Development Documentation

Development Documentation
Connor McKay, Editor
Jorge Ferrer, Editor
Copyright © 2010 by Liferay, Inc.

Contributors:
Joseph Shum, Alexander Chow, Redmond Mar, Ed Shin, Rich Sezov, Samuel Kong, Connor McKay

Table of Contents

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.

1. Introduction

This guide provides a basic introduction to all the aspects of developing with Liferay Portal. The introduction give you the orbital view of the Liferay framework, including its core technologies, libraries, and development options. The second chapter introduces the Plugins Software Development Kit and explains how to setup your development environment. The next four chapters discuss the main development strategies in more detail: portlets, themes, hooks, and Ext plugins. The final chapter is a brief reference for two of the key Liferay technologies, ServiceBuilder and Alloy UI.

Core Technologies

Liferay is first and foremost a Java portlet container, conforming to the JSR168 and JSR286 specifications. Liferay is extremely versatile, and operates as a JavaEE 5 servlet under all commonly used Java servlet containers, including Apache Tomcat, GlassFish, JBoss, Jetty, Jonas, WebSphere, WebLogic, and Resin. Liferay can connect to almost any SQL database server, including IBM DB2, Apache Derby, Firebird, Hypersonic, HSQLDB, IBM Informix, Ingres, MySQL, Oracle, PostgreSQL, SAP DB, Microsoft SQL Server, and Sybase. Internally, Liferay can be configured to use either Hibernate or JPA for managing database connections and handling queries.

To further increase its flexibility, Liferay makes extensive use of the inversion of control (IOC) and aspect oriented programming (AOP) methodologies through the Spring framework. Almost every utility or service in Liferay can be replaced with another implementation simply by altering the appropriate Java Bean reference.

Caching is performed with a combination of distributed and local caching using Ehcache, Memcached, or simple in-memory caching. Developers can add support for any other caching mechanism by writing their own cache manager class and injecting it into Liferay using Spring.

Liferay internally uses many open source projects, including Struts and Spring. This does not mean however that portlets are limited to only these technologies. Liferay is a standard JSR-286 portlet container, and also supports a wide variety of portlet frameworks, including Struts, Spring MVC, Java Server Faces (JSF), and its own framework, MVCPortlet. Portlet templates can be written in Velocity, Freemarker, vanilla JSP, or virtually any other language of the developer's choosing.

To accelerate the development process, Liferay includes many common interface elements and widgets as part of the Alloy UI framework. It also provides its own persistence and service framework called Service Builder, which automatically generates much of the common code required for find, create, update, and delete database operations.

Development Strategies

Extensions to the Liferay platform can take several forms. Some development strategies make it possible to modify the Liferay core behavior, while others only allow the addition of distinct bundles of new functionality. It is important to choose the most appropriate strategy in order to minimize the time and effort involved, while also ensuring forward compatibility with future versions. The six development options available in Liferay are portlets, themes, layout templates, hooks, Ext plugins, and the web application integrator (WAI).

Portlets

Portlets are small web applications 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 extension's functionality into several smaller pieces that can be arranged dynamically on a page. Portlets can be written using any of the frameworks mentioned earlier, or either of the Liferay specific frameworks: MVCPortlet or AlloyPortlet.

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.

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.

Hooks

Hooks are the recommended method of adding to the core functionality of Liferay at many predefined extension points. Hooks can be used to modify portal properties or to perform custom actions on startup, shutdown, login, logout, session creation and session destruction. Using Spring IOC, it is possible with a hook to replace any of the core Liferay services with your own implementation. Hooks 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.

Ext plugins

Ext plugins provide the largest degree of flexibility in modifying the Liferay core, and allow you to replace essentially any class with your own implementation. 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 not recommended unless there is no other way to accomplish the same goal, and only if you are very familiar with the Liferay core. Even though Ext plugins are deployed as plugins, the server must be restarted for changes to take effect.

Ext plugins are a new feature in Liferay 6.0 which replace what was previously known as the extension environment.

The Web Application Integrator (WAI)

The WAI is not a true development option, as it does not require any actual code to be written. Instead it offers a quick way to bring any standard Java Servlet application into Liferay with a decent level of integration. Simply copy a .WAR file into the Liferay deploy directory, and it will automatically be configured with the necessary files to make it available inside an iframe as a portlet. For more information on this topic, please see the Liferay wiki.

2. The Plugins SDK

There are multiple ways to create plugins for Liferay. Many IDEs on the market today support portlet projects natively, and at the time this guide was written, the official Liferay IDE plugin for Eclipse has just been released. However, because Liferay makes every effort to remain tool agnostic, we provide a Plugins Software Development Kit (SDK) which may be used in any environment. In this guide we will use only the Plugins SDK and a text editor, but you may use whatever tool you are comfortable with.

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.

Unzip both archives to a folder of your choosing. Then, in the *liferay-portal-[version]/tomcat-6.0.26/webapps* directory, delete all the directories except for **ROOT** and **tunnel-web** (if you are using the EE edition of Liferay this step is not necessary). To start Liferay, open a terminal, navigate to *liferay-portal-[version]/tomcat-6.0.26/bin*, enter the following command (Linux and Mac OS X):

```
./catalina.sh run
```

On Windows you may simply double click *startup.bat*. To shut the server down later, press Ctrl-c.

