LIFERAY

Enterprise. Open Source. For Life.

Liferay Developer's Guide

Liferay Developer's Guide
Connor McKay, Editor
Jorge Ferrer, Editor
Copyright © 2011 by Liferay, Inc.

This work is offered under the Creative Commons Attribution-Share Alike Unported license.

You are free:

- to share—to copy, distribute, and transmit the work
- to remix—to adapt the work

Under the following conditions:

- **Attribution**. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Share Alike**. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

The full version of this license appears in the appendix of this book, or you may view it online here:
http://creativecommons.org/licenses/by-sa/3.0

Contributors:
Joseph Shum, Alexander Chow, Redmond Mar, Ed Shin, Rich Sezov, Samuel Kong, Connor McKay, Jorge Ferrer, Julio Camarero, Daniel Sanz, Juan Fernández, Sergio González

Table of Contents

Conventions.....5
Publisher Notes.....6
Updates.....6
1. Introduction.....7
Developing applications for Liferay.....7
Portlets.....8
OpenSocial Gadgets.....8
Reusing existing web applications.....9
Supported technologies.....10
Extending and customizing Liferay.....11
Customizing the look and feel: Themes.....11
Adding new predefined page layouts: Layout Templates.....11
Customizing or extending the out of the box functionalities: Hook plugins.....11
Advanced customization: Ext plugins.....12
Choosing the best tool for the job.....12
2. The Plugins SDK.....15
Initial Setup.....15
Ant Configuration.....16
Plugins SDK Configuration.....16
Structure of the SDK.....17
3. Portlet Development.....19
Creating a Portlet.....19
Deploying the Portlet.....20
Anatomy of a Portlet.....20
A Closer Look at the My Greeting Portlet.....22
Writing the My Greeting Portlet.....25
Understanding the two phases of portlet execution: action and render.....27
Passing information from the action phase to the render phase.....30
Developing a portlet with multiple actions.....33
Optional: Adding Friendly URL Mapping to the Portlet.....35
4. Creating Liferay Themes.....37
Introduction.....37
Creating a Theme.....38
Deploying the Theme.....38
Anatomy of a Theme.....38
Thumbnails.....40
JavaScript.....40
Settings.....41
Color Schemes.....42
Portal Predefined Settings.....43
Theme inheritance.....44
5. Hooks.....45
Creating a Hook.....45
Deploying the Hook.....45
Overriding a JSP.....46
Customizing JSPs without overriding the original.....46
Performing a Custom Action.....47
Extending and Overriding portal.properties.....48
Overriding a Portal Service.....48
Overriding a Language.properties File.....50
6. Ext plugins.....51
Creating an Ext plugin.....52

Developing an Ext plugin....54
Set up....54
Initial deployment....54
Redeployment....57
Advanced customization techniques....57
Advanced configuration files....58
Changing the API of an core service....60
Replacing core classes in portal-impl....60
Licencing and Contributing....61
Deploying in production....61
Method 1: Redeploying Liferay's web application....61
Method 2: Generate an aggregated WAR file....62
Migrating old extension environments....63
Conclusions....63
7. Liferay Tools....65
Liferay IDE....65
Installation....66
Requirements....66
Installation steps....66
Alternative installation....69
Set up....69
Liferay Plugins SDK Setup....69
Liferay Portal Tomcat Runtime / Server Setup....72
Importing Existing Projects into Liferay IDE....75
Importing existing Liferay Project from a Plugins SDK....75
Importing an existing Eclipse Project that is not aware of the Liferay IDE....78
Importing an existing Liferay IDE project....79
Verifying that the import has succeeded....79
Setting the Console Encoding....83
Testing the Liferay portal server....84
Create a new Liferay plugin Project....86
Service Builder....88
Define the Model....88
Overview of service.xml....89
Generate the Service....90
Write the Local Service Class....91
Built-In Liferay Services....92
8. Liferay APIs and Frameworks....93
Security and Permissions....93
JSR Portlet Security....93
Liferay's Permission System Overview....95
Implementing Permissions....95
Permission Algorithms....99
Adding a Resource....99
Adding Permission....100
Checking Permissions....101
Asset Framework....105
Adding, updating and deleting assets....106
Entering and displaying tags and categories....108
Publishing assets with Asset Publisher....109
Other frameworks....114
9. Resources for Liferay Developers....117
10. Conclusions....121

PREFACE

This guide was written as a quick reference to getting started developing on the Liferay Portal platform. It is a guide for those who want to get their hands dirty using Liferay's framework and APIs to create fantastic websites.

For a more exhaustive view into Liferay development, we encourage you to check out the complete, official guide to Liferay development, *Liferay in Action*, published by Manning Publications. You can find this book online at http://manning.com/sezov.

The information contained herein has been organized in a format similar to a reference, so that it will be easier to find specific details later.

Conventions

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

Tip: This is a tip. Tips are used to indicate a suggestion or a piece of information that affects whatever is being talked about in the surrounding text. They are always accompanied by this gray box and the icon to the left.

```
Source code and configuration file directives are presented like this.

If source code goes multi-line, the lines will be \
separated by a backslash character like this.
```

Italics are used to represent links or buttons to be clicked on in a user interface and to indicate a label or a name of a Java class.

Bold is used to describe field labels and portlets.

Page headers denote the chapters, and footers denote the particular section within the chapter.

Publisher Notes

It is our hope that this guide will be valuable to you, and that it will be an indispensable resource as you begin to develop on the Liferay platform. If you need any assistance beyond what is covered in this guide, Liferay, Inc. offers training, consulting, and support services to fill any need that you might have. Please see http://www.liferay.com/web/guest/services for further information about the services we can provide.

As always, we welcome any feedback. If there is any way you think we could make this guide better, please feel free to mention it on our forums. You can also use any of the email addresses on our *Contact Us* page (http://www.liferay.com/web/guest/about_us/contact_us). We are here to serve you, our users and customers, and to help make your experience using Liferay Portal the best it can be.

Updates

November 3rd 2010

Extended existing information about the ext plugin and added information about alternatives for deployment to production.

February 27th 2011

- Overall review of the style by Rich Sezov.
- Overall review of the guide and rewrite of some sections by Jorge Ferrer.
- Rewrite of the introduction to make it more welcoming to new developers.
- New section: “Understanding the two phases of portlets: action and render”
- Extended information of the chapter about “Ext Plugins” by Tomas Polesovsky. New section on JSR-286 security by Tomas Polesovsky.
- New chapter about the Asset Framework and quick

introduction to other Liferay frameworks.

- New chapter about Liferay IDE
- New chapter for Conclusions with information about how to learn more after reading this guide.
- New chapter with links to reference documentation.

March 9th 2011

* Minor improvements and fixes based on the comments by community members David H Nebinger and Deb Troxel.

1. INTRODUCTION

Welcome to the Liferay's Developers Guide, the official guide for all developers that want to develop a portal based on Liferay or that want to develop an application that anyone can use in their Liferay installation. This guide will asume that you already know what a portal is and know how to use Liferay from an end user perspective. If you don't, it is recommended that you read the *What is a Portal?* Whitepaper and chapters 1, 5 and 6 of Liferay's Administration Guide.

This first chapter introduces the technologies and tools that you will be able to use to develop applications and also to extend or customise the functionalities that Liferay provides out of the box to adapt them to your needs.

The main mechanism that you as a developer will use to achieve all of the above in a modular way are Liferay Plugins. Liferay Plugins are always distributed as Web Archives (.war files), and deployed through Liferay's deploy mechanisms. There are different types of plugins depending of its purpose. The following sections describe how to develop applications for Liferay and how to perform customizations and the types of plugins that you can use in each case.

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 cohexist with many other applications in a civilized way and also that can reuse many functionalities provided by the portal platform to reduce the development time and deliver a more consistent experience to end users.

If you are reading this, you probably want to (or need to) develop an application that runs on top of Liferay Portal. And you might be looking for an answer to the question of *what is the best and quickest way to do it?* Liferay provides two main ways to develop applications, and both are based on standards:

- Portlets: portlets are small web applications written in Java that follow a certain set of rules to allow cohabitation within the same portal or even within the same page. Portlets can be used to build applications as complex as you want since the full suite of technologies and libraries for the Java platform can be used.
- OpenSocial Gadgets: gadgets are usually small applications, written using browser side technologies such as HTML and Javascript. One interesting benefit of gadgets is that they are easy to develop and for that reason there are thousands of them in repositories such as iGoogle's repository. When the application becomes more complicated you will need a complementary backend technology, such as portlets or regular Java web applications.

The following sections describe these options with some more detail.

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 is only responsible for aggregating the set of portlets that are to appear on any particular page.

Portlets are the least invasive form of extension to Liferay, as they are entirely self contained. Consequentially portlets are also the the most forward compatible development option. They are hot-deployed as plugins into a Liferay instance, 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 portlets development, including Liferay's specific frameworks: MVCPortlet or AlloyPortlet.

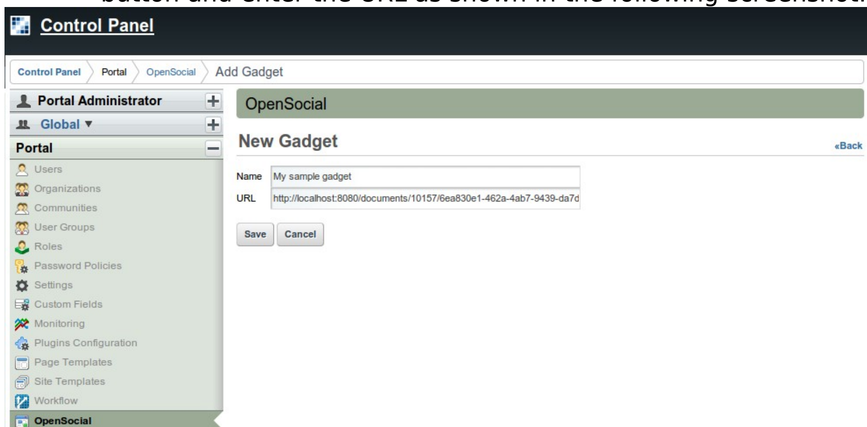
OpenSocial Gadgets

Like portlets, OpenSocial Gadgets are an standard way to develop applications for a portal environment. From a technology perspective one key difference is that they don't mandate an specific backend technology, such as JavaEE, PHP, Ruby, Python, etc. Another difference is that it has been designed specifically to implement social applications, while portlets were designed any type of application.

Because of this, OpenSocial Gadgets, not only provide 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 the user profile, his activities or his friends.

It is possible to deploy OpenSocial Gadgets in Liferay in one of two ways:

- Remote Gadget: A remote gadget is one that is executed in a remote server but is shown to the user in a given page just as if it was another application of the platform. This option is very simple but has the drawback that the portal depends on the remote server for that gadget to work. This might not even be an option in some intranet environments in which there isn't full access to Internet.
- Local Gadget: consists in deploying the gadget in the Liferay server in the similarly as portlets are deployed. Gadgets are defined as an XML file so all you need to do is to upload that file to the server. Some people like to upload them directly through the file system, FTP or similar protocols. In some other cases, just uploading it with theDocument Library portlet and copying the URL is enough. Once you have the URL you can go to the Control Panel > OpenSocial, click the “New Gadget” button and enter the URL as shown in the following screenshot:



After this is done, the gadget will appear as an application that page administrators can add to their pages.

One additional very interesting feature of the latest versions of Liferay is that it is possible to expose any application developed as portlets, as OpenSocial gadgets to the outside world. That is, you can develop a portlet and then let anyone with access to your portlet to add it to the pages of other portals or social networks as a remote gadget.

Note that since an OpenSocial gadget is defined in an XML file, there is no need to create a plugin (that is a .war file) in order to deploy it. All you need to do is make that XML file accessible within the same or another server and let Liferay know the URL.

Reusing existing web applications

Sometimes you don't start from scratch, but there is an existing application that already exists and which has not been implemented using Portlets or OpenSocial Gadgets. What can you do in that situation? There are many options available. Some of the most popular are:

- Rewrite your application as a portlet
- Create simple portlets that interact with your application (possibly using Web Services) and offer all or part of the functionality to end users.
- Create an OpenSocial gadget as a wrapper for your application. The Gadget will use an Iframe to show part of your application in the portal page.
- Create a portlet that integrates the remote application either using an iframe or an HTTP proxy (For example using Liferay's WebProxy portlet). You will also need to find a way to achieve transfer the authentication between the portal and your application.

There are many more options and many reasons why you may want to choose one or another. Reviewing each of them is out of the scope of this guide.

If the existing application has been developed as a JavaEE application, Liferay provides a technology caled Web Application Integrator that allows prototyping the integration and provides several nice features.

In order to use Web Application Integrator all you need to do is deploy the WAR file of your web application as you would do with any Liferay plugin (for example, by using the control panel or by copying it to the deploy directory). As a result Liferay will automatically create a portlet that integrates your application using an iframe.

Supported Technologies

Liferay as a platform strives to provide compatibility with all Java technologies that a developer may want to use to develop their own applications. Thanks to the portlet and the JavaEE specifications each portlet application can use its own set of libraries and technologies regardless of whether they are what Liferay uses itself or not. This section refers mainly to portlet plugins, other plugin types are more

restricted. For example the 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, the task can be daunting to newcomers. This section attempts to provide some advice to help developers choose the best tools for their needs. This advice can be grouped in three:

1. Use what you know: If you already know a framework, that can be your first option (Struts 2, Spring MVC, PHP, Ruby...)
2. Adapt to your real needs: Component based frameworks (JSF, Vaadin, GWT) are specially 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 much simpler than standalone web applications, so, when in doubt use simpler frameworks. (MVC Portlet, Alloy Portlet)

Some of the frameworks mentioned above include their own JavaScript code to provide a very high degree of interaction. That is the case of GWT, Vaadin or JSF implementations such as IceFaces or Rich Faces. In other cases the developers prefer to write their own JavaScript code. In such cases it's most often recommended to use one of the available JavaScript libraries. Some of the most common libraries that can be used with Liferay are jQuery, Dojo, YUI, Sencha (previously known as ExtJs), Sproutcore, etc. Starting with version 6, Liferay also provides its own library called AlloyUI which is based on YUI 3 and provides a large set of components specifically designed for very efficient and interactive applications that work well in portal environments. Liferay's core portlets use AlloyUI for all Javascript code. Developers can also use AlloyUI for their custom portlets or choose any other JavaScript library as long as they make sure that it will not create conflicts with the code of other portlets deployed in the same portal.

Besides the frameworks and libraries mentioned in this section, there are literally thousands more available to Java developers to handle persistence, caching, connections to remote services, and much more. Liferay does not impose specific requirements on the use of any of those frameworks so that portal developers can choose the best tools for their projects.

Extending and customizing Liferay

Liferay provides many out of the box features, included a fully featured content management system, a social collaboration suite and several productivity tools. For some portals those functionalities might be exactly what you need, but for some others you might want to extend them or customize how they work or how they look by default.

Liferay provides several types of plugins that are specialized for an specific type of customization. It is possible to combine several plugin types into a single .war file. For example it is a common practice to combine Themes and Layout Templates. The following sections describe each type of plugin you may need to use.

Customizing the look and feel: Themes

Themes allow the look of the Liferay portal to be changed using a combination of CSS and Velocity templates. In many cases, it is possible to adapt the default Liferay theme to the desired look using only CSS, providing the most forward compatibility. If CSS is not sufficient and more major changes are required, Liferay allows you to include only the templates you modified in your theme, and it will automatically copy the rest from the default theme. Like portlets, themes are hot-deployed as plugins into a Liferay instance.

Adding new predefined page layouts: Layout Templates

Layouts are similar to themes, except that they change the arrangement of portlets on a page rather than its look. Layout templates are also written in Velocity and are hot-deployable.

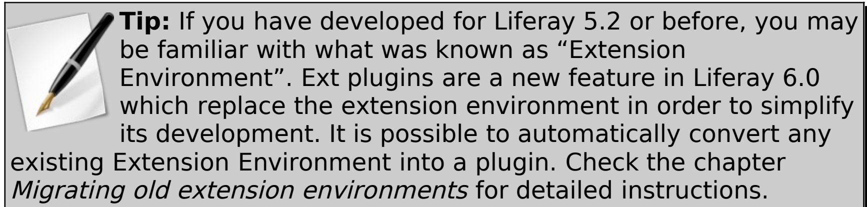
Customizing or extending the out of the box functionalities: Hook plugins

Hook plugins are the recommended method of customizing the the core functionality of Liferay at many predefined extension points. Hook plugins can be used to modify portal properties or to perform custom actions on startup, shutdown, login, logout, session creation and session destruction. Using service wrappers, it is possible for a hook plugin to 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, allowing you to customize their appearance as desired. Best of all, hooks are hot-deployable plugins just like portlets.

Advanced customization: Ext plugins

Ext plugins provide the largest degree of flexibility in modifying the Liferay core, and allow replacing essentially any class with custom implementations. This flexibility comes at a cost however, as 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 where an advanced customization is really needed and there is no other way to accomplish the same goal. Also you should make sure that you are

familiar with the Liferay core to make sure the Ext plugin doesn't have a negative effect on existing funcitonalities.. Even though Ext plugins are deployed as plugins, the server must be restarted for changes to take effect. For this reason, Ext plugins should not be combined with other types of plugins.



Tip: If you have developed for Liferay 5.2 or before, you may be familiar with what was known as “Extension Environment”. Ext plugins are a new feature in Liferay 6.0 which replace the extension environment in order to simplify its development. It is possible to automatically convert any existing Extension Environment into a plugin. Check the chapter *Migrating old extension environments* for detailed instructions.

Choosing the best tool for the job

The Java ecosystem is well know for providing a wide variety of options for almost any developer work that must be done. This is a great advantage because you can find the tool that fits best your needs and the way you work. For that reason once you have found a tool that you are confortable with and have learned to use it you want to keep using it.

On the other hand, the wide variety of tools is often intimidating for newcomers, because they need to choose when they still don't have the experience to decide which option is better.

In Liferay we have taken a pragmatic approach to make sure Liferay developers can benefit from the variety of options while still provide a learning curve that is as soft as possible. To do that we have chosen two Open Source technologies that you can use if you don't have another favorites:

- Apache Ant and the Plugins SDK: Liferay provides a development environment called the Plugins SDK that will allow you to develop all types of plugins by executing a set of predefined commands (also known as targets sin Ant's nomenclature). You can use the Plugins SDK directly from the command line, using editors like Emacs, Vi EditPlus or even the Notepad. You can also integrate it with your favorite IDE, since almost all of them provide support for Apache ant. The next chapter describes how to use the Plugins SDK in detail.
- Eclipse and the Liferay IDE: Eclipse is the most popular and well known Java IDE and provides a wide variety of features. Liferay IDE is a plugin for Eclipse that extend its functionalities to make development of all types of Liferay plugins much easier. Liferay IDE uses the Plugins SDK underneath, but you don't even need to know unless you are trying to perform an

advanced operation not directly supported by the IDE.

This guide will show how to develop for Liferay using the Plugins SDK. We have done so to make sure that it was useful for as many developers as possible even if they don't like IDEs or if they don't use Eclipse.

The guide also has a full chapter on the Liferay IDE. If you are an IDE person and specially if you are an Eclipse user, you may start by reading that chapter first and then go back to reading from the second chapter. It won't be hard to repeat the steps described in the guide using the Liferay IDE.

What about if I don't like Apache Ant and I prefer to use Maven? Many developers prefer one of the 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 so that they can easily be referred from your pom.xml. We are in the process of writing a chapter about using Maven for Liferay development and will be added to this guide in the future. Meanwhile check the following blog entry from Thiago Moreira for more information:

http://www.liferay.com/web/thiago.moreira/blog/-/blogs/liferay-s-artifact-are-now-mavenized

What if I don't like Eclipse and prefer to use Netbeans, IntelliJ IDEA or other IDE? There are many IDEs out there and everyone has its strong aspects. We decided to build Liferay IDE on top of Eclipse because it is the most popular Open Source option. But we also want to make sure developers can use their IDE of choice. In fact quite a few core developers use Netbeans and Intellij IDEA (who has gracefully provided an Open Source license to Liferay's core developers). Both of these IDEs have support for integration with Apache Ant, so you can use the Plugins SDK with them. Additionally, Sun Microsystems developed an extension to Netbeans called the *Portal Pack* that is explicitly designed to develop plugins for Liferay (and their Liferay flavour called WebSpace). You can find more about the Portal Pack in the following URL:

http://netbeans.org/kb/articles/portalpack.html

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

2. The Plugins SDK

Java developers have a wide variety of tools and development environments. Liferay makes every effort to remain tool agnostic, so that you can choose the tools that works best for you. For that reason we provide a Plugins Software Development Kit (SDK) which is based on Apache Ant and may be used along with any editor or Integrated Development Environment (IDE). The chapters of this guide will use the Plugins SDK and a text editor, but you may use whatever tool you are comfortable with. In a later chapter we also introduce Liferay IDE, a plugin for eclipse that simplifies development for Liferay.

Tip: If you are an Eclipse user and prefer to start working with it from be very beginning, you can read that chapter first before reading the rest of the guide..

Initial Setup

Setting up your environment for Liferay development is very straightforward. First, download a fresh copy of Liferay Portal bundled with Tomcat from the Liferay website at http://www.liferay.com/downloads/. We recommend using Tomcat for development, as it is small, fast, and takes up less resources than most other servlet containers. Also download the Plugins SDK from the *Additional Files* page:

1. Unzip both archives to a folder of your choosing. Avoid using a folder name that contains spaces because some operating systems have problems running Java applications in folders with spaces in the name.

2. (Optional) By default Liferay Portal Community Edition comes bundled with many plugins. It's common to remove them to speed up the server startup. To do this, in the *liferay-portal-[version]/tomcat-[tomcat-version]/webapps* directory, delete all the directories except for **ROOT** and **tunnel-web.**

3. Start Liferay:

   - Windows: navigate with the file browser to *liferay-portal-[version]/tomcat-[tomcat-version]/bin and* double click *startup.bat*. To shut the server down later, press Ctrl-C in the terminal window.

   - Linux and Mac OS X: open a terminal, navigate to *liferay-portal-[version]/tomcat-[tomcat-version]/bin*, enter the following command

```
./startup.sh
```

Once Liferay starts your browser should open to http://localhost:8080/ and you can login with the email *test@liferay.com* and password *test*.

Ant Configuration

Before you can begin developing portlets, you must first have some tools installed. Building projects in the Plugins SDK requires that you have Ant 1.7.0 or higher installed on your machine. Download the latest version of Ant from http://ant.apache.org/. Decompress the archive into a folder of your choosing.

