Development Documentation

Development Documentation
Samuel Kong, Editor

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

Table of Contents

Preface

This book was written as a quick guide to getting started developing on the Liferay Enterprise Portal platform. It is a guide for those who want to get their hands dirty developing for Liferay.

The information contained herein has been organized in a way that hopefully makes it easy to locate information.

Conventions

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

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

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

Notes

It is our hope that this book will be valuable to you, and that it will be an indispensable resource as you begin to develop on Liferay's platform. If you need any assistance beyond what is covered in this book, 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 book 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.

1. Hot Deploy Development

This section of the guide is a quick start to plugins development and deployment guide, using Liferay's Plugins Software Development Kit. Plugins (portlets and themes) are now the preferred way to add functionality to Liferay, as they have several benefits over using the extension environment:

● Plugins can be composed of multiple smaller portlet and theme projects. This reduces the complexity of individual projects, allowing developers to more easily divide up project functionality

● Plugins are completely separate from the Liferay core. Portlet plugins written to the JSR-168 standard are deployable on any portlet container

● Plugins can be hot deployed (i.e., deployed while the server is running) and are available immediately. This prevents any server downtime for deployments

There are multiple ways to create portlet and theme plugins for Liferay. Many IDEs on the market today support portlet projects natively, and theme projects are nothing more than standard web modules with style sheets, images, and optional JavaScript and Velocity templates in them. Because of this, there are many tools which can be used to create plugins, from text editors to full blown integrated development environments. If you are already familiar with such a tool, you may use that tool to create plugins.

Because Liferay makes every effort to remain tool agnostic, we provide a Plugins Software Development Kit (SDK) which may be used to create both portlet and theme plugins. This SDK may be used with any text editor or IDE to create plugins for Liferay. Though it is not necessary to use this SDK to create plugins, it is the recommended method.

This section of the document will show you how to create both portlet and theme plugins using Liferay's plugin SDK. In the process, this will also show you the proper project layout for portlet and theme plugins, allowing you to use your own tools to create plugins if you wish to do so.

Initial Setup

Setting up the Plugins SDK is pretty straightforward. Download the archive from Liferay's

Additional Files download page, in the section for developers here:

http://www.liferay.com/web/guest/downloads/additional.

Unzip the file to the location in which you will be doing your work. The two folders you will be working in mostly are the portlets and the themes folders. It is here that you will place your portlet and theme plugin projects.

But first you will need to create a configuration file and make sure you have some tools installed. Building portlet and theme 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. Uncompress the archive into an appropriate 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. Set another environment variable called ANT_OPTS with the proper memory settings for building projects.

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

```
ANT_HOME=/java/apache-ant-1.7.0
ANT_OPTS="-Xms256M -Xmx512M"
PATH=$PATH:$HOME/bin:$ANT_HOME/bin
export ANT_HOME ANT_OPTS PATH
```

Log out and log back in 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-1.7.0), and click *OK*.

Select New again. Make the Variable Name *ANT_OPTS* and the Variable Value "-Xms256M -Xmx512M" and click OK.

Scroll down until you find the **PATH** environment variable. Select it and select *Edit*. Add **%ANT_HOME%\bin** to the end or beginning of the **PATH**. Select OK, and then select *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 to which you unzipped Ant.

You will need a Liferay runtime on which to deploy your plugins to test them. We recommend using the Liferay-Tomcat bundle which is available from Liferay's web site, as Tomcat is small, fast, and takes up less resources than most other containers. Download the latest Liferay-Tomcat bundle and unzip it to a folder on your machine. You can start Tomcat by navigating to the *<Tomcat Home>/bin* folder and running the startup command (i.e., startup.bat for Windows or ./startup.sh for Linux or Mac).

You will notice that the plugins SDK contains a file called *build.properties*. Open this file in the text editor or IDE you will be using to create portlets and themes. At the top of the file is a 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, but you don't want to 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.jsmith.properties*.

You will likely need to customize the following properties:

```
app.server.dir=
java.compiler=
```

app.server.dir: This is the folder into which you have installed your application server..

java.compiler: Defaults to the standard Java compiler, modern, from your JDK. However, you can set this to the Eclipse compiler, ECJ, if you want.

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

The plugins SDK can be used to house all of your portlet and theme projects enterprise-wide, or you can have separate plugins SDK projects for each of your portal projects. For example, if you have an internal Intranet which uses Liferay and which has some custom written portlets for internal use, 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 running for your public Internet web site, you could have a separate plugins SDK with those projects (portlet and theme) in your source code repository. Or you could further separate your projects by having a different plugins SDK project for each portlet or theme project. It's really up to you.

