05

Module 5

Working with Portlet Modules

Liferay

Table of Contents

Module 5: Working with Portlet Modules

Learning Objectives

In this module, you'll understand Portlet Standard concepts like the portlet lifecycle, learn how to create liferay portlet modules, and learn how to set portlet properties.

Tasks to Accomplish

- Implement a basic portlet using Liferay's MVCPortlet

Exercise Prerequisites

- Java JDK Installed to Run Liferay
  - Download here: https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html
  - Instructions on Installation here: https://www.java.com/en/download/help/download_options.xml
- Unzipped module exercise files in the following folder structure:
  - Windows: `C:\liferay`
  - Unix Systems: `[user-home]/liferay`
- Liferay Developer Studio installed with a workspace selected
  - For installation instructions, see module 1
- Be prepared to use the code snippets found in the module's `exercise-src` folder
  - Code snippets for a particular exercise will be found in the folder with the corresponding exercise number: `exercise-[##]`
  - Snippets are named after the exercise set where they are used: `snippet-[mod#]-[exercise#]-[exercise-set-heading]-[optional-step#]`

Java Standard Portlet

Liferay's user interface relies on portlets but not strictly on the standard portlet specification. Although there are many ways of building out your application's user interface, using the MVC pattern along with portlets as the implementation of the Controller and View Layer is still the essence of Liferay development.

Understanding key portlet concepts and the portlet lifecycle is fundamental to learning back-end development in Liferay and fundamental to customizing Liferay's controller layer, discussed in *Module 11 - Override Controller Actions*.

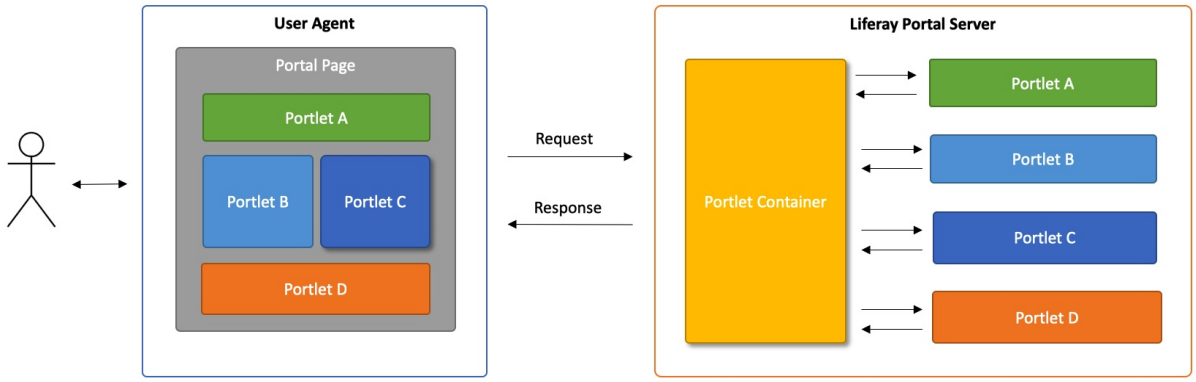
> In Liferay 7.1, the portlet applications were named *Widgets*. Although that term is used everywhere in the user interface, internally, the applications still rely on the portlet paradigm. Portlet is the term you as a back-end developer will encounter when researching Liferay's source code. In the context of this material, we'll be using the term "portlet" unless otherwise needed.

What is a Portlet?

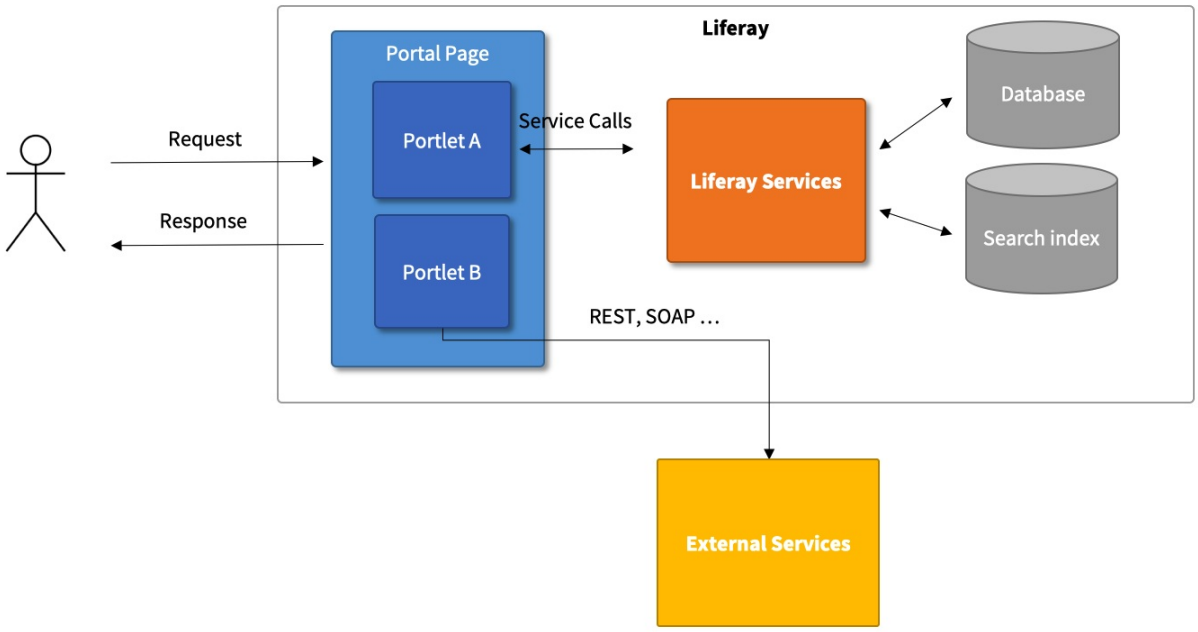
A portlet is a web component or application that produces an HTML fragment of a page. Generally, any application that has a user interface in Liferay uses a portlet. To summarize, a portlet is:

- An application running in a portlet runtime environment called a **portlet container**
- An application in Liferay that has a **user interface** (not all user interfaces are portlets, though)
- An application that follows the standards governed by the Portlet Specification JSR-168: https://www.jcp.org/en/jsr/detail?id=168, JSR-286: https://www.jcp.org/en/jsr/detail?id=286, or JSR-362https://www.jcp.org/en/jsr/detail?id=362.

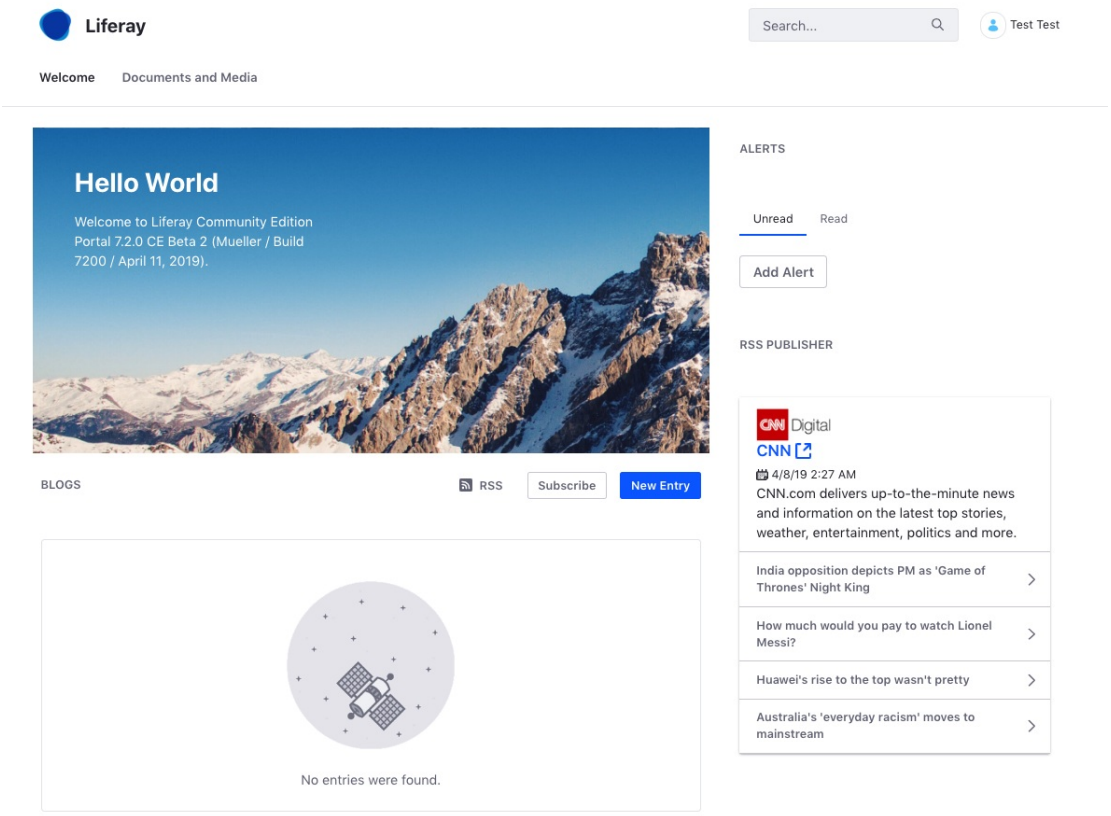
The diagram below demonstrates a user's interaction with portlets on a portal page. When users come to a page in Liferay, they can see the portlets on the page. When a user interacts with a portlet, a request may be sent to Liferay's portlet container, which handles the request and sends a response back. Each one of the portlets on the page is contained and managed by the portlet container in Liferay.