Next, set an environment variable called **ANT_HOME** which points to the folder to which you installed Ant. Use this variable to add the binaries for Ant to your **PATH** by adding $**ANT_HOME/bin** to your **PATH** environment variable.

You can do this on Linux or Mac OS X by modifying your *.bash_profile* file as follows (assuming you installed Ant in */java*):

```
export ANT_HOME=/java/apache-ant-1.8.1
export PATH=$PATH:$ANT_HOME/bin
```

Close and reopen your terminal window to make these settings take effect.

You can do this on Windows by going to ***Start*** -> *Control Panel*, and double-clicking the *System* icon. Go to *Advanced*, and then click the *Environment Variables* button. Under *System Variables*, select *New*. Make the Variable Name **ANT_HOME** and the Variable Value the

path to which you installed Ant (e.g., *c:\java\apache-ant-1.8.1*), and click *OK*.

Scroll down until you find the **PATH** environment variable. Select it and click *Edit*. Add **%ANT_HOME%\bin** to the end of the Variable Value. Click *OK*, and then click *OK* again. Open a command prompt and type **ant -version** and press Enter. You should get and output similar to this:

```
Apache Ant(TM) version 1.8.2 compiled on December 20 2010
```

If not, check your environment variable settings and make sure they are pointing to the directory where you unzipped Ant.

Plugins SDK Configuration

Now that all the proper tools are in place, we must configure the Plugins SDK to be able to deploy into your Liferay instance. You will notice that the Plugins SDK contains a file called *build.properties*. This file 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 should not 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 in the same folder called *build.${user.name}.properties*, where `${user.name}` is your user ID on your machine. For example, if your user name is jsmith (for John Smith), you would create a file called *build.jsmith.properties*.

Edit this file and add the following line:

```
app.server.dir=the directory containing your application server
```

In our case, **app.server.dir** should be the absolute path to your *liferay-portal-[version]/tomcat-[tomcat-version]* directory.

Save the file. You are now ready to start using the Plugins SDK.

Structure of the SDK

Each folder in the Plugins SDK contains scripts for creating new plugins for that type. New plugins are placed in their own subdirectory of the appropriate plugin directory. For instance, a new portlet called "greeting-portlet" would reside in *liferay-plugins-sdk-6/portlets/greeting-portlet*.

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 could keep those portlets and themes in their own Plugins SDK project in your source code repository. If you also have an external instance of Liferay for your public Internet web site, you could

have a separate Plugins SDK with those projects as well. Or you could further separate your projects by having a different Plugins SDK project for each portlet or theme project.

It is also possible to use use the Plugins SDK as a simple cross-platform project generator. You can create a plugin project using the Plugins SDK and then copy the resulting project folder to your IDE of choice. This method requires some manual modification of the ant scripts, but it makes it possible to conform to the strict standards some organizations have for their Java projects.

3. Portlet Development

In this chapter we will 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. Finally we will add friendly URL mapping to the portlet to clean up its URLs.

In developing your own portlets you are free to use any framework you prefer, such as Struts, Spring MVC, or JSF. For this portlet we will use the Liferay MVCPortlet framework as it is simple, lightweight, and easy to understand.

Additionally, Liferay allows for the consuming of PHP and Ruby applications as portlets, so you do not need to be a Java developer in order to take advantage of Liferay's built-in features (such as user management, communities, page building and content management). You can use the Plugins SDK to deploy your PHP or Ruby application as a portlet, and it will run seamlessly inside of Liferay. There are plenty of examples of this; to see them, check out the directory *plugins/trunk* from Liferay's public Subversion repository.

Creating a Portlet

Creating portlets with the Plugins SDK is extremely simple. As noted before, there is a *portlets* folder inside of the Plugins SDK folder. This is where your portlet projects will reside. To create a new portlet, first decide what its name is going to be. You need both a project name (without spaces) and a display name (which can have spaces). When you have decided on your portlet's name, you are ready to create the project. For the greeting portlet, the project name is “my-greeting”, and the portlet title is “My Greeting”. Navigate to the

portlets directory in the terminal and enter the following command (Linux and Mac OS X):

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

On Windows enter the following instead:

```
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 may 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 class path entries in the *build-common.xml* file in the Plugins SDK project. You will be able to determine from the *plugin.classpath* and *portal.classpath entries* which .jar files are necessary to build your newly created portlet project.

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.

Deploying the Portlet

Liferay provides a mechanism called autodeploy that makes deploying portlets (and any other plugin type) a breeze. All you need to do is drop a WAR file into a directory and the portal will take care of making any necessary changes specific to Liferay and then deploy it to the application server. This will be the method used throughout this guide.

Tip: Liferay supports a wide variety of application servers. Many of them, such as Tomcat or Jboss, provide a simple

way to deploy web applications by just copying a file into a folder and Liferay's autodeploy mechanism makes use of that possibility. You should be aware though that some application servers, such as Websphere or Weblogic require the use of specific tools to deploy web applications, so Liferay's autodeploy process won't work for them.

Open a terminal window in your *portlets/my-greeting-portlet* directory and enter this command:

```
ant deploy
```

You should get a BUILD SUCCESSFUL message, which means that your portlet is now being deployed. If you switch to the terminal window running Liferay, and wait for a few seconds, you should see the message “1 portlet for my-greeting-portlet is available for use”. If not, something is wrong and you should double-check your configuration.

Go to your web browser and log in to the portal as explained earlier. Then hover over *Add* at the top of the page, and click on *More*. Select the *Sample* category, and then click *Add* next to *My Greeting*. Your portlet should appear in the page below. Congratulations, you've just created your first portlet!

Anatomy of a Portlet

A portlet project is made up at a minimum of three components:

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

When using Liferay's Plugins SDK these files are stored in a standard directory structure which looks like the following:

```
/PORTLET-NAME/
    build.xml
    /docroot/
        /css/
        /js/
        /WEB-INF/
            /src/ (not created by default)
            liferay-display.xml
            liferay-plugin-package.properties
            liferay-portlet.xml
            portlet.xml
            web.xml
```

```
        icon.png
        view.jsp
```

The portlet we just created is a fully functional portlet which can be deployed to your Liferay instance.

New portlets are configured by default to use the MVCPortlet framework, a very light framework that hides part of the complexity of portlets and makes the most common operations easier. MVCPortlet uses separate JSPs for each page in the portlet.

Portlets created in the SDK are configured by default to use the MVCPortlet framework, a very light framework that hides part of the complexity of portlets and makes the most common operations easier. MVCPortlet uses separate JSPs for each page in the portlet. MVCPortlet uses by default a JSP with the mode name for each of the registered portlet modes. For example edit.jsp for the edit mode, help.jsp for the help mode, etc.

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

The **Configuration Files** are stored in the *docroot/WEB-INF* folder. The standard JSR-286 portlet configuration file *portlet.xml* is here, as well as three Liferay-specific configuration files. The Liferay-specific configuration files are completely optional, but are important if your portlets are going to be deployed on a Liferay Portal server.

liferay-display.xml: This file describes what category the portlet should appear under in the *Add* menu in the dockbar (the horizontal bar that appear at the top of the page to all logged in users).

liferay-portlet.xml: This file describes some optional Liferay-specific enhancements for JSR-286 portlets that are installed on a Liferay Portal server. For example, you can set whether a portlet is instanceable, which means that you can place more than one portlet instance on a page, and each one will have its own separate data. Please see the DTD for this file for further details, as there are too many settings to list here. The DTD may be found in the *definitions* folder in the Liferay source code.

liferay-plugin-package.properties: This file describes the plugin to Liferay's hot deployer. One of the things that can be configured in this file is dependency .jars. If a portlet plugin has dependencies on particular .jar files that already come with Liferay, you can specify them in this file and the hot deployer will modify the .war file on deployment to copy those .jars inside. That way you don't have to include the .jars yourself and the .war will be lighter.

Client Side Files are the .jsp, .css, and JavaScript files that you write to implement your portlet's user interface. These files should go in the docroot folder somewhere—either in the root of the folder or in a folder structure of their own. Remember that with portlets you are only dealing with a portion of the HTML document that is getting returned

to the browser. Any HTML code you have in your client side files should be free of global tags such as `<html>` or `<head>`. Additionally, all CSS classes and element IDs must be namespaced to prevent conflicts with other portlets. Liferay provides tools (a taglib and API methods) to generate the namespace that you should use.

A Closer Look at the My Greeting Portlet

If you are new to portlet development, this section will take a closer look at the configuration options of a portlet.

docroot/WEB-INF/portlet.xml

When using the Plugins SDK, the default content of the portlet descriptor is as follows:

```
<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-jsp</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>
```

Here is a basic summary of what each of the elements represents:

portlet-	The portlet-name element contains the canonical

name	name of the portlet. Each portlet name is unique within the portlet application (that is, within the portlet plugin). This is also referred within Liferay Portal as the portlet id
display-name	The display-name type contains a short name that is intended to be displayed by tools. It is used by display-name elements. The display name need not be unique.
portlet-class	The portlet-class element contains the fully qualified class name that will handle invocations to the portlet.
init-param	The init-param element contains a name/value pair as an initialization param of the portlet.
expiration-cache	Expiration-cache defines expiration-based caching for this portlet. The parameter indicates the time in seconds after which the portlet output expires. -1 indicates that the output never expires.
supports	The supports element contains the supported mime-type. Supports also indicates the portlet modes a portlet supports for a specific content type. All portlets must support the view mode. The concept of "portlet modes" is defined by the portlet specification. Modes are used to separate certain views of the portlet from others. What is special about portlet modes is that the portal knows about them and can provide generic ways to navigate between portlet modes (for example through links in the box surrounding the portlet when it is added to a page). For that reason they are 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.
portlet-info	Portlet-info defines portlet information.
security-role-ref	The security-role-ref element 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, Plugins SDK sets the contents of this descriptor to the following:

```
<liferay-portlet-app>
    <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>
    <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 is a basic summary of what some of the elements represents.

portlet-name	The portlet-name element contains the canonical name of the portlet. This needs to be the same as the portlet-name given in portlet.xml
icon	Path to icon image for this portlet
instanceable	Indicates if multiple instances of this portlet can appear on the same page.
header-portlet-css	The path to the .css file for this portlet to be included in the <head> 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 </body>

There are many more elements that you should be aware of for more advanced development. Please see the DTD for this file in the *definitions* folder in the Liferay source code for more information.

Writing the My Greeting Portlet

Now that you are familiar with the structure of a portlet, it's time to

actually make it do something useful. Our portlet will have two pages. *view.jsp* will display the greeting and provide a link to the edit page. *Edit.jsp* will show a form with a text field allowing the greeting to be changed, along with a link back to the view page. MVCPortlet class will handle the rendering of our JSPs, so for this example we won't have to write a single Java class.

First, we don't want multiple greetings on the same page, so we are going to make the My Greeting portlet non-instanceable. To do this, edit *liferay-portlet.xml* and change the value of the element *instanceable* from true to false so that it looks like this:

```
<instanceable>false</instanceable>
```

Next, we will create our JSP templates. Start by editing *view.jsp* and replacing 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="jspPage" 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="jspPage" 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="jspPage" value="/view.jsp" />
</portlet:renderURL>

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

Deploy the portlet again by entering the command **ant deploy** in 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. Simply 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 notice in 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 *jspPage*, which is used by MVCPortlet to determine which JSP to render for each request. You must always use taglibs to generate URLs to your portlet. This restriction exists because the portlet does not own the whole page, only a fragment of it, so the URL must always go to the portal who is responsible for rendering, not only your portlet but also 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 be able to render the whole page.

Second, notice that the form in *edit.jsp* has the prefix `aui`, signifying that it is part of the Alloy UI tag library. Alloy greatly simplifies the code required to create nice looking and accessible forms, by providing tags that will 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 that you may have noticed is `<portlet:defineObjects/>`. The portlet specification defined this tag in order to be able to insert into the JSP a set of implicit variables that are useful for portlet developers such as renderRequest, portletConfig, portletPreferences, etc.

One word of warning about the portlet we have just built. For the purpose of making this example as simple and easy to follow as possible, we have cheated a little bit. The portlet specification does not allow to set preferences from a JSP, because they are executed in what is known as the render state. There are good reasons for this restriction, that are explained in the next section.

Understanding the Two phases of Portlet Execution

One of the characteristics of portlet development that confuses most developers used to regular servlet development or who are used to other environments such as PHP, Python or Ruby is the need for two phases. The good news is that once you get used to them they become simple and useful.

The reason why two phases are needed is because a portlet does not own a whole HTML page, it only generates a fragment of it. The portal that holds the portlet is the one responsible for generating the page by invoking one or several portlets and adding some additional HTML around them. Usually, when the user interacts with the page, for example by clicking a link or a button, she's doing it within a specific portlet. The portal must forward the action performed by the user to that portlet and after that it must render the whole page, showing the content of that portlet, which may have changed, and also the content of the other portlets. For the other portlets in the page which have not been invoked by the user, what the portal does to get their content is to repeat the last invocation again (since it assumes it will yield the same result).

Now imagine this scenario: we have a page with two portlets, a navigation portlet and a shopping portlet. A user comes to the page and does the following:

1. Load the page
2. Clicks a button on the shopping portlet that automatically

charges an amount on her credit card and starts a process to ship her the product she just bought. After this operation the portal also invokes the navigation portlet with its default view.

3. Click a link in the navigation portlet which causes the content of the portlet to change. After that the portal must also show the content of the shopping portlet, so 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.

I guess that by now you can tell that this is not right. Since the portal doesn't know whether the last operation on a portlet was an action or not, it would have no option but to repeat it over and over to obtain the content of the portlet again (at least until the Credit Card reached its limit).

Fortunately portals don't work that way. In order to prevent situations like the one described above, the portlet specification defines two phases for every request of a portlet, to allow 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 and is usually the result of an user interaction with the portlet. In this phase the portlet can change its status, for instance changing the user preferences of the portlet. It is also recommended that any inserts and modifications in the database or operations that should not be repeated are performed in this phase.

- **Render phase**: The render phase is always invoked for all portlets in 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 executedis 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 value.

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

The file portlet.xml also needs to be changed so that it points to our new class:

```
<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-jsp</name>
        <value>/view.jsp</value>
    </init-param>
    …
```

Finally, you will need to do a minor change in the *edit.jsp* file and change the URL to which the form is sent to let the portal know that it

should execute the action phase. This is the perfect moment for you to know that there are three types of URLs that can be generated by a portlet:

- *renderURL*: this is the type of URL that we have used so far. It invokes a portlet using only its render phase.
- *actionURL*: this type of URL tells the portlet that it should execute its action phase before rendering all the portlets in the page.
- *resourceURL*: this type of URL can be used to retrieve images, XML, JSON or any other type of resource. It is often used to generate images or other media types dynamically. It is very useful also to make AJAX requests to the server. The key difference of this URL type in comparison to the other two is that the portlet has full control of the data that will be sent in response.

So we must change the *edit.jsp* to use an actionURL by using the JSP tag of the same name. We also remove the previous code that was saving the preference:

```
<%@ 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="jspPage" 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="jspPage" value="/view.jsp" />
</portlet:renderURL>
```

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

Try deploying again the portlet after making these changes, everything should work exactly like before.

Well, almost. If you have paid close attention you may have missed something, now the portlet is no longer showing a message to the user to let him know that the preference has been saved right after clicking the save button. In order to implement that we must have a way to pass information from the action phase to the render phase, so that the JSP can know that the preference has just been saved and then show a message to the user.

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 one is through render parameters. Within the implementation in the *processAction* method you can invoke the *setRenderParameter* to add a new parameter to the request that the render phase will be able to read:

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

From the render phase (in our case, the JSP), this value can be read using the regular parameter reading method:

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

It is important to be aware that when invoking an action URL, the parameters specified in the URL will only be readable from the action phase (that is 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 just by setting the following init-param in your portlet.xml:

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

I mentioned there was a second method and in fact it is a better method for what we are trying to do in our example. One final thing you should know about render parameters is that the portal remembers them for all later executions of the portlet until the portlet is invoked again 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 in the page, each time the page is reloaded the portal will render our portlet using the render parameters that we 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 method 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. By 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 that the message is only shown once. Liferay provides a helper class and taglib to do this operation easily. In the processAction 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);
    }
}
```

Also, in the JSP you would need to add the liferay-ui:success JSP tag as shown below (note that you also need to add the taglib declaration at the top):

```
<%@ 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.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.");
%>

<liferay-ui:success key="success" message="Greeting saved successfully!" />

<portlet:actionURL var="editGreetingURL">
    <portlet:param name="jspPage" 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="jspPage" value="/view.jsp" />
</portlet:renderURL>

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

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

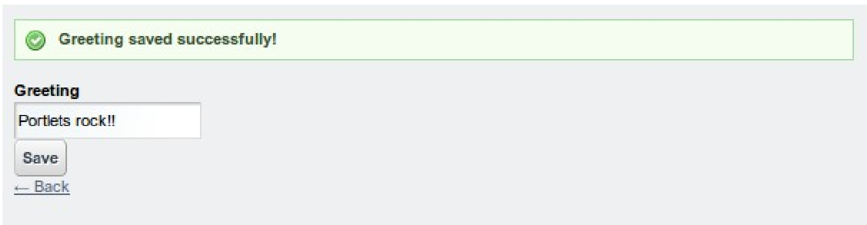


Illustration 1: The sample “My Greetings” portlet showing a success message

There is also an equivalent util class for notifying errors. This is commonly used after catching an exception in the *processAction*. For example.

```
try {
    prefs.setValue("greeting", greeting);
    prefs.store();
}
catch(Exception e) {
    SessionErrors.add(actionRequest, "error");
}
```

And then the error, if it exists, is shown in the JSP using the liferay-ui:error tag:

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

When the error occurs you should see something like this in your portlet:

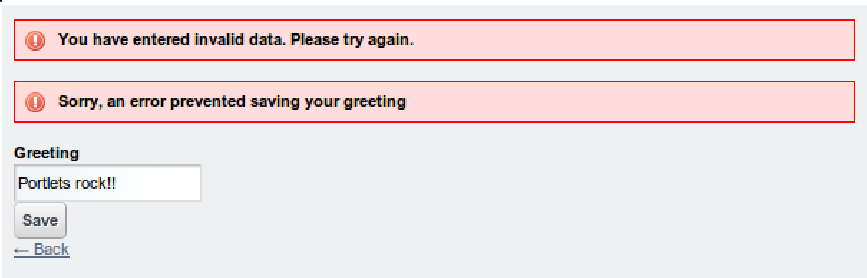


Illustration 2: The sample “My Greetings” portlet showing an error message

The first message is automatically added by Liferay. The second one is the one you entered in the JSP.

Developing a Portlet with Multiple Actions

So far we have developed a portlet that has two different views, the default view and an edit view. Adding more views is easy and all you have to do to link to them is to use the *jspPage* parameter when creating the URL. But we only have one action. How do we add another action, for example for sending an email to the user?

You can have as many actions as you want in a portlet, each of them will need to be implemented as a method that receives two parameters, an ActionRequest and an ActionResponse. The name of the method can be whatever you want since you will be referring to it when creating the URL.

Let's rewrite the example from the previous section to use custom names for the methods of the action to set the greeting and add a second action.

```
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
    }
}
```

Note how we no longer need to invoke the processAction method of the super class, because we are not overriding it.

This change of name also requires a simple change in the URL, to specify the name of the method that should be invoked to execute the action. In the edit.jsp edit the actionURL so that it looks like this:

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

That's it, now you know all the basics of portlets and are ready to use your Java knowledge to build portlets that get integrated in Liferay. The next section explains an extension provided by Liferay to the portlet specification to provide pretty URLs to your portlets that you can use if desired.

Optional: Adding Friendly URL Mapping to the Portlet

You will notice that when you click the *Edit greeting* link, you are taken to a page with a URL similar to 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&_mygreeting_WAR_mygreetingportlet_jspPage=
%2Fedit.jsp
```

In Liferay 6 there is a new feature that requires minimal work to change this into:

```
http://localhost:8080/web/guest/home/-/my-greeting/edit
```

This feature, known as friendly URL mapping, takes unnecessary parameters out of the URL and allows you to place the important parameters in the URL path rather than the query string. To add this functionality, first edit *liferay-portlet.xml* and add the following lines directly after </icon> and before <instanceable>. Be sure to remove the line breaks and the backslashes!