You could also use the plugins SDK as a simple cross-platform new project generator. You can generate the project using the ant scripts in the plugins SDK and then copy the resulting project from the portlets or themes folder to your IDE of choice. You would need to customize the ant script if you wish to do that, but this allows organizations which have strict standards for their Java projects to adhere to those standards.

Introduction to Java Standard Portlets

Portlets are small web applications that run in a portion of a web page. The heart of any portal imIntroduction plementation is its portlets, because portlets are where the functionality of any portal resides. Liferay's core is a portlet container, and this container is only responsible for aggregating the set of portlets that are to appear on any particular page. This means that all of the features and functionality of your portal application must be in its portlets.

Portlet applications, like servlet applications, have become a Java standard which various portal server vendors have implemented. The JSR-168 standard defines the initial portlet specification, while the JSR-286 standard defines the Portlet 2.0 specification. We will refer to these together as Java Standard portlets. A Java Standard portlet should be deployable on any portlet container which complies with the

standard. Portlets are placed on the page in a certain order by the end user and are served up dynamically by the portal server. This means that certain "givens" that apply to servlet-based projects, such as control over URLs or access to the HttpServletRequest object, don't apply in portlet projects, because the portal server generates these objects dynamically.

Portal applications come generally in two flavors: 1) portlets can be written to provide small amounts of functionality and then aggregated by the portal server into a larger application, or 2) whole applications can be written to reside in only one or a few portlet windows. The choice is up to those designing the application. The developer only has to worry about what happens inside of the portlet itself; the portal server handles building out the page as it is presented to the user.

Most developers nowadays like to use certain frameworks to develop their applications, because those frameworks provide both functionality and structure to a project. For example, Struts enforces the Model-View-Controller design pattern and provides lots of functionality, such as custom tags and validating functionality, that make it easier for a developer to implement certain standard features. With Liferay, developers are free to use all of the leading frameworks in the JavaEE space, including Struts, Spring, and Java Server Faces. This allows developers familiar with those frameworks to more easily implement portlets, and also facilitates the quick porting of an application using those frameworks over to a portlet implementation.

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. We have plenty of examples of this; to see them, check out the Plugins SDK from Liferay's public Subversion repository.

Creating A Portlet

Creating portlets with the plugins SDK is a straightforward process. 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. Enter the following command in the *portlets* folder:

```
ant -Dportlet.name=<project name> -Dportlet.display.name="<portlet title>" create
```

For example:

```
ant -Dtheme.name=hello-world -Dtheme.display.name="Hello World" create
```

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.

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.

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/
       /docroot/
       /css/
       /js/
       /WEB-INF/
              /classes/
              /lib/
              /src/
              liferay-display.xml
              liferay-plugin-package.properties
              liferay-portlet.xml
              portlet.xml
              web.xml
       icon.png
       view.jsp
build.xml
```

The example is a fully deployable portlet which can be deployed to your configured Liferay server by running the deploy ant task.

The default portlet is configured as a standard JSR-168 portlet which uses separate JSPs for its three portlet modes (view, edit, and help). Only the view.jsp is implemented in the example; the code will need to be customized to enable the other modes.

The **Java Source** is stored in the *docroot/WEB-INF/src* folder. You can go in and customize (and rename) the portlet class and add any classes necessary to implement your functionality.

The **Configuration Files** are stored in the *docroot/WEB-INF* folder. The two standard JSR-168 portlet configuration files, *web.xml* and *portlet.xml* are 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 for Liferay what category the portlet should appear under in the *Add Application* window.

liferay-portlet.xml: This file describes some optional Liferay-specific enhancements for JSR-168 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 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>.

The default portlet project that is created is a simple JSR-168 portlet, with no bells and whistles. You can use this framework to write your code to the JSR-168 portlet API and implement all the functionality that you need. There are many portlets that are implemented this way. The standard portlet API is easy to use and straightforward.

Many developers, however, prefer to use a particular framework when developing web applications. Several frameworks, such as Struts, Spring, or Java Server Faces, make the development of web applications more straightforward and easier to follow than a standard servlet implementation would be. All three of the frameworks mentioned can also be used to create portlets. Liferay has many examples of how these frameworks would be used in our public Subversion repository at SourceForge. You can grab them by checking them out of the repository or by accessing our Official Plugins page at http://www.liferay.com/web/guest/downloads/official_plugins.

A Closer Look at the Default JSP 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>jsp-portlet</portlet-name>
	<display-name>JSP Portlet</display-name>
```

```
        <portlet-class>com.sample.jsp.portlet.JSPPortlet</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>JSP Portlet</title>
                <short-title>JSP Portlet</short-title>
                <keywords>JSP Portlet</keywords>
        </portlet-info>
        <security-role-ref>
                <role-name>administrator</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. (A Power User can personalize the portal, whereas a User cannot.)

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>jsp-portlet</portlet-name>
                <icon>/icon.png</icon>
                <instanceable>true</instanceable>
                <header-portlet-css>/css/test.css</header-portlet-css>
```

```
                <header-portlet-javascript>/js/test.js</header-portlet-javascript>
        </portlet>
        <role-mapper>
                <role-name>administrator</role-name>
                <role-link>Administrator</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.
struts-path	Tells the portal that all requests with the path ext/library/* are considered part of this portlet's scope. For example, a struts-path of ext/library will give us access to /ext/library/view in struts-config.xml, but not /ext/reports/view.

There are may more elements, so you will probably want to look over the DTD for this file since there are many options you will want to be aware of if you do more advance development. The DTD may be found in the *definitions* folder in the Liferay source code.

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: if the plugin SDK is used, themes become only a list of differences from the default theme. 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).

There is a *themes* folder inside the plugins SDK where all new themes reside. To create a new theme, you run the following command in this folder:

