed in the `#myCarousel <div>` as the first one to display. Now that you've laid the groundwork for your carousel, you can go ahead and style it.



Note: This is only a subset of the attributes that can be modified for a carousel. If you choose not to set values for these attributes, defaults are used. For a full list of the attributes and their defaults, as well as further documentation on the AUI-Carousel, please visit <http://alloyui.com/api/classes/A.Carousel.html>.

2. Open the main.css file and replace its code with the following CSS that styles the carousel:

```
/* styling for the carousel body */
div.carousel-item {
    width: 700px;
    height: 250px;
    border-radius: 6px 6px 0 6px;
    opacity: 100;
}

/* styling for the boundingBox and ContentBox(in this case) */
#myCarousel {
    margin: 0 auto 40px;
}

/* styling for div with id image1 */
#image1 {
    background: url("../img/moon.jpg");
}

/* styling for div with id image2 */
#image2 {
    background: url("../img/thor.jpg");
}

/* styling for div with id image3 */
#image3 {
    background: url("../img/toy.jpg");
}

/* styling for div with id image4 */
#image4 {
    background: url("../img/spock.jpg");
}

/* Pause Button */
#myCarousel menu li a.carousel-menu-item.carousel-menu-pause {
    background-image: url("../img/icons.png");
    background-position: 0 43px;
    height: 20px;
    width: 20px;
    border-radius: 90px;
}

/* Play Button */
#myCarousel menu li a.carousel-menu-item.carousel-menu-play {
```

```

background-image: url("../img/icons.png");
background-position: 20px 43px;
height: 20px;
width: 20px;
border-radius: 90px;
}

/* Prev Button */
#myCarousel menu li a.carousel-menu-item.carousel-menu-prev {
    background-image: url("../img/icons.png");
    background-position: 0 64px;
    height: 20px;
    width: 20px;
    border-radius: 90px;
}

/* Next Button */
#myCarousel menu li a.carousel-menu-item.carousel-menu-next {
    background-image: url("../img/icons.png");
    background-position: 21px 0;
    height: 20px;
    width: 20px;
    border-radius: 90px;
}

/* active index indicator */
#myCarousel menu li a.carousel-menu-item.carousel-menu-item.carousel-menu-index.carousel-menu-active {
    background-image: url("../img/icons.png");
    background-color: rgba(0,0,0,0);
    background-position: 21px 22px;
    width:20px;
    height:20px;
}

/* inactive index indicator */
#myCarousel menu li a.carousel-menu-item.carousel-menu-item.carousel-menu-index {
    background-image: url("../img/icons.png");
    background-color: rgba(0,0,0,0);
    background-position: 0px 22px;
    width:20px;
    height:20px;
}

/* Menu Bar */
#myCarousel menu {
    background: none repeat scroll 0 0 #0000C0;
    border-bottom: 3px solid #00CCE0;
    border-radius: 0 0 15px 15px;
    bottom: auto;
    display: table;
    left: 518px;
    padding: 1% 0;
}

```

```

        right: 0;
        top: 250px;
        width: 26%;
    }

/* List of menu buttons */
#myCarousel menu li {
    float: inherit;
}

```

Some explanation of these styles is in order. The code starts off by setting the width and height for the carousel body, giving it rounded edges with the border-radius property.

```

/*styling for the carousel body*/
div.carousel-item
{
    width: 700px;
    height: 250px;
    border-radius: 6px 6px 0 6px;
    border-radius: 6px 6px 0 6px;
}

```

Next, the carousel is centered in the middle of the portlet by setting the margin property of the boundingBox element (#mycarousel in this case) to auto for the left and right margins. The top margin is set to 0 while the bottom margin is set to 40px to leave room for the menu controls.

```

/* styling for the boundingBox and ContentBox(in this case) */
#myCarousel
{
    /* centers the carousel in the middle of the portlet */
    margin:0 auto 40px;
}

```

Next, the carousel's images are set by pointing the background-image properties of the corresponding <div>s to the location of each respective image file, which reside in the

.../img/ directory.

```

/* styling for div with id image1 */
#image1
{
    background-image: url("../img/moon.jpg");
}

/* styling for div with id image2 */
#image2
{
    background-image: url("../img/thor.jpg");
}

/* styling for div with id image3 */
#image3
{
    background-image: url("../img/toy.jpg");
}

/* styling for div with id image4 */

```

```
#image4
{
    background-image: url("../img/spock.jpg");
}
```

Now that you've seen how to style the carousel's body, it's time to break down the carousel menu's styling. Existing classes are referenced for the menu controls. To understand how to determine the classes for the menu controls, you need to understand the DOM tree:

```
<div id="myCarousel">
<menu>
<li>
    <a class="carousel-menu-item carousel-menu-pause"></a>
    <a class="carousel-menu-item carousel-menu-play"></a>
    <a class="carousel-menu-item carousel-menu-prev"></a>
    <a class="carousel-menu-item carousel-menu-next"></a>
</li>
</menu>
</div>
```

This is a simplified version of the DOM tree, but it gives you an idea of the overall structure. You can see from the DOM tree that the carousel menu controls lie within the `#myCarousel` `<div>`, inside a `menu` tag, inside a `list` tag, inside an `anchor` tag.



Note: You can see the DOM tree by right-clicking the carousel's *Next* menu button and inspecting the element in the browser (Firefox in this case).

Once you understand the DOM tree, you can go ahead and place the styling for each of the menu buttons with their corresponding class. Each menu button has a `background` property set to its image location and a `border-radius` property for rounding edges. If you don't want a rounded edge, you can omit the `border-radius` property from your styling. Another important property to note is the `background-position` property. You used an image sprite for the menu controls, and so you configured a `background-position` property to tell the buttons where exactly on the image they are.

```
/* Pause Button */
#myCarousel menu li a.carousel-menu-item.carousel-menu-pause{
    background-image: url("../img/icons.png");
    background-position: 0 43px;
    height: 20px;
    width: 20px;
    border-radius:90px; /* in this case I have a circular icon */
}

/* Play Button */
#myCarousel menu li a.carousel-menu-item.carousel-menu-play{
    background-image: url("../img/icons.png");
    background-position: 20 43px;
    height: 20px;
    width: 20px;
    border-radius:90px; /* in this case I have a circular icon */
}
```

```

/* Prev Button */
#myCarousel menu li a.carousel-menu-item.carousel-menu-prev{
    background-image: url("../img/icons.png");
    background-color: rgba(0,0,0,0);
    background-position: 0 64px;
    height: 20px;
    width: 20px;
    border-radius:90px; /* in this case I have a circular icon */
}

/* Next Button */
#myCarousel menu li a.carousel-menu-item.carousel-menu-next{
    background-image: url("../img/icons.png");
    background-color: rgba(0,0,0,0);
    background-position: 21px 0;
    height: 20px;
    width: 20px;
    border-radius:90px; /* in this case I have a circular icon */
}

```

Next, you styled the active and inactive index indicators. Once again, in the DOM tree there are existing classes that can be used for styling. The images are set with the `background-image` and `background-position` properties and given the proper height and width.

```

/* active index indicator */
#myCarousel menu li a.carousel-menu-item.carousel-menu-item.carousel-
menu-index.carousel-menu-active{
    background-image: url("../img/icons.png");
    background-position: 21px 22px;
    width: 20px;
    height: 20px;
}

/* inactive index indicator */
#myCarousel menu li a.carousel-menu-item.carousel-menu-item.carousel-
menu-index {
    background-image: url("../img/icons.png");
    background-position: 0 22px;
    width: 20px;
    height: 20px;
}

```

Finally, you styled the menu bar, which holds the menu controls that you also styled. The left edge of the menu bar is set 518 px from the left. The top edge of the menu bar is set 250 px from the top, placing it just beneath the 250 px height carousel. The menu bar's width is scaled down to 26% of the size of the carousel's width. To finish out the CSS, the list, which holds the menu buttons, is set to adjust to the size and shape of the menu bar.

```

/* Menu Bar */
#myCarousel menu {
    background: none repeat scroll 0 0 #0000C0;
    border-bottom: 3px solid #00CCE0;
    border-radius: 0 0 15px 15px;
    bottom: auto;
    display: table;

```

```

left: 518 px;
padding: 1% 0;
right: 0;
top: 250px;
width: 26%
}

/* List of menu buttons */
#myCarousel menu li {
    float: inherit;
}

```



Here is an example of a customized carousel using the configuration above:

You can access a finished version of the customized

portlet at <https://github.com/liferay/liferay-docs/tree/master/devGuide/code/12-working-with-alloyUI/customized-carousel-portlet>

Now that you've gotten your feet wet using some of AlloyUI's components, next you'll see how to work with the AlloyUI source so you can create your own components.

13.3. Working with the AlloyUI project

Liferay bundles AlloyUI with the portal, as it's used throughout the portal and core portlets. Conveniently, you can use AlloyUI in any project--it doesn't have to run on Liferay. When you develop AlloyUI scripts and components for use in the portal, you can reuse them anywhere else. If you're using AlloyUI outside Liferay, you might want to build it yourself.

Here are some other reasons why you might use a local AlloyUI installation or AlloyUI project build:

- Creating and testing your own AlloyUI component modules
- Using the latest AlloyUI project source code that is not yet released
- Using AlloyUI on a closed network
- Contributing and testing a fix or enhancement to AlloyUI

Let's download AlloyUI and set it up for developing AlloyUI scripts and components locally.

13.3.1. Working with an AlloyUI Project Release Zip File

You can download any AlloyUI version as a .zip file from <https://github.com/liferay/alloy-ui/releases>. The file contains the following files and folders:

- alloy-[version] / - AlloyUI project root directory
 - › build/ - Contains the AlloyUI and YUI modules used in Liferay
 - › demos/ - Contains basic examples of the AlloyUI components
 - › src/ - Contains the source code of the AlloyUI modules
 - › .alloy.json - Specifies how to build the modules
 - › LICENSE.md - Defines AlloyUI's the license agreement
 - › README.md - Explains the AlloyUI project

As you did in the initial example, the first thing you'll call is AlloyUI's aui-min.js seed file, in your alloy-[version]/build/aui/ folder. For example, if your AlloyUI project root directory is /home/joe.bloggs/alloy-2.0.0/, you'll refer to the seed file like this:

```
<script src="/home/joe.bloggs/alloy-2.0.0/build/aui/aui-min.js"></script>
```

Likewise, make sure to specify your local bootstrap seed file as well:

```
<link src="/home/joe.bloggs/alloy-2.0.0/build/aui-css/css/bootstrap.min.css" rel="stylesheet"></link>
```

Go ahead and replace the remote seed file references from the example HTML file we used at the beginning of this chapter with references to your local seed files. Except for the paths to your seed files, your HTML content should look similar to this:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Example</title>

<script src="/home/joe.bloggs/alloy-2.0.0/build/aui/aui-min.js"></script>
<link href="/home/joe.bloggs/alloy-2.0.0/build/aui-css/css/bootstrap.min.css" rel="stylesheet"></link>
</head>
<body>
<input type="text" id="some-input" />
<span id="counter"></span> character(s) remaining

<script>
YUI().use(
    'aui-char-counter',
    function(Y) {
        new Y.CharCounter(
            {
                counter: '#counter',
                input: '#some-input',
                maxLength: 10
            }
        );
    }
)
```

```
);  
</script>  
</body>  
</html>
```

123

7 character(s) remaining

The figure below shows what your web page should look like.

Great! Now you know how to use a local set of the AlloyUI tag libraries. Next, we'll show you how to work with the AlloyUI source project. You'll learn how to build the project so you can experiment with the latest AlloyUI code whenever you want.

13.3.2. Working with the AlloyUI Project Source

You may want to work with the latest cutting-edge AlloyUI code from time to time. Liferay makes it easy do get your hands on. We use a public GitHub project named *alloy-ui* to store and share the latest AlloyUI code. You can download the code so that you can build it and try it locally. You can also leverage the *alloy-ui* project to create some AlloyUI modules of your own. We'll show you just how easy it is to install the project and use it.

In this section, we'll demonstrate the following:

- Installing the required software for the AlloyUI project
- Installing the *alloy-ui* project
- Building the project
- Packaging the project in a distribution and using that distribution

Let's get started by installing AlloyUI's dependencies.

13.3.2.1. Setting Up AlloyUI's Required Software

The *alloy-ui* project depends on the following software:

- Node.js is a platform for building applications.
- Ruby is used in the *alloy-ui* project for downloading other software packages.
- Compass is an open-source CSS authoring framework.
- Sass stands for Syntactically Awesome Stylesheets. It is a scripting language used for specifying CSS.

Let's install Node.js first. You can download it from <http://nodejs.org/>. Linux, OS X, or UNIX users can download its source in a `.tar.gz` file, unzip it, un-tar it, and build it per the instructions in its `README.md` file. Windows users can download the `.msi` installer file and run it.



Warning: On Windows, only install to locations that have UNIX-friendly paths. Paths like `C:\Program Files (x86)` that contain space characters and parentheses can prevent software from working properly.

You can download Ruby from <https://www.ruby-lang.org>. Alternatively, on Windows, you can

download RubyInstaller from <http://rubyinstaller.org/> and use it to install Ruby. After installing Ruby, execute the following command from your terminal to get its latest updates:

```
gem update --system
```

Now, let's use Ruby's gem command to install Compass and Sass. Conveniently, Sass comes bundled with Compass. To install both of them, simply execute the following command:

```
gem install compass
```

Great! You've installed all of the software applications that the alloy-ui project requires. Next, let's get our hands on the alloy-ui project.

13.3.2.2. *Installing the AlloyUI Project*

Liferay's AlloyUI developers and AlloyUI community members contribute code to the alloy-ui project on GitHub. To access the alloy-ui project and install it locally, you'll need an account on GitHub and the Git tool on your machine. Visit <https://github.com/> for instructions on setting up the account and see <http://git-scm.com/> for instructions on installing Git.

Here are some simple steps for forking the alloy-ui project on GitHub and installing the project locally:

1. Go to the AlloyUI project repository at <https://github.com/liferay/alloy-ui>.
2. Click *Fork* to copy Liferay's alloy-ui repository to your account on GitHub.
3. In your terminal or in GitBash, navigate to the location where you want to put the alloy-ui project. Then download a clone of the repository by executing the following command, replacing `username` with your GitHub user name:

```
git clone git@github.com:username/alloy-ui
```

Now you have your own personal copies of the project in GitHub and on your local machine. Before you start building the project, let's set it up with the `2.0.x` branch. AlloyUI 2.0 is the version used by Liferay Portal 6.2.

4. Navigate to your new alloy-ui directory in GitBash by running `cd alloy-ui`.
5. To download Liferay's alloy-ui branches, you must first associate a remote branch to Liferay's alloy-ui repository and then fetch all of branches via that remote branch:

```
git remote add upstream git@github.com:liferay/alloy-ui.git  
git fetch upstream
```

6. Lastly, create your own branch named `2.0.x` based on Liferay's `2.0.x` branch, by executing the following command:

```
git checkout -b 2.0.x upstream/2.0.x
```

Great! Now that you have the `2.0.x` branch checked out, we can install and initialize the project's remaining dependencies. Follow these steps:

1. Install the global dependencies (exclude using `[sudo]` on Windows):

```
[sudo] npm install -g grunt-cli shifter yogi yuidocjs phantomjs
```

2. Then install the local dependencies:

```
npm install
```

3. Lastly, the alloy-ui project has a special target called `init` that clones and updates the GitHub software projects on which alloy-ui depends. These projects include yui3, ace-builds, alloy-bootstrap, alloy-apidocs-theme, and alloyui.com. Initialize these projects for alloy-ui by executing this command:

```
grunt init
```

Alright! You have the alloy-ui project and all of its dependencies. Next, we'll build AlloyUI.

13.3.2.3. ***Building the AlloyUI Project***

The alloy-ui project contains source code for AlloyUI, YUI3, and Twitter Bootstrap. The project uses a JavaScript build tool called Grunt to build all kinds of things, including AlloyUI, YUI3, Twitter Bootstrap CSS, and AlloyUI API documentation. The alloy-ui project has targets that simplify building several sources at once and it has granular targets for building individual sets of source code. We've provided a table of these targets below.

The alloy-ui Project Grunt Targets

Target	Command	Description
build	<code>grunt build</code>	Builds YUI and AlloyUI together
build:yui	<code>grunt build:yui</code>	Builds YUI only
build:aui	<code>grunt build:aui</code>	Builds AlloyUI only
bootstrap	<code>grunt bootstrap</code>	Builds and imports Bootstrap's CSS

Let's build everything by executing the following command:

```
grunt all
```

On successfully executing each of these commands, Grunt reports this message: `Done, without errors.` Well done!

Note, to build a single AlloyUI module, you can execute the following (replace `aui-module-name` with the module's name):

```
grunt build:aui --src src/aui-module-name
```

When you're ready to try out your locally built version of AlloyUI, you can package it up and use it. We'll do that next.

13.3.2.4. ***Using Your Locally Built AlloyUI Distribution***

Building a release distribution of your alloy-ui project is easy. And it's just as easy using your distribution in your web pages. We'll do it together.

To create your distribution `.zip` file of AlloyUI, execute the following command:

```
grunt release
```

This creates zip file `alloy-[version].zip`. Unzip this file to an arbitrary location.

You can reference your AlloyUI distribution in your Liferay JSPs in the same manner we demonstrated in the section *Working with an AlloyUI Project Release*. That is, you reference AlloyUI's `aui-min.js` file as a seed file.

For example, you could specify the following seed file, replacing `/home/joe.bloggs/` with the path to your unzipped distribution.