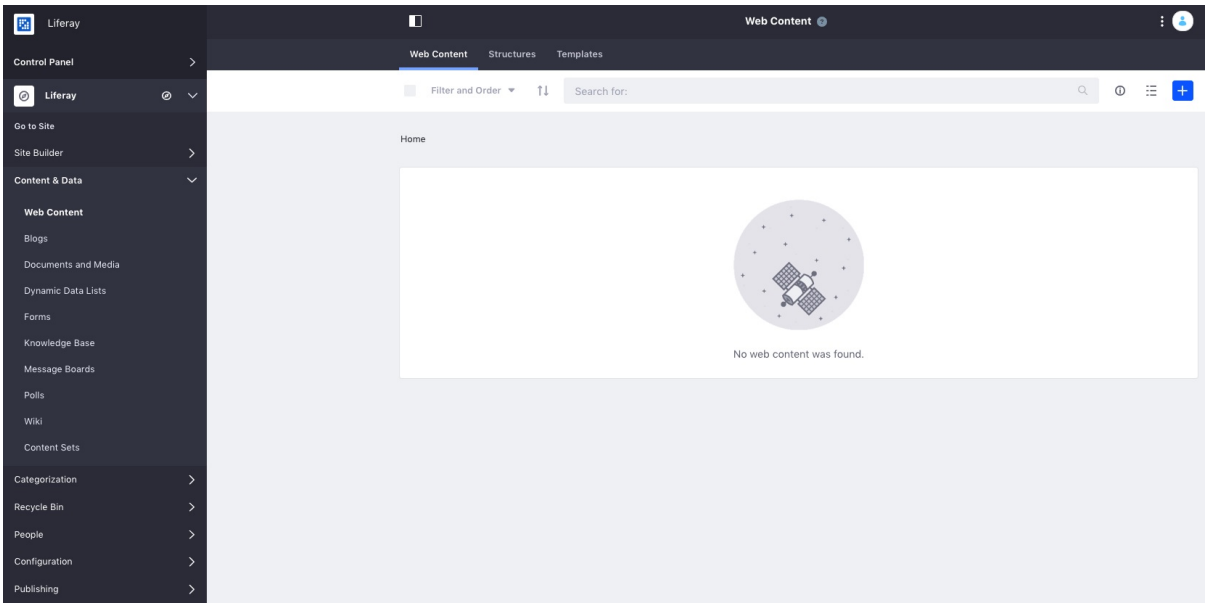
Portlets typically work as a bridge between the user interface and service layer. For example, you submit personal data from the user interface, and the portlet processes it to be sent to the service layer. The service layer then sends feedback, which the portlet provides back to the user interface.



On a Liferay page, the UI elements we see are implemented as portlets. The Blogs, Alerts, Hello World, and RSS Publisher are all portlets on the page:



Elements within the various parts of the *Product Menu* and *Control Panel* are also implemented as portlets. Creating Web Content for example, happens through the Web Content portlet:



Building Blocks of a Standard Portlet

The first key element of a standard Java portlet is a **deployment descriptor**. Two deployment descriptors are necessary:

- **web.xml**: used to declare servlet filters, taglibs, and other configuration parameters
- **portlet.xml**: used to declare various properties like the portlet class

The second key element is the **portlet class** (or Java class) that extends a specific implementation of the Portlet Specification. The default code implementation of a Java Standard javax.portlet.Portlet interface is the javax.portlet.GenericPortlet.

> When creating a Java Standard Portlet, you always extend `javax.portlet.GenericPortlet`.

The third key element of a Java standard portlet is **JSP** files. Most of the time, when we want to see something displayed in a portlet application, we use JSP files. Portlets, by default, come with different modes. With each mode, a JSP can be displayed based on the mode that is selected.

Now that we have the building blocks of a portlet, we're going to take a look at the key functional concepts.

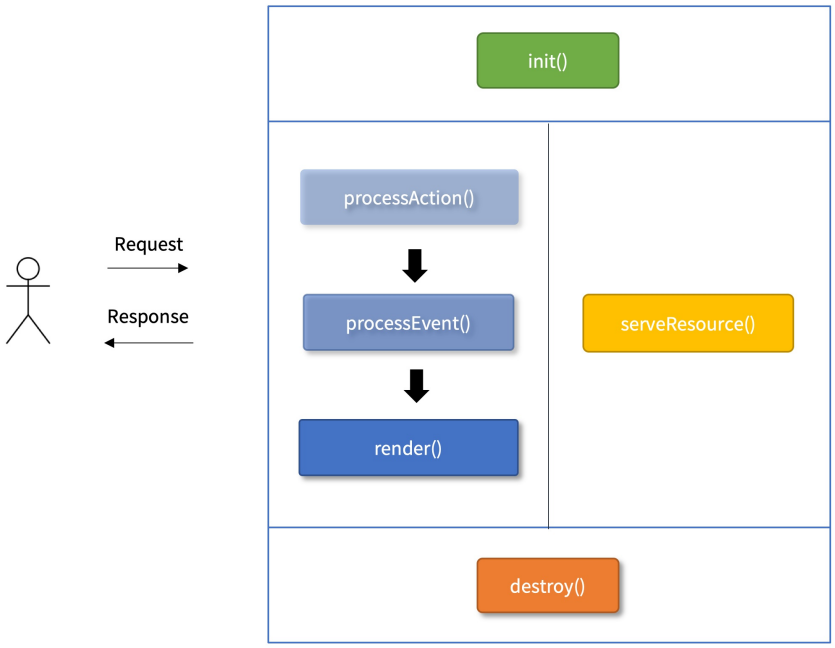
Portlet Lifecycle

The Portlet Lifecycle defines how portlets should handle specific types of requests. The portlet container is the one that will handle the requests that come in and determine what phase of the portlet's lifecycle should be invoked.

There are six phases of the portlet lifecycle:

- Init Phase
- Action Phase
- Event Phase
- Render Phase
- Resource Serving Phase
- Destroy Phase

Each phase of the Portlet Lifecycle has a corresponding method in `javax.portlet.GenericPortlet`.



Init Phase

The Init phase of the portlet is called when the portlet class initializes, or, in other words, when the portlet is deployed. During the Init phase, initialization parameters are read. This phase only occurs once.

Render Phase

The Render phase is used to generate an **HTML fragment**. In order to have something displayed on a JSP, this phase has to be invoked. If the page ever changes or is refreshed, all the portlets on a page will have to go through the Render phase again.

This phase along with the action phase is part of what's known as the **request response lifecycle**. When interacting with this phase, there are wrapper objects, **RenderRequest** and **RenderResponse**, that are used to store and retrieve various attributes that help with our development and control over what's happening throughout the Render phase and other phases.

In the `javax.portlet.GenericPortlet`, this phase is handled by the `render()` method.

Calling the Render Phase Example

The Render phase is automatically called, for example, when a portlet is added to a page, when the page is loaded or reloaded or after the Action phase is finished. When we want to display different JSPs, we usually manually call the Render phase.

In the example below, we first create a Render URL and assign that to a link, so that when the link is clicked, the Render phase is invoked:

```
<portlet:renderURL var="viewEntryUrl">
    <portlet:param name="entryId" value="<%= String.valueOf(entry.getEntryId()) %>" />
</portlet:renderURL>

<a href="<%= viewEntryUrl">Click here to view the entry</a>
```

When the Render URL is invoked, it creates a URL. The URL contains parameters like which portlet is invoking the phase, what lifecycle phase is being invoked, and other information about the portlet:

```
http://localhost:8080/web/guest/home?p_p_id=com_liferay_blogs_web_portlet_BlogsPortlet&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&_com_liferay_blogs_web_portlet_BlogsPortlet_entryId=34403
```

In the `render()` method of a portlet's Java class, we can retrieve parameters that were set in the JSP and stored in the **renderRequest**, and then, for example, call services using those parameters.