```
ant -Dtheme.name=<project name> -Dtheme.display.name="<theme title>" create
```

For example:

```
ant -Dtheme.name=newtheme -Dtheme.display.name="My New Theme" create
```

This command will create a blank theme in your *themes* folder.

Theme Concepts

Custom themes are based on differences between the custom code and the default Liferay theme, called Classic. You will notice that there is a *_diffs* folder inside of your custom theme folder. This is where you will place your theme code. You only need to customize the parts of your theme that will differ from what is already displayed in the Classic theme. To do this, you mirror the directory structure of the Classic theme inside of the _diffs folder, placing only the folders and files you need to customize there.

For example, to customize the Dock (a necessary component of all themes), you would copy just the *dock.vm* file from your Liferay installation (the Classic theme is in *<Tomcat_Home>/webapps/ROOT/html/themes/classic*) into your theme's *_diffs/templates* folder. You can then open this file and customize it to your liking. For example, you might want to change the welcome message to something else, like "Quick Links."

For custom styles, we recommend you create a css 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 to the default Liferay styles. It is best to do it this way because of the order in which the .css files are loaded. Custom.css is loaded last, and so anything inside this file will be guaranteed to override any styles that are in any of the other style sheets.

Anatomy of a Theme

The folders in themes are designed to be easy to navigate and understand. Currently, this is what the new directory structure looks like:

```
/THEME_NAME/
       /css/
              base.css
              custom.css
              main.css
              navigation.css
              forms.css
              portlet.css
              deprecated.css
              tabs.css
              layout.css
       /images/
              (many directories)
       /javascript/
              javascript.js
       /templates/
              dock.vm
              navigation.vm
              portal_normal.vm
              portal_popup.vm
              portlet.vm
/WEB-INF
```

```
/META-INF
```

You can copy any of these files from the default custom theme to your *_diffs* folder in order to customize that portion of the theme.

JavaScript

Liferay now includes the jQuery JavaScript library, and theme developers can include any plugins that jQuery supports. The $ variable, however, is not supported (for better compliance with different portlets). Inside of the *javascript.js* file, you will find three different function calls, like this:

```
jQuery(document).ready(
	function() {
		//Custom javascript goes here
	}
);
Liferay.Portlet.ready(
	function(portletId, jQueryObj) {
		//Custom javascript goes here
	}
);
jQuery(document).last(
	function() {
		//Custom javascript goes here
	}
);
```

- **jQuery(document).ready(fn);**

When this gets passed a function (it can be a defined function, or an anonymous one like above), the function gets executed as soon as the HTML in the page has finished loading (minus any portlets loaded via ajax).

- **Liferay.Portlet.ready(fn);**

When this gets passed a function (it can be a defined function, or an anonymous one like above), the function gets executed after each portlet has loaded. The function that gets executed receives two variables, *portletId* and *jQueryObj*. *portletId* is the id of the current portlet that has loaded, and *jQueryObj* is the jQuery object of the current portlet element.

- **jQuery(document).last(fn);**

When this gets passed a function (it can be a defined function, or an anonymous one like above), the function gets executed after everything—including AJAX portlets—gets loaded onto the page.

Besides theme-wide JavaScript there is also support for page specific JavaScript. The Page Settings form provides three separate JavaScript pieces that you can insert anywhere in your theme. Use the following to include the code from these settings:

```
$layout.getTypeSettingsProperties().getProperty("javascript-1")
```

```
$layout.getTypeSettingsProperties().getProperty("javascript-2")
$layout.getTypeSettingsProperties().getProperty("javascript-3")
```

The content of the JavaScript settings fields are stored in the database as Java Properties. This means that each field can only have one line of text. For multi-line scripts, the newlines should be escaped using \, just as in a normal .properties file.

Settings

Each theme can define a set of settings to make it configurable.

The settings are defined in the *liferay-look-and-feel.xml* using the following syntax:

```
<settings>
	<setting key="my-setting" value="my-value />
	...
</settings>
```

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.

While developing the theme we get to the header. In the *portal_normal.vm* template we write:

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

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="beauty1" name="Beauty 1">
	<root-path>/html/themes/beauty</root-path>
	<templates-path>${root-path}/templates</templates-path>
	<images-path>${root-path}/images</images-path>
	<template-extension>vm</template-extension>
	<settings>
		<setting key="header-type" value="detailed" />
	</settings>
	<color-scheme id="01" name="Blue">
		<css-class>blue</css-class>
		<color-scheme-images-path>
			${images-path}/color_schemes/${css-class}
		</color-scheme-images-path>
```

```
        </color-scheme>
        ...
</theme>
<theme id="beauty2" name="Beauty 2">
        <root-path>/html/themes/beauty</root-path>
        <templates-path>${root-path}/templates</templates-path>
        <images-path>${root-path}/images</images-path>
        <template-extension>vm</template-extension>
        <settings>
                <setting key="header-type" value="brief" />
        </settings>
        <color-scheme id="01" name="Blue">
                <css-class>blue</css-class>
                <color-scheme-images-path>
                        ${images-path}/color_schemes/${css-class}
                </color-scheme-images-path>
        </color-scheme>
        ...
</theme>
```

Color Schemes

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

In your *liferay-look-and-feel.xml* (located in WEB-INF), you would specify the class names like so:

```
<theme id="my_theme" name="My Theme">
        <root-path>/my_theme</root-path>
        <templates-path>${root-path}/templates</templates-path>
        <images-path>${root-path}/images</images-path>
        <template-extension>vm</template-extension>
        <color-scheme id="01" name="Blue">
                <css-class>blue</css-class>
                <color-scheme-images-path>
                        ${images-path}/color_schemes/${cssclass}
                </color-scheme-images-path>
        </color-scheme>
        <color-scheme id="02" name="Green">
                <css-class>green</css-class>
        </color-scheme>
</theme>
```

Inside of your *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 *green.css* and let the default styling handle the first color scheme, or you could have both *blue.css* and *green.css*.

Now, inside of your *custom.css*, you would place the following lines:

```
@import url(color_schemes/blue.css);
```

```
@import url(color_schemes/green.css);
```

You can identify the styling for the CSS by using prefixes. In *blue.css* you would prefix all of your css styles like this:

```
.blue a {color: #O6C;}
.blue h1 (border-bottom: 1px solid #O6C}
```

And in *green.css* you would prefix all of your CSS styles like this:

```
.green a {color: #O6C;}
.green h1 (border-bottom: 1px solid #O6C}
```

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.

- **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**

The value must be a comma separated list of valid bullet styles to be used. The default is an empty list. The navigation portlet will not show any option in the portlet configuration screen.

Example:

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

The bullet styles referred to in the setting are defined in any of the CSS files of the theme following this 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 cool:

```
<div class="nav-menu nav-menu-style-cool">
       <h3><a href="/web/guest/community">Community</a></h3>
       <ul class="layouts">
              <li class=""><a class=""
href="/web/guest/community/documentation">Documentation</a></li>
              <li class=""><a class="" href="http://wiki.liferay.com" target="_blank">
Wiki</a></li>
              <li class=""><a class="" href="/web/guest/community/forums"> Forums</a></li>
       </ul>
</div>
```

Using your CSS skills and, if desired, some unobtrusive Javascript it's possible to implement anytype of menu.

Deployment

You will notice that when your project was created in the Plugins SDK, an ant script was also created for it. To deploy a plugin, you run the deploy ant task in your project.

```
ant deploy
```

This task will compile your plugin (theme or portlet), store it in a dist folder, and deploy your plugin to your local Liferay installation.