```
<friendly-url-mapper-class>com.liferay.portal.kernel.portlet.Default\
FriendlyURLMapper</friendly-url-mapper-class>
<friendly-url-mapping>my-greeting</friendly-url-mapping>
<friendly-url-routes>com/sample/mygreeting/portlet/my-greeting-friendly-url\
-routes.xml</friendly-url-routes>
```

Next, create the file (note the line break):

```
my-greeting-portlet/docroot/WEB-INF/src/com/sample/mygreeting/portlet/my\
-greeting-friendly-url-routes.xml
```

Create new directories as necessary. Place the following content

into the new file:

```
<?xml version="1.0"?>
<!DOCTYPE routes PUBLIC "-//Liferay//DTD Friendly URL Routes 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-friendly-url-routes_6_0_0.dtd">
<routes>
    <route>
        <pattern>/{jspPageName}</pattern>
        <generated-parameter name="jspPage">/{jspPageName}.jsp</generated-
parameter>
    </route>
</routes>
```

Redeploy your portlet, refresh the page, and try clicking either of the links again. Notice how much shorter and more user-friendly the URL is, without even having to modify the JSPs. For more information on friendly URL mapping, you can check full discussion of this topic in *Liferay in Action*.

4. Creating Liferay Themes

Themes are hot deployable plugins which can completely transform the look and feel of the portal. Theme creators can make themes to provide an interface that is unique to the site that the portal will serve. Themes make it possible to change the user interface so completely that it would be difficult or impossible to tell that the site is running on Liferay. Liferay provides a well organized, modular structure to its themes. This allows the theme developer to be able to quickly modify everything from the border around a portlet window to every object on the page, because all of the objects are easy to find. Additionally, theme developers do not have to customize every aspect of their themes. A theme can inherit the styling, images, and templates from any of the built in themes, overriding them only where necessary. This allows themes to be smaller and less cluttered with extraneous data that already exists in the default theme (such as graphics for emoticons for the message boards portlet).

Introduction

Liferay's themes are designed in such way that they can be very easy to create. You can start by making changes in CSS files and as your customization requirements grow you can also make changes to the HTML that controls the page design.

Some of the technologies that you may need to know in order to make the best use of themes are:

- CSS: If desired you can create a whole new theme just by

changing a CSS file.

- Velocity: a simple yet powerful tool to create templates. You will need to use it in order to customize the HTML generated by the theme.
- JavaScript: can be used to add special behaviors.
- XML: each theme has a configuration file written in XML. You will use this file to specify some settings of the theme.

To follow the examples of this guide you will also need some familiarity with using the command line. Alternatively you can use the Liferay IDE and use its menus instead of the commands used in the text.

But let's finish the introduction and get started with our first theme.

Creating a Theme

The process for creating a new theme is nearly identical to the one for making a portlet. You will need both a project name (without spaces) and a display name (which can have spaces). For example, the project name could be “deep-blue”, and the theme title “Deep Blue”. In the terminal, navigate to the *themes* directory in the Plugins SDK and enter the following command (Linux and Mac OS X):

```
./create.sh deep-blue "Deep Blue"
```

On Windows enter the following instead:

```
create.bat deep-blue "Deep Blue"
```

This command will create a blank theme in your *themes* folder. Notice that the Plugins SDK automatically appends “-theme” to the project name when creating this folder.

Deploying the Theme

Open a terminal window in your *themes/deep-blue-theme* directory and enter this command:

```
ant deploy
```

You should get a BUILD SUCCESSFUL message, which means that your theme is now being deployed. If you switch to the terminal window running Liferay, and wait for a few seconds, you should see the message “1 theme for deep-blue-theme is available for use”.

Go to your web browser and login to the portal as explained earlier. Then hover over **Manage** at the top of the page, and click on *Page*. Directly underneath the words **Manage** Pages select the *Look and Feel* tab. Simply click on your theme to activate it.

Anatomy of a Theme

Custom themes are based on differences from one of several built-in Liferay themes.

The structure of a theme is designed to separate different types of resources into easily accessible folders. The full structure of the deep blue theme is shown below:

```
/deep-blue-theme/
    /docroot/
        /WEB-INF/
            liferay-plugin-package.properties
        /_diffs/ (subfolders not created by default)
            /css/
            /images/
            /js/
            /templates/
        /css/
            application.css
            base.css
            custom.css
            dockbar.css
            extras.css
            forms.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
```

You will notice that there is a `_diffs` folder inside the `docroot` directory of your theme; this is where you will place your theme code. You only need to customize the parts of your theme that will differ from the parent theme. To do this, you mirror the directory structure of the parent theme inside of the `_diffs` folder, placing only the folders and files you need to customize there.

You will also notice that there are several other folders inside `docroot`; these were copied over from the parent theme in your Liferay bundle when you deployed your theme. You should use these files as

the basis for your modifications. For example, to customize the navigation, you would copy `navigation.vm` from `deep-blue-theme/docroot/templates/navigation.vm` into `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 would put all of your new styles and all of your overrides of the styles in the parent theme. `Custom.css` is loaded last, and so styles in this file are able to override any styles in the parent theme.

Best practice recommends that you make all your custom themes using only the `custom.css` file, and that you not override any of the templates unless absolutely necessary. This will make future upgrades far easier, as you won't have to manually modify your templates to add support for new Liferay features.

Whenever you make modifications to your theme, redeploy it by opening a terminal in `themes/deep-blue-theme` and entering the command **ant deploy**. Wait a few seconds until the theme deploys, and then refresh your browser to see your changes.

Tip: If you wish to see changes even more quickly, it is also possible to modify you theme directly in your Liferay bundle. In our example, *custom.css* is located in *liferay-portal-[version]/tomcat-6.0.26/webapps/deep-blue-theme/css*. However, for modifications made here to appear in your browser as soon as you refresh the page, you must enable Liferay Developer Mode. See the Liferay wiki for more information.

Also make sure that you copy any changes you make back into your *_diffs* folder, or they will be overwritten when you redeploy your theme.

Thumbnails

You will notice that in the *Look and Feel* settings the *Classic* theme has a thumbnail preview of what it looks like, while our theme has only a broken image. To correct this, take a screenshot of your theme and save it in `_diffs/images` with the name `thumbnail.png`. It must have the exact size of 150 pixels wide by 120 pixels high. You should also save a larger version in the same directory with the name `screenshot.png`. Its size must be exactly 1080 pixels wide by 864 pixels high. After redeploying your theme, it will have a thumbnail preview just like the *Classic* theme.

JavaScript

Liferay now includes its own JavaScript library called Alloy, which is

an extension to Yahoo's YUI3 framework. Developers can take advantage of the full power of either of these frameworks in their themes. Inside of the `main.js` file, you will find definitions for three JavaScript callbacks:

```
AUI().ready(
    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() {
    }
);
```

- **AUI().ready(fn);**

This callback is executed as soon as 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*. p*ortletId* 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—including AJAX portlets—has finished loading.

Settings

Each theme can define settings to make it configurable. These settings are defined in a file named `liferay-look-and-feel.xml` inside `WEB-INF`. This file does not exist by default, so you should now create it with the following content:

```
<?xml version="1.0"?>
<!DOCTYPE look-and-feel PUBLIC "-//Liferay//DTD Look and Feel 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-look-and-feel_6_0_0.dtd">

<look-and-feel>
    <compatibility>
        <version>6.0.0+</version>
    </compatibility>
    <theme id="deep-blue" name="Deep Blue">
        <settings>
            <setting key="my-setting" value="my-value" />
        </settings>
    </theme>
</look-and-feel>
```

You can define additional settings by adding more <setting> elements. These settings can be accessed in the theme templates using the following code:

```
$theme.getSetting("my-setting")
```

For example, say we need to create two themes that are exactly the same except for some changes in the header. One of the themes has more details while the other is smaller (and takes less screen real estate). Instead of creating two different themes, we are going to create only one and use a setting to choose which header we want.

In the `portal_normal.vm` template we could write:

```
#if ($theme.getSetting("header-type") == "detailed")
    #parse ("$full_templates_path/header_detailed.vm")
#else
    #parse ("$full_templates_path/header_brief.vm")
#end
```

Then when we write the `liferay-look-and-feel.xml`, we write two different entries that refer to the same theme but have a different value 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>
```

Color Schemes

Color schemes are specified using a CSS class name, with which you can not only change colors, but also choose different background images, different border colors, and so on.

In your `liferay-look-and-feel.xml`, you can define color schemes like so:

```
<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>
```

Inside of your `_diffs/css` folder, create a folder called `color_schemes`. Inside of that folder, place a `.css` file for each of your color schemes. In the case above, we would could either have just one called `night.css` and let the default styling handle the first color scheme, or you could have both `day.css` and `night.css`.

Assuming you follow the second route, place the following lines at the bottom of your `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 you styling. In `day.css` you would prefix all of your CSS styles like this:

```
body.day { background-color: #ddf; }
.day a { color: #66a; }
```

And in `night.css` you would prefix all of your CSS styles like this:

```
body.night { background-color: #447; color: #777; }
.night a { color: #bbd; }
```

You can also create separate thumbnail images for each of your color schemes. The `<color-scheme-images-path>` element tells Liferay where to look for these images (note that you only have to place this

element in one of the color schemes for it to affect both). For our example, create the folders `_diffs/images/color_schemes/day` and `_diffs/images/color_schemes/night`. In each of these folders place a `thumbnail.png` and `screenshot.png` file with the same sizes as before.

Portal Predefined Settings

The portal defines some settings that allow the theme to determine certain behaviors. So far there are only two predefined settings but this number may grow in the future. These settings can be modified from `liferay-look-and-feel.xml`.

portlet-setup-show-borders-default

If set to false, the portal will turn off borders by default for all the portlets. The default is **true**.

Example:

```
<settings>
    <setting key="portlet-setup-show-borders-default" value="false" />
</settings>
```

This default behavior can be overridden for individual portlets using:

- `liferay-portlet.xml`
- Portlet CSS popup setting

bullet-style-options

This setting is used by the *Navigation* portlet to determine the CSS class name of the list of pages. The value must be a comma separated list of valid bullet styles to be used.

Example:

```
<settings>
    <setting key="bullet-style-options" value="classic,modern,tablemenu" />
</settings>
```

The bullet style can be changed by the user in the *Navigation* portlet configuration. The chosen style will be applied as a CSS class on the <div> containing the navigation. This class will be named in the following pattern:

```
.nav-menu-style-{BULLET_STYLE_OPTION} {
    ... CSS selectors ...
}
```

Here is an example of the HTML code that you would need to add style through CSS code. In this case the bullet style option is **modern**:

```
<div class="nav-menu nav-menu-style-modern">
    <ul class="breadcrumbs lfr-component">
```

```
        ...
    </ul>
</div>
```

Using CSS and/or some unobtrusive Javascript it's possible to implement any type of menu.

Theme inheritance

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 will see the following:

```
<project name="theme" basedir="." default="deploy">
    <import file="../build-common-theme.xml" />
    <property name="theme.parent" value="_styled" />
</project>
```

The `theme.parent` property determines which built-in theme your theme will inherit from. In addition to the **_styled** theme, you may also choose to inherit from the **_unstyled** theme, which contains no styling whatsoever. This involves more work, but in exchange you get full flexibility to design your own CSS files from scratch.

You can also use the default Liferay theme, called **classic**, as the parent of your themes. Using this approach allows you to start with a look and feel that already works and get nice results quickly. The drawback is that since there is so much done already for you, there won't be as much flexibility to build the desired design. It's a compromise between creating a theme as quickly as possible versus having full control of the result. It's your choice.

5. Hooks

Liferay Hooks are the newest type of plugin which Liferay Portal supports. They were introduced late in the development cycle for Liferay Portal 5.1.x, and are now the preferred way to customize Liferay's core features. As with portlets, layout templates, and themes, they are created using the Plugins SDK.

Hooks can fill a wide variety of the common needs for overriding Liferay core functionality. Whenever possible, hooks should be used in place of Ext plugins, as they are hot-deployable and more forward compatible. Some common scenarios which require the use of a hook are the need to perform custom actions on portal startup or user login, overwrite or extend portal JSPs, modify portal properties, or replace a portal service with your own implementation.

Creating a Hook

Hooks are stored within the *hooks* directory of the plugins directory. Navigate to this directory in terminal and enter the following command to create a new hook (Linux and Mac OS X):

```
./create.sh example "Example"
```

On Windows enter the following instead:

```
create.bat example "Example"
```

You should get a BUILD SUCCESSFUL message from Ant, and there will now be a new folder inside of the `hooks` folder in your Plugins SDK. Notice that the Plugins SDK automatically appends “-hook” to the project name when creating this folder.

Deploying the Hook

Open a terminal window in your *hooks/example-hook* directory and enter this command:

```
ant deploy
```

You should get a BUILD SUCCESSFUL message, which means that your hook is now being deployed. If you switch to the terminal window running Liferay, and wait for a few seconds, you should see the message “Hook for example-hook is available for use.” However, unlike portlets or themes, your new hook doesn't actually do anything yet.

Overriding a JSP

One of the simplest tasks a hook can perform is replacing a portal JSP. In this example we will modify the Terms of Use page. First, create the directory `hooks/example-hook/docroot/META-INF/custom_jsps`. Next, edit `hooks/example-hook/docroot/WEB-INF/liferay-hook.xml`, and add the following between `<hook></hook>`:

```
<custom-jsp-dir>/META-INF/custom_jsps</custom-jsp-dir>
```

Now, any JSP you place inside the *custom_jsps* directory will replace its original inside your Liferay instance when your hook is deployed. The directory structure inside this folder must mirror the one within *liferay-portal-[version]/tomcat-6.0.26/webapps/ROOT*. To override the Terms of Use, copy *liferay-portal-[version]/tomcat-6.0.26/webapps/ROOT/html/portal/terms_of_use.jsp* to `hooks/example-hook/docroot/META-INF/custom_jsps/html/portal/terms_of_use.jsp`. You will have to create all the intervening directories first.

Next, open your copy of the `terms_of_use.jsp` and make a few changes. Deploy your hook and wait until it is deployed successfully. Then, create a new user and try to log in. When you get to the Terms of Use page, you will see your version instead of the default. Please note that this is not the recommended way of changing the Terms of Use, it is simply a convenient example. You can actually replace the Terms of Use with web content by setting two properties in `portal-ext.properties`. A hook is not necessary.

If you look inside the `liferay-portal-[version]/tomcat-6.0.26/webapps/ROOT/html/portal` directory you will see that there are now two terms of use files, one called `terms_of_use.jsp` and another called `terms_of_use.portal.jsp`. `terms_of_use.jsp` is the version from your hook, while `terms_of_use.portal.jsp` is the original. If you now undeploy your hook by deleting its directory in `webapps`, you will see that your replacement JSP is removed and the `.portal.jsp` file is renamed again to take its place. In this manner, you can override any JSP in the Liferay core, while also being able to revert your changes by undeploying your hook. Note however that it is not possible to override the same JSP from multiple hooks, as Liferay will not know which version to use.

Customizing JSPs without overriding the original

The drawback of overridding a JSP is that if the original changes (for example to fix a bug) then you have to also change your customized file in order to benefit from the original change.

If you wish to avoid this drawback and make your JSP modifications even less invasive, it is possible to render the original JSP into a string, and then modify it dynamically afterwards. This makes it possible to change minor elements of a JSP, such as adding a new heading or button, without needing to worry modifying your hook every time you upgrade Liferay. Here is an example that customizes the search portlet to remove the ability to a search provider in the browser:

```
<liferay-util:buffer var="html">
    <liferay-util:include page="/html/portlet/search/search.portal.jsp" />
</liferay-util:buffer>

<%
int x = html.indexOf("<div class=\"add-search-provider\">");
int y = html.indexOf("</div>", x);

if (x != -1) {
    html = StringUtil.remove(html, html.substring(x, y + 6), 
StringPool.BLANK);
}
%>

<%= html %>
```

Since this technique involves String manipulation it is mainly useful when the amount of changes desired are small.

Performing a Custom Action

Another common use of hooks is to perform custom actions on certain common portal events, such as user log in or system startup. The actions that are performed on each of these events are defined in `portal.properties`, which means that in order to create a custom action we will also need to extend this file. Fortunately, this is extremely easy using a hook.

First, create the directory `example-hook/docroot/WEB-INF/src/com/sample/hook`, and create the file `LoginAction.java` inside it with the following content:

```
package com.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");
    }
}
```

Next, create the file `portal.properties` inside `example-hook/docroot/WEB-INF/src` with the following content:

```
login.events.pre=com.sample.hook.LoginAction
```

Finally, edit `liferay-hook.xml` inside `example-hook/docroot/WEB-INF` and add the following line above `<custom-jsp-dir>`:

```
<portal-properties>portal.properties</portal-properties>
```

Deploy your hook again and wait for it to complete. Then log out and back in, and you should see our custom message in the terminal window running Liferay.

There are several other events that you can define custom actions for using hooks. Some of these actions must extend from `com.liferay.portal.kernel.events.Action`, while others must extend `com.liferay.portal.struts.SimpleAction`. For more information on these events, see the `portal.properties` configuration file for your version of Liferay in: http://www.liferay.com/community/wiki/-/wiki/Main/Portal+Properties

Extending and Overriding *portal.properties*

In our hook, we modified the `login.events.pre` portal property. Since this property accepts a list of values, our value was appended to the existing values. It is safe to modify these portal properties from multiple hooks, and they will not interfere with one another. Some portal properties only accept a single value, such as the `terms.of.use.required` property, which can be either **true** or **false**. You should only modify these properties from one hook, otherwise Liferay will not know which value to use. You can determine which type a particular property is by looking in `portal.properties`.

Not all portal properties can be overridden in a hook. A complete list of the available properties can be found in the DTD for `liferay-hook.xml` in the `definitions` folder of the Liferay source code. In addition to defining custom actions, hooks can also override portal properties to define model listeners, validators, generators, and content sanitizers.

Overriding a Portal Service

All of the functionality provided by Liferay is encapsulated behind a layer of services that is accessed from the frontend layer (the portlets).

One of the benefits of this architecture is that it is possible to change how a core portlet of Liferay behaves without changing the portlet itself, customizing the backend services that it uses. This section explains how to do that from a hook plugin.

Liferay automatically generates dummy wrapper classes for all of its services, for example `UserLocalServiceWrapper` is created as a wrapper of the `UserLocalService` that is used to add, remove and retrieve user accounts. To modify the functionality of `UserLocalService` from our hook, all we have to do is create a class that extends from `UserLocalServiceWrapper`, override some of its methods, and then instruct Liferay to use our class instead of the original.

First, inside `example-hook/docroot/WEB-INF/src/com/sample/hook` create a new file called `MyUserLocalServiceImpl.java` with the following content:

```
package com.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);
    }
}
```

Tip: Note that the wrapper class (MyUserLocalServiceImpl in this example) will be loaded in the hook's class loader. That means that it will have access to any other class included within the same WAR file, but it won't have access to internal classes of Liferay.

Next, edit `liferay-hook.xml` inside `example-hook/docroot/WEB-INF` and add the following after `</custom-jsp-dir>`:

```
<service>
    <service-type>com.liferay.portal.service.UserLocalService</service-type>
    <service-impl>com.sample.hook.MyUserLocalServiceImpl</service-impl>
</service>
```

Redeploy your hook, then refresh your browser. In the terminal window containing Liferay you should see the messages printed by our hook.

Here are some other services of Liferay that you may need to extend to meet advanced requirements:

- OrganizationLocalService: add, delete and retrieve organizations. Also assign users to organizations and retrieve the list of organizations of a given user.
- GroupLocalService: add, delete and retrieve communities.
- LayoutLocalService: add, delete, retrieve and manage pages of communities, organizations and users.

For a complete list of the services available and the methods of each of them check the javadocs distributed with your version of Liferay.

Overriding a *Language.properties* File

In addition to the three capabilities of hooks already discussed, it is also possible to override `Language.properties` files from a hook, allowing you to change any of the messages displayed by Liferay to suit your needs. The process is extremely similar to any of the ones we have just described. All you need to do is to create a Language file for the language whose messages you want to customize and then refer to it from the `liferay-hook.xml`. For example to override the translations to Spanish and French the following two lines would be added to the file:

```
<hook>
    ...
   <language-properties>content/Language_es.properties</language-properties>
   <language-properties>content/Language_fr.properties</language-properties>
    ...
</hook>
```

6. EXT PLUGINS

Ext plugins provide the most powerful method of extending Liferay. This comes with some tradeoffs in complexity, and so Ext plugins are designed to be used only in special scenarios in which all other plugin types cannot meet the needs of the project.

Before deciding to use an Ext plugin it's important to understand the costs of using such a powerful tool. The main one is the cost in terms of maintenance. Because Ext plugins allow using internal APIs or even overwriting files provided in the Liferay core, it's necessary to review all the changes done when updating to a new version of Liferay (even if it's a maintenance version or a service pack). Also, unlike the other types of plugins, Ext plugins require the server to be restarted after deployment, as well as requiring additional steps when deploying or redeploying to production systems.

The main use cases in which an Ext plugin may be needed are:

- Customizing `portal.properties` that are not supported by Hook Plugins
- Customizing Struts Actions
- Providing custom implementations for any of the Liferay beans declared in Liferay's Spring files (use service wrappers from a hook instead if possible)
- Adding JSPs that are referenced from portal properties that can only be changed from an ext plugin (be sure to check if the property can be modified from a hook plugin before doing this)
- Direct overwriting of a class (not recommended unless it's strictly necessary)

Creating an Ext plugin

Ext plugins are stored within the `ext` directory of the Plugins SDK. Navigate to this directory in a terminal and enter the following command to create a new Ext plugin (Linux and Mac OS X):

```
./create.sh example "Example"
```

On Windows enter the following instead:

```
create.bat example "Example"
```

You should get a BUILD SUCCESSFUL message from Ant, and there will now be a new folder inside of the `ext` folder in your Plugins SDK. Notice that the Plugins SDK automatically appends "-ext" to the project name when creating this folder.

Once the target has been executed successfully you will find a new folder called example-ext with the following structure:

```
/ext-example/
    /docroot/
        /WEB-INF/
            /sql/
            /ext-impl/
                /src/
            /ext-lib/
               /global/
               /portal/
            /ext-service/
               /src/
            /ext-util-bridges/
               /src/
            /ext-util-java/
               /src/
            /ext-util-taglib/
               /src/
            /ext-web/
```

The most significant directories in this structure are the ones inside the `docroot/WEB-INF` directory. In particular you should be familiar with the following directories:

- **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:** Place here any libraries that should be copied to the global classloader of the application server upon deployment of the ext plugin.

- **ext-lib/portal:** Place here any libraries that should be copied inside Liferay's main application. Usually these libraries are needed because they are invoked from the classes added within `ext-impl/src`.
- **ext-service/src:** Place here any classes that should be available to other plugins. When using Service Builder, it will put the interfaces of each service here. Also in advanced scenarios, this directory will contain classes that overwrite the classes of `portal-service.jar`.
- **ext-web/docroot:** Contains configuration files for the web application, including `WEB-INF/struts-config-ext.xml` which will allow customizing Liferay's own core struts actions. You can also place any JSPs needed by your customizations here.
- **Other:** `ext-util-bridges`, `ext-util-java` and `ext-util-taglib` are only needed in advanced scenarios in which you need to customize the classes of three libraries provided with Liferay: `util-bridges.jar`, `util-java.jar` and `util-taglib.jar` respectively. In most scenarios you can just ignore these directories.

By default, several files are added to the plugin. Here are the most significant ones:

- Inside `docroot/WEB-INF/ext-impl/src`:
  - **`portal-ext.properties`**: this file can be used to overwrite any configuration property of Liferay, even those that cannot be overridden by a hook plugin (which is always preferred when possible). Note that if this file is included it will be read instead of any other `portal-ext.properties` in the application server. Because of that you may need to copy into it the properties related to the database connection, file system patches, etc.
- Inside `docroot/WEB-INF/ext-web/docroot/WEB-INF`:
  - **`portlet-ext.xml`**: Can be used to overwrite the definition of a Liferay portlet. In order to do this, copy the complete definition of the desired portlet from `portlet-custom.xml` within Liferay's source code and then apply the necessary changes.
  - **`liferay-portlet-ext.xml`**: Similar to the file above, but for the additional definition elements that are specific to Liferay. In order to override it, copy the complete definition of the desired portlet from `liferay-portlet.xml` within Liferay's source code and then apply the necessary changes.

- `struts-config-ext.xml` **and** `tiles-defs-ext.xml`: Can be used to customize the struts actions used by Liferay's core portlets.

Tip: after creating an Ext plugin, remove all of the files added by default that are not necessary for the extension. This is important because Liferay keeps track of the files deployed by each Ext plugin and it won't allow deploying two Ext plugins if they override the same file to avoid collisions. By removing any files not really necessary from an ext plugin it will be easier to use along with other Ext plugins.