```
<script src="/home/joe.bloggs/alloy-2.0.0/build/aui/aui-min.js"></script>
It's just that easy to use your very own cutting-edge copy of the AlloyUI code!
```

13.4. Summary

In this chapter, we've only scratched the surface of showing you what AlloyUI has to offer you in designing user interfaces in Liferay. We've introduced you to AlloyUI to show you what it is and to explain how it integrates so well with Liferay. We've provided a simple example that demonstrates using AlloyUI components and we've shown you how to set up the AlloyUI project environment so that you can build your own AlloyUI components. You should visit AlloyUI's official website, <http://alloyui.com/>, regularly to get the latest information on the AlloyUI framework. But, we'll continue to add examples to this chapter to demonstrate more ways you can use AlloyUI in your portal. So, make sure to periodically check back with this guide.

Now that you know how to use AlloyUI components to build snazzy UIs, let's consider what it takes to develop your apps for publishing to *Liferay Marketplace*.

14. Liferay Marketplace

The **Liferay Marketplace** is an exciting hub for sharing, browsing and downloading Liferay-compatible applications. As enterprises look for ways to build and enhance their existing platforms, developers and software vendors search for new avenues to reach this market. Marketplace leverages the entire Liferay ecosystem to release and share apps in a user-friendly, one-stop site.

In addition to providing application consumers with Marketplace, Liferay provides a Plugin Security Manager to help protect a consumer's portal from potentially negative side-affects that can possibly be caused by an app. The Plugin Security Manager's job is to only allow an app to use resources that the app has specified up-front in its Portal Access Control List (PACL). As such, we'll explain how to create PACLs for the apps you develop.

This chapter covers the following topics related to developing apps for Liferay Marketplace:

- Marketplace Basics
- Requirements for Publishing to the Marketplace
- Developing and Testing Apps for the Marketplace
- Publishing Apps to the Marketplace
- Maintaining and Updating Apps
- Tracking App Performance
- Understanding Plugin Security Management
- Developing Plugins with Security in Mind
- Enabling the Security Manager
- Portal Access Control List (PACL) Properties

This chapter focuses on the topics of interest to a Liferay developer. It is highly recommended that you first read the Liferay Marketplace chapter of *Using Liferay Portal*, which contains detailed information about the Marketplace from an end-user's perspective.

14.1. Marketplace Basics

Before diving into the details of developing for the Marketplace, it is important that you have a good grasp of the concepts introduced in the Marketplace. The following sections discuss these concepts.

14.1.1. What is an App?

As a Liferay developer, you're undoubtedly already be familiar with the concept of plugins (portlets, hooks, themes, etc). If not, see [Developing Applications for Liferay](#). A *Liferay App* (sometimes just called an *app*) is a collection of one or more of these plugins, packaged together to represent the full functionality of an application on the Liferay platform. In addition to the plugins contained within an app, apps have metadata such as names, descriptions, versions, and other ancillary information used to describe and track the app throughout its lifecycle.

Much like standard Liferay plugins, Liferay apps are also *hot-deployable*. On downloading an app from the Marketplace, you find that it is a special file type with a `.lpkg` extension. This file can be dropped into Liferay's hot-deploy folder (`liferay-portal-[version]/deploy`), like any other plugin, to deploy it into that running instance of Liferay Portal.

As an app developer, you're not required to create the actual Liferay app files. Instead, your app's individual plugins (`.war` files) are uploaded as part of the publication process, along with information (name, description, version, icon, etc) that identifies the app. The publication process is described in detail later.

14.1.2. What is a Version?

The concept of versioning is well known in software, and it is no different here. A version of an app represents the functionality of the app at a given point in time. When you first create an app, you give it an initial version (e.g., `1.0`). On updating the app, you increment its version (e.g., from `1.0` to `1.1`) and you upload new files representing that version of the app. In some cases, you may want to specify additional qualifiers in order to convey a special meaning. For example, you may declare that the version of your app is always in `x.y.z` format (where you've clearly defined the significance of each `x`, `y`, and `z`). Liferay Portal versions and official Liferay app versions, resemble this format.

In any case, you have complete freedom in how you wish to assign version designators to your app. It is highly recommended that you stick to a well known and easily understandable format, such as `1.0`, `1.1`, `1.2`, and so on. Although you may want to include alphabetical characters (e.g., `1.0 Beta 2` or `6.3 Patch 123235-01`), we discourage it, as it can make it difficult for people to understand how the app versions relate to one another.

Keep in mind that the version of your app is completely up to you to specify, but the releases of Liferay with which your app works must be specified using Liferay's versioning scheme, as explained in [Understanding Liferay's Releases](#). See the later section *Specify App Packaging Directives* for details on specifying the releases of Liferay for which your app is designed.

14.1.3. What is a Package?

Apps can be written to work across many different versions of Liferay. For example, suppose you wish to publish version 1.0 of your app, which you're supporting on Liferay 6.1 and 6.2. It may not be possible to create a single binary .war file that works across both Liferay versions, due to incompatibilities between these Liferay versions. In this case, you need to compile your app twice: once against Liferay 6.1 and once against 6.2, producing 2 different *packages* (also called *variations*) of your version 1.0 app. Each package has the same functionality, but they're different files, and it is these packages that you can upload in support of different versions of Liferay, as you will see in a later section. In this guide, packages are sometimes referred to as files that make up your app.

14.1.4. How Do Apps Relate to Users and Companies?

When publishing an app, it is possible to publish it *on behalf of* yourself (an individual) or a *company* with which you are associated. The selection you make determines who has access to the app, once published. To understand the concepts of a Marketplace user, portal administrator, and company, and the ramifications of publishing apps as an individual versus publishing apps as part of a company, see the Leveraging the Marketplace chapter of Using Liferay Portal 6.2.

14.1.5. What Are the Requirements for Publishing Apps?

Liferay apps are "normal" Liferay plugins with additional information about them. Therefore, most of the requirements are the same as those that exist for other Liferay plugins, as explained in Developing Portlet Applications. In addition to those requirements, there are some Marketplace-specific ones to keep in mind:

- *Target the Java 6 JRE*: Your app's byte code must be compatible with Java 6 (i.e., Java 1.6). Liferay's Plugins SDK already targets Java 6 via the build.properties setting ant.build.javac.target=1.6; so don't override this setting. Your app will be rejected if its byte code is not compatible with Java 6.
- *WAR (.war) files*:
 - WARs must contain a WEB-INF/liferay-plugin-package.properties file.
 - WARs must not contain any WEB-INF/liferay-plugin-package.xml file.
 - WAR file names must not contain any commas.
 - WAR file names must conform to the following naming convention:

context_name-*plugin_type*-A.B.C.D.war

Where:

- *context_name* - Alpha-numeric (including – and _) short name of your app. This name is used as the deployment context, and must not duplicate any other app's context (you'll be warned if you use a context name of any other app on the Marketplace).
- *plugin_type* - one of the following: hook, layouttpl, portlet, theme, or web.
- A.B.C.D - The 4 digit version of your WAR file. 4 digits must be used.

Example: myapp-portlet-1.0.0.0.war

- WEB-INF/liferay-plugin-package.properties file:
 - Property recommended.deployment.context must not be set.
 - Setting property security-manager-enabled to true is mandatory for all paid apps on 6.1 CE GA3, 6.1 EE GA3, and later; the setting is optional for free apps. Setting this property to true enables Liferay's Plugin Security Manager. If you're enabling the security manager, you'll also need to define your Portal Access Control List (PACL) in this file. Read Developing Plugins with Security in Mind for information on developing secure apps.
- Deployment contexts:
 - Liferay reserves the right to deny an application if any of its plugin deployment contexts is the same as a context of another plugin in the Marketplace.
 - Liferay reserves the right to replace app plugin WAR files that have the same deployment context as plugins built by Liferay.



Important: If you're developing a paid app or want your free app to satisfy Liferay's Plugin Security Manager, see the section Understanding Plugin Security Management, for details. Give yourself adequate time to develop your app's PACL and time to test your app thoroughly with the security manager enabled.

Now that you've learned the packaging and deployment requirements for your app, let's consider the versions of Liferay you're targetting for your app and how to prepare your app for them.

14.1.6. Things You Need Before You Can Publish

You must first develop your app using your preferred development tool. For example, using Liferay IDE or Liferay Developer Studio, or the Plugins SDK. Your app will consist of one or more Liferay plugins. Ensure your app is designed to work with Liferay 6.1 or later. If you wish to target multiple versions of Liferay (for example, you may wish to support 6.2 EE SP1, 6.2 CE GA1, 6.1 EE GA3, and 6.1 CE GA3), ensure you have built binary images of your app for each supported minor family release, if necessary. If a single set of files will work across all supported Liferay versions, you do not need to build multiple plugins. Liferay guarantees compatibility within a given minor release family, so your users can rest assured that your app will work with the minor release that you specify, along with all future maintenance releases of that minor release.

Next, think of a good name and description of your app, along with a versioning scheme you wish to use. Take some screenshots, design an icon, create web sites for your app (if they do not already exist), and have a support plan in place.

14.1.7. Image and Naming Requirements

Icons for your app *must be* exactly 90 pixels in both height and width and must be in GIF, JPEG/JPG, or PNG format. The image size cannot exceed 512kb. Animated images are

prohibited.

Screenshots for your app *must not exceed* 1080 pixels in width x 678 pixels in height and must be in GIF, JPEG/JPG, or PNG format. The file size of each screenshot *must not exceed* 384KB. Each screenshot should preferably be the same size (each will be automatically scaled to match the aspect ratio of the above dimensions), and it is preferable if they are named sequentially, for example `fluffy-puppies-01.png`, `fluffy-puppies-02.png`, and so on.

Titles of Apps: In some views with Marketplace, titles of applications longer than 18 characters will be shortened with ellipsis. In the Marketplace, titles *must not be* longer than 50 characters.

Description, Tags, Websites and Version Numbers: Descriptions, web sites and version numbers are to be as reflective to the product as possible. Please do not use misleading names, information, or icons. A tags suggestion tool has been provided to aid with tagging your asset. Descriptions should be as concise as possible. Ensure your icons, images, descriptions, and tags are free of profanity or other offensive material.

Above and beyond these basics of creating apps in the form of Liferay plugins, there are additional considerations to take into account when designing and publishing apps.

14.1.8. What Kind of Validations Are Performed by Liferay?

Liferay ensures that apps meet a minimum set of requirements, by performing the following activities:

- Running basic anti-virus checks
- Ensuring titles, descriptions, images, etc. are appropriate
- Doing basic sanity checking of functionality (e.g., deployment testing, etc.)

Liferay does not do source code reviews and will not ask for your source code. Further, Liferay is not responsible for the behavior (or misbehavior) of apps on the Marketplace. For details regarding this, consult the *Liferay Marketplace User Agreement*, *Liferay Marketplace Developer Agreement*, and the individual *End User License Agreements* associated with each app.

14.1.9. What Versions of Liferay Should I Target?

Of course, targeting the widest possible range of Liferay versions in an app typically draws larger audiences to the app. And there are certain features in specific versions of Liferay that you may wish to take advantage of. When uploading apps, you can specify which versions your app is compatible with and you can have multiple files for your app that are designed for different versions of the Liferay Platform.

Note that apps on the Liferay Marketplace must be designed for Liferay 6.1 or later. That's not to say that they can't work with prior versions. However, only Liferay 6.1 and later versions provide support for installing apps directly from the Marketplace and provide safeguards against malicious apps. If you wish to use an app for an earlier version, consult the documentation for that app, as it may or may not be supported on earlier versions of Liferay.

Read the section below for details on how to specify the versions of Liferay your app works with.



Note: If you haven't yet done so, make sure to read the Leveraging the Marketplace chapter of *Using Liferay Portal 6.2*!

Now that we've covered the basics, you're armed with knowledge to start creating apps on the Marketplace, so let's see what that looks like in the next section.

14.2. ***Developing and Publishing Apps***

Let's jump right in with an example. In this section, we'll walk you through the creation and publication steps (but we won't actually publish the app on the Marketplace, since this example app isn't very useful!). After walking through this, you should understand typical Marketplace app development.

14.2.1. **Develop a Sample App**

Before you can publish anything, you first have to create (develop) an app! Since apps are nothing more than collections of individual plugins, your first step in developing a Marketplace app is to develop the functionality in the form of one or more Liferay plugins. To create a sample app that contains a single portlet, follow the detailed instructions in *Developing Portlet Applications*. After creating and deploying your sample app, return here to continue.

In the real world, apps usually consist of multiple components (e.g., multiple .war file plugins), are spread across multiple plugin types, and present non-trivial functionality which in many cases requires some configuration. How these advanced tasks are dealt with is out of scope for this section, but some tips and considerations for Marketplace development can be found in the sections that follow.]

14.2.2. **Specify App Packaging Directives**

When publishing your app, each plugin you upload is packaged into one or more *packages* for each Liferay release you intend to support. When you upload your plugins to the Liferay Marketplace, your app is scanned, and the embedded packaging directives you have specified are extracted and used to create different downloadable *packages* of your app for different Liferay releases. You must insert this information into each plugin in your app before you can publish it to the Marketplace.

The packaging directives are related to the Liferay releases with which your app is compatible. In order to specify which release of Liferay your app is compatible with (and therefore which packages should be created for eventual download on the Marketplace), you first need to understand how Liferay releases are named and how they relate to the underlying Liferay release version. Details can be found in the chapter *Understading Liferay Releases* in *Using Liferay Portal 6.2*. Accordingly, Liferay 6.2 CE GA1 is designated as version 6.2.0. CE GA2 is then 6.2.1, and so on. Liferay 6.2 EE GA1 is designated as 6.2.10. EE versioning follows a slightly different policy given the presence of fix packs and service packs, so 6.2 EE GA2 will be 6.2.20.

For each plugin that makes up your app, packaging directives must be placed in the `liferay-`

`plugin-package.properties` file (located in the `WEB-INF/` directory of your plugin's `.war` file). Within this file, you must specify a comma-separated list of Liferay releases with which your app is compatible and for which packages should be generated using the `liferay-versions` keyword. Marketplace will create packages that contain your plugins based on these packaging directives (and will intelligently group them together as each plugin is uploaded). You should specify CE versions first, followed by EE versions, using this form: `liferay-versions=CE,CE+,EE,EE+` (where `CE` and `EE` are replaced with the corresponding Liferay Releases with which your app is compatible).



Note: If your app is compatible with both CE and EE, you must specify a set of versions for both CE and EE releases. If you only specify compatibility with CE, then your app will not be compatible with (and will fail to deploy to) any EE release.

For example, to specify that a particular plugin in your app is compatible with Liferay 6.1 CE GA3 (and later), and 6.1 EE GA3 (and later), add this line to your `liferay-plugin-packages.properties` file:

```
liferay-versions=6.1.2+,6.1.30+
```

This means that the app works with any 6.1 CE release starting with CE GA3, and and 6.1 EE release starting with EE GA3. Marketplace will create two packages, one that is compatible with the 6.1 CE GA3 release and *later*, and another that is compatible with 6.1 EE GA3 release and *later*.



Note: Any CE or EE versions you include in your packaging directives *must* be terminated with a version using the `+` symbol. This ensures that your app will be deployable onto future versions of Liferay (but does not guarantee your app will work in future versions). So, `liferay-versions=6.1.1,6.1.2` will not work, but `liferay-versions=6.1.1,6.1.2+` will work. Similarly, `liferay-versions=6.1.2+,6.1.30,6.1.31` will not work (as the EE versions are not properly terminated), but `liferay-versions=6.1.2+,6.1.30,6.1.31+` will work.

Here are some additional examples:

```
# works with Liferay 6.1 CE and EE GA3 and later (NOT compatible with 6.1  
# CE or EE GA2). This is most likely what you want to use.  
liferay-versions=6.1.2+,6.1.30+
```

```
# works with Liferay 6.1 CE GA2, GA3, and GA5 (but not GA4), and EE GA2  
# and later  
liferay-versions=6.1.1,6.1.2,6.1.4+,6.1.20+
```