This is done by copying the plugin .war file to your Liferay hot deploy folder. If your local installation of Liferay is running, your plugin will be automatically picked up by the server and deployed. Watch your Liferay console for messages. When you see

```
<plugin> registered successfully.
```

in the console, your plugin has been deployed to the server and is ready for use.

If your plugin is a portlet, you can add it to a page by hovering over the Dock and clicking **Add Content**. Find your portlet in the category you specified in your *liferay-display.xml* file. If you have not yet customized the file, your portlet will be in the Samples category. Simply click the Add button next to it to add it to the page you are currently viewing.

If your plugin is a theme, you can choose it for the page you are viewing by hovering over the Dock and clicking **Manage Pages**. Go to the Look and Feel tab and your theme should be in the list. Select it and it will be applied to the page you are viewing. You can then click the **Return to Full Page** link and see your theme applied to the full page.

Advanced Development

Service Builder

The Service Builder is a tool built by Liferay to automate the creation of interfaces and classes that are used by a given portal or portlet.

Define the Database Structure

Start by creating a *service.xml* file in */docroot/WEB-INF* and type the following:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 5.0.0//EN"
"http://www.liferay.com/dtd/liferay-service-builder_5_0_0.dtd">

<service-builder package-path="com.sample.portlet.library">
	<namespace>Library</namespace>
	<entity name="Book" local-service="true" remote-service="false">

		<!-- 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. Tables 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.

Generate Service Layer

Open up a command prompt, navigate to your portlet's root folder and type the following command.

```
ant build-service
```

The service layer has been generated successfully when you see "BUILD SUCCESSFUL."

Create the Service Layer Class

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 the code, you must regenerate the service layer class

Navigate to the root folder of your portlet, and type

```
ant build-service
```

You can now add a new book to the database by making the following call

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

Security and Permissions Service

Liferay Portal implements a fine-grained permissions system, which developers can use to implement access security into their custom portlets, giving administrators and users a lot more control over their portlets and content. This section of the document will provide a reference for implementing

security into custom portlets.

- **Overview**

Adding fine-grained 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 into the permissions system. This is also known simply as "adding resources."
3. **A**ssociate the necessary permissions to these resources.
4. **C**heck permission before returning resources.

- **Implementing Permissions**

In this section, each of the four main steps in adding Liferay's security feature into custom portlets (built on top of the Liferay portal) will be explained. The following are two definitions that are important to remember.

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 the permission for a portlet resource is implemented a little differently from the 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.

For your custom portlet, Liferay portal needs to know whether there are resources that require permission and whether there are custom permissions. The default configuration is encapsulated in an XML file found in the portal source under the /portal-impl/classes/resource-actions directory; you might use it as a reference to create a similar file for your portlet. There is also a Sample Permissions portlet available in the Plugins project on SourceForge. If your portlet only needs the view and the configuration permission, and that the portlet doesn't use any models with permission, then you do not need to create this XML file. The reason is that all portlets in Liferay automatically inherit these permissions. However, if your portlet does have custom permission and/or uses models that have custom permissions, then you will need to create an XML file defining the resources and actions. Let's take a look at blogs.xml in portal/portal-impl/classes/resource-actions and see how the blogs portlet defined these resources and actions:

```
<?xml version="1.0"?>
<resource-action-mapping>
	<portlet-resource>
```

```
		<portlet-name>33</portlet-name>
		<supports>
			<action-key>ADD_ENTRY</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>ADD_ENTRY</action-key>
		</guest-unsupported>
	</portlet-resource>
	<model-resource>
		<model-name>com.liferay.portlet.blogs.model.BlogsEntry</model-name>
		<portlet-ref>
			<portlet-name>33</portlet-name>
		</portlet-ref>
		<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>
	</model-resource>
</resource-action-mapping>
```

