LIFERAY PLUGIN DEVELOPMENT GUIDE

Extending Liferay with Portlets and Themes

Richard L. Sezov, Jr.

Liferay Plugin Development Guide
by Richard L. Sezov, Jr.

Portions reworked from the Themes document at http://wiki.liferay.com/index.php/Themes. Please see the *History* tab there for author credit for this material.

Table of Contents

1. Introduction

This guide is a quick start 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 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.

What is a Portlet?

Portlets are small web applications that run in a portion of a web page. The heart of any portal im-

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 portlet specification. A JSR-168 standard portlet should be deployable on any JSR-168 portlet container. 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.

What is a Theme?

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).

2. 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. You will see that it has the following directory structure:

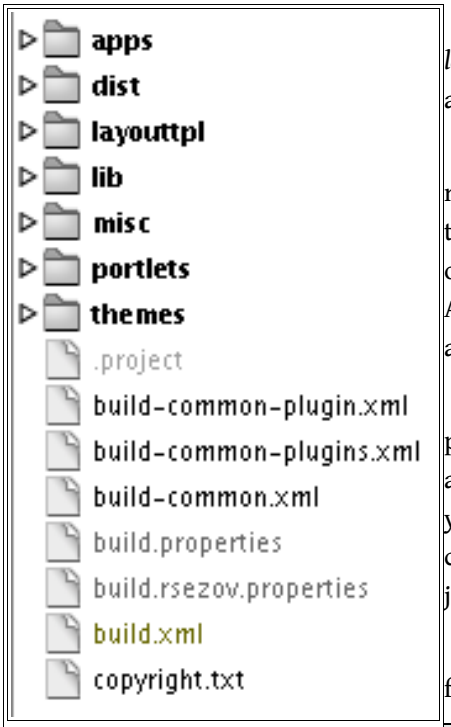

Illustration 1: Plugins SDK folder structure

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.username.properties*, where *username* 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:

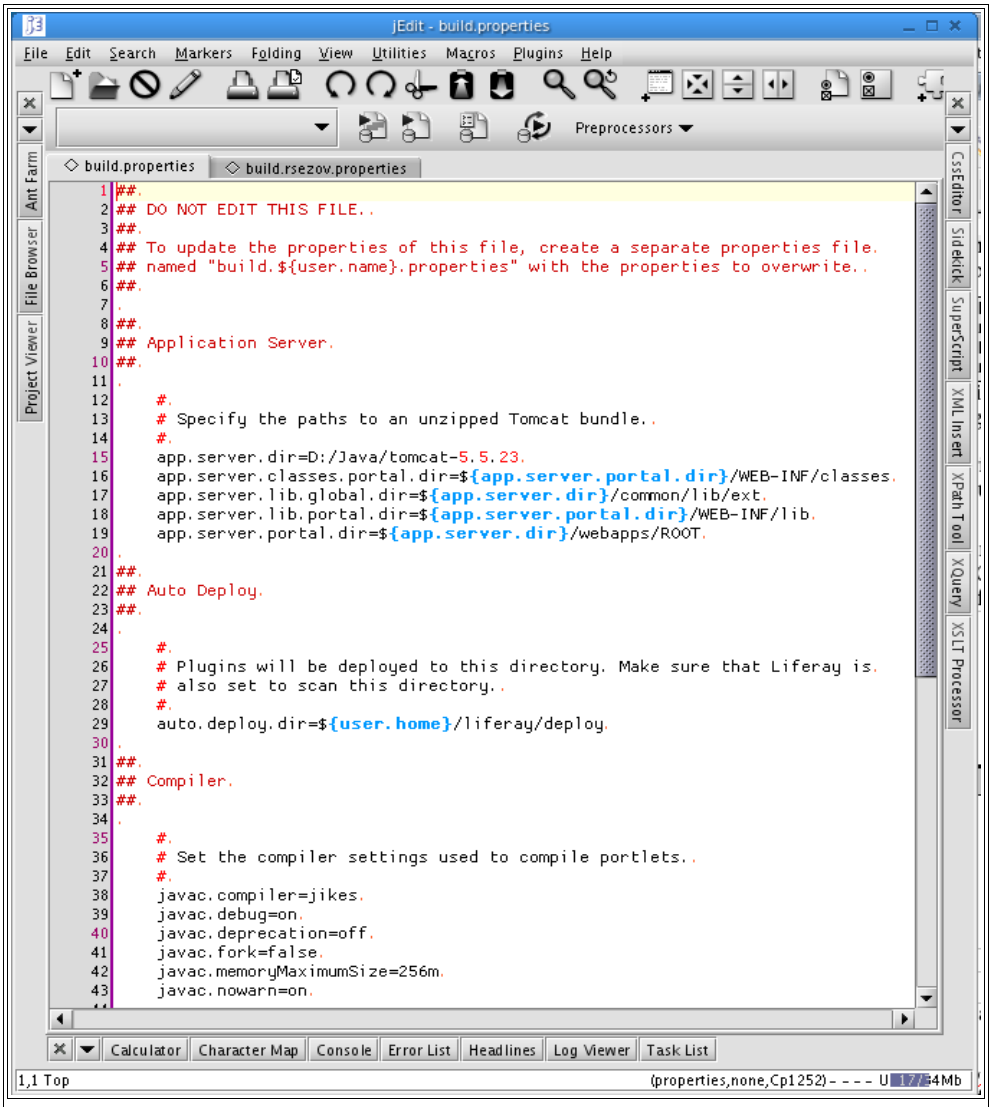

Illustration 2: The build.properties file

```
app.server.dir=
auto.deploy.dir=
app.server.lib.portal.dir=
app.server.portal.dir=
java.compiler=
```

app.server.dir: This is the folder into which you have installed your development version of Liferay.