```
# works with Liferay 6.1 EE GA3 and later (NOT compatible with CE)  
liferay-versions=6.1.30+
```

You may find it advantageous to implement one of your app's plugins in multiple ways, customizing that plugin for different Liferay releases. We'll illustrate this with an example.

14.2.2.1. Example App: Using Different Versions of a Hook

Suppose your app consists of two plugins: a portlet and a hook. The portlet uses standard API calls that work on all Liferay 6.1 releases. Your hook, on the other hand, needs to interact with EE GA3 differently than it does with CE GA3, because you want the hook to take advantage of an exclusive EE feature. For your app, how do you provide one version of your hook plugin for EE and another version of it for CE, while applying your portlet plugin to both EE and CE?

It's easy. In this case, you'd specify versions `liferay VERSIONS=6.1.2+, 6.1.30+` for your portlet plugin, indicating that it is compatible with CE GA3 and later, and EE GA3 and later. As for your hook plugin, you'd create and build *two* versions of it, one version of the hook to use with Liferay EE and the other version of the hook to use with Liferay CE. You'd specify `liferay VERSIONS=6.1.30+` for your EE hook and `liferay VERSIONS=6.1.2+` for your CE hook. The EE hook would work exclusively with EE GA3 and later, while the CE hook would work exclusively with CE GA3 and later. You might think that it's difficult to arrange the packaging for an app that has plugins targeted to different Liferay releases, but it's easy.

Marketplace takes care of it based on the `liferay VERSIONS` values you specified for each plugin. We'll talk about that next.

14.2.2.2. Marketplace Packages Your App's Plugins

When you upload your app's plugins, as demonstrated later on in this chapter, you'll notice that Marketplace groups them into separate packages based on the respective releases each plugin supports. Marketplace copies a plugin into each of the release packages corresponding to its list of `liferay VERSIONS` values. If Marketplace cannot verify the version of Liferay the plugin supports, it rejects the plugin. For example, if you specify `liferay VERSIONS=1.0.0+` for your plugin--perhaps because you are confident it can work on any Liferay release--Marketplace will likely reject it, because it doesn't know of a 1.0 release of Liferay. So take care in specifying the Liferay version information for each your app's plugins.

Now that you've developed your app and specified its packaging directives, it's time to get it to the Marketplace!

14.2.3. Establish a Marketplace Account

Before you can publish anything to the Marketplace, you must first have an account on liferay.com. If you do not have an account, visit <http://liferay.com> and click *Register* in the upper-right corner of the screen. After you've registered, you can visit the Marketplace at <http://liferay.com/marketplace>. The Marketplace home page is shown below:

MARKETPLACE

Welcome James Hinkey!

Purchased

Apps

Communication

Productivity

Security

Utility

Templates and Themes

Page Layouts

Themes / Site Templates

EE Marketplace

Marketplace Overview

Marketplace User Guide

Marketplace Partners

Marketplace Community Forum

Marketplace Developer Portal

Become a Developer

Get Marketplace Plugin

Select a Country

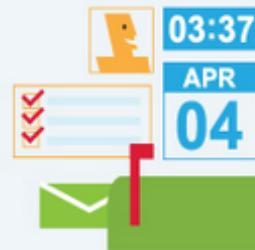
United States

[Home »](#)

Featured App:

Social Office

Get your team on the same page.



Featured Apps

[See All](#)



[Valamis - e...](#)

Arcusys Ltd.

★★★★★☆
Free



[Props 'n' P...](#)

Sébastien L...

★★★★★☆
Free



[Ask IRC](#)

Bijan Vakili

★★★★★☆
Free



[Mercado](#)

CIGNEX Data...

★★★★★☆
Free



[Remote jBPM...](#)

3F Consulting

★★★★★☆
Free



[Workflow Se...](#)

Permeance T...

★★★★★☆
Free

New and Interesting

[See All](#)



[Activiti Wo...](#)

EmDev Limited

★★★★★☆
Free



[Beveled](#)

Innotall GmbH

★★★★★☆
Free

Most Viewed

- [Social Office CE](#)
Liferay, Inc.
- [Zoe Tech](#)
Liferay, Inc.
- [Kaleo Workflow CE](#)
Liferay, Inc.
- [Web Form CE](#)
Liferay, Inc.
- [Zoe Brochure](#)
Liferay, Inc.
- [Social Networking](#)
Liferay, Inc.
- [Zoe Healthcare](#)
Liferay, Inc.
- [Kaleo Forms EE](#)
Liferay, Inc.
- [Zoe Resort](#)
Liferay, Inc.

This is the front page of the Marketplace and is where users go to find new and interesting apps. Since you'll visit here often during the course of development, you may want to bookmark it now.

You can publish Marketplace apps as an individual or as part of a company. Before you can

submit apps to the Marketplace, you must register yourself as an app developer. It's easy. Simply click *Become a Developer* in the *MARKETPLACE* column on the left. You're now in the Marketplace registration wizard. If you're registering with a company and the company is already registered, you can search for it from these Marketplace registration screens and request to *join* the company in publishing apps to the Marketplace. On completing the Marketplace registration, Liferay sends you an email confirming your acceptance as a Marketplace Developer-- Congratulations!

Now that you're a Marketplace Developer, options for adding new apps and viewing your published apps are available to you from your User Profile. Let's go there now. In the upper right corner on <http://www.liferay.com> select your picture → *User Profile*.

The screenshot shows the Liferay Marketplace interface. At the top, there is a navigation bar with links for Products, For Business, Developers, and Control Panel (with User Profile highlighted). Below the navigation bar, the main content area has a header 'MARKETPLACE' and a welcome message 'Welcome James Hinkey!'. There is a 'Home »' link and a large blue button. To the right, there is a sidebar labeled 'Control Panel' with a 'User Profile' link.

In the left side navigation panel of your profile page, there are links to pages related to using apps and developing apps. Links to *Apps* and *App Metrics* are listed in the *Development* section of the navigation panel. You'll use these links heavily during development; so you may want to bookmark this page too. Click *Apps* from within the *Development* section to access your app development page.

Apps



Add New App



Jim's ...

Version: 1.0

0 Downloads

Unsubmitted



Nose-ster E...

Version: 1.0

0 Downloads

Unsubmitted

James Hinkey

Showing 2 results.

My Account

Support

License

Apps

Projects

Development

Apps

Metrics

Now that you know how to get to your app development page, let's publish an app!

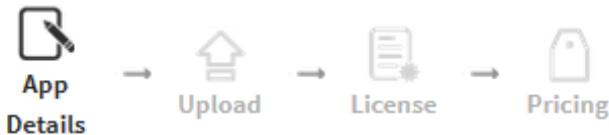
14.2.4. Upload (Publish) Your App

To begin the process of publishing your app, click *Add New App*. A form appears, allowing you to fill in your app's details.

14.2.4.1. Initial App Details

The first step is to enter the basic details about your app.

App Details



App Owner *

You are submitting this app as an individual or as a sole proprietor. [Create a company](#) if you are submitting on behalf of a company, corporation, limited liability company (LLC), non-profit organization, joint venture, partnership, or government organization.

James Hinkey

App Pricing *

You cannot change this after Liferay has approved your app. If you wish to have both a free and paid version of your app, you must submit them separately.

Free

Paid Licensing based on Liferay Instances

Note: In order to price and sell apps in the Liferay Marketplace, you must first [upgrade your developer account](#).

App Information

Title *

My New App

Description *

This is the description of my new app. It does amazing things.

Localized

Icon *

Icons must be a 90 x 90 PNG and cannot exceed 512KB. Icons cannot be animated images.

No file selected.

This screen allows you to enter basic details about the app you are publishing.

App Owner: Choose to whom the app "belongs" once it is uploaded--either yourself (personal) or a company. If you'd like to submit on behalf of a company, click *Create a Company* to go to the *Join a Company* page. From this page, search to see if your company already exists in the Liferay Marketplace. If you wish to publish your app on behalf of your company, but your company does not yet have a Marketplace profile, register your company by clicking the *Register My Company*, filling out the registration form, and submitting the form.

Publishing on behalf of yourself is the default. When you publish on behalf of yourself, your name appears in the Marketplace as the Publisher/Author. You are the only one who can manage your personal app (add new releases to it, add new versions of it, edit its details, etc).

Publishing on behalf of a company effectively hands the keys over to the admins of the company. The app only appears on the company's Marketplace app development page. In addition to yourself, company admins can manage this app (add new releases, new versions, edit details, etc). The app appears to be authored/developed by the company, not you personally. It also appears on the company's public profile page under its list of apps.

App Pricing: Choose whether you want the app to be free or paid. If you choose paid, you'll have the option to specify pricing and licensing details later in the submission process. Once your app is published to the Marketplace, you cannot change this option. If you wish to have both free and paid licenses for your app, you must submit one version of your app for free licenses and another version of your app for paid licenses.

Title: The name of your application. Arguably the most important detail of your app, the name of your app should be a good title that conveys the function of the app but is not overly wordy or misleading. Choosing a good name for your app is key to its success, so choose wisely! Do not include versions in the title unless it is a vital part of the name, for example *Angry Birds 2: More Anger*. Each app on the Marketplace must have a unique name.

Description: Like the name says, this is a description of your app. You can put anything you want here, but a good guideline is no more than 4-5 paragraphs. This field does not allow any markup tags or other textual adornments--it's just text.

Icon: The icon is a small image representation of the app. See the *Marketplace Basics* section of this chapter for detailed requirements for icons.

Screen Captures: You can supply multiple screenshots of your app in action. The screenshots you upload here are displayed when your app is viewed on the Marketplace, using a carousel of rotating images. See the *Marketplace Basics* section of this chapter for detailed requirements for screen captures.

Category: Choose the Marketplace category that most accurately describes what your app does. Users looking for specific types of apps will often browse categories by clicking on a specific category name in the main Marketplace home page. Having your app listed under the appropriate category will help them find your app.

Developer Website URL: This is a URL that should reference the web site associated with the development of the app. For open source apps, this typically points at the project site where the source is maintained. For others, links to a home page of information about this app would be

appropriate.

Support URL: This is a URL that should reference a location where purchasers or users of the app can get support for the app.

Documentation URL: What better way to showcase the amazing capabilities of your app than to have a live, running version of it for potential buyers to see and use? This field can house a URL pointing to exactly that.

Source Code URL: If you'd like to provide a link to the source code of your app, do so here.

Labs: You can denote an app as experimental by flagging the appropriate box.

Security: If your app does *not* use Liferay's PACL Security Manager, flag the appropriate box. Otherwise, make sure to enable the security manager in your app by including the setting `security-manager-enabled=true` in your `liferay-plugin-package.properties` file.

Tags: A set of descriptive words that categorize your app. These tags are free-form and can help potential purchasers find your app through keyword searches, tag clouds, and other search mechanisms. You can click on *Suggestions* to let Marketplace suggest some tags for you based on the data you've already entered. Click *Select* to select from existing tags, or you can manually type in new tags. See the *Marketplace Basics* section of this chapter for detailed requirements for tags.

EULA: You can either use the default end user license agreement or provide your own. There's a link to the minimum terms which custom EULAs must satisfy.

Liferay Only: If you're publishing a CE-only or EE-only app, select *CE Only* or *EE Only* in the *Product Type* dropdown selector. If you're providing an app that runs on both CE and EE, just flag the *Liferay EE Plugin* checkbox. If you're uploading a bug fix or would like to replace a previous version of your app, specify the app entry IDs of the apps to be replaced by the new app in the *Supersedes App Entry IDs* field.

Make up some sample data to use during this example, and enter it into the form. Once you have entered all your app's details, click *Next* to move on to the next screen.

14.2.4.2. *Upload Files (Plugins) for your App*

On this screen, you must specify the version of your app and upload its plugin files. Review the previous section *What is a Version?*, to decide on a good version specifier and enter it here. For our example, since this is the first version, enter `1.0`.

Then upload the different sets of plugin files (variations) to support different Liferay versions you're targeting. You must upload at least one plugin file before advancing beyond this screen. So, click the *Browse* button, and select the plugins that make up your app. Each time you add plugins to the list, they automatically begin uploading and their compatibility information is scanned (read the previous sections in this chapter to understand what compatibility information is read from your plugins).

Edit App Version

[« Back](#)



Version Number *

1.0

Upload Liferay Plugin Packages *

Please be sure to specify Liferay compatibility through the appropriate properties or XML files in your plugins. You can select multiple files for upload.

[Browse](#)

Liferay Portal 6.2 CE GA1+

event-listing-portlet-6.2.0.1.war

[Delete](#)

[Upload Source Code](#)

[Back](#)

[Save as Draft](#)

[Next](#)

As a more complicated example, let's consider an app that consists of a hook and a portlet. The portlet works across all Liferay releases, but the hook is built separately for CE and EE.

Therefore, you would upload 3 plugins that make up the app: 1 portlet plugin for all releases, 1 hook plugin for CE, and 1 hook plugin for EE. Once the files are uploaded, a check mark appears next to each plugin, and the plugins are displayed based on their compatibility information. This indicates that the files were successfully uploaded. The portlet plugin is automatically copied for use in both the EE and CE variations, even though you only uploaded the portlet plugin once.

If you selected *Free* for your app pricing, click *Next* to advance to the final screen. If you selected *Paid*, you'll be presented with additional options for licensing and pricing your app.

14.2.4.3. *Creating your licensing and pricing model*

Carefully consider which licensing structure best meets your needs. Once your app has been approved, these options, with the exception of pricing updates, cannot be changed.

Choose a license term:

License Term

A perpetual license is permitted to run without expiration. A non-perpetual license has an annual expiration.

- Perpetual
- Non-Perpetual

Support

If support is offered, customers can only receive updates during the support contract period. Customers who purchase support may contact you with support requests for your App.

- Offer Support

Choosing *Perpetual* allows the app to continue running without expiration, whereas choosing **Non-Perpetual*** expires the app's license one year from the purchase date. Perpetual License also allows you to offer Support Services, which the customer must renew annually to maintain access to app updates and support. If you choose not to offer Support Services with a Perpetual License, customers will be provided with app updates only, whenever updates are available. You cannot change your app's license terms once the app is approved.

Creating license options:

Standard Licenses

Add Quantity

1 Instance

[Delete](#)

2 Instances

[Delete](#)

5 Instances

[Delete](#)

Developer Licenses

Add Quantity

10 Instances

Creating license options allows you to design license bundles and to specify discounts for customers who purchase more Liferay Instances for your app (a Liferay Instance or Instance refers to a single installation of the Liferay Portal). Also you can designate different pricing for Standard Licenses vs. Developer Licenses. Developer Licenses are limited to 10 unique IP addresses. You must specify at least one license option, but no more than 10 options per type. You'll price these options on the next page.

Paid support: You can offer additional paid Support Services for your app. If you select this option, customers can contact you with support requests and are entitled to regular updates.

Offer a trial: You can offer a free 30-day trial of your app, restricted to 1 Instance and 25 users.

When you're finished selecting all the options for your license, proceed to the next page to determine the app's pricing and availability.

Pricing:

App Pricing

[« Back](#)



Pricing and Availability

Set your prices in table below then drag the countries from the list on the right.

Default Pricing

Currency

Standard Licenses

1	
2	

Developer Licenses

1	
---	--

Countries

Drag countries here to sell at these prices.

[Select All](#) [Deselect All](#)

[Add Pricing Table](#)

Available Countries

[Select All](#) [Deselect All](#)

- Canada
- France
- Germany
- Hong Kong
- Hungary
- Israel
- Italy
- Japan
- South Korea
- Netherlands

Based on your selections from the previous page, you'll have price fields for each license option and for any support option you offered.

Choose the currency to use with the pricing options and then fill in the price fields accordingly. Fill in the renewal cost for any Support Services you offered. The Support Services price is based on per Instance, so for example, if you entered \$100 USD and the customer is running 10 Instances, their annual Support Services renewal cost would be \$1000. Note: This only applies to Perpetual Licenses. For Non-Perpetual Licenses, you should include any Support Services cost in the annual license price.

Once you've specified the prices, you can add the desired countries to this box. If you wanted to specify different prices for different regions, add a new table and complete the fields as desired.

Although the Liferay Marketplace supports major currencies and a broad list of countries, not all

currencies and countries are currently available. Additional currencies and countries may become available at a later time.

When you have completed your app's pricing and availability, click *Next* to advance to the final screen.

14.2.4.4. *Preview and Submit Your App*

Whenever you make a change (app details, adding files, adding new versions), you always wind up at the *App Preview* screen. This allows you to preview your app as it will appear in the Marketplace, so you can confirm your changes.



Nose-ster Events

James Hinkey

Event Listing Portlet

This is the Event Listing Portlet portlet in View mode.

Name	Description	Location	Date	Actions
Tour Museo del Prado	Enjoy the "Captive Beauty" exhibition--"Hidden Beauty. From Fra Angelico to Fortuny."	Museo del Prado	01/11/2013 11:00 AM	
Muffaletta Mania!	Huge amazing sandwiches dressed with "the works!"	New Orleans	04/03/2014 03:30 PM	

Latest Version: 1.0

Total Downloads: 0

[Developer Website](#)

[Documentation](#)

[Support](#)

[Source Code](#)

[License Agreement](#)

Current Requirements

Liferay Portal 6.2 CE GA1+

Security Disabled: This app has [Liferay's PACL Security Manager](#) disabled.

Labs: This app is experimental.

Disclaimer: This app was submitted by a Liferay employee but is in no way affiliated with Liferay, Inc. or its affiliates.

Keeps you updated on all of the Nose-ster happenings around the world.

Uploaded Plugin Files

For this example, review the information. Is it as you expect? If not, click *Edit* to go back and continue making changes until you are satisfied.

Once you are satisfied, click *Submit for Review*. If you're walking through this example on

Liferay's Marketplace, don't do it, since this is only an example app. The next section describes what happens when you submit apps or app changes.

14.2.5. The Review Process

When you submit apps to the Marketplace, they are reviewed by Liferay Marketplace staff to ensure that your app meets the minimum standards described in the previous section *What Are the Requirements for Publishing Apps?*

Each of the following changes require a review by Marketplace staff before the change is published to the Marketplace:

- Submitting a new app
- Changing details of an app (for example, changing the description or the screenshots)
- Adding a new package (set of files) to an existing app, in order to support more Liferay releases
- Adding a new version of an existing app

While your submitted change is under review, you can view the status of your change by visiting *Home → Apps*. You can also cancel your submission by clicking *Cancel Submission* on the *App Preview* screen for each app.

Once your app is approved by Marketplace staff, congratulations! You will receive an email confirmation and at that moment, your app is available on Marketplace. The app is also shown on your public Profile page, which lists all apps that you have personally developed and published.

If your app is rejected, an email will be sent to the email address associated with the app, along with a note explaining the reasons for rejection. At that point, you can make the requested changes, and re-submit the app for approval.

Now that you have successfully published your first app, you'll likely get all kinds of feedback from users and yourself about what's right and wrong with it. In the next section, we'll explore how to make changes once you have published your app.

14.3. Making Changes to Published Apps

After your app is published and approved, you will undoubtedly need to make one or more of these kinds of changes during the life of the app:

- Editing your app details (e.g., description, icon, etc)
- Editing app prices
- Adding support for a new version of Liferay Portal
- Releasing a new version of your app to fix bugs or offer new functionality
- Disabling your apps

Liferay Marketplace supports all of the above operations as described below.

14.3.1. Editing Your App Details

App details include the name, description, icon, screenshots, and other information that you supplied on the first screen during the app creation process. To make changes to this content for

your app, navigate to *Home* → *Apps* and select the app you wish to edit. Click the *Edit* button to edit that app.

This screen shows you what the app looks like on the Marketplace. To edit the detail information, click the *Edit* button at the bottom of the preview. This allows you to edit details (as well as add new files to your existing version). Note that the current values as they appear in your app are used to pre-fill the form. Make any changes as needed on this screen, and click *Next*. If you do not need to edit any more variations, you can continue clicking *Next* until you reach the final preview screen. Click *Submit for Review* to submit your detail changes for review. Once approved, the changes you request appear on the Marketplace.

14.3.2. Editing App Prices

You can change your app's prices, Instance bundles, and regional availability if you wish. This can be for a variety of reasons, whether it's to run a promotional offer, or to adjust your pricing model to better account for app demand. To make changes to your app, navigate to *Company Profile Home* → *Apps*, then click the app you wish to edit, and then at the bottom, click *Edit* → *Pricing*. When you've finished editing your app's prices, click *Next*.

These changes do not require Liferay verification process to approve. Simply make the changes you wish and save your app. These new prices will be reflected immediately.

14.3.3. Adding Support for New Versions of Liferay Portal

If you need to add files in support of another Liferay release, the process is similar. Navigate to *Home* → *Apps* and select the app you wish to edit. Click the *Edit* button to edit that app. Click *Next* to advance past the details screen (making any changes as needed), and click *Next* to advance past the version edit screen (you can't actually edit the version number of an already-approved version, but you can edit the "What's New" information if needed).

Once you advance past the version edit screen, you'll be at the File Upload screen. This screen should look familiar--it's the same workflow used when you initially created your app! The difference is that you can't edit pre-approved files for specific Liferay releases. You can only add *new* files for a different Liferay release (if you actually need to update existing files, you must create a new version of the app--see the later section on adding versions for details on how to do this).

Upload your new files (ensuring that your new plugins have updated compatibility information, see the section on *Specify App Compatibility* for details on versioning), click *Next*, and observe the newly-added files listed at the bottom of the preview screen. Click *Submit for Review* to submit your requested change (adding of files). The files will be reviewed by Liferay, and once approved, the new package is available for download in the Marketplace.

14.3.4. Releasing a New Version of your App

After time passes, you may wish to add new functionality to your app or fix a batch of bugs. This can be accomplished by releasing a new version of your app. New versions offer your users new functionality and bugfixes, and users are generally encouraged to always use the latest version.

In addition, when a new version of your app becomes available, existing users are notified automatically through Liferay's notification system.

New versions of your apps are created similarly to the way the initial version was. To add a new version, navigate to *Home* → *Apps* and select the app you wish to edit. Click the *Edit* button to edit that app. You will be taken to the Details screen. At the bottom of the Details screen, click the *Add New Version* button. This button begins the process of adding a new version, starting with the App Details screen. In this case, the screen is pre-filled with data from the current version of the app.

You can make any changes to the pre-filled data on this screen. Since this is a new version of an existing app making major changes (such as completely changing the name or description) might be unsettling to your existing users. It is common that you'll want to upload new screenshots and refresh the icon. Note that you cannot change the app owner (such as moving from a personally-developed app to a company-developed app).

Clicking *Next* takes you through the same screens you've already seen. On the *Add App Version* screen, you can specify a new version name for this version of your app. Also, note that when adding new versions to an existing app, you have the option to add *What's New* text. This is typically filled in with a list of changes for this version, such as significant new features or bugfix information. Clicking *Next* from here allows you to upload the files associated with the new version of the app. For a new version of the app, you must upload all files for all supported Liferay versions again, even if they have not changed since the last version.

14.3.5. Deactivating Your App

When the time comes to retire your app, you can *Deactivate* it. Deactivating an app causes the app to no longer be downloadable from the Marketplace for new customers and it won't appear in any public Marketplace listings. Existing customers that have already downloaded your app can continue downloading the legacy versions of the app they have already acquired, but they can't download any versions they've not already received. The app remains in your inventory, with all of its history, in case you need to re-activate or reference it in the future.

To deactivate your app, navigate to *Home* → *Apps* and select the app you wish to deactivate. Click the *Deactivate* button.

14.4. Tracking App Performance

One of the main reasons for developing and publishing apps into the Marketplace is to drive downloads and adoption of your app. The Marketplace enables you, as the developer of your app, to get detailed reports about the number of views, downloads, and purchases of your app(s). To access these metrics, navigate to *Home* → *Metrics* (under *Development*).

Metrics

App Performance Top Performers

Interval

Start Date

End Date

Month ▾

January ▾

1 ▾

2013 ▾



January ▾

1 ▾

2014 ▾



Apps

1-2-1 Columns Layout

CE

1-2-1 Columns Layout

EE

1-3-1 Columns Layout

CE

1-3-1 Columns Layout

EE

1-3-2 Columns Layout

CE

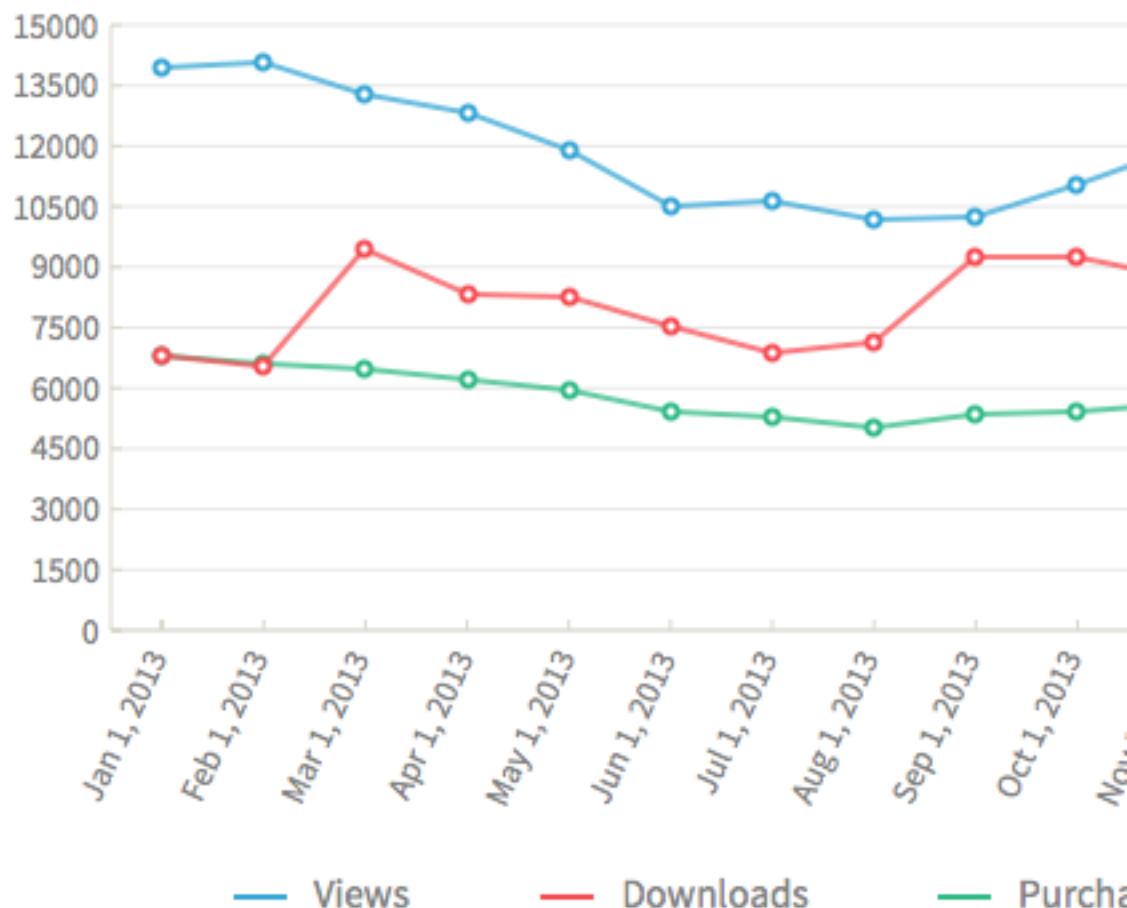
1-3-2 Columns Layout

EE

2-1-2 Columns Layout

CE

2-1-2 Columns Layout



The view shown above is the default metrics view for a single app. Across the top is a list of data series options (*Views*, *Downloads*, or *Purchases*). Below that, a date range can be chosen. In the middle, a graph is shown for the data within the date range. Finally, the same data that is graphed is also shown in tabular format, in case you want to know the exact values making up the graph.

The different types of data available to view are described below.

14.4.1. Views

When someone searches or browses the Marketplace, they click on apps to see detailed views of the apps they're interested in. When this occurs for your app, a *View* is recorded for the app, and this data is what is shown on the App Metrics screen when *Views* is selected at the top. *Views* is also the default view, as shown above. The number of recorded views per day per user is unlimited.

14.4.2. Downloads

A download is recorded for your app when someone downloads a specific package of a specific version of your app. The number of recorded downloads per day per user is unlimited.

14.4.3. Purchases

The Marketplace makes an effort to count the total number of purchases for your app.

Now that you understand how to publish your app and track its performance, let's get a good understanding of Liferay Portal's Plugin Security Manager and the security information you'll need to provide with your app.

14.5. *Understanding Plugin Security Management*

We all wish cyberspace was free of malicious software and unwanted bugs. Since it isn't, we need to guard ourselves and our portals from these evils. Enter Liferay Portal's Plugin Security Manager! It's like a super-hero in a cape and tights, except, well, it's not.

In its quest for peace within your portal, the Plugin Security Manager pledges to:

- Protect your portal and host system from unwanted side affects and malicious software introduced by plugins.
- Control plugin access to your portal, host system, and network by requiring that plugins specify ahead of time the portal resources they intend to access.

Let's go over some scenarios that could apply to you with regard to trying new plugins, and then maybe the importance of this will be clear.

- A flashy new plugin has arrived on Liferay Marketplace and you want to give it a whirl. But naturally, you want to know the parts of your system it will access.
- A colleague finds an interesting plugin after scouring the web for something that can help streamline processes at your workplace. Of course, you don't know whether you can truly trust the plugin creator--this plugin was found outside the Liferay Marketplace. If the plugin isn't open source, you have no way of knowing if it does anything nefarious.
- Upper management requests your corporate branch and other branches use a standard set of plugins on your portal instances. This set of plugins, however, was written by an outside firm, and you must assure that the plugins will not tamper with your proprietary files.

These are just a few scenarios that may ring true for you. When you're responsible for keeping

your system running well 24x7, you can't be too cautious in protecting your portal, system, and network.

When the Plugin Security Manager is enabled for your plugin, it checks your plugin's *Portal Access Control List (PACL)*. This list describes what APIs the plugin accesses, so people deploying the plugin can review what it does without seeing its source code. If the plugin tries to access anything that's not on this list, the plugin's request is stopped dead in its tracks and the security manager logs information on the plugin's attempt to access the unauthorized APIs or resources.

Access to APIs and resources is authorized by means of property values specified in the plugin's `liferay-plugin-package.properties` file. This file must be specified in your plugin's `WEB-INF` directory. These security management properties are collectively known as the plugin's PACL.

As you develop plugins for Liferay Marketplace or for distribution within your organization, you'll need to set the security management properties appropriately. Before we dive into the intricacies of these properties, let's consider a plugin development approach that involves designing an app for the security manager from the ground up.

14.6. *Developing Plugins with Security in Mind*

At the start of plugin development, you may not have a clear picture of all the aspects of the portal you'll need to access, and that's fine. In fact, we suggest you go ahead and develop your plugin first and address your plugin's Portal Access Control List (PACL) later. But, as you develop your plugin there are some common security pitfalls, highlighted in the next section, that you'll want to avoid. After you develop your plugin you'll dig whole-heartedly into security management by generating and fine-tuning your plugin's PACL. Don't worry, we'll guide you through the entire process.

If you're developing a plugin as part of a free app, writing the plugin's PACL and enabling the security manager for the plugin are optional, and you can skip the remainder of this chapter. Otherwise, read on.

Here is the suggested approach for developing secure plugins:

- Consider common security pitfalls.
- Develop your plugin.
- Build your plugin's PACL using Liferay's PACL Policy Generation tool.
- Test your plugin thoroughly, with the security manager enabled.
- Add to your plugin's security policy, as needed.
- Convert your policy's absolute file paths to relative paths.

Let's go over each part of this approach.

14.6.1. *Consider Common Security Pitfalls*

As you develop your plugin, you need to anticipate your plugin's actions in light of Liferay's secured environment. The security manager leverages the Java SE Security Architecture. So understanding Java SE Security and learning the few requirements that Liferay's security

manager adds on top of it will benefit you as you develop plugins. The extensive Java SE Security Architecture documentation is available for you to read at <http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc.html>. But we'll highlight a couple common mistakes developers make that violate Liferay's secured environment:

- Invoking a method, directly or indirectly, without considering whether the method can throw a security exception.
- Using external libraries or external frameworks that access classloaders outside of your plugin.

You wouldn't intentionally make these kinds of mistakes, but you'd be surprised at how easily you can make them if you're not being careful enough. We'll consider scenarios that illustrate both of these mistakes and explain how to avoid making them in your plugin. Let's consider security exceptions first.

When you're running on Liferay Portal with the security manager enabled for your plugin, you must only access authorized resources. If you invoke a method declared as throwing a security exception (i.e., `java.lang.SecurityException`) and you're not authorized to access the resources the method uses, the method throws the security exception and the Security Manager stops your plugin dead in its tracks. Security exceptions are unchecked, meaning that the compiler doesn't require your code to handle them. But since methods that throw security exceptions are declared as throwing them, you should check their signatures while you're designing your plugin. If the methods your plugin uses throw security exceptions, handle them appropriately with try/catch blocks. Keep in mind that you not only need to handle security exceptions of methods your plugin invokes *directly*, but you also need to handle the security exceptions of the underlying methods your plugin invokes *indirectly*.

For example, you may be using a file utility that calls `java.io.File`'s `canRead` method. Since the `canRead` method can throw a `SecurityException`, your plugin will violate security if it invokes the utility on a file that you're not authorized to access. So, be aware of all security exceptions thrown by methods your plugin calls directly and indirectly.

Operations involving reflection, and similar activities, typically can throw security exceptions. The Java SE Security documentation explains how to deal with them. In many cases, you can declare your plugin's permissions to avoid running into these exceptions. We'll go over your plugin's permissions and security policies later in this chapter.

The second common mistake you should avoid is allowing your plugin to bring up classloaders unintentionally, via other frameworks or libraries. Consider the following Spring configuration from a plugin:

```
<bean id="userServiceBeanFactory"
      class="com.liferay.portal.service.UserLocalServiceUtil"
      factory-method="getService"
/>
```

It declares a factory bean that calls a method on a Liferay class. This seems reasonable, right? Unfortunately, Spring tries to grab the classloader for the factory class. Since the factory class does not belong to the plugin, the security manager balks at the plugin's attempt to access the classloader for the factory class. The security manager forbids applications from accessing

arbitrary classloaders because the classloaders can add, access, and modify classes that your plugin is unauthorized to access. Using Spring in this manner violates the secured environment.

How do you get around this issue? You could simply invoke the method directly like this:

```
UserLocalServiceUtil.getService()
```

But if you insist on using a Spring factory bean, you can do the following:

1. Write a class *inside* your plugin to act as a factory. Your factory class should declare a class that wraps the type of instance your factory returns. Your factory should also implement a method that returns the instance, wrapped in the class you declared.
2. Configure a Spring factory bean that uses your plugin's factory class.

Here's what your plugin's new factory class could look like:

```
package test;

// Add imports here ...

public class FactoryUtil {

    public static UserLocalService getUserLocalService() {
        TestUserLocalServiceWrapper localServiceWrapper =
            new TestUserLocalServiceWrapper(
                UserLocalServiceUtil.getService());

        return localServiceWrapper;
    }

    private static class TestUserLocalServiceWrapper
        extends UserLocalServiceWrapper {

        public TestUserLocalServiceWrapper(
            UserLocalService userLocalService) {

            super(userLocalService);
        }
    }
}
```

The code above declares a factory class named `FactoryUtil` that resides in a package named `test`. The factory declares an inner class named `TestUserLocalServiceWrapper` that extends Liferay's `UserLocalServiceWrapper` class. Note,

`UserLocalServiceWrapper` in turn wraps `UserLocalService`--the class you want the factory to return. Lastly, the `getUserLocalService()` method uses the original factory method, `UserLocalServiceUtil.getService()`, to get the `UserLocalService` instance. This instance is wrapped up in your factory's `TestUserLocalServiceWrapper` class. In your plugin, you've implemented a factory class to access the instances you want. That wasn't so difficult, was it?

Your new Spring factory bean would look like the following configuration:

```
<bean id="userServiceBeanFactory"
      class="test.FactoryUtil"
      factory-method="getUserLocalService"
/>
```

Great! Now you know a couple alternatives to using the troublesome Spring factory bean configuration that was accessing a classloader that didn't belong to your plugin.

With regards to both of the use cases we've illustrated, the main point we're emphasizing is that you must be aware of the how the libraries you use behave with respect to your secured environment. The better you understand Java SE Security and Liferay's Plugin Security Management, the easier it will be for you to write security-aware plugins. Keeping this in mind, you can proceed confidently creating your plugin.

14.6.2. Develop Your Plugin

Start creating your plugin the way you normally would. Design your application, write code, unit test your code, and have users beta test your app. In essence, do everything you would normally do. Do all of this with the Plugin Security Manager disabled via your plugin's `liferay-plugin-package.properties` file:

```
security-manager-enabled=false
```

Before the Plugin Security Manager is enabled, you must specify the resources your plugin accesses. Let's build a list of these resources in your plugin's PACL.

14.6.3. Build Your Plugin's PACL

Rather than tediously figuring out all of the resources your plugin accesses, on your own, let Liferay's PACL Policy Generation tool to give you a head start. The generation tool detects resources your plugin accesses and writes corresponding PACL properties to a policy file. You can then merge the PACL properties from this policy file into your plugin's `liferay-plugin-package.properties` file.

Here's how you generate a PACL policy for your plugin:

1. Make sure your Liferay Portal instance has `liferay` set as its security manager strategy value and that the security manager was activated during application server startup.

In your `portal-ext.properties` file, make sure Liferay Portal's security manager strategy is specified as follows:

```
portal.security.manager.strategy=liferay
```

Your app server may require that certain startup arguments be used for activating the security manager. Check the PACL and security manager instructions for your app server in the Installation and Setup chapter of *Using Liferay Portal 6.2*. Some app servers, like Tomcat, output a terminal message, like "Using Security Manager", indicating that it's using the security manager.

Unless you already started Liferay with the security manager enabled and activated as described above, you must restart Liferay with these settings.

2. Enable the security manager to generate a security policy for your plugin by setting the

following property in your plugin's `liferay-plugin-package.properties` file:
`security-manager-enabled=generate`

3. Deploy your plugin.

The PACL Policy Generation tool writes a PACL policy file with the following path:

`[liferay.home]/pacl-policy/[servletContextName].policy`

On deploying your plugin and as you exercise your plugin's features, Liferay Portal's security manager performs security checks on your plugin; but rather than throwing errors on failed checks, the generator tool writes suggested rules that specify access to the resources your plugin accesses.

Unless you've turned off logging for the generator tool, messages like the ones below are logged, reporting the various authorization properties that the tool generated

```
DEBUG [localhost-startStop-2] [GeneratingPACLPolicy:230] my-pacl-portlet
generated authorization property {key=security-manager-properties-read,
value=log4j.configDebug}
DEBUG [localhost-startStop-2] [GeneratingPACLPolicy:230] my-pacl-portlet
generated authorization property {key=security-manager-properties-read,
value=line.separator}
```

4. Lastly, merge the properties that the security manager wrote (i.e., your newly generated PACL policy file `[liferay.home]/pacl-policy/[servletContextName].policy`) into your plugin's `liferay-plugin-package.properties` file. It's just a matter of merging the properties that start with the "security-manager-" prefix.



Note: There is a known issue LPS-41716 in which Liferay may need to be restarted after deploying your plugin, in order for the security manager to detect and write out the complete set of policies for a plugin. If you are using your plugin with the "security-manager-" generated properties the first time and notice security violations, then you may need to turn on policy generation one more time and restart Liferay. This gives the security manager another opportunity to detect additional properties to satisfy your security policy. If you are still seeing security violations on deployment, you'll need to address them per instructions that follow in this chapter. Here are the work-around steps: Remove the previously generated `[servletContextName].policy` file, set `security-manager-enabled=generate` in your `liferay-plugin-package.properties` file, restart Liferay, redeploy your plugin, and merge any new properties from the newly generated `[servletContextName].policy` file into your `liferay-plugin-package.properties` file.

Now that you've thoroughly specified the resources your plugin accesses, let's enable the security manager and do final testing of your PACL properties.

14.6.4. Test the Plugin with the Security Manager Enabled

If you want to distribute plugins, either through the Liferay Marketplace or through your web site, you have to assume potential users will insist the Security Manager be enabled in your plugin. For this reason, you should enable it when testing your plugins.

To enable the Security Manager set the following `liferay-plugin-package.properties` property to true:

```
security-manager-enabled=true
```

Then, re-deploy your plugin and re-test its functionality. The Security Manager throws Java security exceptions, if your plugin accesses resources that are not specified in your plugin's security policy. As you test, keep track of these Java security exceptions, so you can authorize access to the respective resources in the PACL properties of your `liferay-plugin-package.properties` file. Save your changes to the file, re-deploy the plugin, and re-test. Make sure everything works. If not, there are more rules you must declare for your plugin. Refer to the online definition of the Portal Access Control List Properties for the `liferay-plugin-package.properties` file at http://docs.liferay.com/portal/6.2/propertiesdoc/liferay-plugin-package_6_2_0.properties.html and in the PACL properties section of this chapter for additional details.

If you are not finding an adequate way to specify a security rule with PACL, you can specify it in a Java Security Policy file. It's almost impossible for Liferay and PACL to be aware of every possible security implementation check, because developers, libraries, and the Java Security API can always call for new types of security checks. So, Liferay provides a fallback to PACL, that lets you specify operations permissible within the context of your app's plugins.

In case you need it for your plugin, let's get familiar with the Java Security Policy file.

14.6.5. Using a Java Security Policy File

If you cannot find a way to specify PACL permissions for an operation that your plugin must access, you can specify the permission in a Java Security Policy file. You can create the policy file (`java.policy`) in your plugin's WEB-INF folder. The policy file must follow Policy File syntax as described in detail at

<http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html#FileSyntax>. Like the rules you define in your plugin's PACL, the additional rules you define in your plugin's Java Policy File, WEB-INF/`java.policy`, only apply to that plugin. Plugins aren't privy to each other's security policies.

Importantly, the Java policy file should only be used to specify rules Liferay's PACL property implementation does not already support. You should not specify any rules in a Java policy file that you can specify in a PACL.

Here's a scenario that calls for using a Java Security Policy:

Java has a security implementation called

<http://docs.oracle.com/javase/7/docs/api/java/net/NetPermission.html>. It checks a whole bunch of networking operations, that Liferay's implementation doesn't check. In case you want to perform

one of these operations, like using a custom Stream Handler, you can grant your plugin permission to do so in its WEB-INF/java.policy Java Security Policy file. Here's one way to specify that rule:

```
grant codeBase "file:${my-supercool-portlet}${/}-" {  
    permission java.net.NetPermission "specifyStreamHandler";  
};
```

This grant entry defines permission for the plugin's code to access the specifyStreamHandler target operation of the java.net.NetPermission class. The codebase value, in this example, specifies the following:

- file : indicates the code resides on the server's file system.
- \${my-supercool-portlet} represents the context path of a plugin named "My Supercool Portlet". The context path is a system property Liferay generates for the plugin. It maps the context path name to the plugin's fully qualified deployment path.
- \${/} represents the system's path separator.
- - matches files and folders, in this folder and below.

On reading this plugin's .jar file, the JVM creates a codebase for it. The codebase uses properties that Liferay sets for the plugin that say, in effect, "If a file originates within the plugin, then this plugin can perform the specifyStreamHandler operation on it". The codebase narrows the scope for the permission. This plugin is permitted to perform the defined operation, specifyStreamHandler, as long as it is done within the scope the plugin.

How do you add more permissions to a codebase? Just define them on separate lines in the grant entry:

```
grant codeBase "file:${my-supercool-portlet}${/}-" {  
    permission java.lang.RuntimePermission "loadLibrary.test_b";  
    permission java.net.NetPermission "specifyStreamHandler";  
};
```

In this example, we've granted the plugin permission to invoke native code that's in some library (test_b.so). This is another type of operation which Liferay's PACL does not support. So, it makes sense to specify permission for it in the Java Security Policy file.

With Liferay's PACL policy and Java Security Policy files, you can precisely specify all of the resources your plugin needs to access! Next, let's revisit the file path values that the PACL Policy Generation Tool wrote to your liferay-plugin-package.properties file.

14.6.6. Convert PACL Absolute File Paths into Relative Paths

As mentioned earlier in this chapter, we recommend using the PACL generation tool to give you a head start on specifying your plugin's security rules. But The generator is only aware of file paths with respect to the current system, and therefore generates them as absolute file paths. In order to use your security policy in production, it must use only relative file paths. So, as a final step after testing the generated PACL, you must massage the generated file paths into appropriate relative file paths. For example, you can specify paths relative to your Liferay web portal directory:

```
security-manager-files-read=\  
    ${liferay.web.portal.dir}/WEB-INF/tld/-,\
```

```
 ${liferay.web.portal.dir}/html/themes/-
```

In this example, we used a dash (-) character at the end of the paths. We use this as a wildcard character. Oracle defines wildcards for use with Java Security, and Liferay provides some too. Let's consider some helpful wildcards you can use in PACL properties and Java Security policies.

For files and file paths, you can leverage the following wildcard characters:

- Dash (-) matches everything in the current folder and below, like you might expect with the normal GLOB operation in UNIX. The current folder isn't included in the match.
- Star (*) matches every file (*not* folder) in the current folder. The current folder and subfolders are excluded from the match.

Let's say you want to match all of your theme files and folders, specify ...

this:

```
security-manager-files-read=\n    ${liferay.web.portal.dir}/html/themes/-
```

NOT this:

```
security-manager-files-read=\n    ${liferay.web.portal.dir}/html/themes/*
```

The star means "every file in this single directory." The dash, however, matches everything in this folder and below.

One more note. This:

```
 ${liferay.web.portal.dir}/html/themes/-
```

does not include this:

```
 ${liferay.web.portal.dir}/html/themes
```

The dash lets you read the *contents* of the folder, but not the folder itself. Also, when defining the folder, do not include a trailing slash, otherwise the folder itself will not be included. Below, we specify the themes folder and all of the contents under it:

```
security-manager-files-read=\n    ${liferay.web.portal.dir}/html/themes,\n    ${liferay.web.portal.dir}/html/themes/-
```

For file path separators, you can use the \${/} alias.

Example,

```
grant codeBase "file:${my-supercool-portlet}${/}-" {\n    permission java.net.NetPermission "specifyStreamHandler";\n};
```

Congratulations! You now know how to specify your policy's file paths appropriately for deployment on any server. Once you've completed testing your plugin without getting any Java security exceptions, you can distribute it as an app on Liferay Marketplace. You can do so with confidence, because you've specified all of the resources it uses in the application's PACL, and possibly its Java Security Policy, and your application satisfies Liferay Portal's Security Manager.

The sections that follow demonstrated how to enable the Security Manager (which you've already done) and provide descriptions for each type of PACL property.

14.7. Enabling the Security Manager

If you want to distribute plugins, either on the Liferay Marketplace or through your web site, you have to assume users will insist the Security Manager is enabled in your plugin. For this reason, you should enable it when testing your plugins and on packaging it for distribution.

It's very easy to activate the security manager. Set the following `liferay-plugin-package.properties` property to true:

```
security-manager-enabled=true
```

Next, we'll explain the purpose of the PACL properties, show you some of the wildcards you can use for particular property values, and refer you to the file containing the PACL property definitions.

14.8. Portal Access Control List (PACL) Properties

Liferay Portal's Plugin Security Manager checks all your plugin's API access attempts against the security manager properties specified in your plugin's `liferay-plugin-package.properties` file. If your plugin tries to access a portal resource that is not specified in these properties, the Plugin Security Manager prevents it from happening. Consider this a virtual finger waggin'. To prevent this from happening, you have to tell the Plugin Security Manager up-front the access your plugin needs.

The online definitions for the PACL properties can be found at http://docs.liferay.com/portal/6.2/propertiesdoc/liferay-plugin-package_6_2_0.properties.html. If you have the Liferay Portal source code, you can find the `liferay-plugin-package_6_2_0.properties` file in the `liferay-portal/definitions` folder.

Some of the properties accept wildcard characters that have special meaning. Let's investigate the wildcard characters you can use in your plugin's file security properties.

The following properties address file deletion, execution, reading, writing and replacement operations. The * character in a path name indicates all files in the current directory. The - character in a path name indicates all files in the current directory and in its subdirectories.

Here's an example that uses the - character to specify that the plugin is permitted to delete files in the `../webapps/chat-portlet/WEB-INF/src/com/liferay/chat/temp` directory and its subdirectories.

```
security-manager-files-delete=\n    ..../webapps/chat-portlet/WEB-INF/src/com/liferay/chat/temp/-
```

Note, you can use a relative paths in the file security properties.

You can use a mix of UNIX/Linux style paths and Windows style paths as demonstrated in the example below:

```
security-manager-files-execute=\n    /bin/bash,\n    C:\\WINDOWS\\system32\\ping.exe
```

And the following example uses the * character to specify that the plugin is reads files in the `../webapps/chat-portlet/images` and `../webapps/chat-portlet/WEB-`

```
INF/* directories, but not their subdirectories:  
security-manager-files-write=\  
    ..../webapps/chat-portlet/images/*,\  
    ..../webapps/chat-portlet/WEB-INF/*,\  
    ..../webapps/chat-portlet/WEB-INF/src/com/liferay/chat/util/ChatUtil.java
```

For socket security properties the * character represents any hostname. For example, *.liferay.com matches any host ending in .liferay.com, such as docs.liferay.com and issues.liferay.com. And * : * matches every socket and every port.

14.9. Summary

In this chapter, we introduced concepts and instructions for developers to make their apps available on the Liferay Marketplace.

We looked at how to create, publish, maintain, and track Liferay Marketplace apps. You can do this through the App Manager that's available on your personal.liferay.com home page (liferay.com account required!). Then, we covered the requirements for publishing apps, which did not differ significantly from requirements for general Liferay development. Next, we showed how you can publish a sample app on the Marketplace and how you can modify it as the app evolves. Finally, we looked at how to track the adoption of apps using view, download, and install metrics.

Regarding plugin security management, we discussed why plugin security management is necessary, how the Plugin Security Manager checks each plugin against its portal access control list (PACL), and how to specify PACL properties for the plugins you create and deploy. We also explained Liferay's support of the Java Security Policy, in case you need to specify rules above and beyond what PACL properties support.

Now you have a better understanding of how plugin security works and can use Liferay Portal's Plugin Security Manager effectively to specify exactly what services your plugin needs in order to function. Anyone running Liferay Portal with Security Manager turned on will know you're a "law abiding" citizen, because you've specified what services your applications need to access in order to function. We hope this information helps you understand how to develop safe and powerful Liferay apps.

Next, we'll talk about using Ext plugins to make customizations that you can't make with any other Liferay plugin type.

15. Advanced Customization with Ext Plugins

Ext plugins are powerful tools for extending Liferay. Because they increase the complexity of your Liferay instance, you should only use an Ext plugin if you're sure you can't accomplish your goal using a different tool. Check out Customizing and Extending Functionality with Hooks for the available alternatives. If a hook won't suffice, keep reading to discover the use cases for Ext plugins and how to set one up. First let's talk about why you should avoid Ext plugins when possible.

As someone once said, "With great power comes great responsibility" (okay, many people have

said that many times). Before deciding to use an Ext plugin, weigh the cost of using such a powerful tool. Ext plugins allow the use of internal APIs or even overwriting files from the Liferay core. When upgrading to a new version of Liferay (even if it's a maintenance version or a service pack), you have to review all changes and manually modify your Ext plugin to merge your changes with Liferay's. Additionally, Ext plugins aren't hot deployable. To deploy an Ext plugin, you must restart your server. Lastly, with Ext plugins, additional steps are required to deploy or redeploy to production systems.

Now that you know the limitations of Ext plugins, let's look at why you'd want to use them:

- To specify custom classes as portal property values. For example, to specify a property that needs a custom class (e.g., `global.startup.events=my.custom.MyStartupAction`), you need an Ext plugin to add your custom class to the portal class loader.
- To provide custom implementations for any Liferay beans declared in Liferay's Spring files (when possible, use service wrappers from a hook instead of an Ext plugin).
- To add JSPs referenced from portal properties that can only be changed from an Ext plugin (check whether the property can be modified from a hook plugin first).
- To Overwrite a class (not recommended unless you have no other choice).

With these use cases in mind, we'll discuss the following topics:

- Creating an Ext plugin
- Developing an Ext plugin
- Deploying in Production
- Migrating Old Extension Environments

Let's create an Ext plugin.

15.1. ***Creating an Ext Plugin***

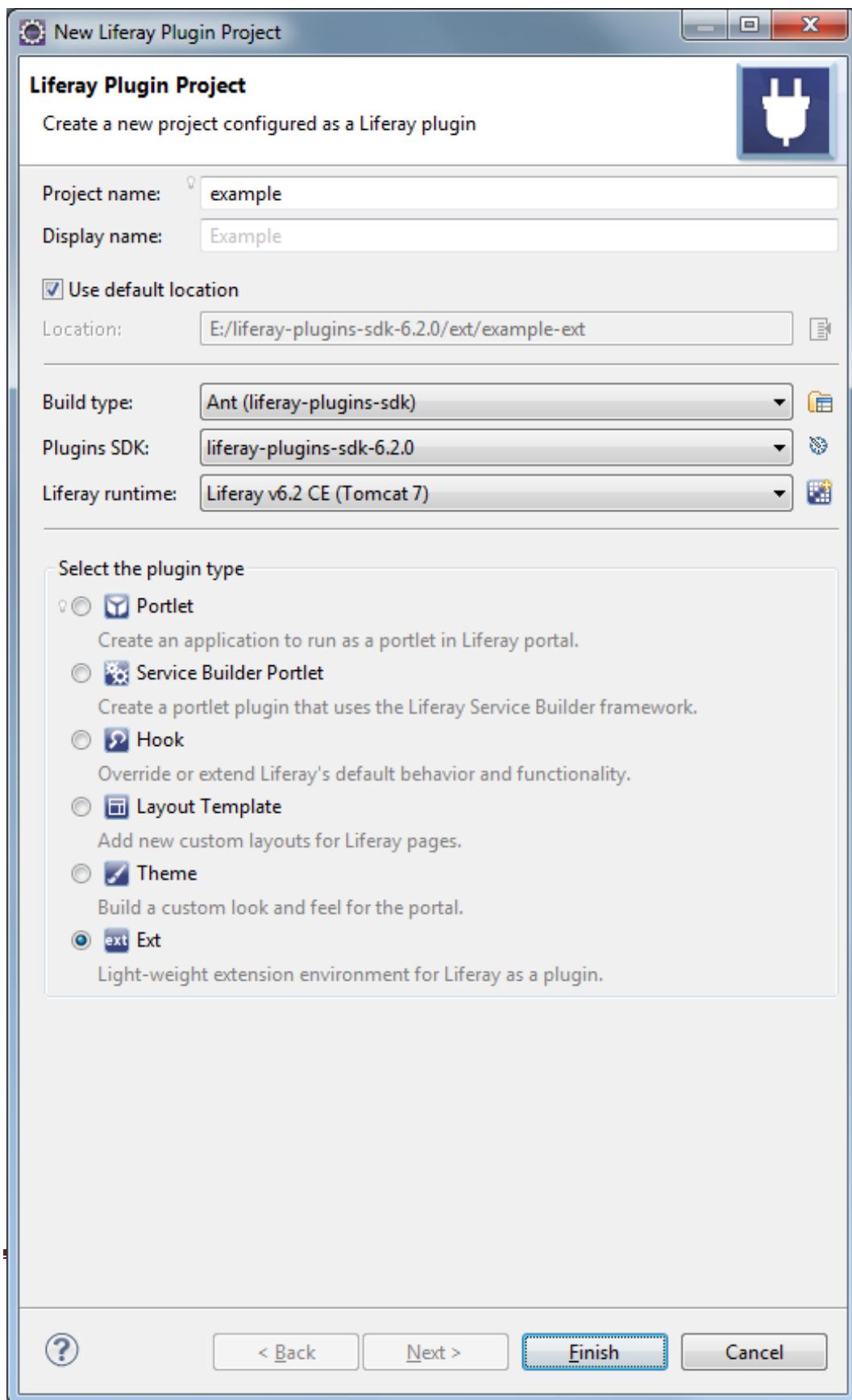
You can create Ext plugins in Liferay Developer Studio or in your terminal environment. The Ext plugin is stored in the `ext` directory of the Plugins SDK (see Leveraging the Plugins SDK).

15.1.1. **Using Developer Studio**

1. Go to File → New → Liferay Project.
2. Fill in *example* for project name and *Example* for the display name.
3. Leave the *Use default location* checkbox checked. By default, the default location is set to your current workspace. If you'd like to change where your plugin project is saved in your file system, uncheck the box and specify your alternate location.
4. Select the *Ant (liferay-plugins-sdk)* option for your build type. If you'd like to use *Maven* for your build type, navigate to the Developing Plugins Using Maven section for details.
5. Your configured SDK and Liferay Runtime should already be selected. If you haven't yet pointed Liferay IDE to a Plugins SDK, click *Configure SDKs* to open the *Installed Plugin SDKs* management wizard. You can also access the *New Server Runtime Environment* wizard if you need to set up your runtime server; just click the *New Liferay Runtime* button next to the *Liferay Portal Runtime* dropdown menu.

6. Select *Ext* for your Plugin type.

7. Click *Finish*.



The Plugins SDK automatically appended *-ext* to the project name when naming the parent folder of your Ext plugin. In Developer Studio, you can either create a completely new plugin or add a new plugin to an existing plugin project.

15.1.2. Using the Terminal

Navigate to the *ext* directory in the Liferay Plugins SDK and enter the appropriate command for your operating

system to create a new Ext plugin:

1. In Linux and Mac OS, enter