In the XML, the first thing defined is the portlet itself. Right under the root element <resource-

action-mapping>, we have a child element called <portlet-resource>. In this element, we define the portlet name, which is 33 in our case. Next, we list all the actions this portlet supports under the <supports> tag. Keep in mind that this is at the portlet level. To understand what should be listed here, developers should ask themselves what actions belong to the portlet itself or what actions are performed on the portlet that may require a security check. In our case, users need permission to add an entry (ADD_ENTRY), configure blogs portlet settings (CONFIGURATION), and view the blogs itself (VIEW). Each of these supported permissions is within its own <action-key> tag. After we've defined all the actions that require a check, we move on to define some of the default permission settings. The community-defaults tag defines what actions are permitted by default for this portlet on the community (group) page upon which the portlet resides. To put it another way, what should a user that has access to the community in which this portlet resides be able to do minimally? For the blogs portlet, a user with access to the community containing the blogs portlet should be able to view it. Likewise, the guest-defaults tag defines what actions are permitted by default to guests visiting a layout containing this portlet. So if a guest has access to the community page that contains a blogs portlet, the guest should, at the very least, be able to view the portlet according to blogs.xml (not necessarily the content of the portlet). Otherwise, the guest will see an error message within the portlet. Depending on your custom portlet, you may add more actions here that make sense. The guest-unsupported tag contains actions that a visiting guest should never be able to do. For example, the guest visiting the blogs portlet should never be able to add a blog entry since the blog belongs to either a user or a group of users. So even if a user wants to grant guests the ability to add a blog entry to her blog, there is no way for her to grant that permission because the blogs.xml doesn't permit such an action for guests.

After defining the portlet as a resource, we move on to define models within the portlet that also require access check. 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 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 in section 3.1, 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 on your way to implement security to your custom Liferay portlets!

2. Customizing Liferay: Developing in the Extension Environment

This section is intended as a guide for those who want to do extensive customization on top of Liferay Portal. The extension environment, also known as the ext environment, is a set of tools that allow developers to build portals and portlets on top of Liferay Portal. It can be seen as a Software Development Kit that is independent of any IDE but integrates well with all of them thanks to its usage of ant, the most popular and supported build tool. Another way to think about the extension environment is as a wrapper for Liferay's core source because, in most cases, it mirrors Liferay's core source directories (i.e. ext-impl/ for portal-impl/, ext-web/ for portal-web/). It allows you to develop on top of Liferay portal, like a platform, providing help when an upgrade to a new version of Liferay Portal is needed. The following instructions explain how to use the environment and are meant to keep custom code separated from the Liferay Portal code so that upgrades can be made easily.

The following instructions will help you get your development environment ready for working with the source code. These instructions are specific to setting up for deployment to Tomcat 5.5 using a MySQL database and developing with Java JDK 1.5. Liferay Portal version 5 requires JDK 1.5 or above, while 4.x is also compatible with Java 1.4. All versions work with a wide array of databases, application servers and containers. You will need to adjust your development environment according to your platform.

Tools Setup

Before we can get started, the following components must be installed on your machine.

Java Development Kit

1. Download and install JDK 1.5 or above at http://java.sun.com/j2se/1.5.0/download.jsp
2. Set an environment variable called **JAVA_HOME** pointing to your JDK directory

ANT

1. Download and unzip the latest version of ANT from http://ant.apache.org/
2. Set an environment variable called **ANT_HOME** to point to your Ant directory
3. Add **ANT_HOME\bin** to your **PATH** environment variable

MySQL

Download and install MySQL from http://www.mysql.com/

Liferay Source

1. Download Liferay source from Liferay's download page at
2. Unzip the source code into a folder called *portal*

Alternatively, you can download the source code from from Liferay's subversion repository at

```
https://lportal.svn.sourceforge.net/svnroot/lportal/portal/branches/5.0.x/
```

Setup Your Application Server and Liferay Portal

The first thing we need to do install Liferay Portal on your application server. You can download and install the application server and Liferay portal separately, however, the easiest thing thing is to download a Liferay Portal bundle. The Liferay Portal bundles can be obtained from http://www.liferay.com/web/guest/downloads/portal. The examples in this guide will assume you are using Tomcat 5.5 bundle, but you can download the bundle that best fits your need.

Once you have downloaded the file, unzip it to a location of your choice. The example in this guide will assume you have unzipped the file to

```
D:/Projects/liferay/tomcat
```

Create the Extension Environment

Go to the root directory of the Liferay Portal sources and create a file called *release.${user.name}.properties* where *${user.name}* is the name of the user in your computer. Be sure to write it in the proper letter cases (even in Windows). Set the property *lp.ext.dir* to the directory where you would like to create the directory (make sure that it does not exist yet). If you expect to be using Eclipse, add also a property called *lp.eclipse.project.name* with the desired name of the project and *lp.eclipse.dir* with the desired Eclipse work space: For example:

```
lp.ext.dir=D:/Projects/liferay/portal/ext
lp.eclipse.dir=D:/Projects/liferay/eclipse/workspace
lp.eclipse.project.name=liferay-ext
```

After creating the file, execute the following ant targets to build the sources and create the

extension directory structure:

```
ant start
ant build-ext
```

You can now go to the directory specified and see the directory structure that has been created. If you use some type of version control repository such as CVS or Subversion you can upload the directory to it. In the next steps you will be creating build products that should not be uploaded to the version control repository.