Developing an Ext plugin

Developing an Ext plugin is slightly different than working with other plugin types. The main reason for the difference is that an Ext plugin when deployed will make changes to the Liferay web application itself, instead of staying as a separate component that can be removed at any time. It's important to remember that *once an Ext plugin has been deployed, some of its files are copied inside the Liferay installation, so the only way to remove its changes is to redeploy an unmodified Liferay application again.*

The Plugins SDK contains several Ant targets that help with the task of deploying and redeploying during the development phase. In order to do this it requires a `.zip` file of a Tomcat 6 based Liferay bundle. The Ant targets will unzip and clean up this installation whenever needed to guarantee that any change done to the Ext plugin during development is properly applied and previous changes that have been removed are not left behind. This is part of the added complexity when using Ext plugins, and so it is recommended to use another plugin type to accomplish your goals if it is at all possible.

Set up

Before attempting to deploy an Ext plugin, it's necessary to edit the file `build.{username}.properties` in the root folder of the Plugins SDK. If this file doesn't exist yet you should create it. Substitute `{username}` with the your user ID on your computer. Once the file is open, add the following three properties to the file, making sure the individual paths point to the right locations on your system:

```
app.server.dir={...}/liferay-portal-6.0.6/tomcat-6.0.26
app.server.zip.name={...}/liferay-portal-tomcat-6.0.6.zip
ext.work.dir={...}/work
```

`app.server.zip.name` should point to a `.zip` with a bundle of Liferay. The directory denoted by the property `ext.work.dir` will be used to unzip the bundle as well as remove it and unzip again as needed. `app.server.dir` should point to the Tomcat directory inside the work

directory.

For example, if `ext.work.dir` points to `C:\ext-work`, and `app.server.zip.name` points to `C:\files\liferay-portal-tomcat-6.0-${lp.version}.zip`, then `app.server.dir` should point to `C:\ext-work\liferay-portal-${lp.version}\tomcat-6.0.18`.

Initial deployment

Once the environment is set up, we are ready to start customizing. We'll show the full process with a simple example, customizing the sections of a user profile. Liferay allows doing that through the `portal-ext.properties` configuration file, but we'll be changing a property that cannot be changed from a hook plugin. In order to make this change, open the `docroot/WEB-INF/ext-impl/src/portal-ext.properties` file and paste the following contents inside:

```
users.form.update.main=details,password,organizations,communities,roles
```

This line removes the sections for user groups, pages and categorizations. We might want to make this change because we don't want them in our portal.

Once we've made this change, we are ready to deploy. Open a terminal window in your `ext/example-ext` directory and enter this command:

```
ant deploy
```

You should get a BUILD SUCCESSFUL message, which means that your plugin is now being deployed. If you switch to the terminal window running Liferay and wait for a few seconds, you should see the message “Extension environment for example-ext has been applied. You must restart the server and redeploy all other plugins.” Redeploying all other plugins is not strictly mandatory, but you should do it if some changes applied through the Ext plugin may affect the deployment process itself.

The `ant deploy` target builds a `.war` file with all the changes you have made 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 content to the appropriate destinations within the deployed and running Liferay inside your application server. You must now restart your application server.

Once the server has started, log in as an administrator and go to *Control Panel -> Users*. Edit an existing user and verify that the right navigation menu only shows the five sections that were referenced from the `users.form.update.main` property.

Once we've applied this simple modification to Liferay, we can go ahead with a slightly more complex customization. This will give us an opportunity to learn the proper way to redeploy an Ext plugin, which is

different from the initial deployment.

For this example we'll customize the *details* view of the user profile. We could do that just by overwriting its JSP, but this time we'll use a more powerful method which also allows adding new sections or even merging the existing ones. Liferay allows referring to custom sections from the `portal-ext.properties` and implementing them just by creating a JSP. In our case we'll modify the property `users.form.update.main` once again to set the following value:

```
users.form.update.main=basic,password,organizations,communities,roles
```

That is, we removed the section *details* and added a new custom one called *basic*. When Liferay's user administration reads this property it looks for the implementation of each section based on the following conventions:

- The section should be implemented in a JSP inside the directory: html/portlet/enterprise_admin/user
- The name of the JSP should be like the name of the section plus the `.jsp` extension. There is one exception. If the section name has a dash sign (“-”), it will be converted to an underscore sign (“_”). For example, if the section is called *my-info*, the JSP should be named `my_info.jsp`. This is done to comply to common standards of JSP naming.
- The name of the section that will be shown to the user will be looked for in the language bundles. When using a key/value that is not already among the ones included with Liferay, you should add it to the `Language-ext.properties` and each of the language variants for which we want to provide a translation. Within the Ext plugin these files should be placed within `ext-impl/src`.

In our example, we'll need to create a file within the Ext plugin in the following path:

```
ext-web/docroot/html/portlet/enterprise_admin/user/basic.jsp
```

For the contents of the file, you can write them from scratch or make a copy of the `details.jsp` file from Liferay's source code and modify from there. In this case we've decided to do the latter and then remove some fields to simplify the creation of a user. The result is this:

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

<%
User selUser = (User)request.getAttribute("user.selUser");
%>

<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:error exception="<%= DuplicateUserScreenNameException.class
%>"
                       message="the-screen-name-you-requested-is-already-
taken" />
    <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" />

    <aui:input name="screenName" />

    <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" />

    <aui:input name="emailAddress" />

    <liferay-ui:error exception="<%= ContactFirstNameException.class %>"
                       message="please-enter-a-valid-first-name" />
    <liferay-ui:error exception="<%= ContactFullNameException.class %>" m
                       essage="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" />
</aui:fieldset>
```

In our case, we don't need to add a new key to `Language-ext.properties`, because “basic” is already included in Liferay's language bundle. We are ready to redeploy.

Redeployment

So far, the process has been very similar to that of other plugin

types. The differences start when redeploying an Ext plugin that has already been deployed. As mentioned earlier, when the plugin was first deployed *some of its files were copied within the Liferay installation.* After making any change to the plugin the recommended steps to redeploy are first to stop the application server, and then to execute the following ant targets:

```
ant clean-app-server direct-deploy
```

These ant targets first remove the work bundle (unzipping the one that was referred to through `build.{username}.properties`), and then deploy all the changes directly to the appropriate directories. The `direct-deploy` target is faster because the changes are applied directly., while the Liferay server does it on start up if you use the `deploy` target. For that reason it is usually preferred during development.

You can deploy several Ext plugins to the same server, but you will have to redeploy each of them after executing the `clean-app-server` target.

Once you have finished the development of the plugin you can execute the following ant target to generate a `.war` file for distribution:

```
ant war
```

The file will be available within the `dist` directory in the root of the plugins SDK.

Advanced customization techniques

This section covers additional customization techniques that are possible through an Ext plugin. As mentioned above, you can change almost everything within Liferay when using the Ext plugin, therefore be careful when using such a powerful tool.

Always keep in mind that with ever new Liferay version, implementation classes may have changed. Thus if you've changed Liferay source code directly, you may have to merge your changes into the newer Liferay version. General approach for minimizing conflicts is – don't change anything, only extend.

The alternative is to extend the class you want to change and override methods needed. Then use some of Liferay's configuration files to reference your subclass as a replacement of the original class.

This and other advanced techniques are described in detail in the following sections.

Advanced configuration files

Liferay uses several internal configuration files for easier maintenance and also to configure the libraries and frameworks it depends on, such as Struts or Spring. For advanced customization

needs it may be useful to override the configuration specified in these files, so Liferay provides a clean way to do that from an Ext plugin without modifying the original files.

Next is a list of all of these files, along with a description and a reference to the original file in the path where they can be found in the source code of Liferay (you may need to look at them for reference):

- ext-impl/src/META-INF/ext-model-hints.xml
  - Description: This file allows overwriting the default properties of the fields of the data models used by Liferay's core portlets. These properties determine how the form to create or edit each model is rendered.
  - Original file in Liferay: portal-impl/src/META-INF/portal-model-hints.xml
- ext-impl/src/META-INF/ext-spring.xml
  - Description: This file allows overwriting the Spring configuration used by Liferay and any of its core portlets. The most common usage is to configure specific datasources or to 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: This file allows overwriting 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: This file allows overwriting the log4j configuration. The most common usage is to increase or decrease the log level of a given package or class to obtain more information or hide unneeded information from the logs respectively.
  - 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: This file allows overwriting 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: This file allows overwriting the declaration of the core portlets included in Liferay. The most common usage is 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: This file allows overwriting the Liferay-specific declaration of the core portlets included in Liferay. Refer to the liferay-portlet-app_6_0_0.dtd file for details on all the available options. Use this file with care since 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: This file allows overwriting the portlets that will be shown in the “Add application” pop-up and the categories in which they'll be organized. The most common usage is to change the categorization, hide some portlets or make some 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: This file allows specifying custom template files for each of the layout templates provided by default with Liferay. You should not need to do this except for very advanced needs.
    - 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: This file allows changing the properties of the default themes provided by default with Liferay. You should not need to do this except for very advanced needs.

- Original file in Liferay: portal-web/docroot/WEB-INF/liferay-look-and-feel.xml

Changing the API of a core service

One advanced customization need that appears in some scenarios is to change the API of a method provided by one of Liferay's services, for example the UserLocalService.

Is it possible to do that? The short answer is no. The long answer is that you can but it will require modifying Liferay's source code directly and make manual changes to quite a few files. The good news is that there is a better alternative to achieve the end goal.

The best way to extend an existing service is to create a complementary custom service, for example 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.

Note that this technique does not require an Ext plugin since it can be done from portlet plugins. In fact, using service-builder for Ext plugin is deprecated but is supported for migration from the old extension environment.

In some advanced circumstances it's desired to change the implementation of the original service to call your custom one, which can only be done from an Ext plugin. To achieve this, override spring definition for UserLocalServiceUtil in ext-spring.xml and point it to your **My**UserLocalServiceImpl (instead of UserLocalServiceImpl). This way both MyUserLocalServiceUtil and UserLocalServiceUtil will use the same spring bean: your new implementation.

Replacing core classes in portal-impl

If you really need to change core portal-impl class and this class that cannot be replaced in any configuration file, then best way to avoid conflicts and easily merge with a new portal version is to:

1. Rename original class (e.g. DeployUtil → MyDeployUtil)
2. Create new subclass with old name (e.g DeployUtil extends MyDeployUtil)
3. Override methods you need to change
4. Delegate static methods
5. Use logger with appropriate class name for both classes (e.g. DeployUtil)

This strategy will help you determine what you will need to merge

in the future when a new version of Liferay is released.

Tip: This is a very advanced technique that may have a high impact on the maintainability of your code, especially if abused. Try to look for alternatives and if you confirm that this is your only alternative try to apply only for the short term and get in touch with Liferay's developers to apply the necessary changes to the product source code.

Licencing 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 either publicly or to an specific customer, you need to make sure that those modifications are compliant with this license. A common way to do this is to make the source code of your modifications available to the community under the same license, but make sure to read the license text yourself to find the best option that fits your needs.

If the goal of the changes was to fix a bug or to make an improvement that could be of interest to a broader audience, consider contributing it back to the project. That would benefit all other users of the product and also would be good for you since you won't have to maintain the changes when new versions of Liferay come out. 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

Deploying in production

In production or pre-production environments it's often not possible to use Ant to deploy web applications. Also, some application servers such as WebSphere or Weblogic have their own deployment tools and it isn't possible to use Liferay's autodeploy process. This section describes two methods for deploying and redeploying Ext plugins in production that can be used in each of these scenarios.

Method 1: Redeploying Liferay's web application

This method can be used in any application server that supports auto deploy, such as Tomcat or JBoss. Its main benefit is that the only artifact that needs to be transferred to the production system is the `.war` file which the Ext plugin produced using the `ant war` target, which is usually a small file. Here are the steps that need to be executed on the

server:

1. Redeploy Liferay. To do this, follow the same steps you used when first deploying Liferay on the app server. If you are using a bundle, you can just unzip the bundle again. If you've installed Liferay manually on an existing application server, you'll need to redeploy the `.war` file and copy the global libraries to the appropriate directory within the application server. If this is the first time the Ext plugin is deployed, you can skip this step.
2. Copy the Ext plugin `.war` into the auto deploy directory. For a bundled Liferay distribution, the *deploy* folder is in the *root* folder.
3. Once the Ext plugin is detected and deployed by Liferay, restart the Liferay server.

Method 2: Generate an aggregated WAR file

This method can be used for application servers that do not support autodeploy, such as WebSphere or Weblogic. Its main benefit is that all Ext plugins are merged before deployment to production, so a single `.war` file will contain Liferay plus the changes from one or more Ext plugins. Before deploying the `.war` file, you'll need to copy the dependency `.jar`s for both Liferay and the Ext plugin to the global application server class loader in the production server. This location varies from server to server; please see the *Liferay Portal Administrator's Guide* for further details for your application server.

To create the aggregated `.war` file, deploy the Ext plugin first to the Liferay bundle you are using in your development environment (using for example, Tomcat). Once it's deployed, restart the server so that the plugin is fully deploy and shut it down again. Now the aggregated file is ready. Create a `.war` file by zipping the `webapps/ROOT` folder of Tomcat. Also, copy all the libraries from the `lib/ext` directory of Tomcat that are associated to all the Ext plugins to your application server's global classpath, as noted above. These steps will be automated with Ant targets in the next version of Liferay, but for now, they need to be done manually.

Once you have the aggregated `.war` file follow these steps on the 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.

Migrating old extension environments

Ext plugins have been created as an evolution of the extension environment provided in Liferay 5.2 and previous versions of Liferay. Because of this a common need for projects upgrading from previous versions might be to migrate Extension environments into Ext plugins. The good news is that this task is automated and thus relatively easy.

Tip: When migrating an extension environment, it's worth considering if all or at least some of its features can be moved into other types of plugins such as portlets and hooks. The benefit of using portlets and hooks is that since they are focused on specific goals they are easier to learn. Additionally they are cheaper to maintain since they often require fewer changes when upgrading to a new version of Liferay.

The process of migrating consists of executing a target within the ext directory from Plugins SDK, pointing to the old extension environment and naming the new plugin:

```
ant upgrade-ext -Dext.dir=/projects/liferay/ext -Dext.name=my-ext
-Dext.display.name="My Ext"
```

Here is a description of the three parameters used:

- `ext.dir` is a command line argument to the location of the old Extension Environment.
- `ext.name` is the name of the Ext plugin that you want to create
- `ext.display.name` is 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 (read the section “Creating an Ext plugin” for an explanation of the main directories within the plugin).

When the migration process is complete, some additional tasks will be needed to upgrade the code to the new version of Liferay. Some of the most typical tasks are:

- Review the uses of Liferay's APIs and adapt them accordingly.
- Review the changes to the JSPs and merge your changes into the JSPs of the new Liferay version.
- When using Service Builder you will need to run `ant build-service` again. It's also recommended to consider moving this code to a portlet plugin, because it is now as powerful and allows for greater modularity and maintainability.
- If you've implemented portlets in Ext, migrate them to portlet plugins, as this capability is deprecated and is not guaranteed to be available in future releases.

Conclusions

Ext plugins are a very powerful way of extending Liferay. There are no limits in what can be customized using them and for that reason they have to

be used carefully. If you find yourself using an Ext plugin, verify if all or part of the desired functionality can be implemented through portlets, hooks or web plugins instead.

If you really need to use an Ext plugin make it as small as possible and make sure you follow the instructions in this guide carefully to avoid issues.

7. LIFERAY TOOLS

Liferay's developers use a variety of tools to develop the product and as a consequence of that they have always tried hard to allow other developers to use any tools they wanted for their own development. Because of this you can develop portals based on Liferay with complex IDEs Eclipse, Netbeans or IntelliJ Idea or just use text editors such as Notepad. You can write your persistence layer directly using SQL and JDBC or you can use advanced object-relational mapping libraries such as hibernate or iBatis.

But while being agnostic is great, specially for more experienced developers who can reuse their existing knowledge, it can be overwhelming for newcomers. For that reason Liferay also offers specific tools that can be used to ease the learning curve when developing portlets with Liferay. Two of the most significant of these tools are Liferay IDE, a fully featured Integrated Development Environment based on Eclipse, and Service Builder, a code generator that encapsulates the complexity of Hibernate and Spring to get you started in minutes. Let's learn more about each of them.

Liferay IDE

Liferay IDE is an extension for the Eclipse platform that supports development of plugin projects for the Liferay Portal platform. It is available as a set of Eclipse plugins installable from an update-site. The latest version supports developing 5 Liferay plugin types: portlets, hooks, layout templates, themes, and ext plugins. Liferay IDE requires the Eclipse Java EE developer package using either Galileo or Helios versions.

The first two sections below show how to install and set-up Liferay

IDE within your environment. If you are using a copy of Liferay Developer Studio, which comes with Liferay Portal Enterprise Edition, you can skip directly to the section titled "Testing the Liferay portal server" since it comes already preconfigured.

Installation

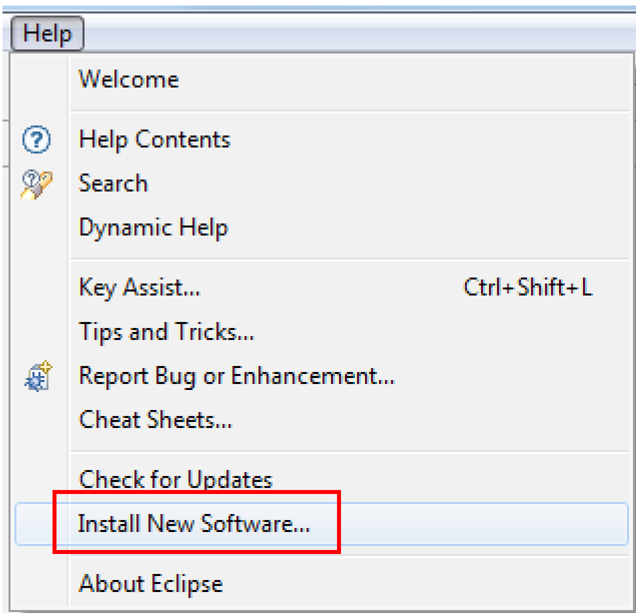
This section is a detailed guide to install Liferay IDE.

Requirements

- Java 5.0 JRE
- Eclipse Helios (3.6.0) IDE for Java EE Developers OR Eclipse Galileo SR2 - IDE for Java EE Developers

Installation steps

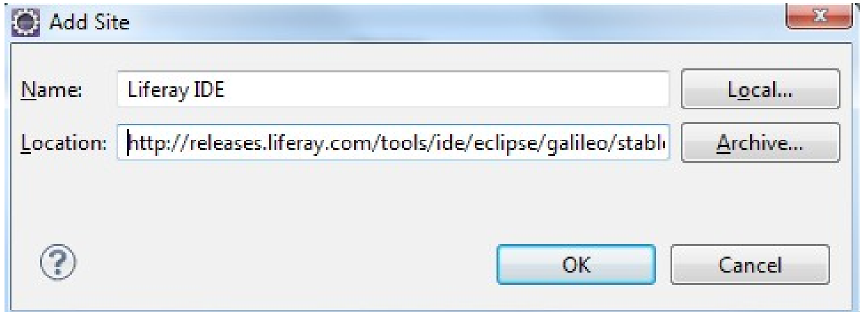
1. Install Eclipse Helios or Galileo (unzip download file from above)
2. Run eclipse.exe
3. When eclipse opens, go to Help > Install New Software...



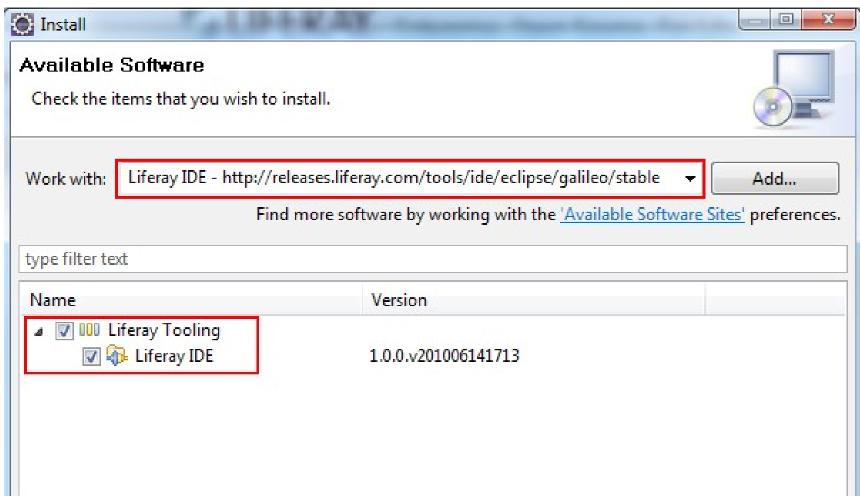
4. Click the "Add..." button to open Add Site dialog
5. Type in "Liferay IDE" for name and as the location use one of the

following URLs:

- **For Eclipse Helios**: http://releases.liferay.com/tools/ide/eclipse/helios/stable/
- **For Eclipse Galileo:** http://releases.liferay.com/tools/ide/eclipse/galileo/stable/

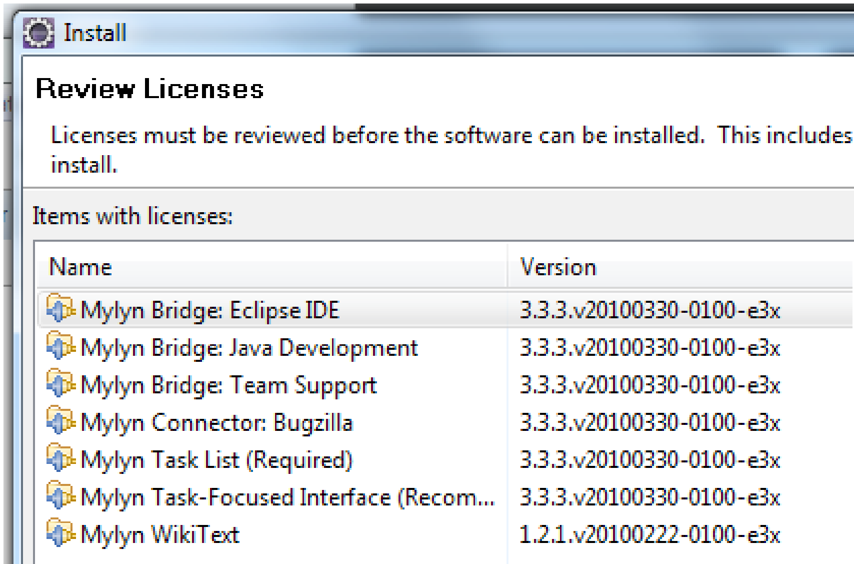


6. Select OK
7. Select the Liferay IDE site from the combo selection box.
8. When the table refreshes you should see Liferay Tooling category and one entry for Liferay IDE feature, select the checkbox to install the feature.



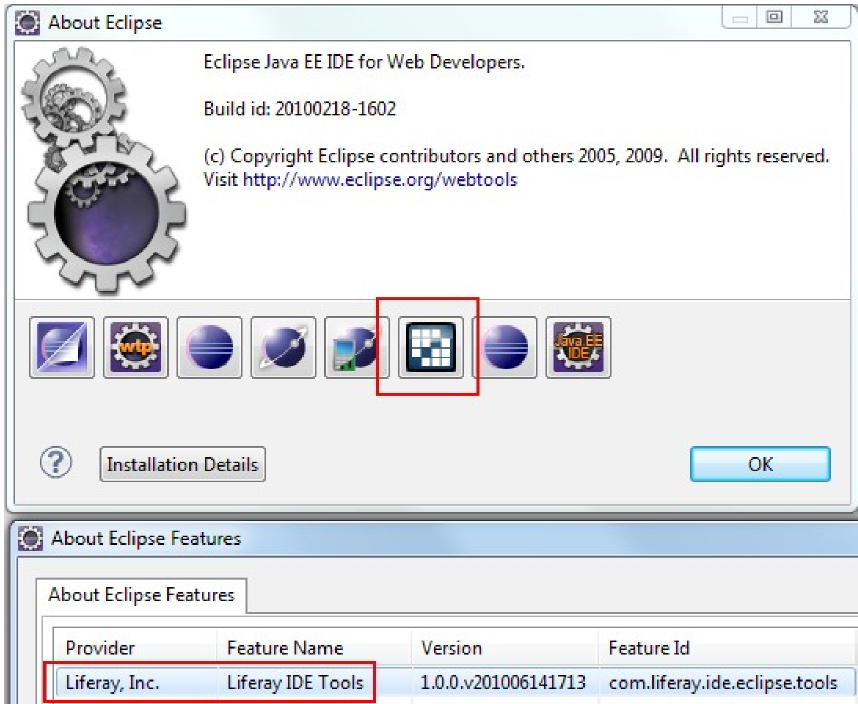
9. Click Next and then click Finish to begin the install
10. (Note) If you are using Galileo and before you click Finish you will see several Mylyn features that need to be installed as well, that is

expected, as there has been a Mylyn update since the Galileo SR2 release so Mylyn will be updated along side of the Liferay IDE installation.



11. After plugins download and install you will have to accept that the content is unsigned and then restart eclipse.

12. After you restart, go to Help > About Eclipse and you should see a Icon badge for Liferay IDE that shows you have it properly installed.



Alternative installation

5a. Instead of typing in a URL, you can download the the archived update site from this link liferay-ide-eclipse-updatesite-1.0.1.zip

5b. In Add Site dialog, click the "Archive" button and browse to the location of the downloaded zip file.

5c. Installation proceeds normally

Set up

This section describes the setup necessary to begin doing Liferay development and testing your developments. In order to do that the instructions below will ask you to download and install Liferay Portal 6.0 and the Liferay Plugins SDK 6.0, if you haven't done it yet.

Note: earlier versions of Liferay, such as 5.2 and earlier are not supported by the Liferay IDE.

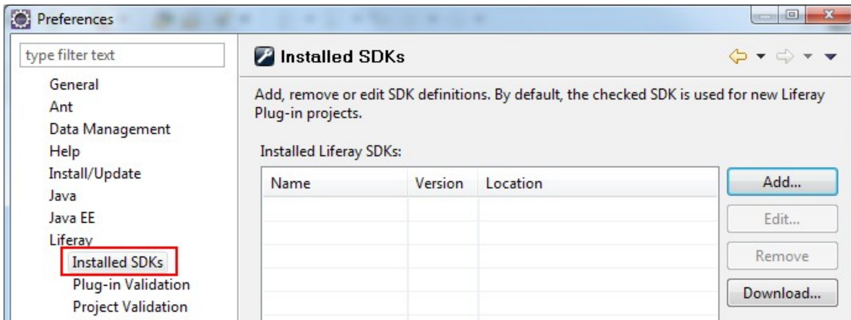
Liferay Plugins SDK Setup

Before you can begin creating new Liferay plugin projects, a Liferay

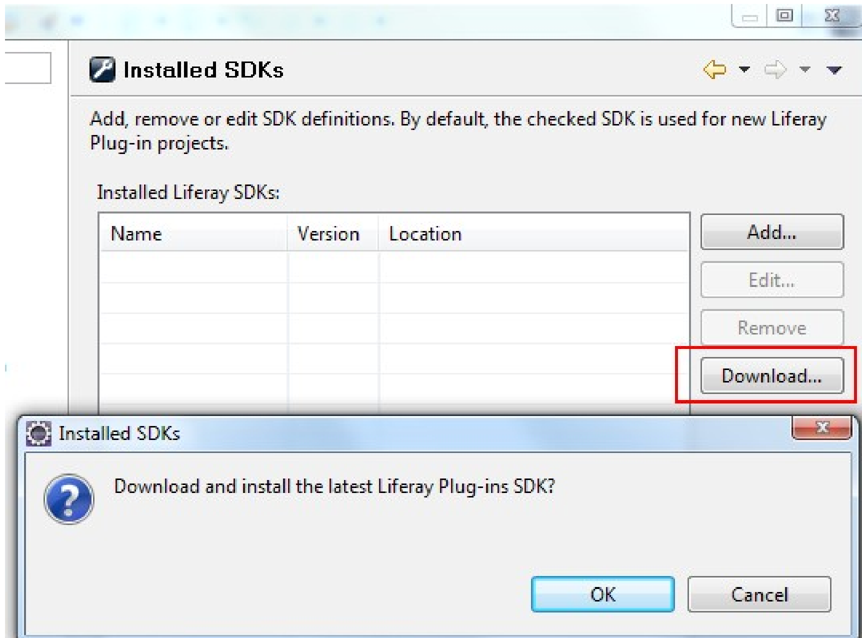
Plugins SDK and Liferay Portal must be installed and configured in the IDE.

1. Open Eclipse with Liferay IDE installed.

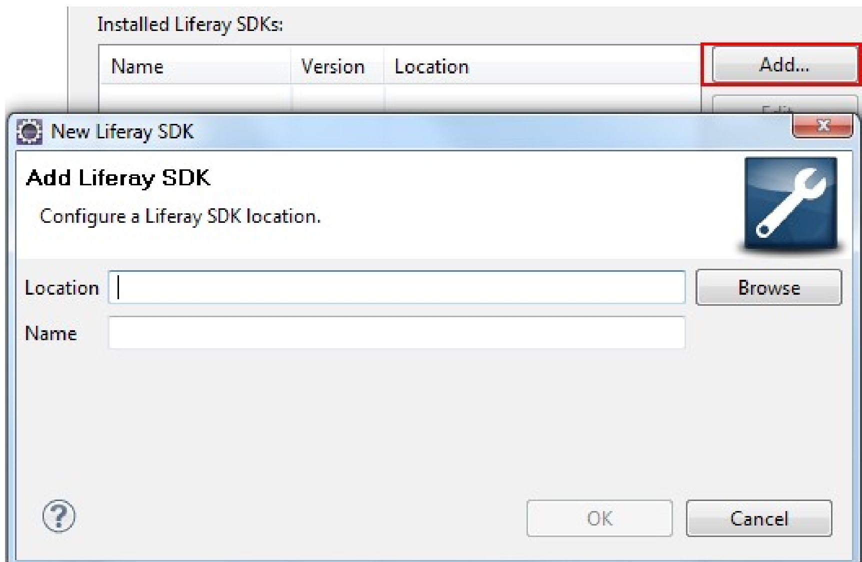
2. Open Preference page for Liferay > Installed SDKs (Go to Window > Preferences > Liferay > Installed SDKs)



3. If you have not already downloaded the Liferay Plugins SDK for your portal version you can download it using the "Download..." button on the Installed SDK page and it will automatically download and install the latest Liferay Plugins SDK for you.
Alternatively you can download manually any specific version yourself from the sourceforge.net Liferay downloads page. Look for liferay-plugins-sdk-[version].zip where [version] is the desired version.



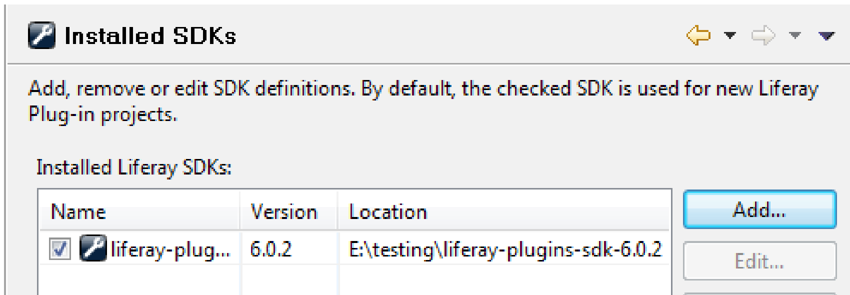
4. If you decided to download it manually, you will need to add your SDK using the Add button which brings up the Add SDK Dialog. Otherwise skip to step 7.



5. Browse to the location of your Plugins SDK installation.

6. The default name is the name of the directory but you can change it if you wish.

7. Select OK and you should see your SDK in the list of Installed SDKs.

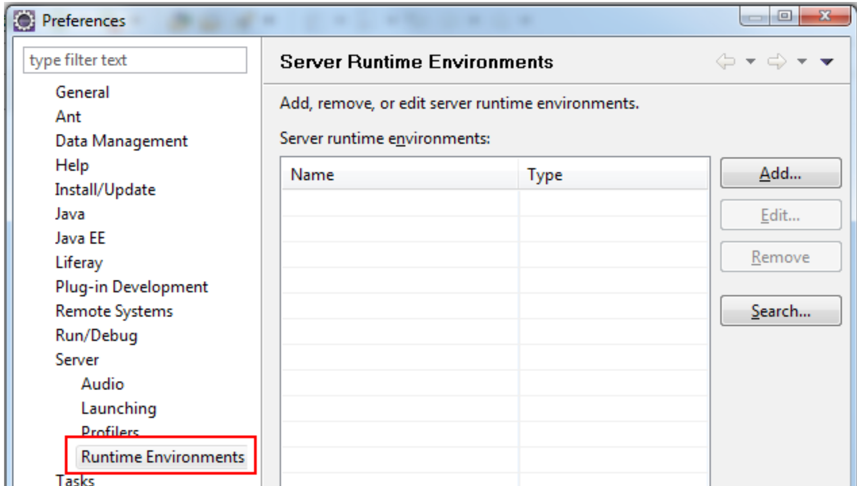


Note that multiple SDKs can be added to the preferences but you will need to select at least one SDK to be the default which is represented by the SDK that has the checkbox selected.

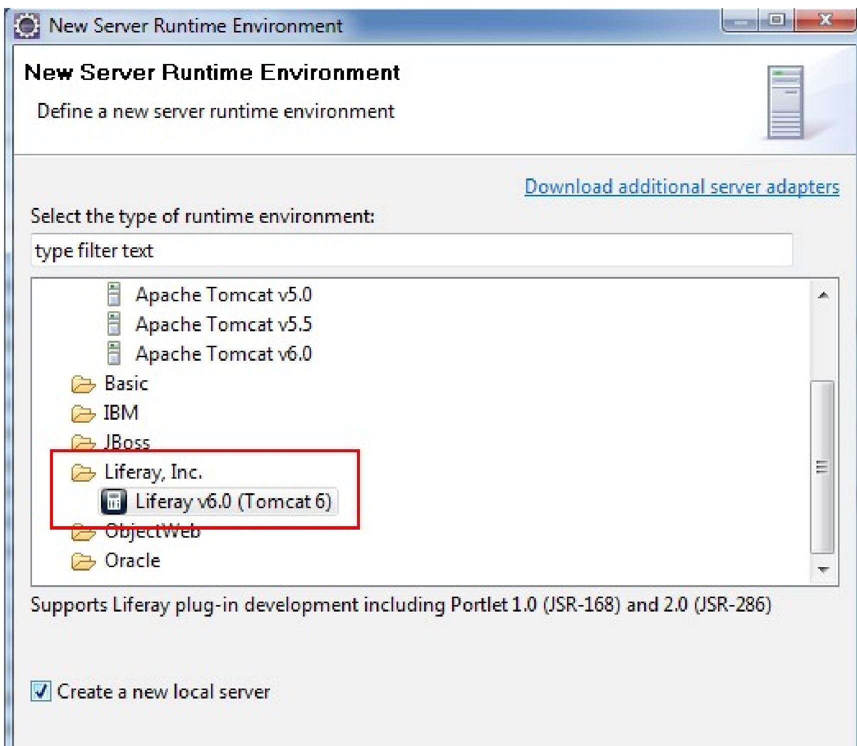
> Note: There is a known issue with version 6.0.4. It is recommended that you upgrade to version 6.0.6 or the latest maintenance version of Liferay 6.0 to avoid it. If that is impossible in your case, you will need to change your build.properties file. The SDK plugin assumes you have installed the SDK at the same level (same directory) as the portal directory. In the line
>
> app.server.dir=${project.dir}/../bundles/app_server_name
>
> of build.properties you have to change bundles with the name of the Liferay portal installation directory e.g. liferay-portal-6.0.4.
>
> Otherwise, portlets created with the SDK plugin will not be deployed by Ant.

Liferay Portal Tomcat Runtime / Server Setup

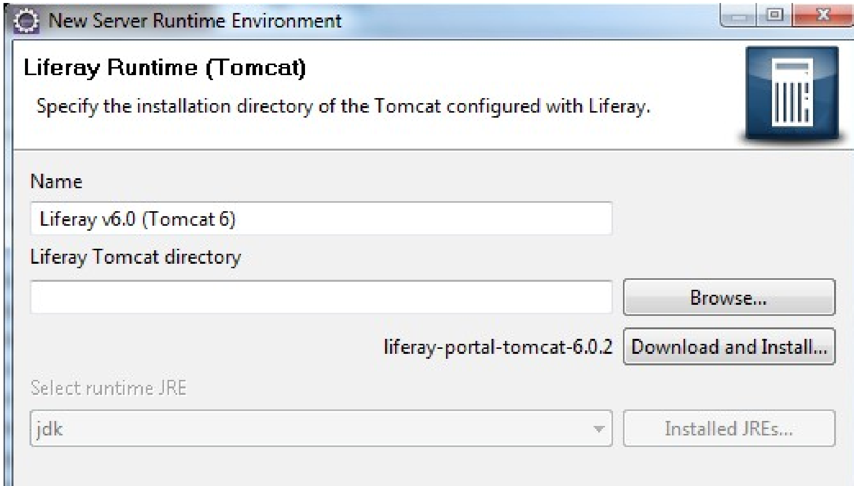
1. In eclipse open the Runtime environments preference page (Go to Window > Preferences > Server > Runtime environments)



2. Click Add to add a new Liferay runtime and find “Liferay v6.0 (Tomcat 6)” under the Liferay, Inc. category and click Next.



3. If you have not already downloaded and installed a copy of the Liferay Portal Tomcat bundle then you can download the latest Liferay Portal Tomcat bundle by clicking the "Download and Install..." button within the wizard.



4. If you used the download option you can skip this step, if not, click Browse and select the location of the liferay-portal-[version] directory.

Importing Existing Projects into Liferay IDE

If you have been following the examples of this guide using the Plugins SDK or have previous plugins developed with it that you want to keep developing with Liferay IDE then this section is for you. It also shows other options to import. Specifically it shows how to import from:

1. Existing Liferay projects that are not in Eclipse workspace
2. Projects already in Eclipse but not Liferay IDE (don't have Liferay facet or target runtime)
3. Existing Liferay IDE projects from another version of Liferay IDE or workspace

The following subsections describe the steps for each of them with more detail.

Importing existing Liferay Project from a Plugins SDK

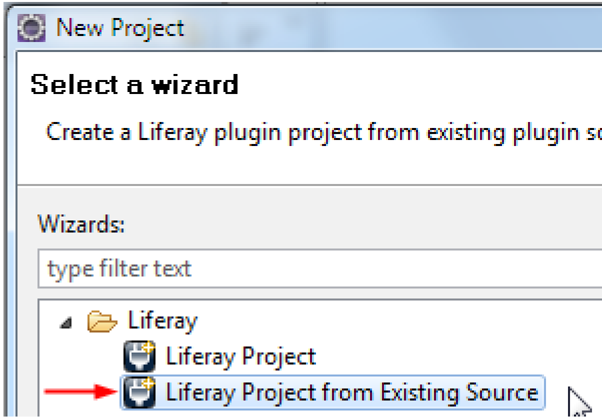
An existing Liferay project that has been created by the Plugins SDK but has not yet been added to an Eclipse workspace or have not been added to the current workspace open in Eclipse/Liferay IDE. These projects may or may not have .project or .classpath files. Whether they do or don't we will follow the same steps and use the same wizard.

There are two options to create projects from existing sources, depending on whether you want to create one single project or multiple projects from the same SDK. Let's see both in detail.

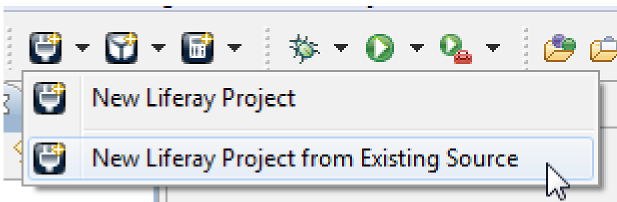
Create one single project from one plugin in an existing sources

This option will create one project for a single plugin that already exists inside a Plugins SDK.

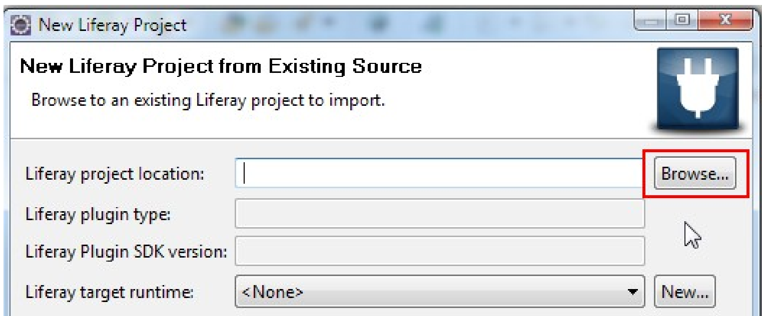
1. In Eclipse, go to File > New > Project... > Liferay > Liferay Project from Existing Source



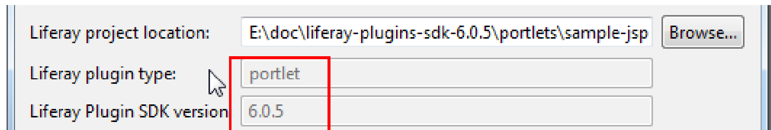
Or you can invoke the same wizard from the Liferay shortcut toolbar.