```
./create.sh example "Example"
```

2. In Windows, enter

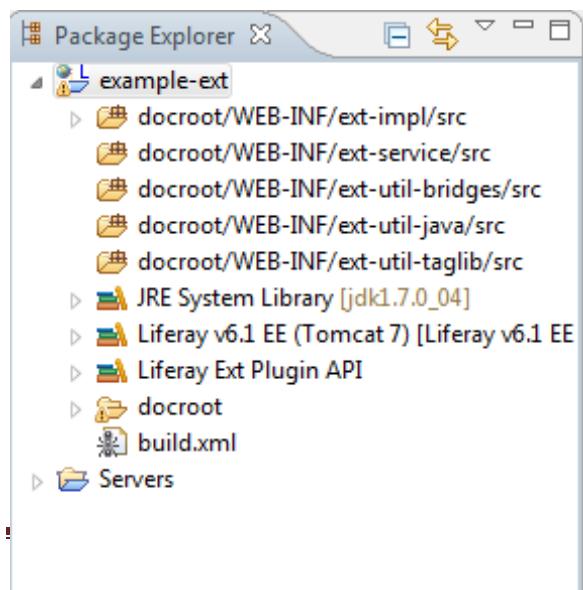
```
create.bat example "Example"
```

A BUILD SUCCESSFUL message from Ant tells you there's a new folder named `example-ext` inside the `ext` folder in your Plugins SDK. The Plugins SDK automatically named the EXT by appending `-ext` to the project name.

15.1.3. Anatomy of the Ext Plugin

The structure of your new `example-ext` folder looks like this:

- `example-ext/`
 - `docroot/`
 - `WEB-INF/`
 - `ext-impl/`
 - `src/`
 - `ext-lib/`
 - `global/`
 - `portal/`
 - `ext-service/`
 - `src/`
 - `ext-util-bridges/`
 - `src/`
 - `ext-util-java/`
 - `src/`
 - `ext-util-taglib/`
 - `src/`
 - `ext-web/`
 - `docroot/`



Let's look at a few of the `/docroot/WEB-INF/` subdirectories in more detail:

`ext-impl/src`: Contains the `portal-ext.properties` configuration file, custom implementation classes, and in advanced scenarios, classes that override core classes within `portal-impl.jar`.

`ext-lib/global`: Contains libraries that should be copied to the application server's global classloader upon deployment of the Ext plugin.

`ext-lib/portal`: Contains libraries to be

copied inside Liferay's main application. These libraries are usually necessary because they are invoked from the classes added in `ext-impl/src`.

`ext-service/src`: Contains classes that should be available to other plugins. In advanced scenarios, this directory contains classes that overwrite the classes of `portal-service.jar`. Service Builder puts the interfaces of each service here.

`ext-web/docroot`: Contains the web application's configuration files, including `WEB-INF/struts-config-ext.xml`, which allows you to customize Liferay's core struts classes. However, hooks are recommended for customizing a struts action. Any JSPs that you're customizing also belong here.

`ext-util-bridges`, `ext-util-java` and `ext-util-taglib`: These folders are needed only in scenarios where you need to customize the classes of three libraries provided with Liferay: `util-bridges.jar`, `util-java.jar` and `util-taglib.jar`, respectively. Otherwise you can ignore these directories.

By default, several files are added to the plugin. Here are the most significant:

- `build.xml`: The Ant build file for the Ext plugin project.
- `docroot/WEB-INF/liferay-plugin-package.properties`: Contains properties of the plugin, including display name, version, author, and license type.
- `docroot/WEB-INF/ext-impl/src/portal-ext.properties`: Overrides Liferay's configuration properties--use a hook plugin to override properties whenever it's possible. An example where an Ext plugin is necessary to override a property is when specifying a custom class as a portal property value. You can use a `portal-ext.properties` file with each of your Ext plugins, but don't override the same property from multiple `portal-ext.properties` files--the loading order isn't assured, and you can cause unintended system behavior as a result.
- `docroot/WEB-INF/ext-web/docroot/WEB-INF` files:
 - `portlet-ext.xml`: Used to overwrite the definition of a Liferay portlet. To do this, copy the complete definition of the desired portlet from `portlet-custom.xml` in Liferay's source code, then apply the necessary changes.
 - `liferay-portlet-ext.xml`: This file is similar to `portlet-ext.xml`, but is for additional definition elements specific to Liferay. To override these definition elements, copy the complete definition of the desired portlet from `liferay-portlet.xml` within Liferay's source code, then apply the necessary changes.
 - `struts-config-ext.xml` and `tiles-defs-ext.xml`: These files are used to customize the struts actions used by Liferay's core portlets.



Tip: After creating an Ext plugin, remove the files you don't need to customize from `docroot/WEB-INF/ext-web/docroot/WEB-INF`. Liferay keeps track of the files deployed by each Ext plugin and won't let you deploy multiple Ext plugins that override the same file. If you remove unnecessary (uncustomized) files, you'll avoid collisions with Ext plugins deployed alongside yours.

You've now created an Ext plugin and are familiar with its directory structure and its most significant files. Let's use your Ext plugin to customize Liferay Portal.

15.2. **Developing an Ext Plugin**

An Ext plugin changes Liferay itself when deployed; it's not a separate component that can be easily removed at any time. For this reason, the Ext plugin development process is different from other plugin types. It's important to remember that once an Ext plugin is deployed, some of its files are *copied* inside the Liferay installation; the only way to remove its changes is by *redeploying* an unmodified Liferay application.

The Plugins SDK lets you deploy and redeploy Ext plugins during your development phase. Redeployment involves *cleaning* (i.e. removing) your application server and unzipping your specified Liferay bundle to start from scratch. That way any changes made to the Ext plugin during development are properly applied, and files removed from your plugin by previous changes aren't left behind in the Liferay application. This added complexity is why we recommend using another plugin type to accomplish your goals, whenever possible.

Before digging in to the details, here's an overview of the steps required to develop Ext plugins:

- We'll show you how to *configure* your Plugins SDK environment to develop Ext plugins for Liferay Portal on your application server.
- We'll show you how to *deploy* and *publish* your Ext plugins for the first time.
- We'll show you how to *redeploy* normally or use a *clean redeployment* process after making changes to your Ext plugins.
- We'll show you how to package your Ext plugins for distribution.
- We'll show you examples of Liferay Portal customizations that require advanced customization techniques.

Now let's look at each step of the development process in more detail.

15.2.1. **Set Up**

Before deploying an Ext plugin, you must edit the `build.[username].properties` file in the root folder of your Plugins SDK. If the file doesn't yet exist, create it now. Substitute `[username]` with your user ID on your computer. Once you've opened your build properties file, add the following properties--make sure the individual paths reflect the right locations on your system:

```
ext.work.dir=[work]
```

```
app.server.dir=[work]/liferay-portal-[version]/[app server]
```

```
app.server.zip.name=[...]/liferay-portal-[app server].zip
```

Your `app.server.zip.name` property should specify the path to your Liferay bundle `.zip` file. Your `work` directory, specified by the `ext.work.dir` property, is where you've unzipped your Liferay bundle runtime. The `app.server.dir` property should point to your application server's directory in your `work` directory. Look in your Liferay bundle at the path to the application server directory to determine the value to use for your `app.server.dir` property.

For example, C:/work could be your ext.work.dir value. If we have a Liferay bundle .zip file C:/downloads/liferay-portal-tomcat-6.2.0-ce-ga1-[timestamp].zip which we set as the value for our app.server.zip.name property, the *relative path* to the application server *within* our Liferay bundle .zip file is liferay-portal-6.2.0-ce-ga1\tomcat-7.0.40. We'd then specify C:/work/liferay-portal-6.2.0-ce-ga1/tomcat-7.0.40 as our app.server.dir property value.



Note: Some Liferay bundles come installed with a sample website. It's useful for showcasing certain features of Liferay, but if you removed it, you likely don't want it reinstalled each time your bundle is unzipped. To prevent the reinstallation of 7-Cogs, unzip your bundle, delete the [work]/liferay-portal-[version]/[app server]/webapps/welcome-theme folder, then re-zip your bundle.

Next we'll change our newly created Ext plugin and deploy it.

15.2.2. Initial Deployment

Our environment is set up and we're ready to start customizing. First let's look at a simple example that customizes the sections of a user profile. The following example can be done using a hook and the users.form.add.main property, but for demonstration purposes, we'll make the customization by overriding portal properties using an Ext plugin. Open the docroot/WEB-INF/ext-impl/src/portal-ext.properties file and paste in the following contents:

```
users.form.update.main=details,password,organizations,sites,roles
```

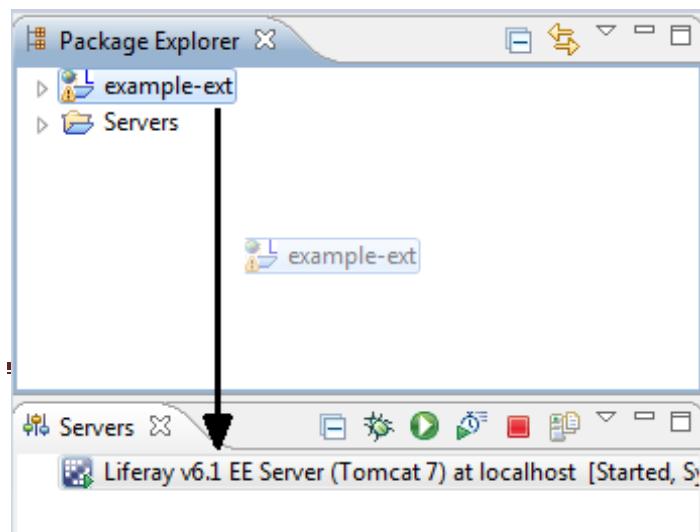
We've removed the sections for user groups, personal sites, and categorizations from the user profile page. This ensures that these sections won't be used in our portal.

Now we're ready to deploy.

15.2.2.1. Deploy the Plugin

You can deploy your plugin from Liferay Developer Studio or the terminal.

Deploying In Developer Studio: Drag your example-ext project from your Package Explorer onto your server.



Deploying In the terminal: Open a terminal window in your ext/example-ext directory and enter one of these commands:

```
ant deploy
```

```
ant direct-deploy
```

The direct-deploy target deploys

all plugin changes directly to the appropriate directories in the Liferay application. The deploy target creates a .war file with your changes and then deploys it to your server. Either way, your server must be restarted after the deploy occurs. Using direct-deploy is usually preferred for deploying Ext plugins during development. However, direct-deploy does not work in WebLogic Server or WebSphere application server environments.

A BUILD SUCCESSFUL message indicates your plugin is now being deployed. If you switch to the console window running Liferay, in few seconds you should see the message

Extension environment for example-ext has been applied. You must reboot the server and redeploy all other plugins

If any changes applied through the Ext plugin affect the deployment process itself, you must redeploy all other plugins. Even if the Ext plugin doesn't affect the deployment process, it's a best practice to redeploy all your other plugins following initial deployment of the Ext plugin.

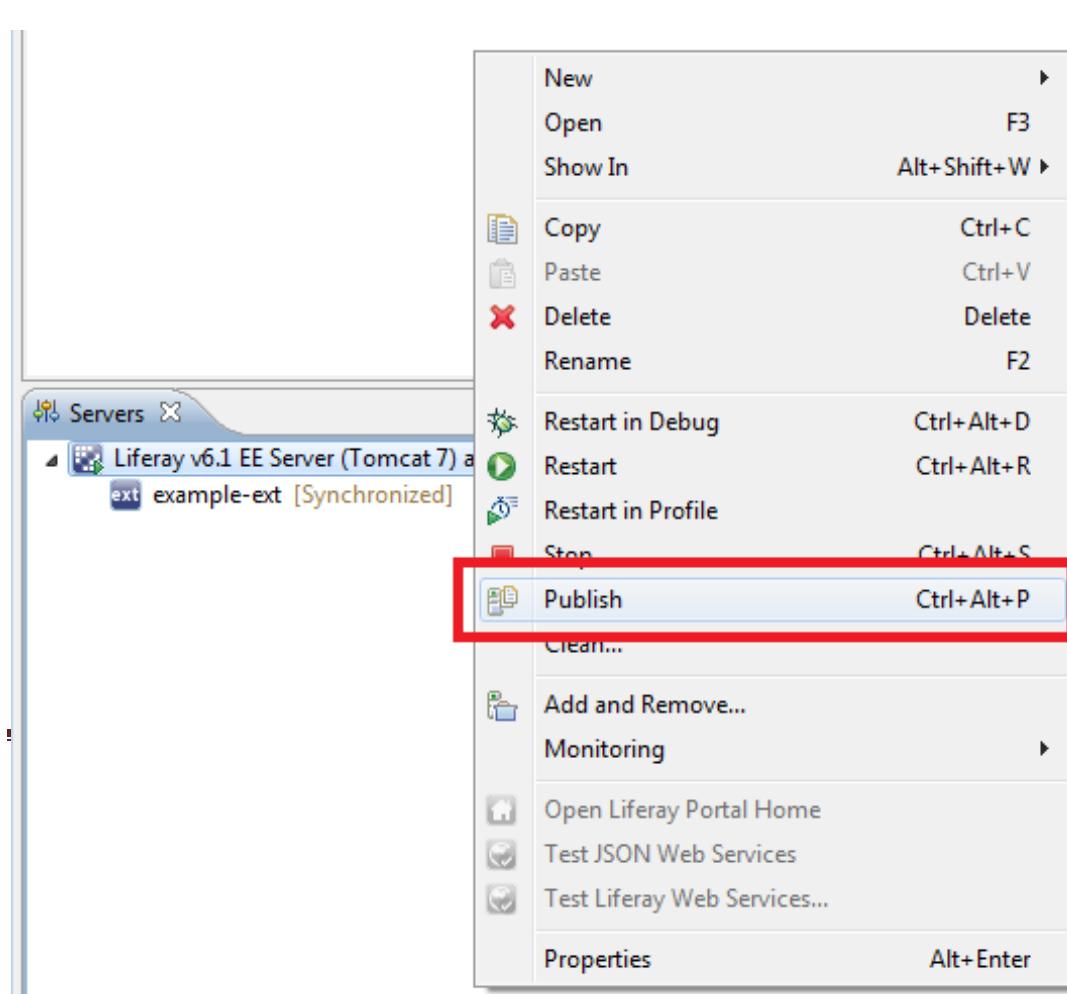
The ant deploy target builds a .war file with your changes and copies them to the auto-deploy directory inside the Liferay installation. When the server starts, it detects the .war file, inspects it, and copies its contents to the appropriate destinations inside the deployed and running Liferay application.

Restart your application server, and let's find out about *publishing* your changes.

15.2.2.2. Publish the Plugin

To complete the deployment process, your Ext plugin must be published to the Liferay server. As with deployment, you can publish using Liferay Developer Studio or your terminal.

Publishing in Developer Studio:

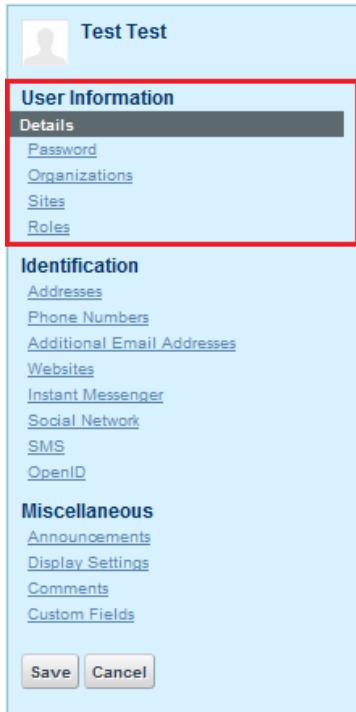


1. Select the Liferay server in the *Servers* view.

2. Select the server's *Publish* option.

Publishing in the terminal:
Restart the Liferay server.

Let's try Liferay portal, customized by your Ext plugin. Once your server restarts, log in as an administrator and go to Control Panel → Users and Organizations. Edit an existing user and verify that the right navigation menu only shows the five sections that we referenced from the `users.form.update.main` property.



That was a simple application of an Ext plugin. Let's proceed with a more complex customization that illustrates the proper way to *redeploy* an Ext plugin, which is different from *initial deployment*.

Let's customize the *details* section of the user profile. Rather than override its JSP, we'll use a more powerful method that lets us add new sections or even merge existing ones. With Liferay we can refer to custom sections from the `portal-ext.properties` and implement them just by creating a JSP. We'll modify the property `users.form.update.main` again and the property `users.form.add.main` to set the following:

```
users.form.add.main=basic,organizations,personal-site  
users.form.update.main=basic,password,organizations,sites,roles
```

We removed the original *details* section and added a custom one called *basic*. When Liferay Portal's user administration reads the property, it looks for the implementation of each section based on the following conventions:

- The section is implemented in a JSP inside the following directory:

```
ext-web/docroot/html/portlet/users_admin/user/
```

- The name of the JSP uses the name of the section, with the `.jsp` extension. If the section name has a dash, (-), replace it with an underscore (_). For example, if the section is called *my-info*, the JSP should be named `my_info.jsp` to comply with JSP naming standards.
- The section name that's shown to the user comes from the language bundles. When using a key/value that is not included with Liferay, add it to both your Ext plugin's `Language-ext.properties` file and the language-specific properties file for each language variant you're providing a translation for. These files go in the `ext-impl/src` directory of your Ext plugin.

For our example, we'll create a file in the Ext plugin with the following path:

```
ext-web/docroot/html/portlet/users_admin/user/basic.jsp
```

We can write the contents of the file from scratch or just copy the `details.jsp` file from Liferay's source code and modify from there. Let's do the latter and then remove some fields, leaving the screen name, email address, first name, and last name fields to simplify user creation and user update. Here's the resulting JSP code. Note, make sure to remove the line escape character \ instances:

```

<%@ include file="/html/portlet/users_admin/init.jsp" %>

<%
User selUser = (User)request.getAttribute("user.selUser");
Contact selContact = (Contact)request.getAttribute("user.selContact");
%>

<liferay-ui:error-marker key="errorSection" value="details" />

<aui:model-context bean="<%= selUser %>" model="<%= User.class %>" />

<h3><liferay-ui:message key="details" /></h3>

<aui:fieldset column="<%= true %>" cssClass="aui-w50">
    <liferay-ui:success
        key="verificationEmailSent"
        message="your-email-verification-code-has-been-sent-and-the-new-\
                  email-address-will-be-applied-to-your-account-once-it-has-been-\
                  verified"
    />

    <liferay-ui:error
        exception="<%= DuplicateUserScreenNameException.class %>"
        message="the-screen-name-you-requested-is-already-taken"
    />

    <liferay-ui:error exception="<%= GroupFriendlyURLException.class %>">

        <%
        GroupFriendlyURLException gfurle =
            (GroupFriendlyURLException)errorException;
        %>