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** and press Enter. If you get a build file not found error, you have correctly installed Ant. If not, check your environment variable settings and make sure they are pointing to the directory in which 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*. Open this file in your editor of choice. At the top of the file is the message “DO NOT EDIT THIS FILE”. This file contains the settings for where you have Liferay installed and where your deployment folder is going to be. However, you should not customize this file. Instead, 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.j-smith.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-6.0.26* 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 generating that type of plugin. 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 project. 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 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.

Deploying the Portlet

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.)

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, which uses separate JSPs for each page in the portlet, and for each of the three portlet modes: view, edit, and help.

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.

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 instance on a page, and each portlet 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 so that those .jars are on the class path.

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 with the portlet ID to prevent conflicts with other portlets.

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

```
<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-name	The portlet-name element contains the canonical name of the portlet. Each portlet name is unique within the portlet application. (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 of 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.
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 role's 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.

```
<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 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 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 line that says:

```
<instanceable>true</instanceable>
```

From **true** to **false**.

Next, we will create our 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="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 = ParamUtil.getString(renderRequest, "greeting");
if (Validator.isNotNull(greeting)) {
    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-6.0.26* 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. 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 the form, by providing tags that will render both the label and the field at once.

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 not followed any of the best practices of portlet design in creating it. Please do not use this portlet as a template for creating your own portlets, but only as a demonstration of how to use MVCPortlet's page flow.

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!

```
<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, please see the fuller 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).

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. 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.

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-6.0.4/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 style through the 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.

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. 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 both 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`. You will need to create the `META-INF` directory as well. 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.

Tip: If you wish to 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. For an example of this technique, checkout `plugins/trunk` from the Liferay public Subversion repository, and look in the hook *so-hook*.

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. The log out and back in, and you should see our custom message in the terminal window running Liferay.

Custom actions are not the only extensions that can be made by

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 `portal.properties` in the Liferay source code.

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

In Liferay, the service classes used to manage database interaction are accessed through utility classes that enable static access the the service methods. For instance, the user service is defined in the interface `UserLocalService` and implemented in `UserLocalServiceImpl`. `UserLocalServiceUtil` contains a single instance of a class that implements `UserLocalService`, which by default is `UserLocalServiceImpl`. This instance is injected dynamically at runtime using a Spring IOC bean reference. You can see the definition of this reference in the file `portal-spring.xml` in the Liferay source code:

```
<bean id="com.liferay.portal.service.UserLocalService"
class="com.liferay.portal.service.impl.UserLocalServiceImpl" />
```

The advantage of this design is that it makes it possible to replace `UserLocalServiceImpl` (or any other portal service) with your own implementation in a hook.

To make this task easier, Liferay automatically generates dummy wrapper classes for all of its services – `UserLocalServiceWrapper` in this case. 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 inject our class into `UserLocalServiceUtil`.

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);
    }
}
```

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.

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. To see an example of this technique, checkout `plugins/trunk` from the Liferay public Subversion repository, and examine the hook `so-hook`.

6. Ext plugins

Ext plugins provide the most powerful methods 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 rebooted after deployment, as well as requiring additional steps when deploying or redeploying to production systems.

The main use cases in which an Ext plugin can 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 (use hook plugin if possible)
- 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 this 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-6.0.5/tomcat-6.0.26
app.server.zip.name={...}/liferay-portal-tomcat-6.0.5.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 reboot 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 shut down the 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.

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. In a bundle, this directory is in the root of the unzipped bundle called `deploy`.
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 auto deploy, 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're using in your development environment. Once it's deployed, 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 are not as affected by changes in the Liferay platform when new versions are released.

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 JSPs and merge them with the changes done to those JSPs in 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 Frameworks

This final chapter will provide you with a brief overview of several of the essential frameworks and services in Liferay. For more detailed information about any of these topics, please see *Liferay in Action*, or even check out a copy of the Liferay source code to see how they are used in practice.

Service Builder

Service Builder is a tool built by Liferay to automate the creation of interfaces and classes for database persistence, local and remote services. Service Builder will generated 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.

Tip: A “service” in Liferay is simply a class or set of classes designed to handle retrieving and storing data classes. A local service is used by code running in the local instance of Liferay, while a remote service can be accessed from anywhere over the internet or your local network. Remote services support SOAP, JSON, and Java RMI.

Define the Database Structure

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.

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:

- Persistance
  - BookPersistence - book persistence interface @generated
  - BookPersistenceImpl - book persistence @generated
  - BookUtil - book persistence util, instances BookPersistenceImpl @generated
- Local Service
  - BookLocalService - local service interface @generated
  - BookLocalServiceBaseImpl - local service base @generated @abstract
  - BookLocalServiceImpl - local service
  - BookLocalServiceUtil - local service util, instances BookLocalServiceImpl @generated
  - BookLocalServiceWrapper - local service wrapper, wraps BookLocalServiceImpl @generated
- Remote Service
  - BookService - remote service interface @generated
  - BookServiceBaseImpl - remote service base @generated @abstract
  - BookServiceImpl - remove service
  - 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
   - BookWrapper - book wrapper, wraps Book @generated

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.

Security and Permissions

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 Liferay permissions system, and how to implement it in a your own portlets.

Overview

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

1. **D**efine 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
```

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.

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. Simply 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());
```

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!