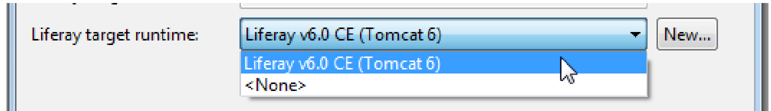
2. Browse to the location of the project folder. Note: the project folder should be a sub-directory of one of the plugin types, e.g. portlets, hooks, themes, etc. If not it will not be able to be imported correctly.



3. Once you select the plugin project folder you should see the plugin type and SDK version values get updated to correct values. If the SDK is not recent enough or project type is not correct it will be marked with an error.



4. Next you will need to select a Liferay Runtime to configure on the project once it is imported. If you don't have a Liferay Runtime, use the New... button to create a new Liferay portal runtime (tomcat bundle only supported).

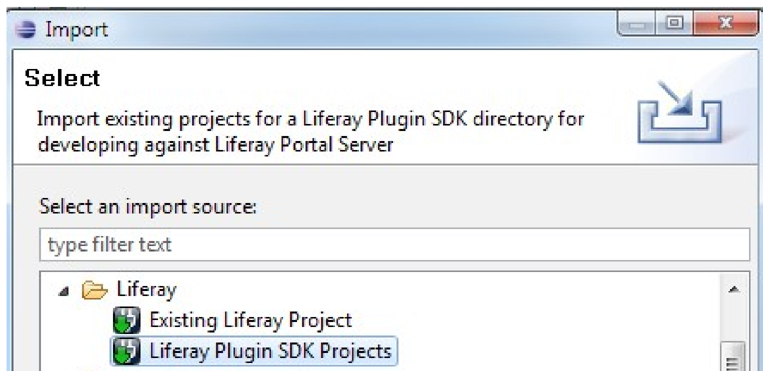


5. Click Finish to perform the import
6. Read the section below on verifying the success of an import process to make sure that your project was configured correctly as a Liferay IDE project.

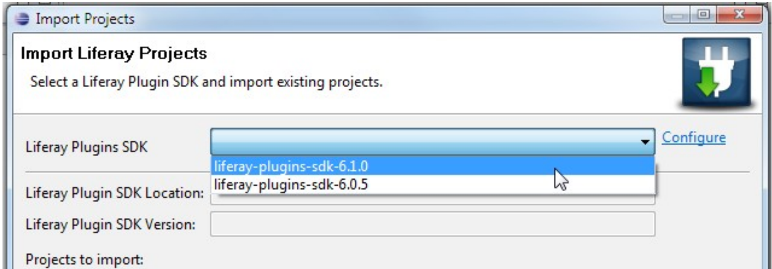
Create multiple projects for all plugins in a Plugins SDK

This option will transverse an existing Plugins SDK and will allow creating one project for each of the plugins it finds inside in one single step.

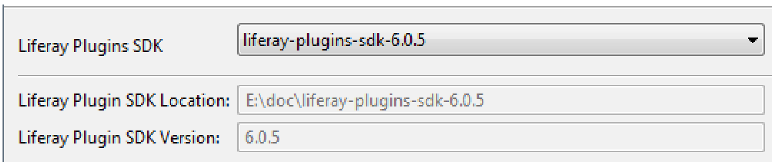
1. In Eclipse go to File > Import... > Liferay > Liferay Plugin SDK projects



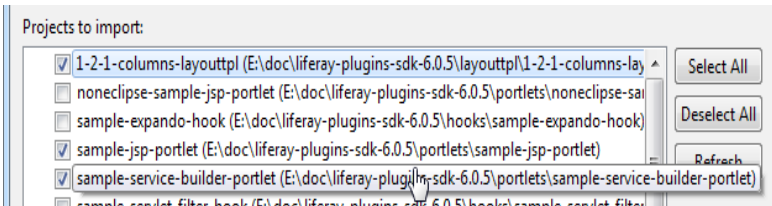
2. First you must select the Plugins SDK that you want to import projects from in the combo box at the top of the wizard.



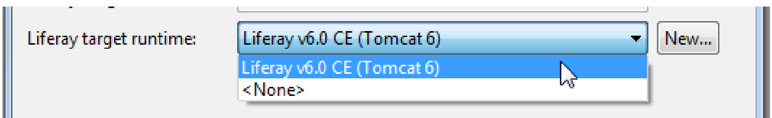
3. If you don't have any SDKs configured in Liferay IDE use the "configure" link to add a setting that points to the Plugins SDK that you want to import projects from. To configure a Plugins SDK on the Installed SDKs on the pref page just click "Add" and then Browse to the directory of the Plugins SDK root directory.
4. Once you have a configured Plugins SDK, you can select it in the Combo box and then the SDK location and version will be filled in. If either are not valid it will be marked with an error.



5. After the SDK is selected the list of projects that are available for import will be shown in the table. If the projects are already in the workspace they will be disabled. If the project is available for import it will have a empty checkbox that can be selected.



6. Select which projects that you wish to import.
7. Select the Liferay runtime that you want to setup for the imported projects. If you don't have a liferay runtime you can add one with the "New..." button.



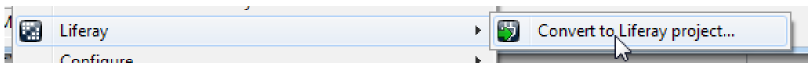
8. Click Finish.
9. Read the section below on verifying the success of an import

process to make sure that your project was configured correctly as a Liferay IDE project.

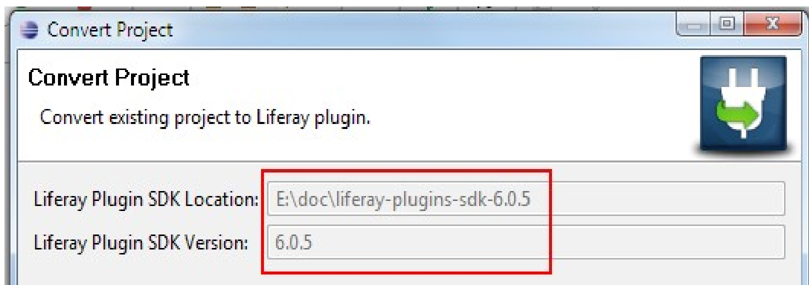
Importing an existing Eclipse Project that is not aware of the Liferay IDE

If your project is not in your Eclipse workspace, you can use the first set of steps above. If your project is already in your workspace (see it in project explorer) but is not yet a Liferay IDE project, the following steps can be used to convert the project.

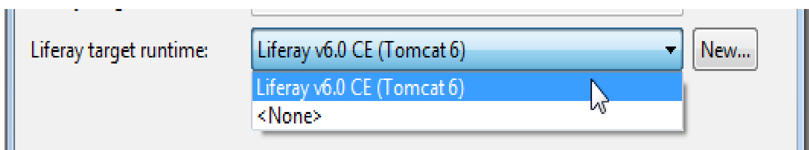
1. In Eclipse, right click the eclipse project that you want to convert, select Liferay > Convert to Liferay plug-in project. If you don't have a convert action available it means the project is either already a Liferay IDE project or it is not a faceted project with Java and Dynamic Web project facets configured and will need to be configured accordingly.



2. When the convert dialog wizard opens your project should be auto-selected and the SDK location and SDK version should be auto-detected. If they are not valid an error message will be displayed.



3. Select the Liferay runtime that you wish to set on the project. If you don't have a Liferay Runtime define use the "New..." action to create one.



4. Click Finish
5. Read the section below on verifying the success of an import process to make sure that your project was configured correctly as

a Liferay IDE project.

Importing an existing Liferay IDE project

This section describes the steps that can be followed if you have previously created or converted a Liferay IDE project in your workspace but it is no longer in the current workspace there are a couple of options for importing this project.

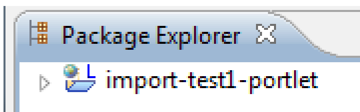
1. Open Liferay IDE, go to File > Import ... > General > Existing Projects into Workspace
2. Use option Select root directory, then click Browse
3. Select the directory of the previous Liferay IDE project
4. Then in the list of projects you should see the one project you selected
5. Click Finish
6. Read the section below on verifying the success of an import process to make sure that your project was configured correctly as a Liferay IDE project.

If you have any errors, it may be that either the SDK name used in that project or the runtime id used doesn't exist in your new workspace. You can modify the SDK name in the Project Properties > Liferay page and you can modify the targeted runtime in the Project properties > Targeted Runtimes page.

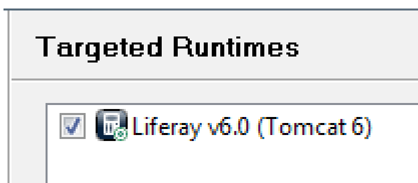
Verifying that the import has succeeded

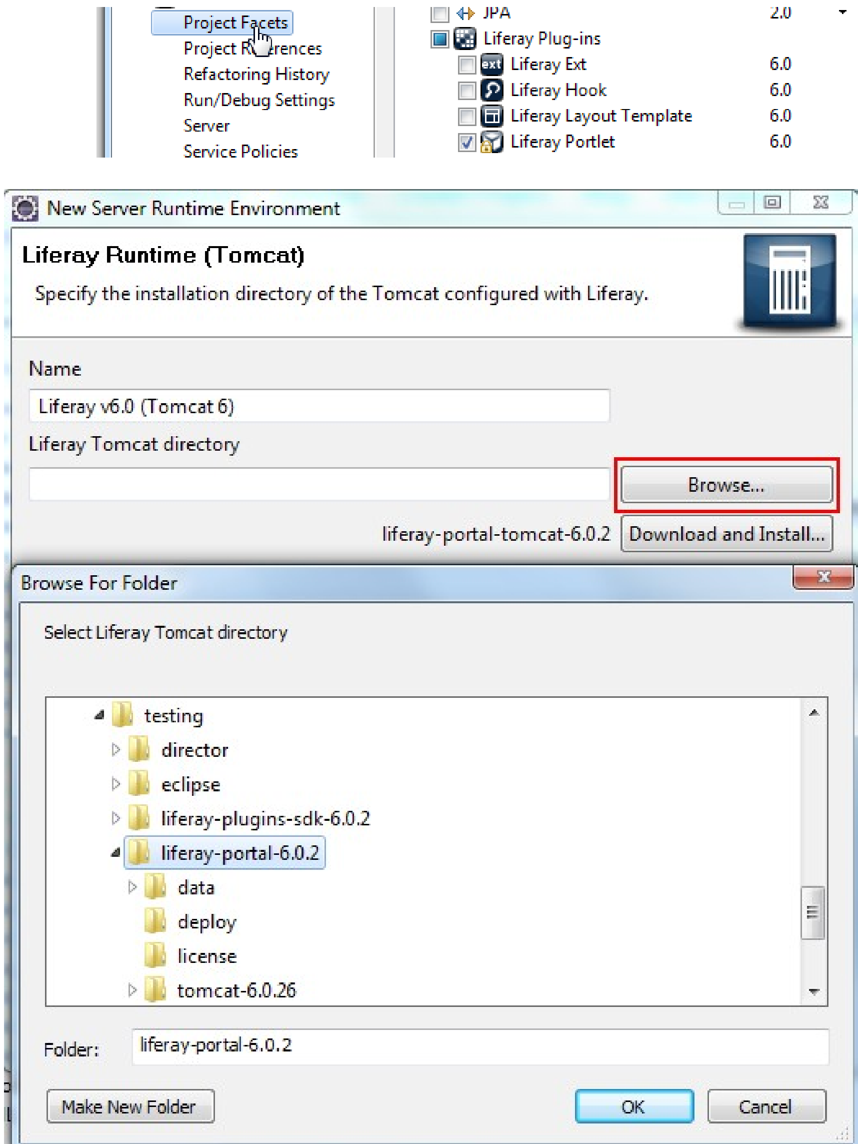
Follow the following steps to verify that either of the previous import processes has been successful.

1. Once the project import process is finished, you should see a new project inside Eclipse and it should have a "L" overlay image to show its a Liferay project.

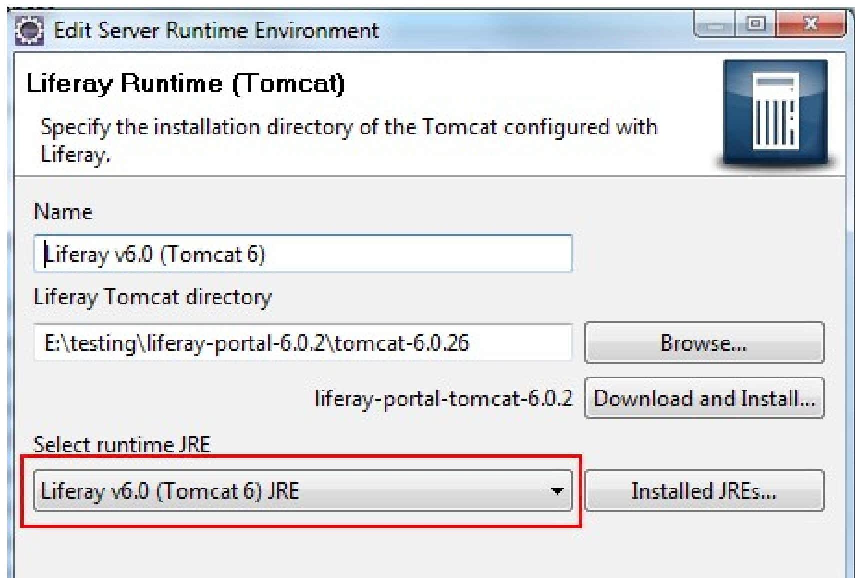


2. Secondly, to make sure the project is now a "Liferay IDE" project is to check the target runtime project property (right-click project > properties > target runtimes) and also check the project facets to make sure both Liferay runtime and Liferay plug-in facets are properly configured.

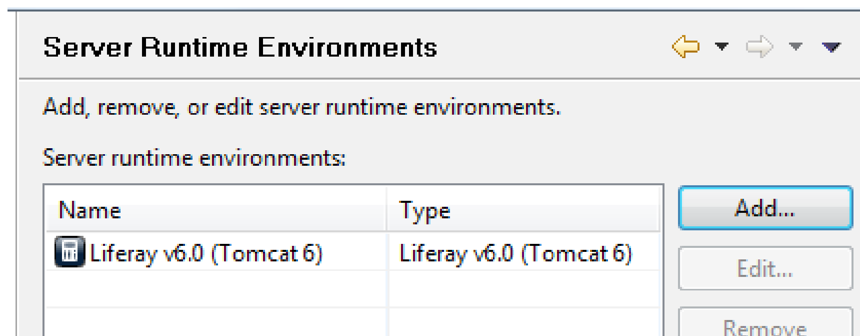




5. Once you have selected the Liferay portal directory if it has a bundled JRE then that bundled JRE will be automatically selected as the JRE to use for launching the server. However, if there is no bundled JRE (Mac and Linux users) then you will need to select the JRE to use for launch.

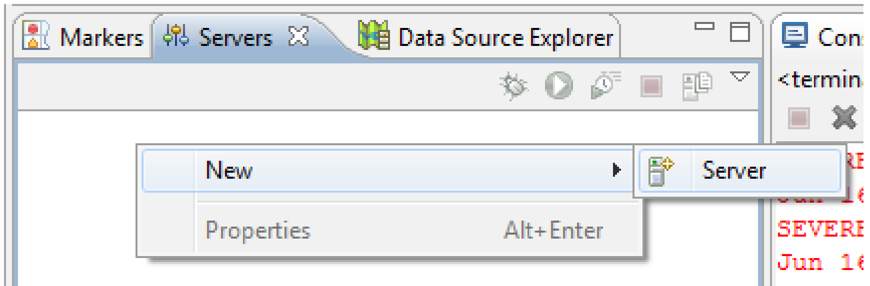


6. Click finish and you should see the Liferay portal runtime in the list of runtimes in the preference page.

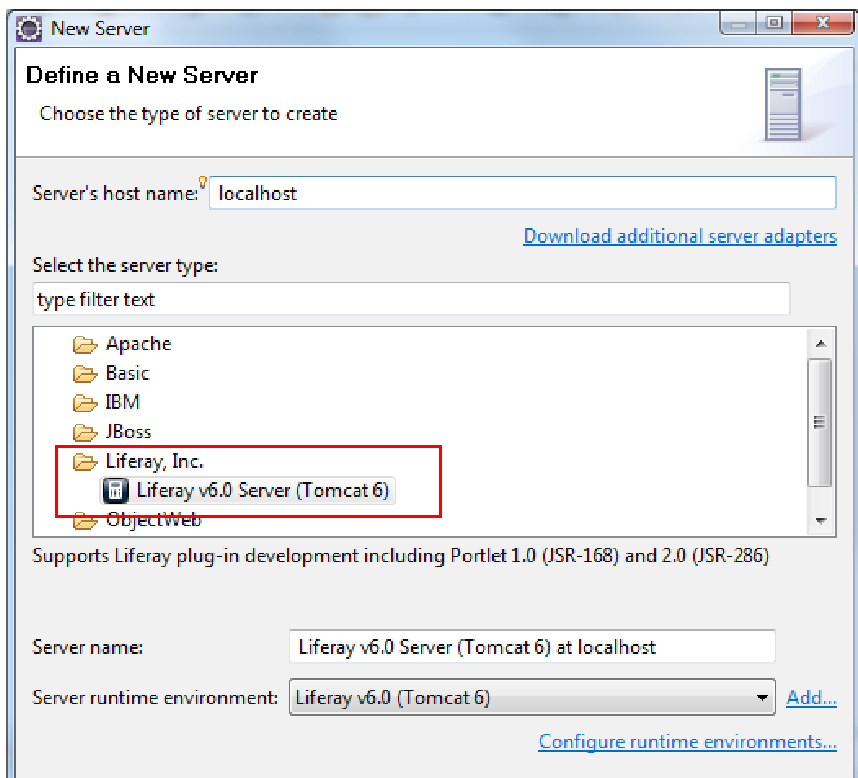


7. Click OK to save the runtime preferences.

8. If you didn't choose to create a server you will need to create one from the servers view before you can test the server.



9. Find the Liferay, Inc category and select the Liferay v6 Server and choose the Liferay v6 Runtime that you had previously created.



Setting the Console Encoding

These steps are not necessary if Liferay is installed on an OS whose encoding is latin1 or UTF-8 (most US and European OS). Otherwise, it is necessary to specify the console encoding to properly display console

messages.

1. Select "Run" - "Run Configuration..." from the Eclipse menu.

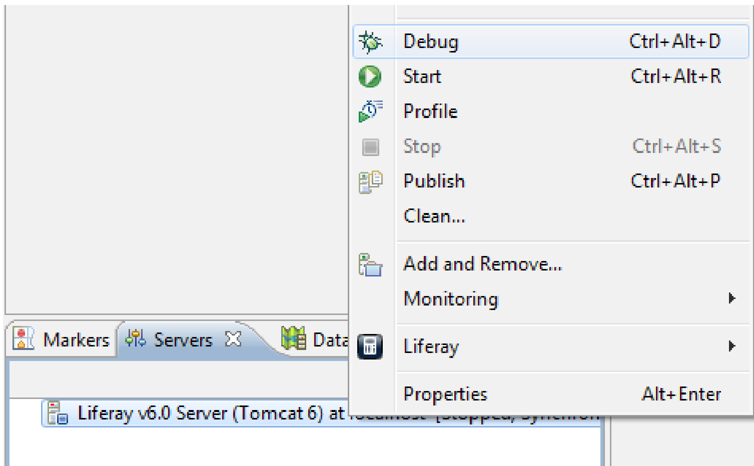
2. Select "Liferay v6.0 Server" from the menu.

3. Select the "Common" tab.

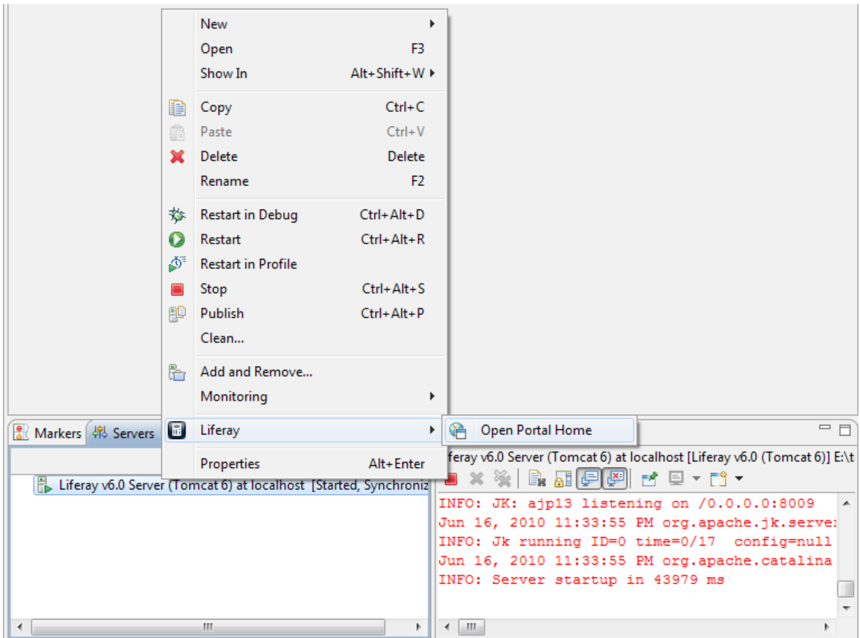
4. In the encoding section, select "Other" and "UTF-8". Select "Apply" - "Close".

Testing the Liferay portal server

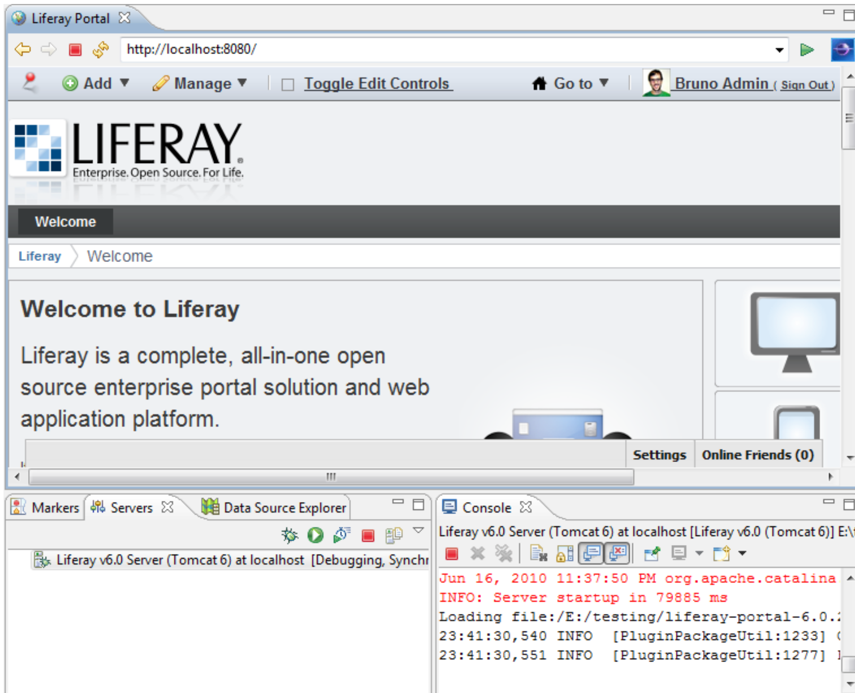
1. Go to the servers view and you should see the new server that was created. right click and choose "Start" or "Debug"



2. You should see messages appear in the Console view and once it starts, the servers view will update to show that it is "Started" and then, right-click the server and select the (Liferay Portal > Open Portal Home) action.

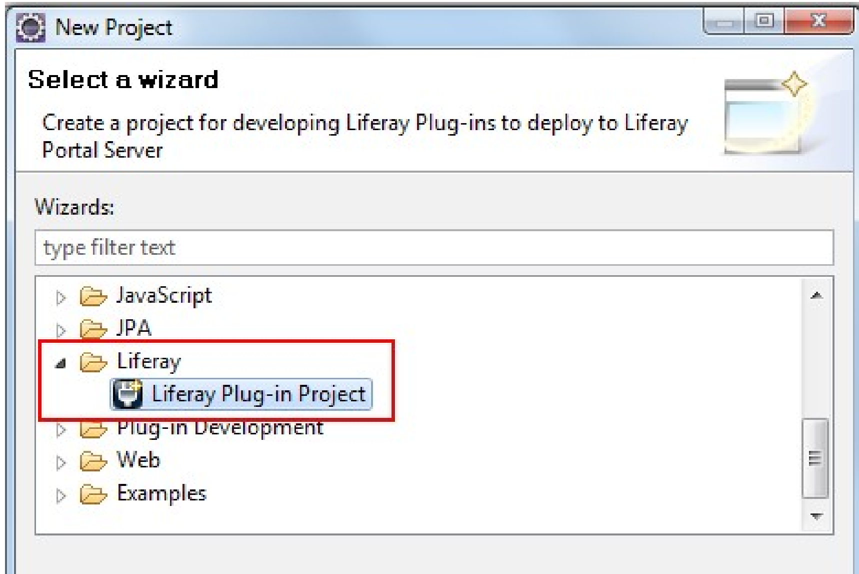


3. The eclipse browser should open to the portal home at http://localhost:8080

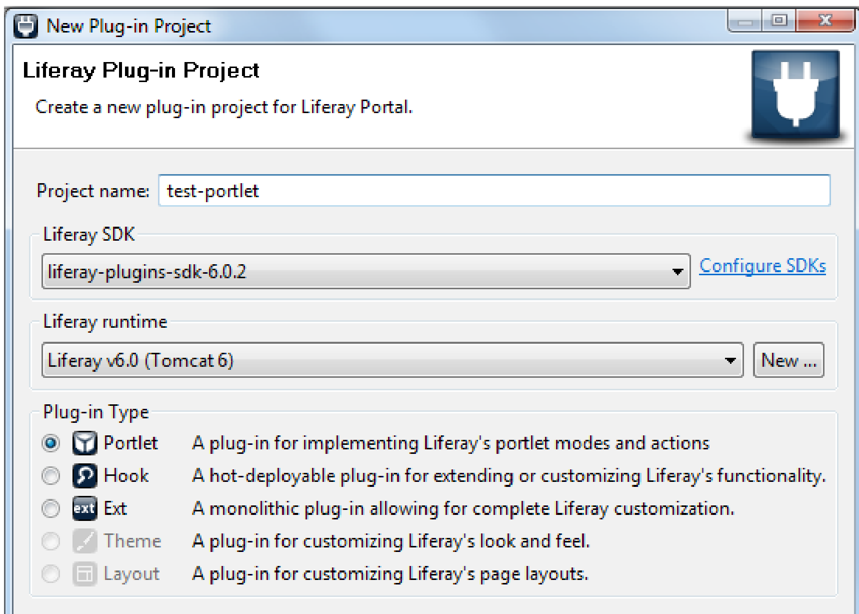


Create a new Liferay plugin Project

1. Now that a SDK and portal server have been configured you can create a new Liferay plugin project. Go to File > New Project... > Liferay > Liferay plugin Project

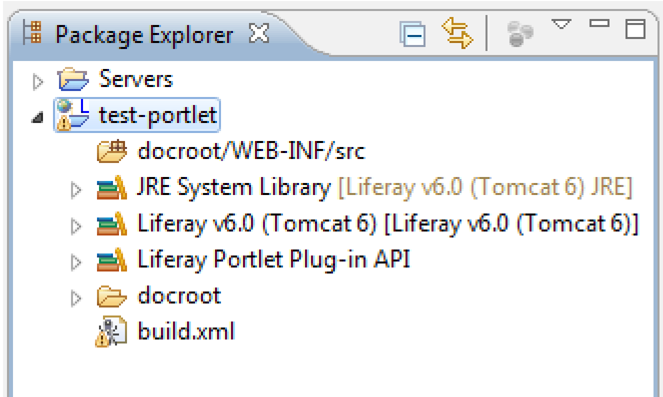


2. In the Liferay Plugin project wizard page, select the SDK and Liferay runtime and then select the plugin type (portlet is default) and now you can create a new plugin project, by clicking Finish.



3. If it worked you should see a new plugin project in the package

explorer, so you are ready to begin Plugin Development.



That's all you need to learn about Liferay IDE. At this point take a few minutes to try out creating plugins of the different types and check how Liferay IDE provides wizards to help with the most common Liferay development tasks.

Service Builder

Service Builder is a model-driven code generation tool built by Liferay to automate the creation of interfaces and classes for database persistence and a service layer. Service Builder will generate most of the common code needed to implement find, create, update, and delete operations on the database, allowing you to focus on the higher level aspects of service design.

The service layer generated by Service Builder, has an implementation class that is responsible to handle retrieving and storing data classes and adding the necessary business logic around them. This layer can optionally be composed of two layers, the local service and the remote service. The local service contains the business logic and accesses the persistence layer. It can be invoked by client code running in the same Java Virtual Machine. The remote service usually ads a code to check security and is 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 using SOAP, JSON and Java RMI.

Define the Model

The first step in using Service Builder is to define your model classes and their attributes in a `service.xml` file. For convenience, we will define the service within the **my-greeting** portlet, although it should be placed inside a new portlet. Create a file named `service.xml`

in `portlets/my-greeting-portlet/docroot/WEB-INF` inside the Plugins SDK and add the following content:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 6.0.0//EN"
"http://www.liferay.com/dtd/liferay-service-builder_6_0_0.dtd">
<service-builder package-path="com.sample.portlet.library">
    <namespace>Library</namespace>
    <entity name="Book" local-service="true" remote-service="true">

        <!-- PK fields -->

        <column name="bookId" type="long" primary="true" />

        <!-- Group instance -->

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

        <!-- Audit fields -->

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

        <!-- Other fields -->

        <column name="title" type="String" />
    </entity>
</service-builder>
```