        <c:if
            test="
                <%
                    gfurle.getType() ==
                    GroupFriendlyURLException.DUPLICATE
                %>"
        >
            <liferay-ui:message
                key="the-screen-name-you-requested-is-associated-with-an-\
                      existing-friendly-url"
            />
        </c:if>
    </liferay-ui:error>

    <liferay-ui:error
        exception="<%= ReservedUserScreenNameException.class %>"
        message="the-screen-name-you-requested-is-reserved"
    />
    <liferay-ui:error
        exception="<%= UserScreenNameException.class %>"
        message="please-enter-a-valid-screen-name"

```

```

/>

<c:if
    test=""
    <%
        !PrefsPropsUtil.getBoolean(
            company.getCompanyId(),
            PropsKeys.USERS_SCREEN_NAME_ALWAYS_AUTOGENERATE) ||
        (selUser != null)
    %>"

>
<c:choose>
    <c:when
        test=""
        <%
            PrefsPropsUtil.getBoolean(
                company.getCompanyId(),
                PropsKeys.USERS_SCREEN_NAME_ALWAYS_AUTOGENERATE)
            ||
            (
                (selUser != null) &&
                !UsersAdminUtil.hasUpdateScreenName(
                    permissionChecker, selUser)
            )
        %>"

    >
        <aui:field-wrapper name="screenName">
            <%= selUser.getScreenName() %>

            <aui:input
                name="screenName"
                type="hidden"
                value="<%= selUser.getScreenName() %>"
            />
        </aui:field-wrapper>
    </c:when>
    <c:otherwise>
        <aui:input name="screenName" />
    </c:otherwise>
</c:choose>
</c:if>

<liferay-ui:error
    exception="<%= DuplicateUserEmailAddressException.class %>"
    message="the-email-address-you-requested-is-already-taken"
/>
<liferay-ui:error
    exception="<%= ReservedUserEmailAddressException.class %>"
    message="the-email-address-you-requested-is-reserved"
/>
<liferay-ui:error
    exception="<%= UserEmailAddressException.class %>"
    message="please-enter-a-valid-email-address"
/>

```

```

<c:choose>
    <c:when
        test=""
            <%= (selUser != null) &&
                !UsersAdminUtil.hasUpdateEmailAddress(
                    permissionChecker, selUser)
            %>">
        <aui:field-wrapper name="emailAddress">
            <%= selUser.getDisplayEmailAddress() %>

            <aui:input name="emailAddress" type="hidden" value="<%=
selUser.getEmailAddress() %>" />
        </aui:field-wrapper>
    </c:when>
    <c:otherwise>

        <%
        User displayEmailAddressUser = null;

        if (selUser != null) {
            displayEmailAddressUser = (User)selUser.clone();

            displayEmailAddressUser.setEmailAddress(
                displayEmailAddressUser.getDisplayEmailAddress());
        }
        %>

        <aui:input
            bean="<%= displayEmailAddressUser %>"
            model="<%= User.class %>" name="emailAddress"
        >
        <c:if test="<%=
PrefsPropsUtil.getBoolean(
    company.getCompanyId(),
    PropsKeys.USERS_EMAIL_ADDRESS_REQUIRED)%>">
            <aui:validator name="required" />
        </c:if>
    </aui:input>
</c:otherwise>
</c:choose>

<liferay-ui:error
    exception="<%=
ContactFirstNameException.class %>"
    message="please-enter-a-valid-first-name"
/>
<liferay-ui:error
    exception="<%=
ContactFullNameException.class %>"
    message="please-enter-a-valid-first-middle-and-last-name"
/>

<aui:input name="firstName" />

```

```

<liferay-ui:error
    exception="<% ContactLastNameException.class %>"
    message="please-enter-a-valid-last-name"
/>

<aui:input name="lastName">
    <c:if test="<%= PrefsPropsUtil.getBoolean(
        company.getCompanyId(),
        PropsKeys.USERS_LAST_NAME_REQUIRED,
        PropsValues.USERS_LAST_NAME_REQUIRED) %>">
        <aui:validator name="required" />
    </c:if>
</aui:input>
</aui:fieldset>

```

We don't need to add a new key to `Language-ext.properties`, because an entry for the key named `basic` is already included in Liferay's language bundle.

Let's redeploy our Ext plugin to review the changes we made.

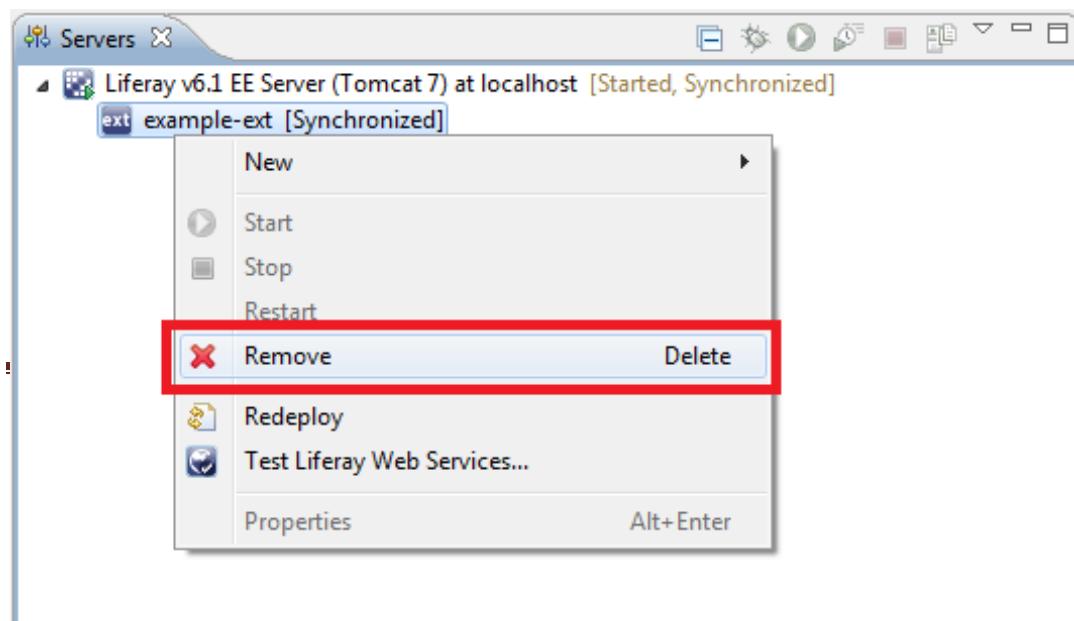
15.2.3. Redeployment

So far, Ext plugin development has been similar to the development of other plugin types. You've now reached the point of divergence. When the plugin is first deployed, some of its files are *copied* into the Liferay installation. After changing an Ext plugin, you'll either *redeploy* or *clean redeploy*, depending on the specific modifications you made to your plugin following the initial deployment. Let's talk about each redeployment method and when to use each one.

Clean Redeployment: If you removed part(s) of your plugin, there are changes to your plugin that can affect the deployment of plugins, or you simply want to start with a clean Liferay environment, *undeploy* your plugin and *clean* your application server before redeploying your Ext plugins. By cleaning the application server, the existing Liferay installation is removed and the bundle specified in your Plugins SDK environment (e.g., the value of `app.server.zip.name` in `build.[username].properties`) is unzipped in its place. The exact steps you take differ based on whether you're developing in Liferay Developer Studio or your terminal:

Using Developer Studio:

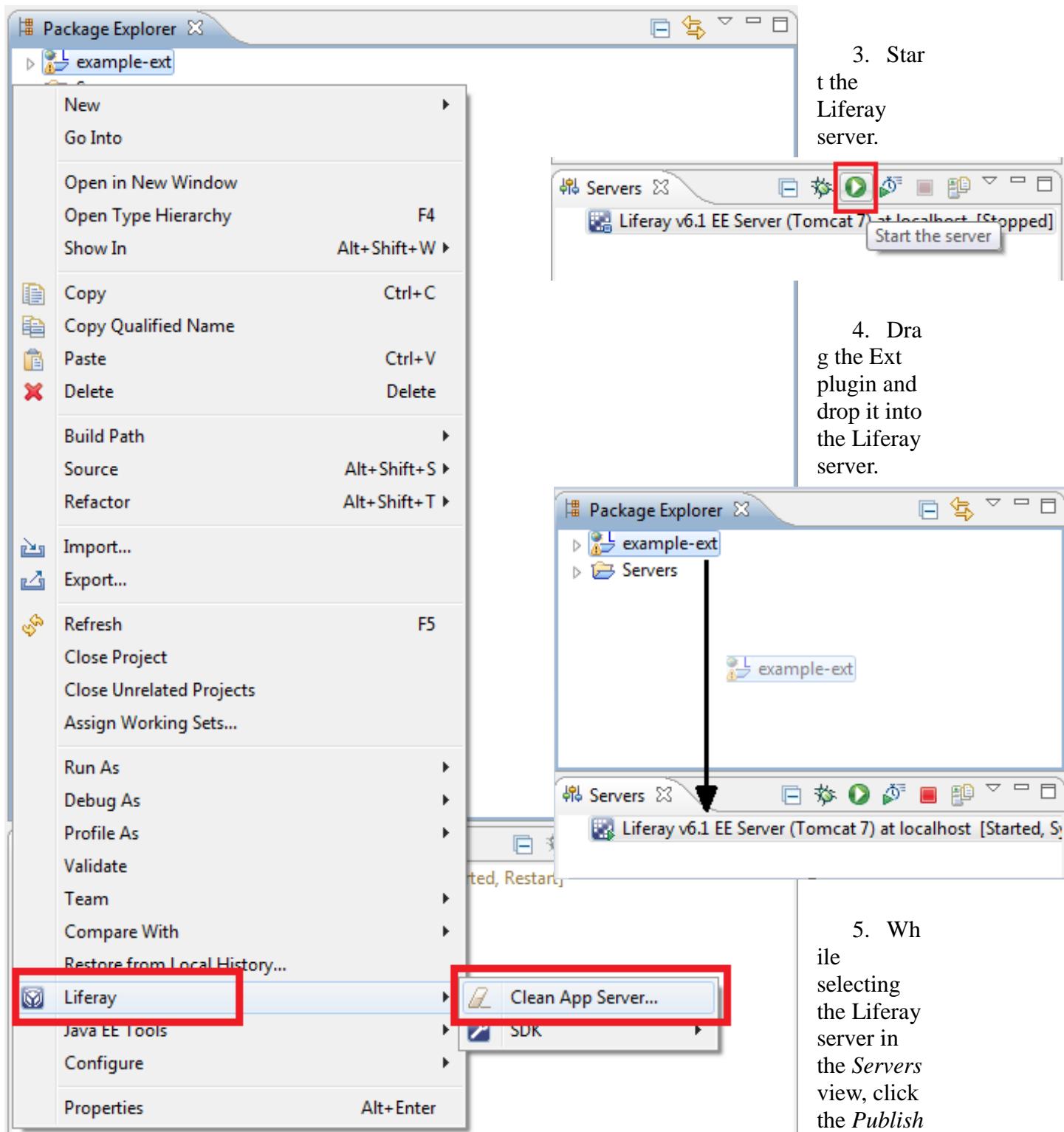
1. Remove the plugin from the server. While selecting the Ext plugin in the *Servers* view,



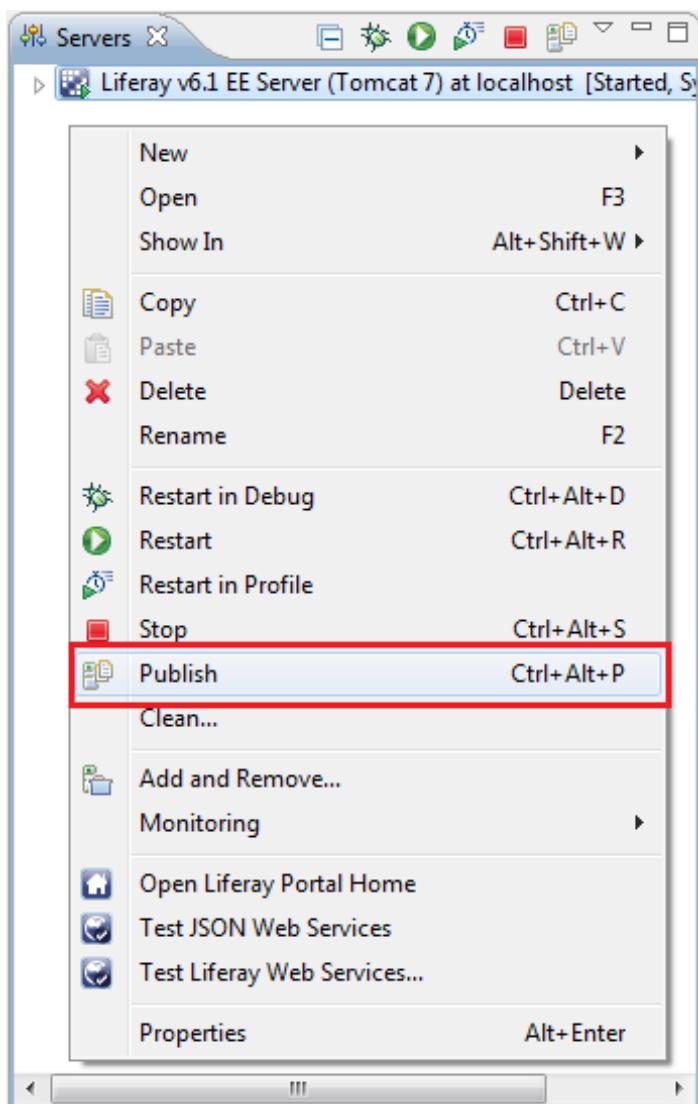
select the plugin's *Remove* option.

2. Clean the application server--

while selecting the Ext plugin project in the *Package Explorer* view, select the plugin's *Liferay → Clean App Server...* option.



option.



your server and select *Redeploy*.

- **Using the terminal:** Redeploy in the terminal using the same procedure as for initial deployment. Open a terminal window in your ext/example-ext directory and execute either ant deploy or ant direct-deploy.

See above in the *Initial deployment* section if you're not sure which command to use.

After your example-ext plugin is published to Liferay Portal, check out your *basic* details page by choosing to add a user or view an existing user.

Using the terminal:

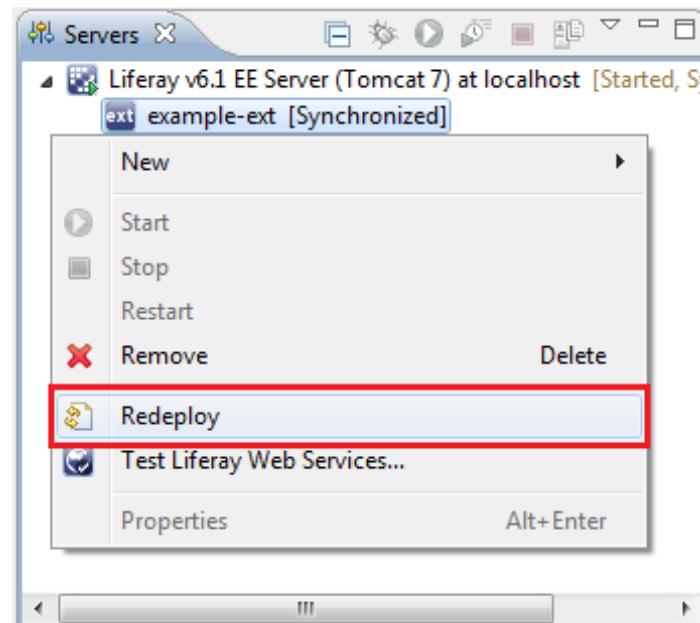
1. Stop the Liferay server.
2. For each Ext plugin you're deploying, enter the following into your console:

```
cd [your-plugin-ext]
ant clean-app-server
ant direct-deploy
```

3. Start the Liferay server.

Redeployment: If you only added to your plugin or made modifications that don't affect the plugin deployment process, you can redeploy using the following steps:

- **Using Developer Studio:** Right-click your plugin located underneath



Users and Organizations

View All Add Export All Users

Test Test

Details

Screen Name (Required)
test

Email Address (Required)
test@liferay.com

First Name (Required)
Test

Last Name
Test

User Information

Basic
Password
Organizations
Sites
Roles

Identification
Addresses
Phone Numbers
Additional Email Addresses
Websites

« Back

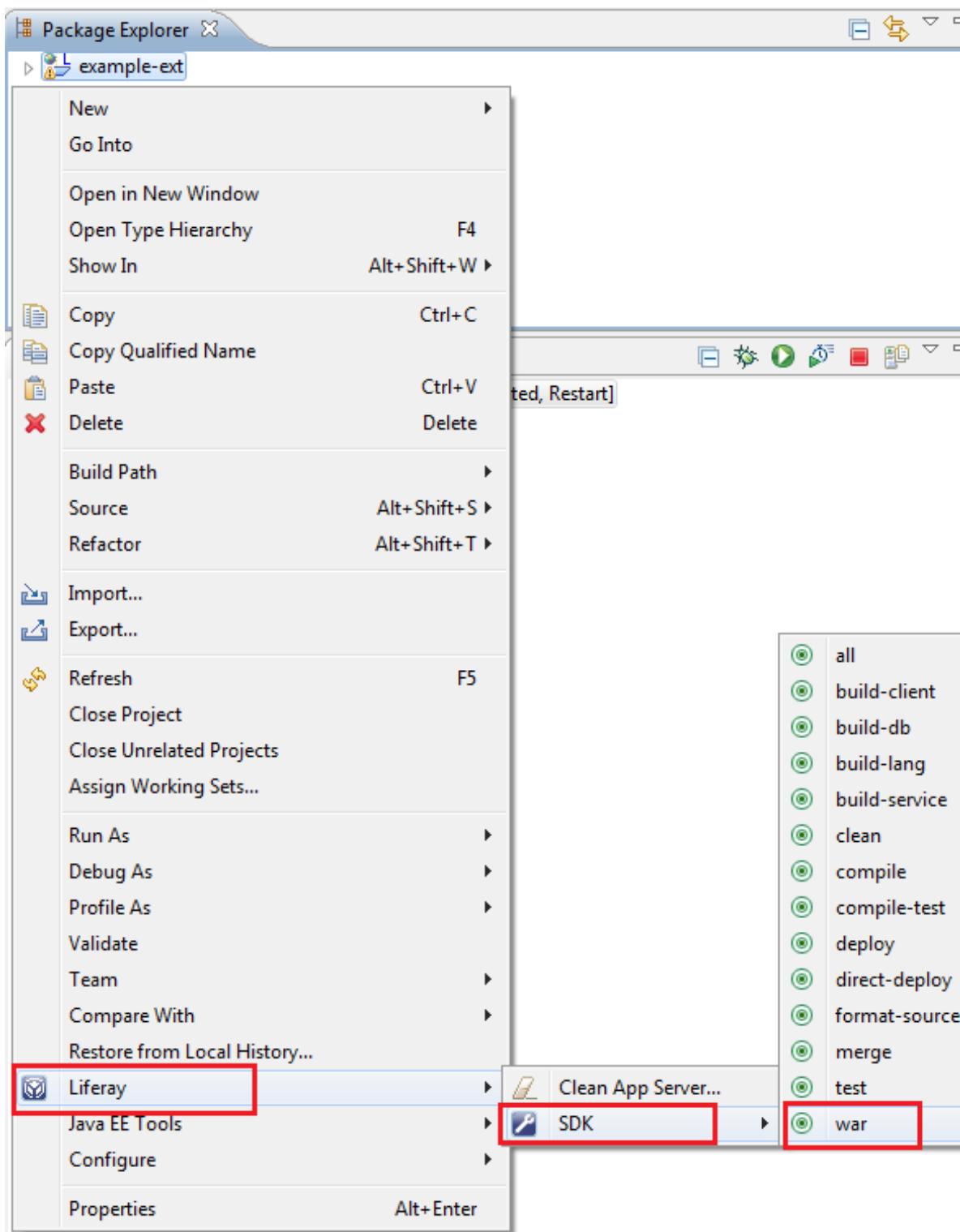
in the *Package Explorer* view, select the project's *Liferay → SDK → war* option.

That completes the development process. Let's learn how you can package your Ext plugin for distribution and production.

15.2.4. Distribution

Once you're finished developing the plugin, you can package it in a *.war* file for distribution and production deployment.

Using Developer Studio: With your Ext plugin project selected



Using the terminal: From your Ext plugin's directory (e.g., ext/example-ext), enter

```
ant war
```

The .war file is written to your [liferay-plugins]/dist directory.

You really have the hang of building and packaging your Ext plugins! Our next section covers JBoss 7 requirements for packaging up an Ext plugin containing a new taglib. If this doesn't apply to you, feel free to skip over it and to start reading about advanced customization techniques.

15.2.5. Ext Plugin Packaging Requirements for JBoss 7

If you're developing an Ext plugin that defines a new taglib, you need to take JBoss's classloading behavior into account. Before packaging this kind of Ext plugin, create a `jboss-deployment-structure.xml` file in the Ext plugin's WEB-INF/ folder and add the following contents to it:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
    <deployment>
        <dependencies>
            <module name="deployment.util-taglib"></module>
            <module name="com.liferay.portal"></module>
        </dependencies>
    </deployment>
    <module name="deployment.util-taglib">
        <resources>
            <resource-root path="WEB-INF/ext-util-
taglib/ext-util-taglib.jar" />
            <resource-root path="WEB-INF/lib/util-
taglib.jar"></resource-root>
            <resource-root path="WEB-INF/lib/util-
java.jar"></resource-root>
        </resources>
        <dependencies>
            <module name="javax.faces.api"></module>
            <module name="javax.servlet.api" />
            <module name="javax.servlet.jsp.api" />
            <module name="com.liferay.portal"></module>
        </dependencies>
    </module>
</jboss-deployment-structure>
```

Also, add the following line to your Ext plugin's `liferay-plugin-package.properties` file, setting the `util-taglib.jar` as a dependency:
`portal-dependency-jars=util-taglib.jar`

Once you've made these updates, you can package your plugin and deploy it, per the normal process described previously in this chapter.