In the example below, we first retrieve the *entryId* parameter, then fetch the entry, and, lastly, set the object back to the request and send it over to the JSP to be displayed:

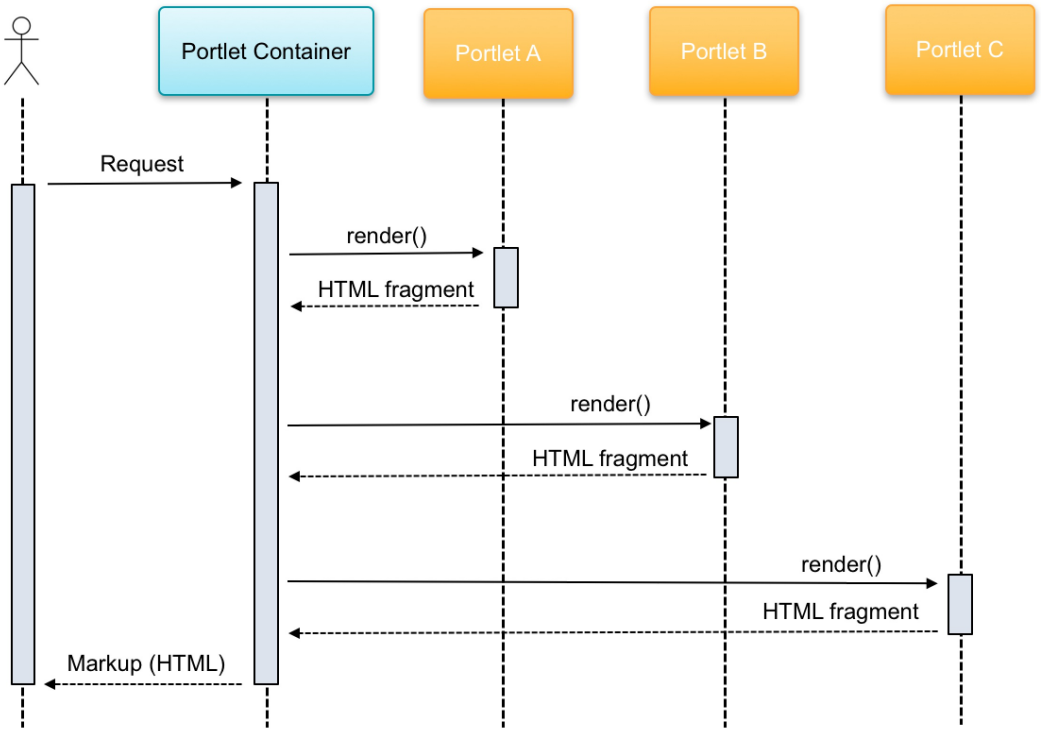
```
@Override
public void render(RenderRequest renderRequest, RenderResponse renderResponse)
    throws IOException, PortletException {

    String entryId = ParamUtil.getString(renderRequest,"entryId");

    MyEntry entry = myService.getEntry(entryId);

    renderRequest.setAttribute("entry", entry);

    super.render(renderRequest, renderResponse);
}
```

Render Phase Flow

The Render phase is always invoked for all portlets on the page after the Action phase (which may or may not exist). It’s important to note that the order in which the Render phase of the portlets in a page gets executed is not guaranteed by the portlet specification. Liferay has an extension to the specification

through the attribute `render-weight`. Portlets with a higher render weight will be rendered before those with a lower weight.

After all the portlets go through the Render phase, the collection of the HTML fragments are assembled together in the render response, containing the Markup for the page:



Action Phase

The Action phase is used to respond to actions a user performs. Typically, it's used to handle an HTML form submit.

The corresponding `javax.portlet.GenericPortlet` method is `processAction()`. Like the Render phase, there are **ActionRequest** and **ActionResponse** objects available.

During the Action phase, **events** can be triggered to invoke the Event phase. After the Action phase, the Render phase is called and all the portlets on the page will render.

Calling Action Phase Example

To call the Action phase, an ActionURL has to be created. In the example below, clicking on a *Submit* button invokes the Action phase. The action `name` attribute makes it possible to handle different actions in a single `processAction()` method:

```
<portlet:actionURL name="/blogs/edit_entry" var="editEntryURL" />

<aui:form action="<%= editEntryURL %>" method="post" name="fm">
```

```
    <aui:input name="entryId" type="hidden" value="<%= entryId %>" />
    <aui:input name="title" type="text" />
</aui:form>
```

As with the Render URL, the actionURL will generate a literal URL containing control parameters:

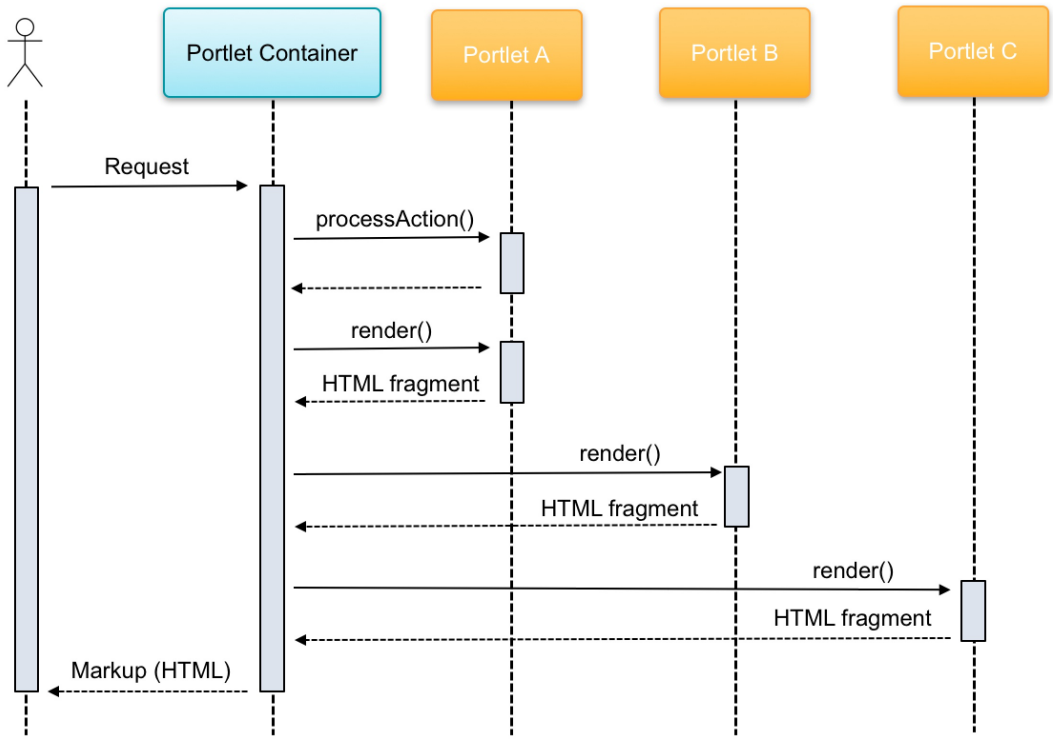
```
http://localhost:8080/web/guest/home?p_p_id=com_liferay_blogs_web_portlet_BlogsPortlet&p_p
_lifecycle=1&p_p_state=maximized&p_p_mode=view&_com_liferay_blogs_web_portlet_BlogsPortlet
_javax.portlet.action=%2Fblogs%2Fedit_entry&_com_liferay_blogs_web_portlet_BlogsPortlet_en
tryId=34403&_com_liferay_blogs_web_portlet_BlogsPortlet_cmd=update&_com_liferay_blogs_web_
portlet_BlogsPortlet_title=Working+Example&p_auth=7uOmzvxB
```

In the example below, we retrieve parameters from the ActionRequest object and update the entry title by doing a service call:

```
@Override
public void processAction(ActionRequest actionRequest, ActionResponse actionResponse)
    throws IOException, PortletException {

    String entryId = ParamUtil.getString(actionRequest,"entryId");
    String title = ParamUtil.getString(actionRequest,"title");

    myService.updateEntryTitle(entryId, title)

    super.processAction(actionRequest, actionResponse);
}
```

Action Phase Flow

Once a user makes an action request, the portlet container will invoke the Action phase for the appropriate portlet. Once the Action phase is finished, the Render phase is invoked, and all the portlets on a page will re-render:



Event Phase

The Event phase was introduced as a way for portlets to communicate with each other. The concept of portlets talking to each other is called **Inter-Portlet Communication** (IPC).

The Event phase is made up of publishers and receivers. For both publishers and receivers, you have to configure the supported events. Event publishing happens in the Action phase, and event processing happens in the Event phase. All portlets configured to process an event will process the event once the event is set.

Once the Event phase is finished, the Render phase will be called. The corresponding method for the Event phase in `javax.portlet.GenericPortlet` is `processEvent()`.