Overview of *service.xml*

```
<service-builder package-path="com.sample.portlet.library">
```

This specifies the package path that the class will generate to. In this example, classes will generate to `WEB-INF/src/com/sample/portlet/library/`

```
<namespace>Library</namespace>
```

The namespace element must be a unique namespace for this component. Table names will be prepended with this namepace.

```
<entity name="Book" local-service="true" remote-service="false">
```

The entity name is the database table you want to create.

```
<column name="title" type="String" />
```

Columns specified in `service.xml` will be created in the database with a data type appropriate to the Java type. Accessors will be

automatically generated for these attributes in the model class.

Tip: Always consider adding two long fields called *groupId* and *companyId* to your data models. These two fields will allow your portlet to support the multi-tenancy features of Liferay so that each community or organization (for each portal instance) can have its own independent data.

Generate the Service

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

```
ant build-service
```

The service has been generated successfully when you see "BUILD SUCCESSFUL." In the terminal window, you should see that a large number of files have been generated. An overview of these files is provided below:

- Persistence
  - BookPersistence - book persistence interface @generated
  - BookPersistenceImpl - book persistence @generated
  - BookUtil - book persistence util, instances BookPersistenceImpl @generated
- Local Service
  - **BookLocalServiceImpl** - local service implementation. This is the only class within the local service that you will be able to modify manually. Your business logic will be here.
  - BookLocalService - local service interface @generated
  - BookLocalServiceBaseImpl - local service base @generated @abstract
  - BookLocalServiceUtil - local service util, instances BookLocalServiceImpl @generated
  - BookLocalServiceWrapper - local service wrapper, wraps BookLocalServiceImpl @generated
- Remote Service
  - **BookServiceImpl** - remove service implementation. Put here the code that adds additional security checks and invokes the local service.
  - BookService - remote service interface @generated
  - BookServiceBaseImpl - remote service base @generated

@abstract

- BookServiceUtil - remote service util, instances BookServiceImpl @generated
- BookServiceWrapper - remote service wrapper, wraps BookServiceImpl @generated
- BookServiceSoap - soap remote service, proxies BookServiceUtil @generated
- BookSoap - soap book model, similar to BookModelImpl, does not implement Book @generated
- BookServiceHttp - http remote service, proxies BookServiceUtil @generated
- BookJSONSerializer - json serializer, converts Book to JSON array @generated

- Model
  - BookModel - book base model interface @generated
  - BookModelImpl - book base model @generated
  - Book - book model interface @generated
  - **BookImpl** - book model implementation. You can use this class to add additional methods to your model other than the autogenerated field getters and setters.
  - BookWrapper - book wrapper, wraps Book @generated

Out of all of these classes only three can be manually modified: BookLocalServiceImpl, BookServiceImpl and BookImpl.

Write the Local Service Class

In the file overview above, you will see that **BookLocalService** is the interface for the local service. It contains the signatures of every method in **BookLocalServiceBaseImpl** and **BookLocalServiceImpl**. **BookLocalServiceBaseImpl** contains a few automatically generated methods providing common functionality. Since this class is generated, you should never modify it, or your changes will be overwritten the next time you run Service Builder. Instead, all custom code should be placed in **BookLocalServiceImpl**.

Open the following file:

```
/docroot/WEB-INF/src/com/sample/portlet/library/service/impl/BookLocalServiceImpl.java
```

We will be adding the database interaction methods to this service layer class. Add the following method to the **BookLocalServiceImpl** class:

```
public Book addBook(long userId, String title)
        throws PortalException, SystemException {
    User user = UserUtil.findByPrimaryKey(userId);
    Date now = new Date();
    long bookId = CounterLocalServiceUtil.increment(Book.class.getName());

    Book book = bookPersistence.create(bookId);

    book.setTitle(title);
    book.setCompanyId(user.getCompanyId());
    book.setUserId(user.getUserId());
    book.setUserName(user.getFullName());
    book.setCreateDate(now);
    book.setModifiedDate(now);
    book.setTitle(title);

    return bookPersistence.update(book);
}
```

Before you can use this new method, you must add its signature to the **BookLocalService** interface by running service builder again.

Navigate to the root folder of your portlet in the terminal and run:

```
ant build-service
```

Service Builder looks through **BookLocalServiceImpl** and automatically copies the signatures of each method into the interface. You can now add a new book to the database by making the following call

```
BookLocalServiceUtil.addBook(userId, “A new title”);
```

Built-In Liferay Services

In addition to the services you create using Service Builder, your portlets may also access a variety of services built into Liferay. These include `UserService`, `OrganizationService`, `GroupService`, `CompanyService`, `ImageService`, `LayoutService`, `OrganizationService`, `PermissionService`, `UserGroupService`, and `RoleService`. For more information on these services, see *Liferay in Action* and Liferay's Javadocs.

8. Liferay APIs and Frameworks

This chapter provides you with a brief overview of several of the essential APIs and frameworks provided by Liferay to developers. An API is any programing interface that you can invoke from your own code either directly through a Java invocation or through web services to perform a certain action. A framework, in this context, is a set of APIs and configuration that is designed for an specific purpose such as enhancing your applications with a permission system, with tags, with categories, comments, etc.

This chapter will keep evolving with more information about the existing APIs and frameworks and how to use it. So look back for more information often.

Security and Permissions

JSR-286 (and JSR-168) define simple security scheme using portlet roles and their mapping to portal roles. On top of that Liferay implements a fine-grained permissions system, which developers can use to implement access security in their custom portlets. This section of the document provides an overview of the JSR-286 (JSR-168) security system, Liferay's permission system, and how to implement them in your own portlets.

JSR Portlet Security

The JSR specification defines the means to specify the roles that will be used by each portlet within its definition in portlet.xml. For

example, the Blogs portlet definition included in Liferay references 3 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>
        <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>
    </portlet>
```

These roles need to be mapped to specific roles within the portal. The reason for this mapping is to allow the deployer of a portlet to solve conflicts if two portlets from two different developers use the same role name for different purposes.

Tip: Liferay provides an additional behavior to the roles referenced in the portlet.xml file using the *security-role-ref* element. Each of those roles will be given the permission to “View” those portlets by default. For example, if you want all users regardless of whether they are logged in or not to be able to view a certain portlet when it's first added to a page, make sure you specify the role “Guest” in that list.

In order to do the mapping it is necessary to use portal-specific configuration files. In the case of Liferay you can define mapping in liferay-portlet.xml. For example see definition of mapping inside `liferay-portlet.xml` 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>
```

This means that if a portlet definition references the role “power-user” it will be mapped to the Liferay role in its database called “Power User”.

In your portlet's code you can then use methods as defined in portlet specification:

- getRemoteUser ()
- isUserInRole()
- getUserPrincipal ()

For example to check if the current user has the “Power User” role the following code could be used:

```
if (renderRequest.isUserInRole(“power-user”)) {
     // ….
}
```

Note that Liferay doesn't use the isUserInRole() method in any of the portlets provided by default. Instead it uses Liferay's permission System directly, to achieve more finegrained security. The next section describes this system and how to use it in your portlets, so that you can decide which option fits your needs better.

Liferay's Permission System Overview

Adding permissions to custom portlets consists of four main steps (also known as DRAC):

1. Define all resources and their permissions.
2. **R**egister all the resources defined in step 1 in the permissions system. This is also known as “adding resources.”
3. **A**ssociate the necessary permissions with resources.
4. **C**heck permission before returning resources.

Implementing Permissions

Before you can add permissions to a portlet, two critical terms must be defined.

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 acting on a resource. For example, the *view* in “viewing the calendar portlet” is defined as a permission in Liferay.

Keep in mind that permissions for a portlet resource are implemented a little differently from other resources such as Java classes and files. In each of the subsections below, the permission implementation for the portlet resource is explained first, then the model (and file) resource.

The first step in implementing permissions is to define your resources and permissions. You can see examples of how this is accomplished for the built-in portlets by checking out a copy of the Liferay source code and looking in the `portal-impl/src/resource-actions` directory. For an example of how permissions work in the context of a portlet plugin, checkout `plugins/trunk` from the Liferay public Subversion repository, and look in the portlet *sample-permissions-portlet*.

Let’s take a look at `blogs.xml` in `portal-impl/src/resource-actions` and see how the blogs portlet defines these resources and actions.

```
<?xml version="1.0"?>
<resource-action-mapping>
    <portlet-resource>
        <portlet-name>33</portlet-name>
        <permissions>
            <supports>
                <action-key>ACCESS_IN_CONTROL_PANEL</action-key>
                <action-key>ADD_TO_PAGE</action-key>
                <action-key>CONFIGURATION</action-key>
                <action-key>VIEW</action-key>
            </supports>
            <community-defaults>
                <action-key>VIEW</action-key>
            </community-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-ref>
        <permissions>
            <supports>
                <action-key>ADD_ENTRY</action-key>
                <action-key>PERMISSIONS</action-key>
                <action-key>SUBSCRIBE</action-key>
            </supports>
            <community-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-ref>
        <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>
            <community-defaults>
                <action-key>ADD_DISCUSSION</action-key>
                <action-key>VIEW</action-key>
            </community-defaults>
            <guest-defaults>
                <action-key>VIEW</action-key>
            </guest-defaults>
            <guest-unsupported>
                <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>
            </guest-unsupported>
        </permissions>
        ...
    </model-resource>
</resource-action-mapping>
```

Permissions in the blogs portlet are defined at several different levels, coinciding to the different sections of the XML file. First, in the `<portlet-resource>` section, actions and default permissions are defined on the portlet itself. Changes to portlet level permissions are performed on a per-community basis. The settings here affect whether users can add the portlet to a page, edit its configuration, or view the portlet at all, regardless of content. All these actions are defined inside the `<supports>` tag. The default portlet-level permissions for members of the community are defined inside the `<community-defaults>` tag. In this case, members of a community should be able to view any blogs in that community. Likewise, default guest permissions are defined in `<guest-defaults>`. `<guest-unsupported>` contains permissions that a guest may never be granted, even by an administrator. For the blogs portlet, guests can never be given permission to configure the portlet or access it in the control panel.

The next level of permissions is based on the scope of an individual instance of the portlet. These permissions are defined in the first `<model-resource>` section. Notice that the `<model-name>` is not the name of an actual Java class, but simply of the blogs package. This is the recommended convention for permissions that refer to an instance of the portlet as a whole.

Tip: A “scope” in Liferay is simply a way of specifying how widely the data from an instance of a portlet is shared. For instance, if I place a blogs portlet on a page in the guest community, and then place another blogs portlet on another page in the same community, the two blogs will share the same set of posts. This is the default or “community-level” scope. If I then configure one of the two blogs and change its scope to the current page, it will no longer share content with any of the other blogs in that community. Thus, with respect to permissions, an “instance” of a blogs portlet could exist on only one page, or span an entire community.

The difference between the portlet instance permissions defined in this section, and the portlet permissions in the `<portlet-resource>` block is subtle, but critical. You will notice that permissions such as adding an entry or subscribing are defined at the portlet instance level. This makes it possible to have multiple distinct blogs within a community,

each with different permissions. For instance, a food community could have one blog that every community member could post recipes to, but also have a separate blog containing updates and information about the site itself that only administrators can post to.

After defining the portlet and portlet instance as resources, we move on to define models within the portlet that also require permissions. The model resource is surrounded by the `<model-resource>` tag. Within this tag, we first define the model name. This must be the fully qualified Java class name of the model. Next we define the portlet name that this model belongs to under the `portlet-ref` tag. Though unlikely, a model can belong to multiple portlets, which you may use multiple `<portlet-name>` tags to define. Similar to the portlet resource element, the model resource element also allows you to define a supported list of actions that require permission to perform. You must list out all the performable actions that require a permission check. As you can see for a blog entry, a user must have permission in order to add comments to an entry, delete an entry, change the permission setting of an entry, update an entry, or simply to view an entry. The `<community-defaults>` tag, the `<guest-defaults>` tag, and the `<guest-unsupported>` tag are all similar in meaning to what's explained above for a portlet resource.

After defining your permission scheme for your custom portlet, you then need to tell Liferay the location of this file. For Liferay core, the XML file would normally reside in `portal/portal-impl/classes/resource-actions` and a reference to the file would appear in the `default.xml` file. For a plugin, you should put the file in a directory that is in the class path for the project. Then create a properties file for your portlet (the one in the Sample Permissions Portlet is simply called `sample-permissions-portlet.properties`) and create a property called `resource.actions.configs` with a value that points to the the XML file. Below is an example from the Sample Permissions Portlet:

```
resource.actions.configs=resource-actions/sample-permissions-portlet.xml
```

Permission Algorithms

There are 6 permission algorithms that Liferay has used during time for checking permissions. Liferay 5 introduced algorithm 5 that is based on RBAC system. Liferay 6 optimized algorithm 5 into version 6, which included important performance improvements by using a reduced set of database tables.

It's important to note that once a permission algorithm is configured and resources are created it cannot be changed, or the existing permissions will be lost (and some system features may fail).

For all new deployments it is strongly recommended to use algorithm 6. For deployments that were using other algorithms it's recommended to use the migration tools provided from the Control

Panel > Server Administration > Migration.

For more info see permissions.user.check.algorithm option in portal.properties file.

Adding a Resource

After defining resources and actions, the next task is to write code that adds resources into the permissions system. A lot of the logic to add resources is encapsulated in the `ResourceLocalServiceImpl` class. So adding resources is as easy as calling the add resource method in ResourceLocalServiceUtil class.

```
public void addResources(
    String companyId, String groupId, String userId, String name,
    String primKey, boolean portletActions,
    boolean addCommunityPermissions, boolean addGuestPermissions);
```

For all the Java objects that require access permission, you need to make sure that they are added as resources every time a new one is created. For example, every time a user adds a new entry to her blog, the addResources(...) method is called to add the new entry to the resource system. Here's an example of the call from the `BlogsEntryLocalServiceImpl` class.

```
ResourceLocalServiceUtil.addResources(
    entry.getCompanyId(), entry.getGroupId(), entry.getUserId(),
    BlogsEntry.class.getName(), entry.getPrimaryKey().toString(),
    false, addCommunityPermissions, addGuestPermissions);
```

The parameters `companyId`, `groupId`, and `userId` should be self explanatory. The name parameter is the fully qualified Java class name for the resource object being added. The `primKey` parameter is the primary key of the resource object. As for the `portletActions` parameter, set this to true if you're adding portlet action permissions. In our example, we set it to false because we're adding a model resource, which should be associated with permissions related to the model action defined in `blogs.xml`. The `addCommunityPermissions` and the `addGuestPermissions` parameters are inputs from the user. If set to true, `ResourceLocalService` will then add the default permissions to the current community group and the guest group for this resource respectively.

If you would like to provide your user the ability to choose whether to add the default community permission and the guest permission for the resources within your custom portlet, Liferay has a custom JSP tag you may use to quickly add that functionality. Simply insert the `<liferay-ui:input-permissions />` tag into the appropriate JSP and the checkboxes will show up on your JSP. Of course, make sure the tag is within the appropriate `<form>` tags.

When removing entities from database it is also good to remove permissions mapped directly to the entity. To prevent having a lot of

dead resources taking up space in the Resource_ database table, you must remember to remove them from the Resource_ table when the resource is no longer applicable. To perform this operation call the `deleteResource(…)` method in `ResourceLocalServiceUtil`. Here's an example of a blogs entry being removed:

```
ResourceLocalServiceUtil.deleteResource(
    entry.getCompanyId(), BlogsEntry.class.getName(),
    Resource.TYPE_CLASS, Resource.SCOPE_INDIVIDUAL,
    entry.getPrimaryKey().toString());
```

Liferay Community Edition 6.0.5 has a known bug (http://issues.liferay.com/browse/LPS-14135) that causes this method to leave some data behind in the database. This error does nos affect the latest Enterprise Edition (6.0.11 and later) and it has also been fixed in the latest release of the Community Edition (6.0.6 at the time of this writing)

Adding Permission

On the portlet level, no code needs to be written in order to have the permission system work for your custom portlet. Your custom portlet will automatically have all the permission features. If you've defined any custom permissions (supported actions) in your `portlet-resource` tag, those are automatically added to a list of permissions and users can readily choose them. Of course, for your custom permissions to have any value, you'll need to show or hide certain functionality in your portlet. You can do that by checking the permission first before performing the intended functionality.

In order to allow a user to set permissions on the model resources, you will need to expose the permission interface to the user. This can be done by adding two Liferay UI tags to your JSP. The first one is the `<liferay-security:permissionsURL>` tag which returns a URL that takes the user to the page to configure the permission settings. The second tag is the `<liferay-ui:icon>` tag that shows a permission icon to the user. Below is an example found in the file `view_entry_content.jspf`.

```
<liferay-security:permissionsURL
    modelResource="<%= BlogsEntry.class.getName() %>"
    modelResourceDescription="<%= entry.getTitle() %>"
    resourcePrimKey="<%= entry.getPrimaryKey().toString() %>"
    var="entryURL"
/>

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

The attributes you need to provide to the first tag are `modelResource`, `modelResourceDescription`, `resourcePrimKey`, and `var`. The `modelResource` attribute is the fully qualified Java object class name. It then gets

translated in `Language.properties` to a more readable name.

As for the `modelResourceDescription` attribute, you can pass in anything that best describes this model instance. In the example, the blogs title was passed in. The `resourcePrimKey` attribute is simply the primary key of your model instance. The `var` attribute is the variable name this URL String will get assigned to. This variable is then passed to the `<liferay-ui:icon>` tag so the permission icon will have the proper URL link. There's also an optional attribute redirect that's available if you want to override the default behavior of the upper right arrow link. That is all you need to do to enable users to configure the permission settings for model resources.

Checking Permissions

The last major step to implementing permission to your custom portlet is to check permission. This may be done in a couple of places. For example, your business layer should check for permission before deleting a resource, or your user interface should hide a button that adds a model (e.g., a calendar event) if the user does not have permission to do so.

Similar to the other steps, the default permissions for the portlet resources are automatically checked for you. You do not need to implement anything for your portlet to discriminate whether a user is allowed to view or to configure the portlet itself. However, you do need to implement any custom permission you have defined in your resource-actions XML file. In the blogs portlet example, one custom supported action is ADD_ENTRY. There are two places in the source code that check for this permission. The first one is in the file `view_entries.jsp`. The presence of the add entry button is contingent on whether the user has permission to add entry (and also whether the user is in tab one).

```
<%
boolean showAddEntryButton = tabs1.equals("entries") &&
PortletPermission.contains(permissionChecker, plid, PortletKeys.BLOGS,
ActionKeys.ADD_ENTRY);
%>
```

The second place that checks for the add entry permission is in the file `BlogsEntryServiceImpl`. (Notice the difference between this file and the BlogsEntryLocalServiceImpl.) In the `addEntry(…)` method, a call is made to check whether the incoming request has permission to add entry.