Now that you've learned the basics of Ext plugin development have covered this requirement for JBoss customizations, let's look at some advanced customizations that you can do.

15.2.6. Advanced Customization Techniques

With Ext plugins, you can change almost everything in Liferay. Let's look at some additional customization techniques made possible by Ext plugins. As always, be careful when using Ext

plugins.

With each new version of Liferay, there can be changes to the implementation classes. If you change Liferay's source code directly, you'll have to merge your changes into the newer Liferay version. To minimize such conflicts, the best approach is not to change anything. Rather, you can extend the class you want to change and override the required methods. Then use the requisite Liferay configuration files to reference your subclass as a replacement for the original class.

In the following subsections, we'll cover these topics:

- Using advanced configuration files
- Changing the API of a core service
- Replacing core classes in portal-impl

Let's learn to use advanced configuration files next.

15.2.6.1. Using Advanced Configuration Files

Liferay uses several internal configuration files for its own architecture; in addition, there are configuration files for the libraries and frameworks Liferay depends on, like Struts and Spring. Configuration could be accomplished using fewer files with more properties in each, but maintenance and use is made easier by splitting up the configuration properties into several files. For advanced customization needs, it may be useful to override the configuration specified in multiple configuration files. Liferay provides a clean way to do this from an Ext plugin without modifying the original files.

Below we list all the configuration files in Liferay by their path in your Ext plugin folder. We provide a description of what the file is for and the path to the original file in Liferay Portal:

- ext-impl/src/META-INF/ext-model-hints.xml
 - Description: Allows overriding the default properties of the fields of the data models used by Liferay's core portlets. These properties determine how the form fields for each model are rendered.
 - Original file in Liferay: portal-impl/src/META-INF/portal-model-hints.xml
- ext-impl/src/META-INF/ext-spring.xml
 - Description: Allows overriding the Spring configuration used by Liferay and any of its core portlets. It's most commonly used to configure specific data sources or swap the implementation of a given service with a custom one.
 - Original files in Liferay: portal-impl/src/META-INF/*-spring.xml
- ext-impl/src/content/Language-ext_*.properties
 - Description: Allow overriding the value of any key used by Liferay's UI to support *I18N*.
 - Original file in Liferay: portal-impl/src/content/Language-* .properties
- ext-impl/src/META-INF/portal-log4j-ext.xml
 - Description: Allows overriding the log4j configuration. It's most commonly used to increase or decrease the log level of a given package or class, to obtain more information, or hide unneeded information from the logs.

- › Original file in Liferay: portal-impl/src/META-INF/portal-log4j.xml
- ext-
 - impl/src/com/liferay/portal/jcr/jackrabbit/dependencies/\
repository-ext.xml
 - › Description: Allows overriding the configuration of the Jackrabbit repository. Refer to the Jackrabbit configuration documentation for details (<http://jackrabbit.apache.org/jackrabbit-configuration.html>)
 - › Original file in Liferay: portal-
 - impl/src/com/liferay/portal/jcr/jackrabbit/dependencies/\
repository.xml
- ext-web/docroot/WEB-INF/portlet-ext.xml
 - › Description: Allows overriding the declaration of the core portlets included in Liferay. It's most commonly used to change the init parameters or the roles specified.
 - › Original file in Liferay: portal-web/docroot/WEB-INF/portlet-
custom.xml
- ext-web/docroot/WEB-INF/liferay-portlet-ext.xml
 - › Description: Allows overriding the Liferay-specific declaration of the core portlets included in Liferay. Refer to the liferay-portlet-app_6_1_0.dtd file for details on all the available options. Use this file with care; the code of the portlets may be assuming some of these options to be set to certain values.
 - › Original file in Liferay: portal-web/docroot/WEB-INF/liferay-
portlet.xml
- ext-web/docroot/WEB-INF/liferay-display.xml
 - › Description: Allows overriding the portlets that are shown in the Add Application pop-up and the categories in which they're organized. It's most commonly used to change the categorization, hide certain portlets, or make specific Control Panel portlets available to be added to a page.
 - › Original file in Liferay: portal-web/docroot/WEB-INF/liferay-
display.xml
- ext-web/docroot/WEB-INF/liferay-layout-templates-ext.xml
 - › Description: Allows specifying custom template files for each of Liferay's standard layout templates. This is rarely necessary.
 - › Original file in Liferay: portal-web/docroot/WEB-INF/liferay-
layout-templates.xml
- ext-web/docroot/WEB-INF/liferay-look-and-feel-ext.xml
 - › Description: Allows changing the properties of Liferay's default themes. This is rarely used.
 - › Original file in Liferay: portal-web/docroot/WEB-INF/liferay-look-
and-feel.xml

Let's learn how to configure a Lucene Analyzer next.

15.2.6.2. Configuring Lucene Analyzers

Liferay uses Lucene to facilitate search and indexing within the portal. In old versions of Liferay,

you could configure Lucene analyzers from `portal-ext.properties`. While convenient, it was problematic to have only a single analyzer for all portal-indexed fields. For example, it was difficult to provide correct behaviors for handling both keyword and text fields.

Since Liferay 6.1, you no longer configure Lucene from `portal-ext.properties`. Instead, Lucene analyzers are defined in Spring configuration files. The default analyzer configuration is defined in `/portal-impl/src/META-INF/search-spring.xml`. Liferay 6.1 introduced per-field analyzers, allowing Lucene's query parser to identify the correct analyzer to handle any field.

There are two common scenarios where it's useful to configure Lucene analyzers: when creating custom language analyzers to override a Liferay language analyzer, and when creating a custom indexer to index new fields. Liferay provides a large number of language analyzers in `search-spring.xml` out of the box and uses regular expression matching to map localized fields to specific analyzers. If your language is not included among the defaults, or you're not satisfied with one of the default language analyzers, you can override it with a custom analyzer. If you've created a custom indexer to index new fields, you can use the default analyzer for your new fields, select a specific one (such as one of the `KeywordAnalyzers`), or define a custom analyzer.

To customize the Lucene analyzer configuration, you must create an Ext plugin. The Analyzer classes reference the Lucene APIs directly, so it's not possible to configure Lucene analyzers from a hook plugin. In your Ext plugin, create a `/docroot/WEB-INF/ext-impl/src/META-INF/ext-spring.xml` file and declare the beans you'll override with a custom configuration.

Here are a few of the pre-configured analyzers from the `com.liferay.portal.search.lucene.PerFieldAnalyzerWrapper` bean configuration in `search-spring.xml`:

```
<entry key="tag"
      value-ref="com.liferay.portal.search.lucene.LikeKeywordAnalyzer" />
<entry key="templateId"
      value-ref="org.apache.lucene.analysis.KeywordAnalyzer" />
<entry key="treePath"
      value-ref="com.liferay.portal.search.lucene.LikeKeywordAnalyzer" />
<entry key="type"
      value-ref="org.apache.lucene.analysis.KeywordAnalyzer" />
<entry key="userName"
      value-ref="com.liferay.portal.search.lucene.LikeKeywordAnalyzer" />
<entry key=".*_ar"
      value-ref="org.apache.lucene.analysis.ar.ArabicAnalyzer" />
<entry key=".*_de_DE"
      value-ref="org.apache.lucene.analysis.de.GermanAnalyzer" />
<entry key=".*_el_GR"
      value-ref="org.apache.lucene.analysis.el.GreekAnalyzer" />
<entry key=".*_fa_IR"
      value-ref="org.apache.lucene.analysis.fa.PersianAnalyzer" />
<entry key=".*_fr_[A-Z]{2}"
      value-ref="org.apache.lucene.analysis.fr.FrenchAnalyzer" />
```

You can define custom analyzers for any fields, including custom fields.

Let's learn to change the API of a core service next.

15.2.6.3. *Changing the API of a Core Service*

Sometimes you might need to change the API of a method provided by one of Liferay's services (e.g., `UserLocalService`). This is an advanced customization need.

Is it even possible to change the API of a core service? Not directly. Don't worry, we didn't put this section here just to tell you it's not possible. Changing a core service API under normal circumstances requires modifying Liferay's source code directly and making manual changes to a slew of files. But that's not the Liferay way: there's a better way to do it.

The best way to extend an existing service is by creating a custom service that's complementary (e.g., a `MyUserLocalService` that includes all the new methods). Your custom code can invoke this service instead of the default service, and the implementation of your service can invoke the original service as needed.

This technique doesn't require an Ext plugin since it can be done from portlet plugins. In fact, using Service Builder for an Ext plugin is deprecated, but it's supported for migration from the old extension environment.

Sometimes it's desirable to change the implementation of the original service to call your custom one; that's when you'll need an Ext plugin. Override the Spring definition for `UserLocalServiceUtil` in `ext-spring.xml` and point it to your `MyUserLocalServiceImpl` (instead of `UserLocalServiceImpl`). Now both `MyUserLocalServiceUtil` and `UserLocalServiceUtil` will use the same Spring bean: your new implementation.

You can also replace core classes in `portal-impl`. Keep reading to find out how.

15.2.6.4. *Replacing Core Classes in portal-impl*

If you're sure you need to change a core `portal-impl` class, and certain it can't be replaced in a configuration file, here's the best way to do it while avoiding conflicts when merging with a new portal version:

1. Rename the original class (e.g., `DeployUtil` → `MyDeployUtil`).
2. Create a new subclass with the old name (e.g., `DeployUtil` extends `MyDeployUtil`).
3. Override any methods you need to change.
4. Delegate static methods.
5. Use a logger with an appropriate class name for both classes (e.g., `DeployUtil`).

This strategy will help you determine what you'll need to merge when a new version of Liferay is released.



Tip: This is an advanced technique; it may have a large impact on the maintainability of your code, especially if abused. Seek alternatives, and if you're sure this is your only option, think of it as a short term solution. Contact Liferay's developers about applying the necessary changes to the product's source code.

That's it for advanced customization techniques. Let's talk about deploying in production next.

15.3. *Deploying in Production*

Often times you can't use Ant to deploy web applications in production or pre-production environments. Additionally, some application servers such as WebSphere or WebLogic have their own deployment tools, and Liferay's autodeploy process won't work. Let's look at two methods for deploying and redeploying Ext plugins in these scenarios.

15.3.1. Method 1: Redeploying Liferay's Web Application

You can use this method in any application server that supports auto-deploy; Tomcat and Glassfish are two examples. What's the benefit? The only artifact that needs to be transferred to the production system is your Ext plugin's .war file, produced using the `ant war` target. This .war file is usually small and easy to transport. Execute these steps on the server:

1. Redeploy Liferay:

If this is your first time deploying your Ext plugin to this server, skip this step. Otherwise, start by executing the same steps you first used to deploy Liferay on your app server. If you're using a bundle, unzip it again. If you installed Liferay manually on an existing application server, you'll need to redeploy the Liferay .war file and copy both the libraries required globally by Liferay and your Ext plugin to the appropriate directory within the application server.

2. Copy the Ext plugin .war into the auto-deploy directory. For a bundled Liferay distribution, the `deploy` folder is in Liferay's `root` folder of your bundle (e.g., `liferay-portal-6.2.0-ce-ga1/`).
3. Once the Ext plugin is detected and deployed by Liferay, restart your Liferay server.

15.3.2. Method 2: Generate an Aggregated WAR File

Some application servers don't support auto-deploy; WebSphere and WebLogic are two examples. With an aggregated WAR file, all Ext plugins are merged before deployment to production. A single .war file will contain Liferay plus the changes from all your Ext plugins. Before you deploy the Liferay .war file, copy the dependency .jar files for Liferay and all Ext plugins to the global application server class loader in the production server. The precise location varies from server to server; see Using Liferay Portal 6.2 to get the details for your application server.

The first step in creating the aggregated .war file is to deploy your Ext plugin. The remaining steps can differ depending on your application server; let's proceed by assuming you're using a Liferay Tomcat bundle. Deploy your plugin, restart the server, then shut it down. The files are

now aggregated in your app server. Create a `.war` file by zipping the `webapps/ROOT` folder of Tomcat, then copy all the libraries from the `lib/ext` directory of Tomcat to your application server's global classpath--these files are associated with your Ext plugins.

Once your `.war` file is aggregated, perform these actions on your server:

1. Redeploy Liferay using the aggregated WAR file.
2. Stop the server and copy the new version of the global libraries to the appropriate directory in the application server.

Next we'll show you how to migrate your extension environment (from older versions of Liferay) into Ext plugins.

15.4. *Migrating Old Extension Environments*

Because Ext plugins are an evolution of the extension environment provided in Liferay 5.2 and earlier, you might need to migrate your extension environment into Ext plugins when upgrading Liferay. If you need to do this, we have good news; migrating is automated and relatively easy.



Tip: When migrating an extension environment, first consider whether any of the extension environment's features can be moved into other types of plugins. Portlets and hooks are designed to meet specific needs and they're easier to learn. Additionally, they're easier to maintain since they often require fewer changes when upgrading to a new version of Liferay.

To successfully migrate, execute an Ant target within the `ext` directory of the Plugins SDK, pointing to the old extension environment and naming the new plugin. Be sure to remove the line escape character `\` from the following example:

```
ant upgrade-ext -Dext.dir=/projects/liferay/ext -Dext.name=my-ext\  
-Dext.display.name="My Ext"
```

Let's look at the three parameters we used above:

- `ext.dir`: The location of the old extension environment.
- `ext.name`: The name of the Ext plugin that you want to create.
- `ext.display.name`: The display name.

After executing the target, you should see the logs of several copy operations that will take files from the extension environment and copy them into the equivalent directory within the Ext plugin (see the section *Creating an Ext plugin* for an explanation of the main directories within the plugin).

With the migration process finished, you can upgrade your code to the new version of Liferay by completing a few additional tasks. Most commonly, you should do the following:

- Review the uses of Liferay's APIs and adapt them accordingly.
- Review any changes to the new version of Liferay's JSPs. Merge your changes into the JSPs of the new Liferay version.
- Run `ant build-service` again, to use Service Builder. It's also recommended to

- consider moving this code to a portlet plugin, because Service Builder is deprecated in Ext, and plugins allow for greater modularity and maintainability.
- If you implemented any portlets in the old extension environment, migrate them to portlet plugins; extension environment portlets have been deprecated since Liferay Portal 6.0, and support isn't guaranteed in future Liferay Portal releases.

15.4.1.1. Licensing and Contributing

Liferay Portal is Open Source software licensed under the LGPL 2.1 license (<http://www.gnu.org/licenses/lgpl-2.1.html>). If you reuse any code snippet and redistribute it, whether publicly or to a specific customer, make sure your modifications are compliant with the license. One common way is to make the source code of your modifications available to the community under the same license. Make sure you read the license text yourself to find the option that best fits your needs.

If your goal in making changes is fixing a bug or improving Liferay, it could be of interest to a broader audience. Consider contributing it back to the project. That benefits all users of the product, including you since you won't have to maintain the changes with each newly released version of Liferay. You can notify Liferay of bugs or improvements in issues.liferay.com. There is also a wiki page with instructions on how to contribute to Liferay:

<http://www.liferay.com/community/wiki/-/wiki/Main/Contributing>

15.5. Summary

Ext plugins are a powerful way to extend Liferay. There are no limits to what you can use them to customize, so use them carefully. Before using an Ext plugin, see if you can implement all or part of the desired functionality through a different plugin type: portlets, hooks, and web plugins offer you a lot of extension capabilities themselves, without introducing the complexity that's inherent with Ext plugins. If you need to use an Ext plugin, make it as small as possible and follow the instructions in this guide carefully to avoid issues.

Next, we'll take a look at some helpful plugin developer references. So get ready to bookmark plenty of links!

16. What's New in Liferay 6.2 APIs?

Liferay Portal 6.2 offers a host of new features and updates to the previous release. Our guide to *Using Liferay Portal 6.2* shows you how to use these features and updates and this guide shows you how to leverage them in the applications you develop. In this chapter, we want to highlight some of the changes to Liferay Portal's application programming interface (API). We've added APIs for the new features and improved APIs for previously existing features. In some cases we've modified portal's API and removed previously deprecated interfaces. The Javadoc for Liferay Portal's entire API is available at <http://docs.liferay.com/portal/6.2/>. But we'll describe some of the most notable additions and changes here in this chapter.

To start things off, we'll take a look at the new things you can do with the Application Display Templates API.

16.1. Application Display Templates

A portlet's Display Settings (*Options → Configuration → Setup → Display Settings*) let you customize its display. They come built in with Liferay, so you don't have to do anything special to enable them for your custom portlets. But what if you need settings in addition to Liferay's default display settings? You could develop a theme or hook with the display options you need, but it'd be nice if you could apply particular display options to specific portlet instances without having to redeploy any plugins. Ideally, you should be able to provide authorized portal users the ability to apply custom display settings to portlets. This saves you from having to change portlet configuration code every time you need new settings.

Be of good cheer! That's precisely what Application Display Templates (ADTs) provide-- the ability to add custom display settings to your portlets from the portal. This isn't actually a new concept in Liferay. In some portlets (e.g., *Web Content*, *Documents and Media*, and *Dynamic Data Lists*), you can already add as many display options (or templates) as you want. Now you can add them to your custom portlets, too.

You can use the Application Display Templates API to add this new feature to your plugins. Let's get started learning how.

16.1.1. Using the Application Display Templates API

To leverage the ADT API, there are several steps you need to follow. These steps involve registering your portlet to use ADTs, defining permissions, and exposing the ADT functionality to users. We'll demonstrate these steps by enabling Application Display Templates for our Location Listing Portlet. Be aware that your specific implementation will look slightly different.

1. Create and register your custom `PortletDisplayTemplateHandler` class.

To join the exclusive ADT club, your portlet must sign a contract, committing itself to fulfill all the necessary Application Display Template requirements. In other words, you have to create your own `PortletDisplayTemplateHandler` implementation by extending the `BasePortletDisplayTemplateHandler` methods. You can check the `TemplateHandler` interface Javadoc to learn about each method. Here's what our template handler class looks like for the Location Listing portlet:

```
package com.nosester.portlet.eventlisting.template;

import java.util.List;
import java.util.Locale;
import java.util.Map;

import com.liferay.portal.kernel.language.LanguageUtil;
import com.liferay.portal.kernel.portletdisplaytemplate.BasePortletDisplayTemplateHandler;
import com.liferay.portal.kernel.template.TemplateVariableGroup;
import com.liferay.portal.kernel.util.StringPool;
import com.liferay.portlet.portletdisplaytemplate.util.PortletDisplayTemplateConstants;
```

```

import com.nosester.portlet.eventlisting.model.Location;
import com.nosester.portlet.eventlisting.util.PortletKeys;

public class LocationListingPortletDisplayTemplateHandler extends
    BasePortletDisplayTemplateHandler {

    public String getClassName() {
        return Location.class.getName();
    }

    public String getName(Locale locale) {
        String locations = LanguageUtil.get(locale, "locations");

        return locations.concat(StringPool.SPACE).concat(
            LanguageUtil.get(locale, "template"));
    }

    public String getResourceName() {
        return PortletKeys.LOCATION_LISTING_PORTLET_ID;
    }

    @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(
            "locations", List.class, PortletDisplayTemplateConstants.ENTRIES,
            "location", Location.class, "curlocation", "name");

        return templateVariableGroups;
    }
}

```

Each of the methods in this class have a significant role in defining and implementing ADTs for your custom portlet. View the list below for a detailed explanation for each method defined specifically for ADTs:

- › **getClassName():** Defines the type of entry your portlet is rendering.
- › **getName():** Declares the name of your ADT type (typically, the name of the portlet).
- › **getResourceName():** Specifies which resource is using the ADT (e.g., a portlet) for permission checking.
- › **getTemplateVariableGroups():** Defines the variables exposed in the template editor.