Below is an example of a sender publishing an event `message;http://www.liferay.com` and a receiver receiving it:

Sender

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Event Publisher Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + LifecyclePortletKeys.EVENT_PUBLISHER,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user",
        "javax.portlet.supported-publishing-event=message;http://www.liferay.com",
```

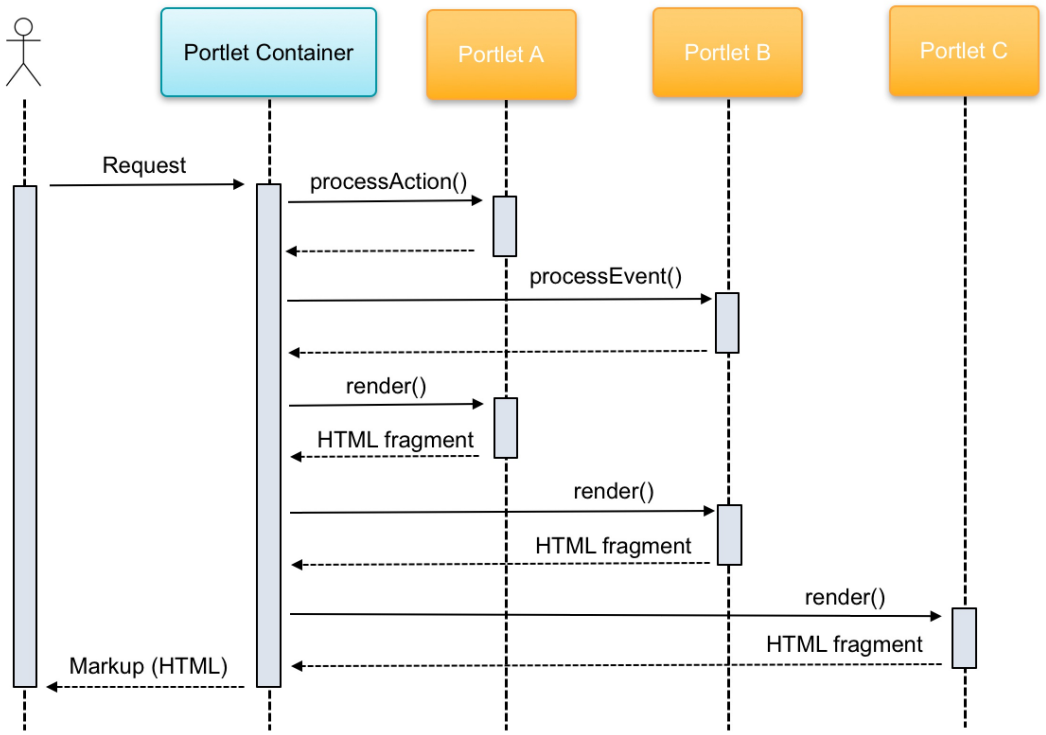
```
    },
    service = Portlet.class
)
public class EventPublisherPortlet extends MVCPortlet {

    @Override
    public void processAction(ActionRequest actionRequest,
        ActionResponse actionResponse)
            throws IOException, PortletException {

        String message = ParamUtil.getString(actionRequest, "message");

        QName qName = new QName("http://www.liferay.com", "message");
        actionResponse.setEvent(qName, message);

        super.processAction(actionRequest, actionResponse);
    }
}
```

Receiver

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Event Receiver Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + LifecyclePortletKeys.EVENT_RECEIVER,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user",
        "javax.portlet.supported-processing-event=message;http://www.liferay.com",
    },
    service = Portlet.class
)
public class EventReceiverPortlet extends MVCPortlet {

    @ProcessEvent(qname = "{http://www.liferay.com}message")
    public void handleProcesseuserEmailAddressEvent(EventRequest request,
        EventResponse response)
            throws javax.portlet.PortletException, java.io.IOException {

        Event event = request.getEvent();
        String message = (String) event.getValue();

        response.setRenderParameter("messageReceived", message);
    }
}
```

Event Phase Flow

A user invokes the Action phase of Portlet A. In the `processAction()` method, the event is set, and Portlet B's Event phase will be invoked. Once the Event phase of Portlet B is finished, all of the portlets on the page will enter the Render phase.



Resource Serving Phase

The Resource Serving phase will provide a way to serve resources to the client without entering the Action or Render phase. Often, the Resource Serving phase is called with Ajax for:

- Autocompletion of a search field
- Refreshing the news content area without a page refresh
- Doing any background operation without a page refresh

Below is an example of a Resource Serving phase being called from a JSP page via Ajax and handled in the `serveResource()` method in the portlet class. Once the method is executed without error, we'll parse the JSON from the response and display it:

JSP

```
<liferay-portlet:resourceURL var="resourceRequestURL" >
    <liferay-portlet:param name="entryId" value="<%=entryId %>" />
</liferay-portlet:resourceURL>

<a href="#" onclick="fetchData()">Fetch Data</a>

<div id="data-element"></div>

<aui:script>
    function fetchData() {
        AUI().use('aui-io-request', function(A) {
            A.io.request('<%= resourceRequestURL %>', {
                method: 'post',
                on: {
```

```
                    success: function() {

                        var data = JSON.parse(response.responseText);
                        A.one('#data-element').html(data);
                    }
                }
            });
        });
    }
</aui:script>
```

Portlet class

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Lifecycle Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + LifecyclePortletKeys.LIFECYCLE,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user",
    },
    service = Portlet.class
)
public class LifecyclePortlet extends GenericPortlet {

    @Override
    public void serveResource(ResourceRequest resourceRequest,
            ResourceResponse resourceResponse) throws IOException,
            PortletException {

        String entryId = ParamUtil.getString(resourceRequest, "entryId");

        String entryData = getEntryData(entryId);

        JSONObject json = JSONFactoryUtil.createJSONObject();
        json.put("entryData", entryData);

        JSONPortletResponseUtil.writeJSON(
                resourceRequest, resourceResponse, json);
    }
    ...
```

Destroy Phase

When a portlet is undeployed, the Destroy phase is invoked to do any clean-up before the portlet is removed. Resources are cleaned up and the portlet itself is released from the portlet container to be eligible for garbage collection. The method that is called when the Destroy phase is invoked is `destroy()`.

Portlet Modes

The portlet specification makes it possible for the portlet to have different perspectives or modes. Each mode, when selected, will render a JSP configured for that mode. The typical naming convention is `view.jsp` for the VIEW mode, `edit.jsp` for the EDIT mode, and `help.jsp` for the HELP mode.

There are three standard modes, each with their respective uses:

- VIEW: Standard mode used as a general point of view
- EDIT: Configuration mode used to customize the behavior of the portlet
- HELP: Displays portlet's help information

Liferay also provides custom modes that can be leveraged:

- About
- Config
- Edit default
- Edit guest
- Print
- Preview

> The only required mode is the `VIEW` mode.

`javax.portlet.GenericPortlet` has a handler method for each of the standard modes. For the VIEW mode, `GenericPortlet` will call `doView()`, the EDIT mode will call `doEdit()`, and the HELP mode will call `doHelp()`. These methods are called from `render()` via `doDispatch()`, which figures out which portlet mode is selected. Once the portlet mode is selected, the corresponding method is called, but it's up to us as the developers to customize `doView`, `doEdit`, and `doHelp` if we are creating a Java standard portlet.

There are two different ways you can set portlet modes, either in the JSP or in the Portlet class:

JSP

```
<portlet:renderURL portletMode="VIEW" var="renderURL" />
```

Portlet class

```
actionResponse.setPortletMode(PortletMode.EDIT);
```

Window States

Window State defines how much space the portlet is going to take up on a page once it renders. There are three window states that the Portlet Specification defines:

- **NORMAL**: default window state where this portlet may share the page with other portlets
- **MAXIMIZED**: portlet will take up the whole page
- **MINIMIZED**: In Liferay, this will only display the title bar of a portlet.

Window states, just like portlet modes, can be set in either the JSP or in the Portlet class:

JSP

```
<portlet:renderURL windowState="<%= WindowState.NORMAL.toString() %>" var="renderURL">
```

Portlet class

```
actionResponse.setWindowState(WindowState.NORMAL);
```

Inter-Portlet Communication

One of the limitations of the first Portlet Specification, JSR-168, was the lack of a standard way for portlets to communicate with each other. In the second Portlet Specification, JSR-268, two methods of Inter-Portlet Communication (IPC) were established: Events and Public Render Parameters. We've already discussed Events, so let's take a look at the Public Render Parameters.

Public Render Parameters

Declaring a Public Render Parameter in a Java Standard Portlet happens in `portlet.xml`. We define the Public Render Parameter and what Public Render Parameters we want to use/support. We are then able to use Public Render Parameters like any other portlet parameter:

portlet.xml

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

<portlet-app>
    ...
    <portlet>
        <portlet-name>trainingPortlet</portlet-name>
        <display-name>Training Portlet</display-name>
        ...
        <supported-public-render-parameter>tag</supported-public-render-parameter>
    </portlet>

    <public-render-parameter>
        <identifier>tag</identifier>
        <qname xmlns:x="http://www.liferay.com/public-render-parameters">x:tag</qname>
    </public-render-parameter>
</portlet-app>
```

JSP

```
<portlet:actionURL var="editEntryURL" />

<aui:form action="<%= editEntryURL %>" method="post" name="fm">
        <aui:input name="tag" type="text" />
```

```
</aui:form>
```

Java

```
String name = ParamUtil.getString(request, "tag","");
```

Non-Standard Communication Methods

Client-Side IPC

We can use Ajax JavaScript calls to communicate with other portlets on the same page. These legacy methods are provided in Liferay's JavaScript library. As with all IPC, the portlets that are communicating with each other have to be on the same page.