```
PortletPermission.check(
    getPermissionChecker(), plid, PortletKeys.BLOGS,
    ActionKeys.ADD_ENTRY);
```

If the check fails, it throws a `PrincipalException` and the add entry request aborts. You're probably wondering what the `PortletPermission` and the `PermissionChecker` classes do. Let's take a look at these two

classes.

The `PermissionChecker` class has a method called `hasPermission(…)` that checks whether a user making a resource request has the necessary access permission. If the user is not signed in (guest user), it checks for guest permissions. Otherwise, it checks for user permissions. This class is available to you in two places. First in your business logic layer, you can obtain an instance of the `PermissionChecker` by calling the `getPermissionChecker()` method inside your `ServiceImpl` class. This method is available because all `ServiceImpl` (not `LocalServiceImpl`) classes extend the `PrincipalBean` class, which implements the `getPermissionChecker()` method. The other place where you can obtain an instance of the `PermissionChecker` class is in your JSP files. If your JSP file contains the portlet tag `<portlet:defineObjects />` or includes another JSP file that does, you'll have an instance of the `PermissionChecker` class available to you via the `permissionChecker` variable. Now that you know what the `PermissionChecker` does and how to obtain an instance of it, let's take a look at Liferay's convention in using it.

`PortletPermission` is a helper class that makes it easy for you to check permission on portlet resources (as opposed to model resources, covered later). It has two static methods called `check(…)` and another two called `contains(…)`. They are all essentially the same. The two differences between them are:

1. One `check(…)` method and one `contains(…)` method take in the portlet layout ID variable (`plid`).
2. The `check(…)` methods throw a new `PrincipalException` if user does not have permission, and the `contains(…)` methods return a boolean indicating whether user has permission.

The `contains(…)` methods are meant to be used in your JSP files since they return a boolean instead of throwing an exception. The `check(…)` methods are meant to be called in your business layer (`ServiceImpl`). Let's revisit the blogs portlet example below. (The `addEntry(…)` method is found in `BlogsEntryServiceImpl`.)

```
public BlogsEntry addEntry(
        long plid, String title, String content, int displayDateMonth,
        int displayDateDay, int displayDateYear, int displayDateHour,
        int displayDateMinute, String[] tagsEntries,
        boolean addCommunityPermissions, boolean addGuestPermissions,
        ThemeDisplay themeDisplay)
    throws PortalException, SystemException {

    PortletPermissionUtil.check(
        getPermissionChecker(), plid, PortletKeys.BLOGS,
        ActionKeys.ADD_ENTRY);

    return blogsEntryLocalService.addEntry(
```

```
        getUserId(), plid, title, content, displayDateMonth, displayDateDay,
        displayDateYear, displayDateHour, displayDateMinute, tagsEntries,
        addCommunityPermissions, addGuestPermissions, themeDisplay);
}
```

Before the addEntry(…) method calls BlogsEntryLocalServiceUtil.addEntry(…) to add a blogs entry, it calls PortletPermission.check(…) to validate user permission. If the check fails, a PrincipalException is thrown and an entry will not be added. Note the parameters passed into the method. Again, the getPermissionChecker() method is readily available in all ServiceImpl classes. The plid variable is passed into the method by its caller (most likely from a PortletAction class). PortletKeys.BLOGS is just a static String indicating that the permission check is against the blogs portlet. ActionKeys.ADD_ENTRY is also a static String to indicate the action requiring the permission check. You're encouraged to do likewise with your custom portlet names and custom action keys.

Whether you need to pass in a portlet layout ID (plid) depends on whether your custom portlet supports multiple instances. Let's take a look at the message board portlet for example. A community may need three separate page layouts, each having a separate instance of the message board portlet. Only by using the portlet layout ID will the permission system be able to distinguish the three separate instances of the message board portlet. This way, permission can be assigned separately in all three instances. Though in general, most portlets won't need to use the portlet layout ID in relation to the permission system.

Since the ServiceImpl class extends the PrincipalBean class, it has access to information of the current user making the service request. Therefore, the ServiceImpl class is the ideal place in your business layer to check user permission. Liferay's convention is to implement the actual business logic inside the LocalServiceImpl methods, and then the ServiceImpl calls these methods via the LocalServiceUtil class after the permission check completes successfully. Your PortletAction classes should make calls to ServiceUtil (wrapper to ServiceImpl) guaranteeing that permission is first checked before the request is fulfilled.

Checking model resource permission is very similar to checking portlet resource permission. The only major difference is that instead of calling methods found in the PortletPermission class mentioned previously, you need to create your own helper class to assist you in checking permission. The next section will detail how this is done.

It is advisable to have a helper class to help check permission on your custom models. This custom permission class is similar to the PortletPermission class but is tailored to work with your custom models. While you can implement this class however you like, we encourage you to model your implementation after the PortletPermission class, which contains four static methods. Let's take a look at the

`BlogsEntryPermission` class.

```
public class BlogsEntryPermission {

    public static void check(
            PermissionChecker permissionChecker, long entryId, String
actionId)
        throws PortalException, SystemException {

        if (!contains(permissionChecker, entryId, actionId)) {
            throw new PrincipalException();
        }
    }

    public static void check(
            PermissionChecker permissionChecker, BlogsEntry entry,
            String actionId)
        throws PortalException, SystemException {

        if (!contains(permissionChecker, entry, actionId)) {
            throw new PrincipalException();
        }
    }

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

        BlogsEntry entry = BlogsEntryLocalServiceUtil.getEntry(entryId);

        return contains(permissionChecker, entry, actionId);
    }

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

        return permissionChecker.hasPermission(
            entry.getGroupId(), BlogsEntry.class.getName(),
entry.getEntryId(),
            actionId);
    }
}
```

Again, the two `check(…)` methods are meant to be called in your business layer, while the two `contains(…)` methods can be used in your JSP files. As you can see, it's very similar to the `PortletPermission` class. The two notable differences are:

1. Instead of having the `portletId` as one of the parameters, the methods in this custom class take in either an `entryId` or a `BlogsEntry` object.
2. None of the methods need to receive the portlet layout ID (`plid`) as a parameter. (Your custom portlet may choose to use the portlet layout ID if need be.)

Let's see how this class is used in the blogs portlet code.

```
public BlogsEntry getEntry(String entryId) throws PortalException,
SystemException {
    BlogsEntryPermission.check(
        getPermissionChecker(), entryId, ActionKeys.VIEW);
    return BlogsEntryLocalServiceUtil.getEntry(entryId);
}
```

In the `BlogsEntryServiceImpl` class is a method called `getEntry(…)`. Before this method returns the blogs entry object, it calls the custom permission helper class to check permission. If this call doesn't throw an exception, the entry is retrieved and returned to its caller.

```
<c:if test="<%= BlogsEntryPermission.contains(permissionChecker, entry,
ActionKeys.UPDATE) %>">
    <portlet:renderURL windowState="<%= WindowState.MAXIMIZED.toString() %>"
var="entryURL">
        <portlet:param name="struts_action" value="/blogs/edit_entry" />
        <portlet:param name="redirect" value="<%= currentURL %>" />
        <portlet:param name="entryId" value="<%= entry.getEntryId() %>" />
    </portlet:renderURL>

    <liferay-ui:icon image="edit" url="<%= entryURL %>" />
</c:if>
```

In the `view_entry_content.jsp` file, the `BlogsEntryPermission.contains(…)` method is called to check whether or not to show the edit button. That's all there is to it!

Let's review what we've just covered. Implementing permission into your custom portlet consists of four main steps. First step is to define any custom resources and actions. Next step is to implement code to register (or add) any newly created resources such as a `BlogsEntry` object. The third step is to provide an interface for the user to configure permission. Lastly, implement code to check permission before returning resources or showing custom features. Two major resources are portlets and Java objects. There is not a lot that needs to be done for the portlet resource to implement the permission system since Liferay Portal has a lot of that work done for you. You mainly focus your efforts on any custom Java objects you've built. You're now well equipped to implement security in your custom Liferay portlets!

Asset Framework

The asset framework provides a set of functionalities that are common to several different content types. It was initially created to be able to add tags to blog entries, wiki pages, web content, etc without having to reimplement this same functionality over and over. Since then, it has grown to add more functionalities and it has been made possible to use the framework for custom applications even if they are implemented within a plugin.

The term *asset* is used as a generic way to refer to any type of content regardless of whether it's purely text, an external file, a URL, an image, an record in an online book library, etc. From now on, whenever the word asset is used, think of it as a generic way to refer to documents, blog entries, bookmarks, wiki pages, etc.

Here are the main functionalities that you will be able to reuse thanks to the asset framework:

- Associate tags to custom content types (new tags will be created automatically when the author assigns them to the content).
- Associate categories to custom content types (authors will only be 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 hierachies).
- Keep track of the number of visualizations of an asset.
- Publish your content using the Asset Publisher portlet. Asset Publisher is able to publish dynamic lists of assets or manually selected lists of assets. It is also able to show a summary view of an asset and offer a link to the full view if desired. Because of this it will save you time since for many use cases it will make it unnecessary to develop custom portlets for your custom content types.

If these functionalities seem useful for your case, then you might be wondering, what do I have to do to benefit from them?

The following subsections describe the steps involved in using the asset framework. The first one is mandatory and consists on letting the framework know whenever one of your custom content entries is added, updated or deleted. The second part is optional but can save a lot of time so most developers will probably make use of it. It consists on using a set of taglibs to provide widgets that allow authors to enter tags and categories as well as how to show the entered tags and categories along with the content. The rest of the sections are also

optional but offer interesting functionalities such as how to allow your custom assets to be published through the Asset Publisher.

Adding, updating and deleting assets

Whenever one of your custom content is created you need to let the asset framework know. Don't worry, it is simple. You just need to invoke a method of the asset framework. When invoking this method you will also let the framework know about the tags and/or categories of the content that was just authored.

All the methods that you will need to invoke are part of the AssetEntryLocalService. In particular you should access these methods using either the static methods of AssetLocalServiceUtil or by using an instance of the AssetEntryLocalService injected by Spring. To make this section simpler we will be using the former, since it doesn't require any special setup in your application.

The method that you need to invoke when one of your custom content has been added or updated is the same and is called updateAsset. Here is the full signature:

```
AssetEntry updateEntry(
    long userId, long groupId, String className, long classPK, String 
classUuid, long[] categoryIds,
    String[] tagNames, boolean visible, Date startDate, Date endDate,
    Date publishDate, Date expirationDate, String mimeType, String title,
    String description, String summary, String url, int height, int width,
    Integer priority, boolean sync)
throws PortalException, SystemException
```

Here is an example invocation to this method extracted out from the blogs portlets that comes bundled with Liferay:

```
assetEntryLocalService.updateEntry(
    userId, entry.getGroupId(), BlogsEntry.class.getName(),
    entry.getEntryId(), entry.getUuid(), assetCategoryIds,
    assetTagNames, visible, null, null, entry.getDisplayDate(), null,
    ContentTypes.TEXT_HTML, entry.getTitle(), null, summary, null, 0, 0,
    null, false);
```

Here is 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 in which the content has been created. If your content does not support scopes, you can just pass 0 as the value.
- *className*: identifies the type of asset. By convention we recommend that it is 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 any other of the same type. It is usually the primary key of the table where the custom content is stored. The *classUuid* parameter can optionally be used to specify a

secondary identifier that is guaranteed to be unique universally. Having this type of identifier is specially useful if your contents will be exported and imported across separate portals.

- *assetCategoryIds* and *assetTagNames*: represent the categories and tags that have been selected by the author of the content. The asset framework will sotre them for you.
- *visible*: specifies whether this content should be shown at all by Asset Publisher.
- *title, description* and *summary:* are descriptive fields that will be used by the Asset Publisher when displaying entries of your content type.
- *publishDate* and *expirationDate:* can be optionally specified to let Asset Publisher know that it should not show the content before a given publication date of after a given expiration date.
- All other fields are optional and might not make sense in all cases. The *sync* parameter should always be false unless you are doing something very advanced (look at the code if you are really curious).

When one of your custom content is deleted you should also let the Asset Framework know, to clean up the information stored and also to make sure that the Asset Publisher doesn't show any information for a content that has been deleted. The signature of method to do this is:

```
void deleteEntry(
    String className, long classPK)
throws PortalException, SystemException
```

Here is an example invocation extracted again from the blogs portlet:

```
assetEntryLocalService.deleteEntry(
    BlogsEntry.class.getName(), entry.getEntryId());
```

Entering and displaying tags and categories

The previous section showed how you can let the asset framework know about the tags and categories that have been associated with a given asset, but how does a content author specify such tags and categories?

The answer is that you can choose any method that you prefer, but Liferay provides a set of JSP tags that you can use to make this task very easy. The following tags can be used within the form you have created to create your type of content to allow entering tags or selecting a predefined category:

```
<label>Tags</label>
<liferay-ui:asset-tags-selector
    className="<%= entry.getClass().getName() %>"
```

```
    classPK="<%= entry.getPrimaryKey() %>"
/>

<label>Categories</label>
<liferay-ui:asset-categories-selector
    className="<%= entry.getClass().getName() %>"
    classPK="<%= entry.getPrimaryKey() %>"
/>
```

These two taglibs will create appropriate form controls that allow the user to enter any tag (even if it doesn't exist) or search and select one of the existing categories.

Tip: If you are using Liferay's Allow Form taglibs, then creating a field to enter tags or categories is even simpler. You just need to use `<aui:input name="tags" type="assetTags" />` and `<aui:input name="categories" type="assetCategories" />` respectively.

Once the tags and categories have been entered you will want to show them somewhere along with the content of the asset, there are another pair of tags that you can use to do so:

```
<label>Tags</label>
<liferay-ui:asset-tags-summary
    className="<%= entry.getClass().getName() %>"
    classPK="<%= entry.getPrimaryKey() %>"
/>

<label>Categories</label>
<liferay-ui:asset-categories-summary
    className="<%= entry.getClass().getName() %>"
    classPK="<%= entry.getPrimaryKey() %>"
/>
```

In both tags you can also use an optional parameter called portletURL. When specifying this parameter each of the tags will be a link built with the provided URL and adding a “tag” parameter or a “categoryId” parameter. This is very useful in order to provide support for tags navigation and categories navigation within your portlet. But you will need to take care of implementing this functionality yourself in the code of the portlet 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.

Publishing assets with Asset Publisher

One of the nice benefits of using the asset framework is the possibility of using the Asset Publisher portlet, which is part of the Liferay distribution, to publish lists of your custom asset types. These lists can be dynamic (for example based on the tags or categories that the asset has) or manually selected by an administrator.

In order to be able to display your assets the Asset Publisher needs to know how to access some metadata of them and also needs you to provide templates for the different type of views that it can display (abstract and full view).

You can provide all this information to the Asset Publisher through a pair of classes that implement the AssetRendererFactory interface and the AssetRenderer interface:

- AssetRendererFactory: this is the class that knows how to retrieve specific assets from the persistent storage from the classPK (that is usually the primary key, but can be anything you have chosen when invoking the updateAsset method used to add or update the asset). This class should be able to grab the asset from a groupId (that identifies an scope of data) and a urlTitle (which is a title that can be used in friendly URLs to refer uniquele to the asset within a given scope). Finally, it can also provide a URL that the Asset Publisher can 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, but you can avoid implementing them by extending from BaseAssetRendererFactory. Extending from this class, instead of implementing the interface directly will also make your code more robust if there are changes in the interface in future versions of Liferay, since the base implementation will provide custom implementations for them.

- AssetRenderer: this is the class that provides metadata information about one specific asset and is also able to check for permissions to edit or view it for the current user. It is also reponsible for rendering the asset for the different templates (abstract, and full content), by forwarding to an specific JSP. It is also recommended that instead of implementing the interface directly, you extend from the BaseAssetRenderer class, that provides with nice defaults and more robustness for methods that could be added to the interface in the future.

Let's seen an example of these two classes. Again we will pick Liferay's Blogs portlet. Lets start with the implementation for the 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);
    }

    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;
    }

    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;
    }

    public boolean hasPermission(
        PermissionChecker permissionChecker, long classPK, String actionId)
        throws Exception {

        return BlogsEntryPermission.contains(
            permissionChecker, classPK, actionId);
    }

    protected String getIconPath(ThemeDisplay themeDisplay) {
        return themeDisplay.getPathThemeImages() + "/blogs/blogs.png";
    }

}
```

And here is the AssetRenderer implementation:

```
public class BlogsEntryAssetRenderer extends BaseAssetRenderer {

    public BlogsEntryAssetRenderer(BlogsEntry entry) {
        _entry = entry;
    }

    public long getClassPK() {
        return _entry.getEntryId();
    }

    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.getContent());
    }

```

```
    public String getTitle(Locale locale) {
        return _entry.getTitle();
    }

    public PortletURL getURLEdit(
        LiferayPortletRequest liferayPortletRequest,
        LiferayPortletResponse liferayPortletResponse) {

        PortletURL portletURL = liferayPortletResponse.createRenderURL(
            PortletKeys.BLOGS);

        portletURL.setParameter("struts_action", "/blogs/edit_entry");
        portletURL.setParameter(
            "entryId", String.valueOf(_entry.getEntryId()));

        return portletURL;
    }

    public String getUrlTitle() {
        return _entry.getUrlTitle();
    }

    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();
    }

    public boolean hasEditPermission(PermissionChecker permissionChecker) {
        return BlogsEntryPermission.contains(
            permissionChecker, _entry, ActionKeys.UPDATE);
    }
```

```
    public boolean hasViewPermission(PermissionChecker permissionChecker) {
        return BlogsEntryPermission.contains(
            permissionChecker, _entry, ActionKeys.VIEW);
    }

    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;
        }
    }

    protected String getIconPath(ThemeDisplay themeDisplay) {
        return themeDisplay.getPathThemeImages() + "/blogs/blogs.png";
    }

    private BlogsEntry _entry;

}
```

Note that in the render method, there is a forward to a JSP in the case of the abstract and the full content templates. The abstract is not mandatory and if it is not provided, the Asset Publisher will show the title and the summary obtained through the appropriate methods of the renderer. The full content template should always be provided. Here is how it looks like 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-attribute-list
        className="<%= BlogsEntry.class.getName() %>"
        classPK="<%= (entry != null) ? entry.getEntryId() : 0 %>"
        editable="<%= false %>"
        label="<%= true %>"
    />
</liferay-ui:custom-attributes-available>
```

That's about it. It wasn't that hard, right? Now you can start enjoying the benefits of the asset framework in your custom portlets.

Other frameworks

Liferay has a wide variety of frameworks that make it much easier to develop complex functionalities for your own applications with little effort. These frameworks have evolved from the applications bundled with Liferay out of the box so they have been proven in the real world, even in very high performance portals.

This chapter is a placeholder that provides a quick description to the main frameworks provided with Liferay 6. Note that what follows is a work in progress since more sections will be added over time and some of the current sections will evolve into its own chapter as we add more information and detailed instructions on how to use them over time.

- File Storage Framework: Allows storing files using the backend of the Document Library. By using this framework you won't have to worry yourself about clustering or backups since that will already be taken care of for the Document Library itself. This framework is used, for example, by the wiki and the message boards of Liferay to store attached files in pages and posts respectively. You can check the sourcecode of these two portlets for great real-life examples of how to use the framework.
- Workflow Framework: Allows adding Workflow functionality to your own portlets. One great benefit of using this framework is that you will be able to reuse all of the workflow management UIs provided by Liferay. Also you will be able to abstract your code from the specific workflow engine that will be used (JBPM, Liferay Kaleo, ...). Many Liferay portlets use this framework. If you want a simple example to learn how to use it, the blogs portlet is a good start.
- Comments Framework: Allows adding comments easily in any portlet without any database code. Many Liferay portlets use

this functionality, for example the blogs portlet for the comments of each entry.

- Custom fields: A portlet that uses custom fields will allow the end user to extend the fields of its data entries with custom ones defined by the end user. To see a list of data types in Liferay that support this functionality just go to the Control Panel > Custom Fields.
- Report abuse: Allow end users to report that some information published in a page should not be there.
- Inline permissions Framework: Allows enhancing your SQL queries so that the database takes care of checking for view permissions. This is particularly useful when doing queries for data entries that could result in a large number of items (and thus checking of permissions afterwards would be very ineficient) or when you want to implement pagination (which would not work fine if permissions are checked afterwards and an item is removed). The Document Library or the Message Boards of Liferay are examples of portlets that use this functionality.
- ServiceContext: The ServiceContext object contains a set of fields that are common to many different services. It is used, for example to carry tags, categories, permissions information, ... It is not a framework in itself but rather a utility object that helps usage of the other frameworks.

Check in the near future for new editions of the Developer's Guide for extended information on each of these frameworks.

9. Resources for Liferay Developers

The following are useful reference resources for developers working with the Liferay Platform:

- Liferay specific resources:
  - **What is a portal?** http://www.liferay.com/products/what-is-a-portal
  - **Platform Javadocs:** http://docs.liferay.com/portal/6.0/javadocs/
  - **Reference documentation for Liferay's XML files:** http://docs.liferay.com/portal/6.0/definitions/
  - **Reference documentation for Liferay's taglibs:** http://docs.liferay.com/portal/6.0/taglibs/
  - Sources for version 6 (use your liferay.com account to access them): http://svn.liferay.com/repos/public/portal/branches/6.0.6/
  - Sources of the development version: http://svn.liferay.com/browse/portal
- Related specifications and standards:
  - Java 5 Javadocs: http://download.oracle.com/javase/1.5.0/docs/api/
  - JavaEE 5 Javadocs: http://download.oracle.com/javaee/5/api/

- JavaEE Overview: http://www.oracle.com/technetwork/java/javaee/tech/index.html
- Portlet Specification 2.0 (JSR-286): http://jcp.org/en/jsr/detail?id=286
- Web Services for Remote Portlets (WSRP): http://www.oasis-open.org/committees/wsrp/
- Java Content Repository (JSR-170): http://jcp.org/en/jsr/detail?id=170
- Java Server Faces 1.2 (JSR-252): http://www.jcp.org/en/jsr/detail?id=252
- Java Server Faces 2.0 (JSR-314): http://www.jcp.org/en/jsr/detail?id=314
- OpenSocial: http://www.opensocial.org/
- Sitemap protocol: http://sitemaps.org/
- WebDAV: http://webdav.org/
- SOAP: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/
- HTML 5: http://html5.org/
- WCAG 2.0: http://www.w3.org/WAI/intro/wcag20.php

- Related and commonly used technologies:
  - Spring Framework: http://www.springsource.org/
  - Hibernate: http://www.hibernate.org/
  - Struts 1: http://struts.apache.org/1.x/
  - Lucene: http://lucene.apache.org/
  - Quartz: http://www.quartz-scheduler.org
  - Alloy UI: http://alloy.liferay.com/
  - YUI 3: http://developer.yahoo.com/yui/3/
  - jQuery: http://jquery.com/
  - IceFaces: http://www.icefaces.org/main/home/
  - PortletFaces: http://www.portletfaces.org/

- Vaadin: http://vaadin.com/home
- OpenXava: http://www.openxava.org/web/guest/liferay
- Apache ant: http://ant.apache.org/
- Maven: http://maven.apache.org/
- Selenium: http://seleniumhq.org/
- Tomcat: http://tomcat.apache.org/
- JBoss Application Server: http://www.jboss.org/

10. 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 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.