2. Now that we've created the template handler, declare it with the <template-

handler>...</template-handler> tags in the Location Listing Portlet's <portlet> element of your liferay-portlet.xml file:

```
<?xml version="1.0"?>
<!DOCTYPE liferay-portlet-app PUBLIC "-//Liferay//DTD Portlet Application 6.2.0//EN" "http://www.liferay.com/dtd/liferay-portlet-app_6_2_0.dtd">

<liferay-portlet-app>
    ...
    <portlet>
        <portlet-name>locationlisting</portlet-name>
        <icon>/icon.png</icon>
        <configuration-action-class>com.liferay.portal.kernel.portlet.DefaultConfigurationAction</configuration-action-class>
        <template-handler>com.nosester.portlet.eventlisting.template.LocationListingPortletDisplayTemplateHandler</template-handler>
            <instanceable>false</instanceable>
            ...
        </portlet>
    ...
</liferay-portlet-app>
```

3. Since the ability to add ADTs is new to your portlet, we need to configure permissions so that administrative users can grant permissions to the roles that will be allowed to create and manage display templates. Just add the action key

ADD_PORTLET_DISPLAY_TEMPLATE to your portlet's docroot/WEB-INF/src/resource-actions/default.xml file:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action Mapping 6.2.0//EN" "http://www.liferay.com/dtd/liferay-resource-action-mapping_6_2_0.dtd">
<resource-action-mapping>
    ...
    <portlet-resource>
        <portlet-name>locationlisting</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>
```

4. Now that your portlet officially supports ADTs, you'll want to expose the ADT option to your users. Just include the liferay-ui:ddm-template-selector taglib in the

JSP file you're using to control your portlet's configuration mode (e.g., config.jsp if you choose to have it created through Liferay Developer Studio's New Portlet wizard), providing the required information. We'll add the display settings to the Location Listing Portlet's configuration.jsp file:

```
<%@ include file="../init.jsp" %>
...
<aui:form action="<% configurationURL %>" method="post" name="fm">
    <aui:input name="<% Constants.CMD %>" type="hidden" value="<%=
Constants.UPDATE %>" />

    <aui:fieldset>
        <div class="display-template">

            <%
                TemplateHandler templateHandler =
TemplateHandlerRegistryUtil.getTemplateHandler(Location.class.getName()) ;
            %>

            <liferay-ui:ddm-template-selector
                classNameId="<%=
PortalUtil.getClassNameId(templateHandler.getClassName()) %>"
                displayStyle="<% displayStyle %>"
                displayStyleGroupId="<% displayStyleGroupId %>"
                refreshURL="<% PortalUtil.getCurrentURL(request) %>"
                showEmptyOption="<% true %>">
            />
        </div>
    </aui:fieldset>
...
</aui:form>
```

In this JSP, the TemplateHandler object is initialized. Then, we specify the liferay-ui:ddm-template-selector taglib, which implements the Display Template drop-down menu in the Location Listing Portlet's Configuration menu.

5. You're almost finished, but you still have to extend your view code to render your portlet with the selected ADT. Here is where you decide exactly which part of your view will be rendered by the ADT and what will be available in the template context. To do this, add the following code outlined below to your Location Listing Portlet's view.jsp file:

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

This is the <b>Location Listing Portlet</b> in View mode.
...
<%
String displayStyle =
GetterUtil.getString(portletPreferences.getValue("displayStyle",
StringPool.BLANK));
long displayStyleGroupId =
GetterUtil.getLong(portletPreferences.getValue("displayStyleGroupId", null),
scopeGroupId);
```

```

long portletDisplayDDMTemplateId =
PortletDisplayTemplateUtil.getPortletDisplayTemplateDDMTemplateId(displayStyle-
eGroupId, displayStyle);

boolean showLocationAddress_view =
GetterUtil.getBoolean(portletPreferences.getValue("showLocationAddress",
StringPool.TRUE));
%>

<c:choose>
    <c:when test="<%= portletDisplayDDMTemplateId > 0 %>">
        <% List<Location> locations =
LocationLocalServiceUtil.getLocationsByGroupId(scopeGroupId); %>

            <%= PortletDisplayTemplateUtil.renderDDMTemplate(pageContext,
portletDisplayDDMTemplateId, locations) %>
    </c:when>
    <c:otherwise>
        <liferay-ui:search-container emptyResultsMessage="location-empty-
results-message">
            <liferay-ui:search-container-results
results="<%=
LocationLocalServiceUtil.getLocationsByGroupId(scopeGroupId,
searchContainer.getStart(), searchContainer.getEnd()) %>
total="<%=
LocationLocalServiceUtil.getLocationsCountByGroupId(scopeGroupId) %>" />
            ...
        </liferay-ui:search-container>
    </c:otherwise>
</c:choose>

```

In this code snippet, we initialized variables dealing with the display settings (`displayStyle`, `displayStyleGroupId`, and `portletDisplayDDMTemplateId`), and then used a do-otherwise statement to choose between rendering the ADT, or displaying what was originally in the `view.jsp`. If the `portletDisplayDDMTemplateId` exists, the locations list is initialized and the ADT is rendered using the page context, template ID, and locations.

Now that our portlet supports ADTs, you can create your own scripts to change the display of your portlet. We'll experiment by adding our own custom ADT.

1. Navigate to *Admin → Configuration → Application Display Templates*. Then select *Add → Locations Template*. Give your ADT a name and insert the following FreeMarker code into the editor, and click *Save*:

```

<if entries?has_content>
    Quick List:
    <ul>
        <#list entries as curLocation>
            <li>${curLocation.name} - ${curLocation.streetAddress},
${curLocation.city}, ${curLocation.stateOrProvince}</li>
        </#list>
    </ul>
</if>

```

2. Go back to your Location Listing Portlet and select *Options* → *Configuration* and click the *Display Template* drop-down. Select the ADT you created, and click *Save*.

This is the Location Listing Portlet in View mode.

Add Location

Quick List:

- Eiffel Tower - 5 Avenue Anatole France, Paris, France
- Golden Gate Bridge - 1345 North Way, San Francisco, CA
- Wrigley Field - 1060 W Addison St, Chicago, IL

For the Location Listing Portlet, we've created a basic FreeMarker script that takes our locations from the default table format and displays them and their selected fields in a bullet list format.

Once your script is uploaded into the portal and saved, users with the specified roles can select the template when they're configuring the display settings of your portlet on a page. You can visit the Using Application Display Templates section in *Using Liferay Portal* for more details on using ADTs.

Next, we'll provide some recommendations for using ADTs in Liferay Portal.

16.1.2. Recommendations

You've harnessed a lot of power by learning to leverage the ADT API. Be careful, for with great power, comes great responsibility! To that end, let's talk about some practices you can use to optimize your portlet's performance and security.

First let's talk about security. You may want to hide some classes or packages from the template context, to limit the operations that ADTs can perform on your portal. Liferay provides some portal properties to define the restricted classes, packages, and variables. You can override the following portal properties via the `portal-ext.properties` file.

```
freemarker.engine.restricted.classes  
freemarker.engine.restricted.packages  
freemarker.engine.restricted.variables  
velocity.engine.restricted.classes  
velocity.engine.restricted.packages  
velocity.engine.restricted.variables
```

In particular, you may want to add `serviceLocator` to the list of default values assigned to the `freemarker.engine.restricted.variables` and `velocity.engine.restricted.variables` portal properties. Make sure to only add to the classes, packages, and variables restricted by default by `portal.properties`.

Descriptions of Liferay Portal's FreeMarker engine and Velocity engine properties are available on docs.liferay.com.

Application Display Templates introduce additional processing tasks when your portlet is rendered. To minimize negative effects on performance, make your templates as minimal as possible by focusing on the presentation, while using the existing API for complex operations.

The best way to make Application Display Templates efficient is to know your template context well, and understand what you can use from it. Fortunately, you don't need to memorize the context information, thanks to Liferay's advanced template editor!

The template editor provides fields, general variables, and util variables customized for the portlet on which you decide to create an ADT. These variable references can be found on the left-side panel of the template editor. You can use them by simply placing your cursor where you'd like the variable placed, and clicking the desired variable to place it there. You can learn more about the template editor in the Using Application Display Templates section of *Using Liferay Portal*.

Finally, don't forget to run performance tests and tune the template cache options by overriding the following portal properties:

```
freemarker.engine.resource.modification.check.interval  
velocity.engine.resource.modification.check.interval
```

The cool thing about ADTs is the power they provide to your Liferay portlets, providing infinite ways of editing your portlet to provide new interfaces for your portal users. We stepped through how to configure ADTs for a custom portlet like the Location Listing Portlet, tried out a sample template, and ran through important recommendations for using ADTs, which included security and performance.

Next, we'll show you some of the changes to consider in using AlloyUI 2.0 and Twitter® Bootstrap.

16.2. AlloyUI 2.0 / Bootstrap Migration

Liferay 6.2 uses Twitter® Bootstrap-based theming for a slick, vibrant look and feel with instant access to the Twitter® Bootstrap (Bootstrap) theme library. But there are a number of changes that needed to be made to AlloyUI in order to accommodate and properly use Bootstrap. In this section, we'll explain the reasoning behind the changes to AlloyUI and we'll explain how to migrate plugins to use AlloyUI 2.0 and Bootstrap.

Here is an outline of the types of changes you'll need to understand and handle in migrating your plugins:

- Removal of the "aui-" prefixes from all classes
- Module deprecations
- CSS classes replaced with Bootstrap equivalents
- Component output and markup changes
- Icon removals, in favor of using Bootstrap icons

The good news is that Liferay provides a tool for making these changes. But before we show you that tool, we'll explain the impact of each of these AlloyUI changes with respect to Liferay 6.1 plugins. First, let's look at the removal of the "aui-" class name prefix.

16.2.1. Removal of the "aui-" Prefix from All Classes

The "aui-" class prefix was hindering developers from copying and pasting examples from Bootstrap's site into their Liferay plugin code. So we've removed the prefix from all of AlloyUI's

CSS and JavaScript classes. You'll need to update any references to the classes that have been removed. For example, you should remove the "aui-" prefix from the class reference `.aui-ace-autocomplete`, converting the reference to `.ace-autocomplete`. There are plenty more class references like this one that you'll need to update.

There are a number of HTML tags that AlloyUI 1.5 styled by defining custom CSS classes. For example, AlloyUI previously styled the HTML `<fieldset>` tag in a class named `.aui-fieldset`. But since Bootstrap provides styling for these tags, we now leverage the styling by wrapping the Bootstrap code (see `aui.css`). For migrating such classes as `.aui-fieldset` to AlloyUI 2.0, simply remove the "aui-" prefix but append the `.aui` parent class name.

For example, you'd replace this ...

```
.aui-fieldset {  
    // Styling  
}  
... with this ...  
.aui .fieldset {  
    // Styling  
}
```

You can check Bootstrap's `_forms.scss` file for the HTML tags that Bootstrap styles.

Next, let's consider the modules that have been deprecated in AlloyUI 2.0.

16.2.2. AlloyUI Module Deprecations

Because extensive changes were needed for a number of AlloyUI modules, many of the original modules were deprecated. In some cases the original modules were deprecated with no replacement; in other cases we used the original name for the new module implementation and have simply renamed the old module by adding a "-deprecated" suffix to it. AlloyUI 2.0's module API is documented at <http://alloyui.com/api/>, but we've listed the deprecated modules here:

```
aui-autocomplete-deprecated  
aui-autosize-deprecated  
aui-button-item-deprecated  
aui-chart-deprecated  
aui-color-picker-base-deprecated  
aui-color-picker-deprecated  
aui-color-picker-grid-plugin-deprecated  
aui-color-util-deprecated  
aui-data-set-deprecated  
aui-datasource-control-base-deprecated  
aui-datasource-control-deprecated  
aui-datepicker-base-deprecated  
aui-datepicker-deprecated  
aui-datepicker-select-deprecated  
aui-delayed-task-deprecated  
aui-dialog-iframe-deprecated  
aui-editable-deprecated  
aui-form-base-deprecated  
aui-form-comboobox-deprecated  
aui-form-deprecated  
aui-form-field-deprecated
```

```
aui-form-select-deprecated  
aui-form-textarea-deprecated  
aui-form-textfield-deprecated  
aui-input-text-control-deprecated  
aui-io-deprecated  
aui-io-plugin-deprecated  
aui-io-request-deprecated  
aui-live-search-deprecated  
aui-loading-mask-deprecated  
aui-overlay-base-deprecated  
aui-overlay-context-deprecated  
aui-overlay-context-panel-deprecated  
aui-overlay-deprecated  
aui-overlay-manager-deprecated  
aui-overlay-mask-deprecated  
aui-panel-deprecated  
aui-resize-base-deprecated  
aui-resize-constrain-deprecated  
aui-resize-deprecated  
aui-scroller-deprecated  
aui-simple-anim-deprecated  
aui-skin-deprecated  
aui-state-interaction-deprecated  
aui-swf-deprecated  
aui-template-deprecated  
aui-textboxlist-deprecated  
aui-tooltip-deprecated  
aui-tpl-snippets-base-deprecated  
aui-tpl-snippets-checkbox-deprecated  
aui-tpl-snippets-deprecated  
aui-tpl-snippets-input-deprecated  
aui-tpl-snippets-select-deprecated  
aui-tpl-snippets-textarea-deprecated
```

Note, some of these modules have new implementations with the same name, excluding the suffix "-deprecated". Liferay's AlloyUI Upgrade Tool tacks the "-deprecated" suffix onto module references it finds. It's up to you to migrate to the new AlloyUI 2.0 modules.

Next, let's consider the CSS classes that have been replaced by Bootstrap equivalent components.

16.2.3. CSS Classes Replaced with Bootstrap Equivalents

Many of the CSS classes used in AlloyUI 1.5 were replaced with Bootstrap classes or were removed because they didn't blend well with Bootstrap. You may find this with CSS classes in your plugins. Consider replacing your classes with Bootstrap's CSS classes. See <http://liferay.github.io/alloy-bootstrap/base-css.html> for more information on these CSS classes.

Next, let's consider the component output and markup changes in AlloyUI 2.0.

16.2.4. Component Output and Markup Changes

AlloyUI 2.0 introduces appealing new changes in its output and some practical changes to its markup. These changes help facilitate building UIs with a consistent look and feel, and they help

improve UI performance. You can try many of these component changes via the pages of examples and tutorials found on <http://alloyui.com/>. You'll have to take a look at the AlloyUI 2.0 API documentation to understand a number of the markup changes--but here are some common changes:

- Buttons work a little differently in AlloyUI 2.0. By default, they now submit the form. If you don't want that default behavior, you should prevent it by using a DOM event call like this: `event.domEvent.preventDefault();`.
- In `A.Modal`, you now use `close-panel` instead of the old `aui-btn-cancel` tablib.
- To delegate selectors for buttons, simply use `.selector-button` instead of `.selector-button input`.

There are plenty more changes, but at least these are a few to get you started. And remember that the Liferay AlloyUI Upgrade Tool--that we'll introduce shortly--will help you out as well. Next, let's consider the changes in the icons available.

16.2.5. Icon Removals, in Favor of Using Bootstrap Icons

We replaced many icons with those provided by Bootstrap. We also added icons from the Font Awesome project. These icons look great and provide a consistent look and feel throughout Liferay and our plugins. You'll need to update your plugin's references for icons that have been removed. In cases where you use the `liferay:icon` taglib, you simply need to change the value of its `image` attribute to that of a different icon. Consider using the new icons available in Bootstrap, such as their icons from Glyphicons.

We've given you the "dime" tour of the types of changes you'll need to accommodate in the plugins you're migrating from Liferay 6.1 to 6.2, but to really jump-start your migration process, we'll show you Liferay's AlloyUI Upgrade Tool. So, put on your work gloves and get ready to power through migrating your plugins!

16.2.6. Upgrading Plugins with the Liferay AlloyUI Upgrade Tool

To access the `liferay-aui-upgrade-tool` project and install it locally, you'll need an account on GitHub and the Git tool on your machine. Visit <https://github.com/> for instructions on setting up the account and see <http://git-scm.com/> for instructions on installing Git.

Here are some simple steps for forking the `liferay-aui-upgrade-tool` project on GitHub and installing the project locally:

1. Go to the AlloyUI project repository at <https://github.com/liferay/liferay-aui-upgrade-tool>.
2. Click *Fork* to copy Liferay's `liferay-aui-upgrade-tool` repository to your account on GitHub.
3. In your terminal or in GitBash, navigate to the location where you want to put the `liferay-aui-upgrade-tool` project. Then download a clone of the repository by executing the following command, replacing `[username]` with your GitHub user name:

```
git clone git@github.com:[username]/liferay-aui-upgrade-tool
```

4. Navigate into your new.liferay-aui-upgrade-tool repository directory and associate a remote branch to Liferay's.liferay-aui-upgrade-tool repository so you'll be able to fetch its latest changes from time to time:

```
cd.liferay-aui-upgrade-tool  
git remote add upstream git@github.com:liferay/liferay-aui-upgrade-tool
```

5. Lastly, create your own branch named 2.0.x based on Liferay's 2.0.x branch, by execute the following command:

```
git checkout -b 2.0.x upstream/2.0.x
```

You now have all of the.liferay-aui-upgrade-tool project's source code. The project's tool you use to upgrade plugins to AlloyUI 2.0 is called *laut*, which stands for Liferay AUI Upgrade Tool. You build the upgrade tool using Node.js, which is a platform for building applications. You can download it from <http://nodejs.org/>. Linux, OS X, or UNIX users can download its source in a .tar.gz file, unzip it, un-tar it, and build it per the instructions in its README.md file.

Windows users can download the .msi installer file and run it.



Warning: On Windows, only install to locations that have UNIX-friendly paths. Paths like C:\Program Files (x86) that contain space characters and parentheses can prevent software from working properly.

To build the upgrade tool with NodeJS, execute the following command (exclude [sudo] on Windows):

```
[sudo] npm install -g laut
```

To get the usage summary of the upgrade tool, run it with the --help option.

```
laut --help
```

By default, the upgrade tool expects to convert files with extension js, jsp, jspf, and css. You can specify a list of file extensions as arguments to the tool's -e option. With the -f option you can specify individual files or directories to search through and convert. It's common for users to simply specify a single directory for the tool to search and convert all of the files with the default extensions.

```
laut -f some-directory
```

16.2.7. Example: Upgrading the Microblogs Portlet to AlloyUI 2.0

Let's use the upgrade tool to upgrade Liferay's CE 6.1 Microblogs portlet from using AlloyUI version 1.5 to using AlloyUI 2.0. Here's how to run it on the Microblogs portlet in its.liferay-plugins repository:

```
laut -f /home/joe.bloggs/liferay-plugins/portlets/microblogs-portlet
```

Let's take a look at the changes the upgrade tool made to the portlet's JSPs.

In the view.jsp, the upgrade tool renamed the aui-io module to aui-io-deprecated. The tool replaces module references, even if a 2.0 module exists with the same name, for a

couple of different reasons. First, assuming that you look at a diff of the modifications the tool makes, you'll notice that the module has been deprecated. Knowing that, you can investigate whether there is a new AlloyUI 2.0 module that you should start using instead. You should investigate the API for the 2.0 module to find out how it works and to determine how you might use it. Second, by using the deprecated module, you're assured that your code will not run into interpretation errors; it may even exhibit the same behavior as before. You must investigate if the deprecated module's behavior has changed. It's up to you as to when and how to start using a 2.0 module.

In 2.0 many of the "aui-" prefixes were dropped and in some cases modules were completely renamed. For example, the upgrade tool modified the Microblogs portlet's `edit_microblogs_entry.jsp` replacing module reference `aui-helper-hidden` with its new 2.0 module named `hide`.

Let's take a look at a different type of change done in the Microblogs portlet's `edit_microblogs_entry.jsp`. Notice that the "aui-" prefix is deleted from all AlloyUI class names, replacing `aui-button-holder`, `aui-button-disabled`, and `aui-button-submit` class references with `button-holder`, `button-disabled` and `button-submit`, respectively. Keep this type of change throughout your portlet's files.



Warning: Make sure to add the `.aui` parent class reference in front of a classname if you're extending the styling of a class that Bootstrap already styles. See the previous section on *Removal of the aui- Prefix from All Classes* for details.

Lastly, consider the changes done to the Microblogs portlet's `main.js` file. The script now uses the `liferay-util-window` module in place of the old `aui-dialog` module. The upgrade tool took things a step further changing so that the script properly uses the `liferay-util-window` module to get pop-up windows for the portlet. Below are code snippets of what it like before and after running the upgrade tool.

The Microblogs portlet's `main.js` code **before** upgrading:

```
AUI().use(
    'aui-base',
    'aui-dialog',
    'aui-io-plugin',
    ...
    getPopup: function() {
        var instance = this;

        if (!instance._popup) {
            instance._popup = new A.Dialog(
                {
                    centered: true,
                    constrain2view: true,
                    cssClass: 'microblogs-portlet',
                    modal: true,
```

```

        resizable: false,
        width: 475
    }
).plug(
    A.Plugin.IO,
{
    autoLoad: false
}
).render();
}

return instance._popup;
},
...
);

```

The Microblogs portlet's main.js code **after** upgrading:

```

AUI().use(
    'aui-base',
    'liferay-util-window',
    'aui-io-plugin-deprecated',
    ...
    getPopup: function() {
        var instance = this;

        if (!instance._popup) {
            instance._popup = Liferay.Util.Window.getWindow(
            {
                dialog: {
                    centered: true,
                    constrain2view: true,
                    cssClass: 'microblogs-portlet',
                    modal: true,
                    resizable: false,
                    width: 475
                }
            }
        ).plug(
            A.Plugin.IO,
{
            autoLoad: false
}
        ).render();
    }

    return instance._popup;
},
...
);

```

The Liferay AlloyUI Upgrade Tool gives you a great jump-start on migrating your plugins to AlloyUI 2.0. Of course, you should review the upgrade changes and test the changes before redeploying your plugin into a production environment. And remember that the tool may not pick up all of the changes that need to be made to your plugin. But you'll be happy that the tool

does a good bit of the monotonous conversion work for you.

No matter which Liferay APIs you're using, you'll need to understand Liferay's deprecation policy. That way you'll know when methods from our API's are deprecated, and you can make any necessary changes. We'll describe the deprecation policy next.

16.3. *Liferay's Deprecation Policy*

Methods in Liferay's APIs are deprecated when they're no longer called by Liferay internally. Method deprecation occurs during major and minor releases of Liferay. A change in the first or second digits of consecutive Liferay releases indicates a major or minor release, respectively. For example, the release of Liferay Portal 6.2.0 after 5.2.0 was a major release; whereas the release of 6.2.0 after 6.1.30 was a minor release. Major and minor releases can have API deprecations.

APIs should not be deprecated between maintenance releases. Maintenance releases are signified by a change in the third digit of the release number. For example, the release of Liferay Portal 6.1.30 after 6.1.20 was a maintenance release and therefore should have no API deprecations.

To understand Liferays releases, see Using Liferay Portal 6.2

16.4. *Summary*

That about wraps up our chapter on Liferay's APIs. Next, we'll reflect on what we've learned in this guide and conclude our journey together.

17. Conclusions

Liferay Portal is a very flexible platform that allows creating a wide variety of portals and websites. It is the developer through custom applications and customizations who gives it the shape desired by the end users of the portal. Liferay provides several tools (Plugins SDK and Liferay IDE) to ease this task. It also provides the foundations and frameworks to either implement completely new applications (portlet plugins) or customize the core functionalities and applications provided with Liferay (hook plugins and ext plugins).

As the official Developer's Guide for Liferay, this document has offered a description of each of the tools and frameworks that you as a developer can use to build the bests portals out there. Of course, while this document is large, it is just the beginning, the more you learn, the more efficient you will be while developing and the more interesting applications and customizations you will create. Here are some suggestions to learn more after reading this guide:

- Read the "Liferay in Action" book. This book, written by Rich Sezov, Liferay's Knowledge Manager, provides a very extensive step by step guide of Liferay's development technologies.
- Use Liferay's Community Forums, not only to ask questions but also to answer them. You will be surprised how much you can learn while trying to help others.
- Read the source. Liferay is Open Source, and you can leverage that to learn as much as you want about it. Download the code if you haven't done it yet and read it. Link it within your IDE so that you can enter Liferay's code while debugging your own code. It will give you a great opportunity to learn as much as the greatest expert of Liferay in the

world.

- Go to the websites of the standards and libraries that Liferay is based on and read their documentation. Some examples are: Spring, Hibernate, Portlet Specification, etc.