Sender

```
<script type="text/javascript"/>
    Liferay.fire('eventName',{

        parameterName1:parameterValue1,
        parameterName2:parameterValue2

    });
</script>
```

Receiver

```
<script type="text/javascript"/>
    Liferay.on('eventName',function(event) {

        console.log(event.parameterName1)
        console.log(event.parameterName2)

    });
</script>
```

Portlet Sessions

A portlet can share the otherwise private session data by declaring the following in liferay-portlet.xml:

```
<private-session-attributes>false</private-session-attributes>
```

Wrapping It Up

- Entering the Action or the Render phase in a single portlet forces all the portlets on a page to re-render.
- The Render phase is for producing the HTML fragment. It cannot do redirection.
- A window state can only be set in the Action phase.
- By default, ActionRequest parameters are not available in the Render phase, but must be set programmatically through the respective methods of the ActionResponse object. Liferay's MVCPortlet simplifies this process by copying all ActionRequest parameters to the Render phase. This behavior can be disabled in the `portlet.xml` .

Knowledge Check

- A _____ is a web component or application that produces an HTML fragment of a page.
- Java Standard Portlet creation requires several key elements including _____, a _____, and _____.
- There are six phases of the portlet lifecycle:
  - _____
  - _____
  - _____
  - _____
  - _____
  - _____
- There are three Portlet Modes: _____, _____ and _____.
- JSR-268 established two methods of Inter-Portlet Communication: _____ and _____.

Exercises

Create a Liferay MVC Portlet Module

Exercise Goals

- Implement a basic portlet using Liferay's MVCPortlet
  - Create a Liferay MVC portlet module
  - Deploy the portlet
  - Implement `view.jsp`
  - Create a form in `view.jsp`
  - Implement an MVC Action Command for handling the form submission
  - Test the module

The MVCPortlet class can be found here: https://github.com/liferay/liferay-portal/blob/7.2.x/portal-kernel/src/com/liferay/portal/kernel/portlet/bridges/mvc/MVCPortlet.java

Create a Liferay MVC Portlet Module

Option 1: Use the Command Line Blade tools

1. **Open** the command line shell in your Liferay Workspace `modules` folder.
2. **Run** command:

```
blade create -t mvc-portlet -p com.liferay.training.portletmodule.portlet -c SimpleMVCPortlet mvc-portlet-module
```

3. **Run** Gradle refresh on the IDE.

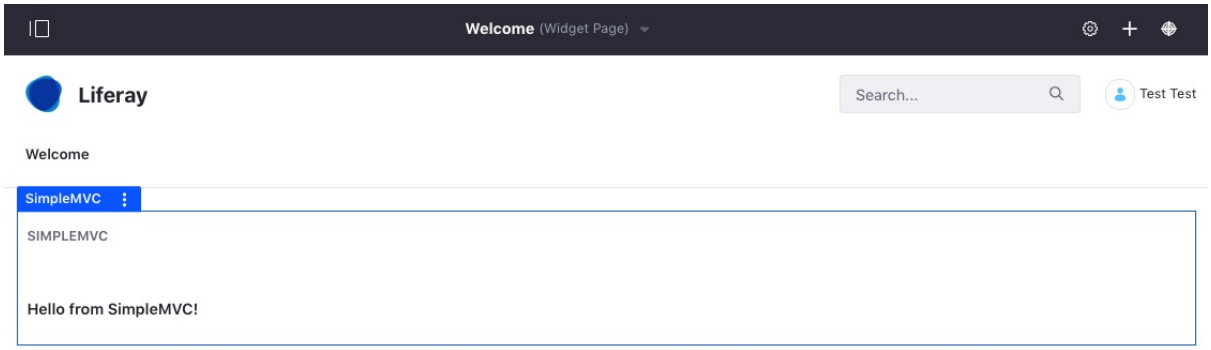
Option 2: Use Developer Studio Wizard

1. **Launch** the *Liferay Module Project* wizard in Developer Studio.
2. **Use** the following information for the first step:
   - **Project Name**: "mvc-portlet-module"
   - **Build Type**: Gradle
   - **Liferay Version**: 7.2
   - **Project Template**: mvc-portlet

3. **Click** *Next* and use the following information in the second step:
    - **Component Class Name**: "SimpleMVCPortlet"
    - **Package Name**: "com.liferay.training.portletmodule.portlet"
4. **Click** *Finish* to close the wizard.

Deploy the Portlet

1. **Drag** the *mvc-portlet-module* module onto your running server in the *Servers* view.
2. **Open** your browser to http://localhost:8080 and sign in.
3. **Click** on the plus button in the upper-right corner to add Widgets.
4. **Expand** the *Sample* category in the *Widgets* menu.
5. **Add** the *SimpleMVC* to the page. The page should look like this:



Implement the view.jsp

1. **Open** the `src/main/resources/META-INF/resources/view.jsp` and implement as follows:

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

<%
    String cssStyle = "";
    String backgroundColor = renderRequest.getParameter("backgroundColor");
    if (backgroundColor != null && !backgroundColor.isEmpty()) {
            cssStyle = "background-color: " + backgroundColor + ";";
    }
%>

<div style="<%= cssStyle %>;">
    <p class="caption">
        <liferay-ui:message key="simplemvc.caption" />
    </p>
</div>

<portlet:renderURL var="viewRedURL">
    <portlet:param name="backgroundColor" value="red"/>
</portlet:renderURL>
<portlet:renderURL var="viewYellowURL">
    <portlet:param name="backgroundColor" value="yellow"/>
</portlet:renderURL>
```

```
<div class="btn-group">
    <a class="btn btn-default" href="<%= viewRedURL %>">Set red</a>
    <a class="btn btn-default" href="<%= viewYellowURL %>">Set yellow</a>
</div>
```

2. **Save** the file and refresh the page in your browser.

> The `view.jsp` is configured as the default view-template in the portlet component's properties.

Create a Form in view.jsp

Add a form for our color test:

1. **Add** the form to the end of the `view.jsp` :

```
<portlet:actionURL name="handleForm" var="actionURL"/>

<aui:form action="<%= actionURL %>" style="margin-top: 2rem;">
    <aui:select name="backgroundColor">
        <aui:option label="aqua"/>
        <aui:option label="gray"/>
        <aui:option label="lime" />
        <aui:option label="olive" />
        <aui:option label="silver" />
    </aui:select>
    <aui:button-row>
        <aui:button type="submit" value="send"/>
    </aui:button-row>
</aui:form>
```

Implement an MVC Action Command

Create an MVC Action Command component to catch the form submit:

1. **Create** a class
   `com.liferay.training.portlet.portletmodule.action.HandleFormMVCActionCommand` .
2. **Implement** the class as follows:

```
package com.liferay.training.portlet.portletmodule.action;

import com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand;
import com.liferay.training.portletmodule.portlet.constants.SimpleMVCPortletKeys;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.PortletException;
```

```
import org.osgi.service.component.annotations.Component;

@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + SimpleMVCPortletKeys.SimpleMVC,
        "mvc.command.name=handleForm"
    },
    service = MVCActionCommand.class
)
public class HandleFormMVCActionCommand implements MVCActionCommand {
    @Override
    public boolean processAction(
            ActionRequest actionRequest, ActionResponse actionResponse)
        throws PortletException {

        _handleActionCommand(actionRequest);

        return true;
    }

    private void _handleActionCommand(ActionRequest actionRequest) {

        System.out.println("HandleFormMVCActionCommand.doProcessAction()");

        String backgroundColor = actionRequest.getParameter("backgroundColor");

        System.out.println("backgroundColor = " + backgroundColor);

    }
}
```

Test the Module

1. **Refresh** the portal page.
2. **Click** *Send* on the exercise portlet.
3. **Watch** the console log. You should see messages from the MVC Action Command component:

```
2019-04-16 21:33:59.602 INFO  [pipe-start 1074][BundleStartStopLogger:39] STARTED com.life
ray.training.portletmodule.portlet_1.0.0 [1074]
HandleFormMVCActionCommand.doProcessAction()
backgroundColor = aqua
```

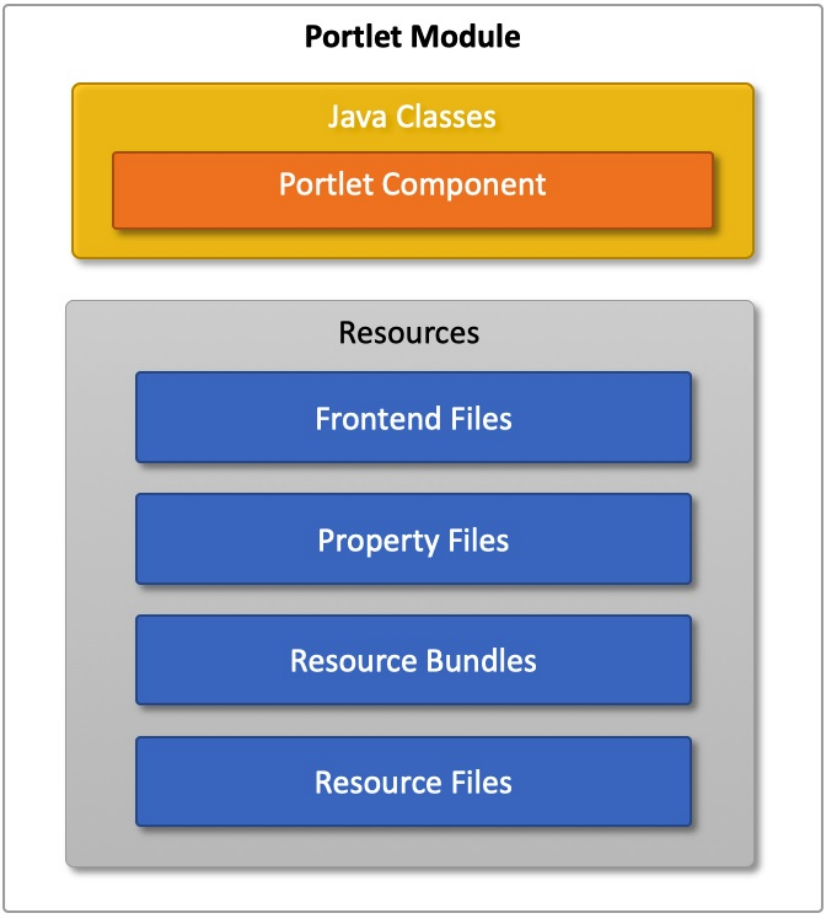
Working with Liferay Portlet Modules

Liferay has a portlet container that supports the JSR-168, JSR-286, and JSR-362 portlet specifications. If you created a portlet using the standard portlet methodology, you can get it to work and deploy in Liferay.

While Liferay's native OSGi portlets are based on the Java portlet standards, they do not follow them completely. Liferay has its own implementation of a portlet that extends the javax.portlet.GenericPortlet. Many of the concepts discussed in the Java standard portlet sections are in effect, but implemented differently. In this section, we'll explain the most important concepts of a Liferay OSGi portlet.

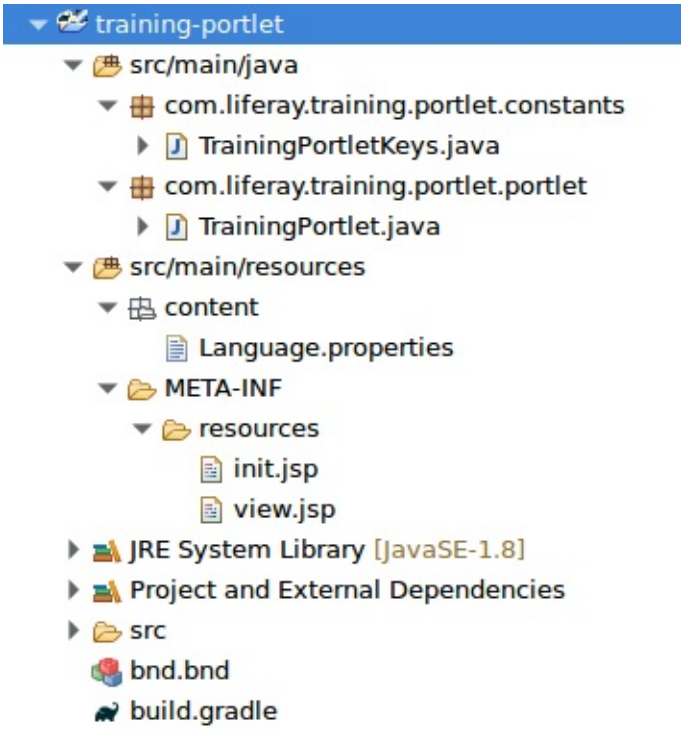
Introducing the Portlet Module

A portlet module is like any other Liferay module, with the exception of having a portlet component class. We'll create the portlet class as an OSGi service component and move all of the portlet properties from the XML descriptor files to portlet component properties:



OSGi Portlet Module Structure

Below is an example of the OSGi portlet module structure. Compare to legacy WAR-style portlets:



Let's have a look at the `bnd.bnd`, `build.gradle`, and `TrainingPortlet.java` of the portlet module.

Bnd File

Headers in `bnd.bnd` are just like any other OSGi module:

```
Bundle-Name: Training Portlet
Bundle-SymbolicName: com.liferay.training.portlet
Bundle-Version: 1.0.0
```

build.gradle

The `build.gradle` contains the portlet, servlet, and JSTL dependencies:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel"
    compileOnly group: "javax.portlet", name: "portlet-api"
    compileOnly group: "javax.servlet", name: "javax.servlet-api"
    compileOnly group: "jstl", name: "jstl"
    compileOnly group: "org.osgi", name: "osgi.cmpn"
}
```

TrainingPortlet.java

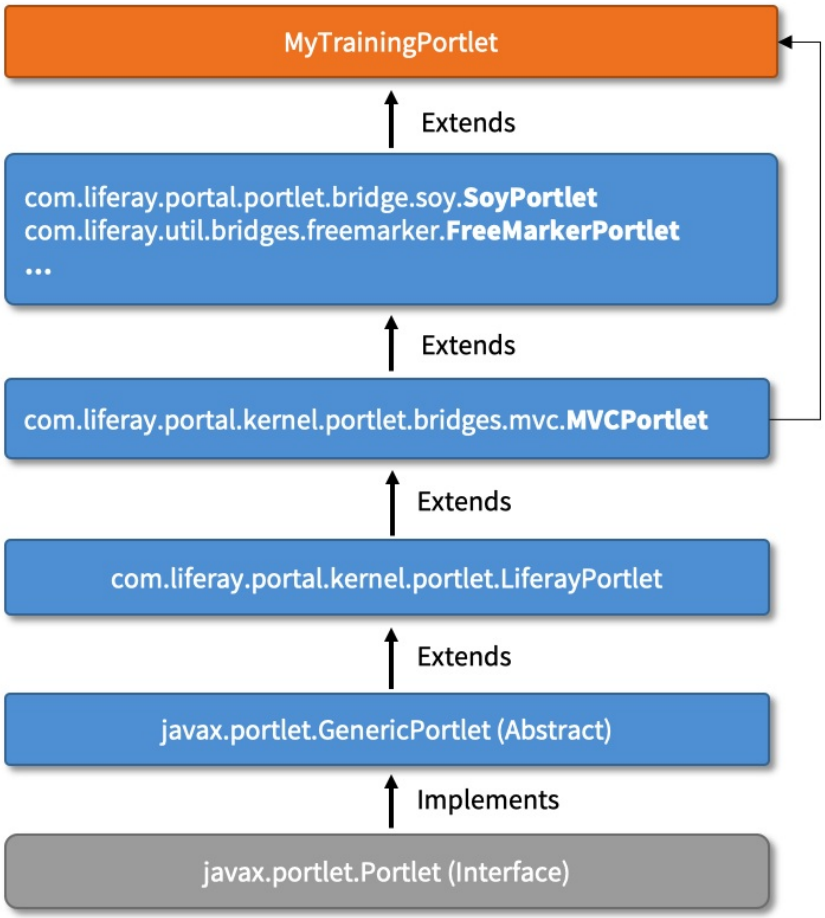
The portlet class is an OSGi service component, in this case extending the Liferay `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet`. We call this "a portlet component". See how the portlet properties, which were previously in XML descriptor files, are here component properties:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.header-portlet-css=/css/main.css",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Training Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + TrainingPortletKeys.TRAINING,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class TrainingPortlet extends MVCPortlet {