auto.deploy.dir: This is the folder into which plugins should be placed in order for them to be hot deployed to Liferay. By default, this folder is in *$HOME/liferay/deploy*.

app.server.lib.portal.dir: This folder is where Liferay's libraries are installed. If you are using the Liferay-Tomcat bundle, for example, you would set this to *$*

{app.server.dir}/webapps/ROOT/WEB-INF/lib.

app.server.portal.dir: This folder is the folder to which Liferay is installed inside of your application server. For the Liferay-Tomcat bundle, set this to *${app.server.dir}/webapps/ROOT*.

java.compiler: The default value for this is the Eclipse compiler, ECJ. ECJ is an alternate Java compiler which performs very fast. If you do not have ECJ installed, the ant script will install it for you by copying *ecj.jar* to your Ant folder. If you do not wish to use ECJ to compile your code, you can set this to *modern*. This will cause the ant scripts in the plugins SDK to use the default Java compiler from your JDK.

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

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.

3. Portlets

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. On Linux and Mac, from the *portlets* directory, enter the following command:

```
./create.sh <project name> "<portlet title>"
```

For example, to create a portlet with a project folder of *hello-world* and a portlet title of *Hello World,* type:

```
./create.sh hello-world "Hello World"
```

On Windows, you would type:

```
create.bat hello-world "Hello World"
```

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. At this point, if you wish, you can check your Plugins SDK into a source code repository in order to share your project with others.

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 Project

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

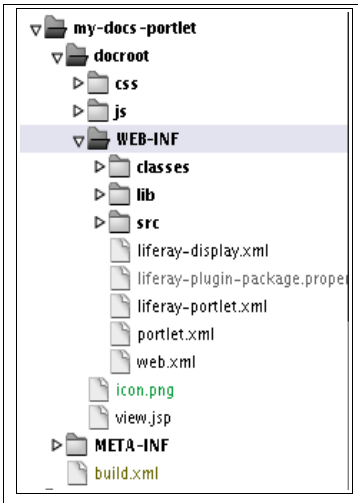

Illustration 3: Portlet project structure

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 illustration to the left. 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. These 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 Content* 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.

You can also check out our *Portlet Development Guide,* which you can find in the same location you found this document. That guide gives you step-by-step instructions for creating portlets.

4. Themes

Creation of themes is done in a similar manner to the creation of portlets. There is a *themes* folder inside the plugins SDK where all new themes reside. To create a new theme, you run a command in this folder similar to the one you used to create a new portlet. For Linux and Mac, type:

```
./create.sh <project name> "<theme title>"
```

For example, to create a theme with a project folder of *hello-world* and a theme title of *Hello World,* type:

```
./create.sh hello-world "Hello World"
```

On Windows, you would type:

```
create.bat hello-world "Hello World"
```

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-im-
ages-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-im-
ages-path>
  </color-scheme>
  ...
</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* (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>${images-path}/color_schemes/${css-
class}</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 any type of menu.

For further information about themes, please see http://wiki.liferay.com.

5. 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. 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 Liferay installation is running in debug mode inside of your IDE, you can set breakpoints in your code and your debugger will stop the execution of the portlet in the specified place so you can step through the code. If you find a problem, fix the problem and then run the *deploy* ant task again to redeploy your portlet to the server for further testing.

If your plugin is a theme, you can choose it for the page you are viewing by hovering over the Dock and clicking *Page Settings*. 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 *Go back to full view* link and see your theme applied to the full page.

When you are finished implementing your features and debugging your code, create a final build for it by running the *deploy* ant task again. The plugin will reside in a *dist* folder in the project folder. You can then take the .war file stored there and deploy it to any running Liferay portal. All that is necessary for this is to copy the file into the hot deploy folder defined for the Liferay portal server. The file can also be

uploaded to a server housing a Liferay Software Catalog. Your portal administrator can then point the Plugin Installer portlet to this software catalog and through a simple point and click interface install any plugins that the development team makes available on their Software Catalog. For more information on how to configure this, please see the *Liferay Administration Guide*.

Conclusion

You can see that the Plugins SDK provides a full development lifecycle for both portlet projects and theme projects. It is tool-agnostic, so developers are free to use the tools of their choice to create portlet projects and theme projects. It provides a method to create and deploy theme projects and portlet projects with little effort and which supports the majority of application server platforms. And finally, it provides a structure to group the portlets and themes that go together with an overall portal project. We hope that you will enjoy using the Plugins SDK and that it will be a useful tool for you to create your portlets and your themes.

And remember, if you create open source portlets and themes, you can make them available on Liferay's community Software Catalog (http://www.liferay.com/web/guest/community/community_plugins).

Happy coding!