The rest of the document will be referring to the root directory where the extension environment has been download as /ext.

Configuring the Environment

The extension environment is built around a set of ant build scripts that are highly configurable through properties files and build scripts. You should create customized versions of the properties files by creating a version with the same name than the original one but inserting your user name before the extension. The two files that should be customized are *app.server.properties* and *build.properties*.

Create a file customized *app.server.properties* files be creating a *app.server.$(user.name}.properties* where *${user.name}* is the name of the user in your computer. Set the following property:

```
app.server.type=tomcat
app.server.tomcat.dir=D:/Projects/liferay/tomcat
```

Read *app.server.properties* to find other properties whose values you may want to override.

Create a file customized build*.properties* files by creating a build*.$(user.name}.properties* where *${user.name}* is the name of the user in your computer. In this file you can set the compiler you want to use (the default is *modern*), the amount of memory that it will have, some extra class paths, specific WARs that you may want to deploy, etc. Here is an example that sets these properties:

```
javac.compiler=modern
javac.debug=off
javac.memoryMaximumSize=128m
classpath.ext=/sharedlibs/mycompanylib.jar
deploy.specific.wars=/sharedportlets/mycompanyportlet.war
```

You can also set the property *jsp.precompile* to on in *build.properties* to precompile JSPs if you are using Jetty, JBoss+Jetty, JBoss+Tomcat, JOnAS+Jetty (other AP. servers do not support this functionality). This takes a few minutes and should only be used when deploying to a production server. Read *build.properties* to look for other properties that you might be interested in overriding.

Setup the Database

Navigate to your application server find the *conf/Catalina/localhost* folder. Open ROOT.xml. Comment out the Hypersonic database resource tag and uncomment the MySQL database resource tag. A resource tag looks like the following:

```
<Resource
       name="jdbc/LiferayPool"
       auth="Container"
       type="javax.sql.DataSource"
       driverClassName="com.mysql.jdbc.Driver"
       url="jdbc:mysql://localhost/lportal?useUnicode=true&amp;characterEncoding=UTF-8"
       username=""
       password=""
       maxActive="20"
/>
```

Be sure the set the *username* and/or *password* if needed.

Deploy the Extension Environment

Navigate to your /ext folder and issue the following command

```
ant clean deploy
```

Start up your application server. If you are using Tomcat, navigate to the *bin* folder and type

```
startup
```

Liferay will start as usual, but since no modification has been made in the extension environment, you won't notice any difference.

Customizing Liferay

File Structure Overview

The following sections explain the purpose of each of the subdirectory and the tasks that can be performed from them.

/ext

The root directory. From this directory you can build and deploy the whole application to the application server by running ant deploy. You should not place any extra files here.

/ext/classes

This directory is for internal use and can be ignored.

/ext/ext-impl

This is the folder that will contain all your sources and configuration files (except those related to the web application). Upon creating of the ext environment several important files will be placed in its subdirectories. The most significant are:

- */ext/ext-impl/classes/portal-ext.properties*: this file can be used to override the values of the properties in the portal.properties configuration file that ships with Liferay Portal.
- */ext/ext-impl/classes/system-ext.properties*: this file can be used to override the values of the

properties in the system.properties configuration file that ships with Liferay Portal.

- */ext/ext-impl/classes/content/Language-ext.properties*: this file can be used to add you're own internationalized text messages or to override the messages that ship with Liferay Portal. You can add variations for other languages using the Java convention for message bundles. For example the translation to Spanish should be named Language-ext_es.properties.

Run *ant deploy* from */ext/ext-impl* to compile your source and to deploy your classes to your specified application server's deployment directory.

/ext/ext-lib

Place any extra dependent libraries in here. They will be copied over to the deployment directory when you run ant deploy from /ext.

/ext/ext-service

This folder contains the classes that are generated by Liferay's service builder. You can read more about Liferay's service builder in the Advanced Development section.

/ext/ext-web

This directory will contain your JSPs, HTMLs, images, and all the web application related files inside the docroot subdirectory. Here are some of the most common activities you will be performing from this directory:

- Run ant deploy to deploy changes in the directory. The script will copy everything from */ext/ext-web/docroot* over to */ext/ext-web/tmp* and then copy everything from */ext/ext-web/tmp* to the deployment directory of your specified application server. Do not manipulate the contents of */ext/ext-web/tmp* manually. This provides an easy way to extend the portal without changing the source and makes upgrading very easy.
- Run ant fast-deploy to deploy only changed JSPs. Use this version for quick deployment in the process of developing a JSP or set of JSPs
- To add entries to the web application configuration file edit */ext/ext-web/docroot/WEB-INF/web.xml.*
- To add a portlet, edit */ext/ext-web/docroot/WEB-INF/portlet-ext.xml*, */ext/ext-web/docroot/WEB-INF/liferay-portlet-ext.xml*, */ext/ext-web/docroot/WEB-INF/liferay-display.xml* and */ext/ext-impl/classes/content/Language-ext.properties*. These *-ext.xml files are read after their parent files are read and override their parent values. Note that we do not recommend adding portlet in the ext environment; it is far better to write a portlet plugin instead.
- If you are using StrutsPortlet to develop your portlets, the following files will also be of interest to you: */ext/ext-web/docroot/WEB-INF/struts-config.xml* and */ext/ext-web/docroot/WEB-INF/tiles-defs.xml.* Again, it is far better to write a hot deployable Struts Portlet plugin than to add it to the extension environment.

/ext/lib

This directory contains all of the portal's deployment time dependent libraries and compile time dependent libraries. It contains the following subdirectories:

- development: contains the libraries that will be used during development but will not be deployed. It also contains some JDBC drivers that you can manually deploy to the application server for the specific database in use.
- global: contains the libraries that will be deployed to the global library directory of the application server.
- portal: contains the libraries that will be placed inside the portal EAR.

/ext/modules

This folder contains the Liferay portal source code in .war or .jar format.

/ext/sql

This directory contains all of the portal's database scripts.

/ext/tools

This directory is for internal use and can be ignored.

Advanced Development

Liferay Services

Portlet applications may invoke the services provided by Liferay Portal by using the portal-client.jar client library.

User Service

The User service allows the management of the portal user and it's communities (aka Groups), Roles and UserGroups. It can be accessed through the static methods of UserServiceUtil. Following is a description of it's most important methods:

```
public static com.liferay.portal.model.User addUser(long companyId,
       boolean autoPassword, java.lang.String password1,
       java.lang.String password2, boolean autoScreenName,
       java.lang.String screenName, java.lang.String emailAddress,
       java.util.Locale locale, java.lang.String firstName,
       java.lang.String middleName, java.lang.String lastName, int prefixId,
       int suffixId, boolean male, int birthdayMonth, int birthdayDay,
       int birthdayYear, java.lang.String jobTitle, long[] organizationIds,
       boolean sendEmail)
       throws com.liferay.portal.PortalException,
              com.liferay.portal.SystemException, java.rmi.RemoteException;
```

Add a new user inserting in its profile the provided information

```
public static com.liferay.portal.model.User updateUser(long userId,
       java.lang.String oldPassword, java.lang.String newPassword1,
       java.lang.String newPassword2, boolean passwordReset,
       java.lang.String screenName, java.lang.String emailAddress,
       java.lang.String languageId, java.lang.String timeZoneId,
       java.lang.String greeting, java.lang.String comments,
       java.lang.String firstName, java.lang.String middleName,
       java.lang.String lastName, int prefixId, int suffixId, boolean male,
       int birthdayMonth, int birthdayDay, int birthdayYear,
       java.lang.String smsSn, java.lang.String aimSn, java.lang.String icqSn,
       java.lang.String jabberSn, java.lang.String msnSn,
       java.lang.String skypeSn, java.lang.String ymSn,
       java.lang.String jobTitle, long[] organizationIds)
       throws com.liferay.portal.PortalException,
              com.liferay.portal.SystemException, java.rmi.RemoteException;
```

Update a user's profile with the provided information.

```
public static void addGroupUsers(long groupId, long[] userIds)
       throws com.liferay.portal.PortalException,
              com.liferay.portal.SystemException, java.rmi.RemoteException;
```

Add a set of users to a give community (aka Group) identified by the groupId.

```
public static void addRoleUsers(long roleId, long[] userIds)
              throws com.liferay.portal.PortalException,
                     com.liferay.portal.SystemException, java.rmi.RemoteException;
```

Add a set of users to a give Role identified by the roleId.

For more information check the Portal Javadocs.