}
```

Introducing the Liferay MVC Portlet

There are quite a few things to do to get a standard `javax.portlet.GenericPortlet` portlet up and running.

Liferay's MVCPortlet inherits from `javax.portlet.Portlet` and extends `javax.portlet.GenericPortlet` so all the rules and patterns for `javax.portlet.GenericPortlet` still apply. Using the MVCPortlet or any of its extensions as the base class removes the need for boilerplate code:



Portlet Configuration in OSGi

In OSGi portlets, portlet configuration is no longer done in `portlet.xml` or `liferay-portlet.xml`. All the properties are set in the `property` element of the `@Component` annotation. When using Liferay Workspace to create a portlet component, many of the properties are automatically set.

Below is a comparison of portlet configuration in a standard `portlet.xml` and in an OSGi portlet's component properties:

portlet.xml

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

<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd" xmlns:xsi="htt
p://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/por
tlet/portlet-app_2_0.xsd http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd" version="
2.0">
    <portlet>
        <portlet-name>1</portlet-name>
        <display-name>Training Portlet</display-name>
        <portlet-class>com.liferay.training.portlet.TrainingPortlet</portlet-class>
        <init-param>
            <name>template-path</name>
            <value>/</value>
```

```
        </init-param>
        <init-param>
            <name>view-template</name>
            <value>/view.jsp</value>
        </init-param>
        <expiration-cache>0</expiration-cache>
        <supports>
            <mime-type>text/html</mime-type>
        </supports>
        <resource-bundle>content.Language</resource-bundle>
        <portlet-info>
            <title>Training Portlet</title>
            <short-title>Training Portlet</short-title>
            <keywords>Training Portlet</keywords>
        </portlet-info>
        <security-role-ref>
            <role-name>administrator</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>guest</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>power-user</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>user</role-name>
        </security-role-ref>
    </portlet>
</portlet-app>
```

Component properties

```
@Component(
        immediate = true,
        property = {
            "com.liferay.portlet.display-category=category.sample",
            "com.liferay.portlet.header-portlet-css=/css/main.css",
            "com.liferay.portlet.instanceable=true",
            "javax.portlet.display-name=Training Portlet",
            "javax.portlet.init-param.template-path=/",
            "javax.portlet.init-param.view-template=/view.jsp",
            "javax.portlet.name=" + TrainingPortletKeys.TRAINING,
            "javax.portlet.resource-bundle=content.Language",
            "javax.portlet.security-role-ref=power-user,user"
        },
        service = Portlet.class
    )
public class TrainingPortlet extends MVCPortlet {

}
```

> Java standard portlet properties are prefixed with `javax.portlet.*`, whereas Liferay-specific portlet properties are prefixed with `com.liferay.portlet.*`. You can find the [OSGi property mapping here: https://dev.liferay.com/develop/reference/-/knowledge_base/7-2/portlet-descriptor-to-osgi-service-property-map.

Portlet Lifecycle Methods in OSGi

Although we still have the classical ways of supporting the various portlet lifecycle methods, the preferred way is to use **MVC Commands**. MVC Commands are Liferay's way of implementing the Render, Action, and Resource Serving phases of a portlet. MVC Commands are implemented as components and provide a more modular way of handling the portlet lifecycle. Rather than putting everything in the portlet class as is the traditional way, the Liferay way keeps the portlet class lean and makes the various phases of the portlet more manageable.

Below is an example of how the Render phase is handled by an MVC Command specific to the Render phase: MVCRenderCommand. A `renderURL` containing the `mvcRenderCommandName` parameter is created in the JSP. An `MVCRenderCommand` component registered to the portlet and listening to the named command handles the call:

view.jsp

```
<portlet:renderURL var="viewEntryUrl">
    <portlet:param name="mvcRenderCommandName" value="/training_portlet/view_entry" />
    <portlet:param name="entryId" value="<%= String.valueOf(entry.getEntryId()) %>" />
</portlet:renderURL>

<a href="<%= viewEntryUrl">Click here to view the entry</a>
```

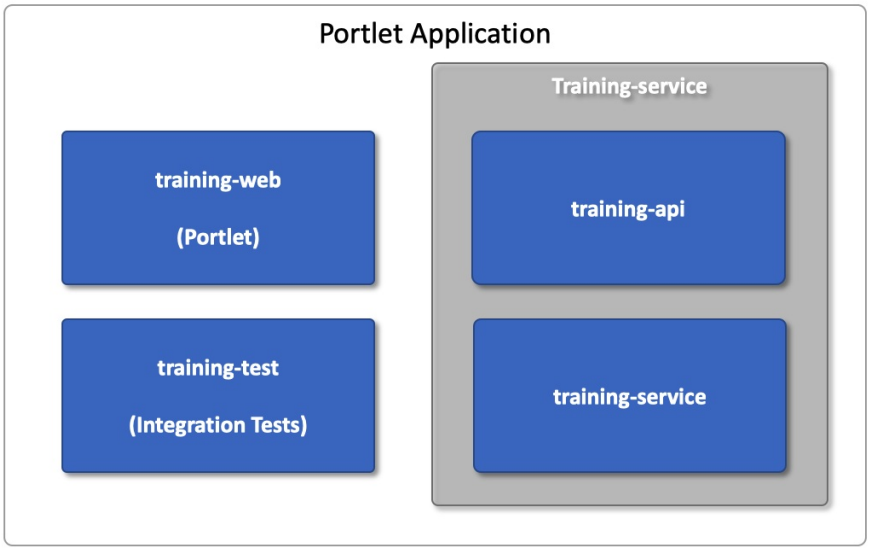
ViewMVCRenderCommand.java

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + PortletKeys.TRAINING_PORTLET,
        "mvc.command.name=/training_portlet/view_entry"
    },
    service = MVCRenderCommand.class
)
public class ViewMVCRenderCommand implements MVCRenderCommand{

    @Override
    public String render(
        RenderRequest renderRequest, RenderResponse renderResponse) {

        ...
    }
```

Modular Portlet Applications

In a traditional portlet application, everything that was needed for a portlet to run was typically packaged as one war file. In a Liferay application, a portlet application typically consists of multiple modules.

In the example diagram below, the portlet component and user interface are located inside the `training-web` module. The service API and implementation are in the `training-api` and `training-service` modules. Additionally, there is a separate testing module `training-test` :



Below is the module structure of a Liferay core Blogs application, demonstrating the multi-module approach:

blogs-analytics	LPS-74544 Auto SF	12 hours ago
blogs-api	LPS-77699 Update translations	19 hours ago
blogs-demo-data-creator-api	LPS-75049 Auto SF	a month ago
blogs-demo-data-creator-impl	LPS-74544 Auto SF	29 days ago
blogs-demo	LPS-75049 Auto SF	a month ago
blogs-editor-configuration	LPS-79621 Remove unused code	22 days ago
blogs-item-selector-api	LPS-75049 Auto SF	a month ago
blogs-item-selector-web	LPS-74544 Auto SF	29 days ago
blogs-layout-prototype	LPS-77699 Update translations	5 days ago
blogs-reading-time	LPS-75049 Auto SF	a month ago
blogs-recent-bloggers-api	LPS-77425 Increment all major versions	2 months ago
blogs-recent-bloggers-test	LPS-77425 Auto SF	2 months ago
blogs-recent-bloggers-web	LPS-79848 Include cancel button	9 days ago
blogs-rest	LPS-75049 Auto SF	a month ago
blogs-service	LPS-78354 Apply clay management toolbar to blog entries	a day ago
blogs-test-util	LPS-75049 Auto SF	a month ago
blogs-test	LPS-79874 Tests are no longer necessary	5 days ago
blogs-uad-test	LPS-80386 blogs-uad-test - removes *UADAggregator* classes	a day ago
blogs-uad	LPS-80466 blogs-uad - autogenerated	a day ago
blogs-web	LPS-79700 Fixing sorting order issue	18 hours ago
source-formatter.properties	LPS-76110 Add temporary exclusion	5 months ago
subsystem.bnd	LPS-78803 Add Collaboration subsystem.bnds	2 months ago
test.properties	LRQA-40064 Add properties for functional module group tests	15 days ago

Available User Interface Technologies

Liferay provides several MVC Portlet extensions to leverage technologies other than JSP in the user interface. Take a look at some of the available Blade examples:

- NPM portlet: https://github.com/liferay/liferay-blade-samples/tree/7.1/liferay-workspace/apps/npm
- Kotlin: https://github.com/liferay/liferay-blade-samples/tree/7.1/liferay-workspace/apps/kotlin-portlet
- Freemarker: https://github.com/liferay/liferay-blade-samples/tree/7.1/liferay-workspace/apps/freemarker-portlet

Knowledge Check

- Liferay has its own implementation of a portlet that extends _____.
- Portlets in Liferay are created as _____ that extend the Liferay _____.
- _____ are Liferay's way of implementing the _____, _____, and _____ resource phase of a portlet.
- In a Liferay app, the _____ only makes up one part of the application.
- The _____ and the _____ for a Liferay app are usually contained in a _____.

Optional Exercise

Implement a Basic JSR-286-Compliant Portlet

Exercise Goals

- Implement a basic JSR-286-compliant portlet that illustrates the most basic concepts of the Portlet Specification 2.0, like the portlet lifecycle, the portlet modes, and window states.
  - Create a portlet WAR module project
  - Implement the portlet class
  - Configure the portlet class in `portlet.xml`
  - Build and deploy your JSR-286 portlet to Liferay
  - Deploy the portlet and monitor the output in the server log
  - Include a JSP for your response output

Create a Portlet WAR Module Project

Option 1: Use the Command Line Blade tools

1. **Open** the command line shell in your Liferay Workspace `modules` folder.
2. **Run** command:

```
blade create -t war-mvc-portlet -p com.liferay.training.portlet.jsr286  jsr-286-portl
et
```

3. **Run** Gradle refresh on the IDE.

Option 2: Use Developer Studio Wizard

1. **Launch** the *Liferay Module Project* wizard in Developer Studio.
2. **Use** the following information for the first step:
   - **Project Name**: "jsr-286-portlet"
   - **Build Type**: Gradle
   - **Liferay Version**: 7.2
   - **Project Template**: war-mvc-portlet
3. **Click** *Next* and use the following information in the second step:
   - **Component Class Name**: (*leave empty because we don't need component class*)

- **Package Name**: "com.liferay.training.portlet.jsr286"

4. **Click** *Finish* to close the wizard.

Deploy and Test the Module

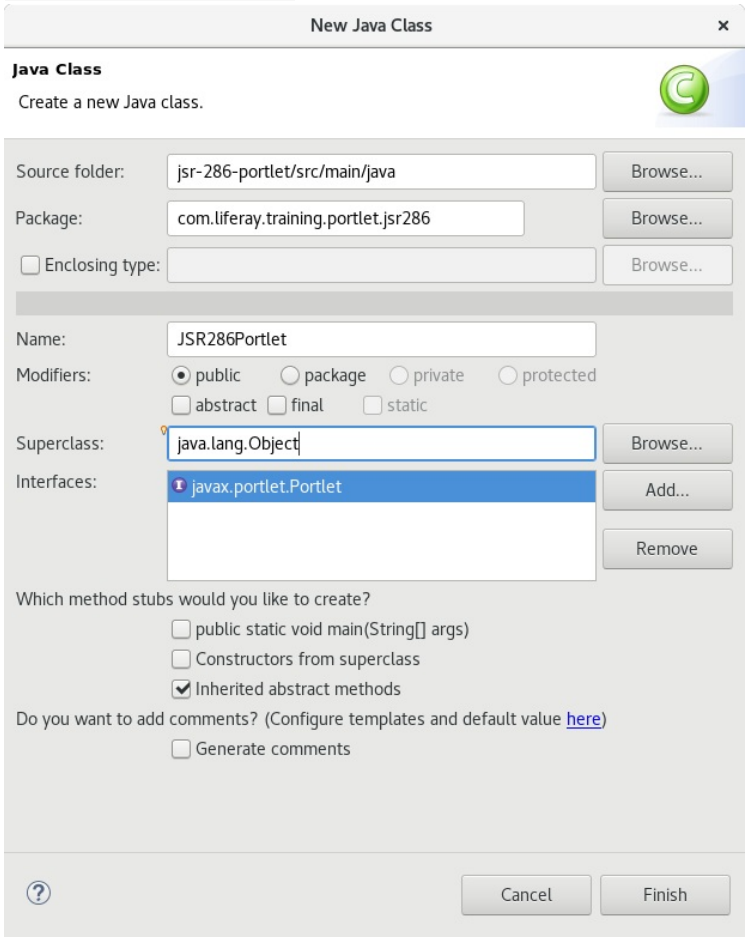
1. **Drag** the `jsr-286-portlet` folder from the *Project Explorer* onto the Liferay server in the *Servers* panel.
2. **Watch** the console. The module is successfully deployed when you see a message like:

```
 2019-04-03 18:19:14.042 INFO  [pipe-start 975][BundleStartStopLogger:39] STARTED com
.liferay.training.portlet.jsr286 [975]
```

3. **Open** your browser to http://localhost:8080 and sign in.
4. **Click** on the *Add* icon on the top right corner of the page to open the *Add Menu*.
5. **Find** the *jsr-286-portlet* portlet in the *Sample* Widget category.
6. **Drag and drop** the portlet on the page.

Implement the Portlet Class

1. **Create** a new class `com.liferay.training.portlet.jsr286.JSR286Portlet`.
   - Implement the `javax.portlet.Portlet` interface.



2. **Add** status messages to the portlet lifecycle methods implementing the class as follows:

```
package com.liferay.training.portlet.jsr286;

import java.io.IOException;
import java.io.PrintWriter;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.Portlet;
import javax.portlet.PortletConfig;
import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

public class JSR286Portlet implements Portlet {

    @Override
    public void init(PortletConfig config) throws PortletException {
        System.out.println("JSR286Portlet.init()");
    }

    @Override
    public void processAction(ActionRequest actionRequest, ActionResponse actionResp
onse)
        throws PortletException, IOException {

        System.out.println("JSR286Portlet.processAction()");
    }

    @Override
    public void render(RenderRequest renderRequest, RenderResponse renderResponse)
        throws PortletException, IOException {

        System.out.println("JSR286Portlet.render()");

        PrintWriter printWriter = renderResponse.getWriter();
        printWriter.write("Output from the JSR286Portlet's render() method.");

    }

    @Override
    public void destroy() {
        System.out.println("JSR286Portlet.destroy()");
    }
}
```

3. **Save** the file and refresh the page on your browser:

Configure the Portlet Class in portlet.xml

1. **Open** the `src/main/webapp/WEB-INF/portlet.xml` for editing.
2. **Replace** the contents of the `<portlet-class>` tag as follows:

```
<portlet-class>com.liferay.training.portlet.jsr286.JSR286Portlet</portlet-class>
```

3. **Save** the file and refresh the page on your browser:

Include a JSP for Your Response Output

If you want to format your `render()` method's output, you can add HTML markup to the string passed to the PrintWriter's `write()` method. But because this approach quickly becomes cumbersome, the Portlet API provides the `PortletRequestDispatcher` interface, which defines an object that receives requests from the client and sends them to the specified resource, e.g., a JSP file on the server. The `PortletRequestDispatcher`'s `include()` method allows you to include the contents of a resource (your JSP file) in the response. In essence, this method enables programmatic server-side includes.

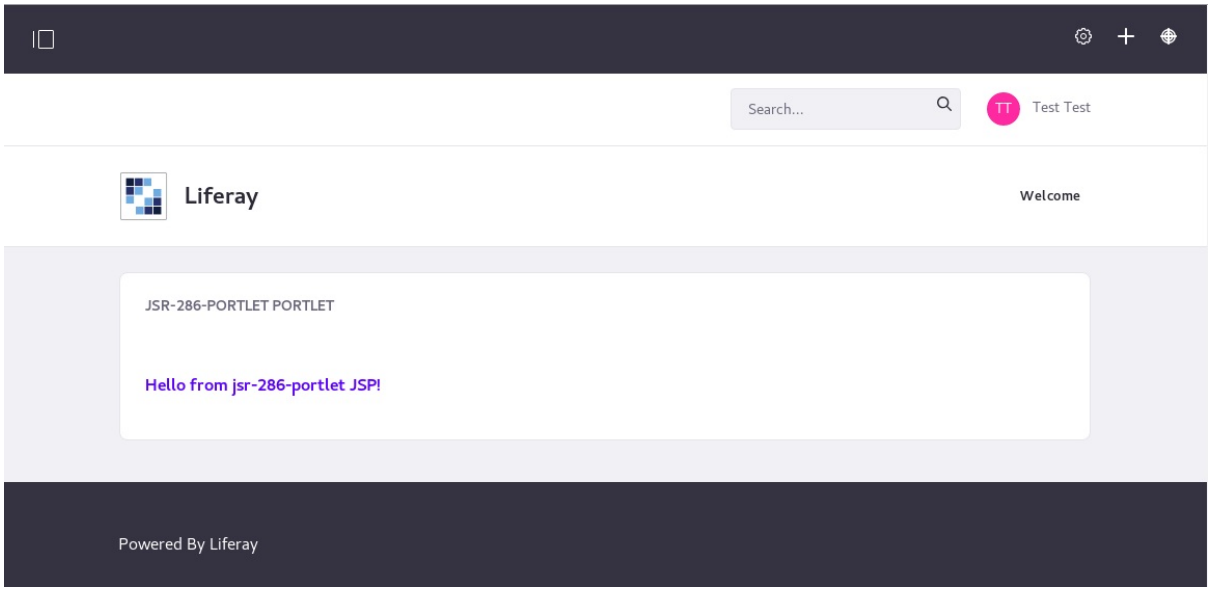
Let's modify the `render()` method to use the `view.jsp` file:

1. **Open** the portlet class `com.liferay.training.portlet.jsr286.JSR286Portlet.java`.
2. **Implement** the `render()` method as follows:

```
@Override
public void render(RenderRequest renderRequest, RenderResponse renderResponse)
    throws PortletException, IOException {
    System.out.println("JSR286Portlet.render()");

    // PrintWriter printWriter = renderResponse.getWriter();
    // printWriter.write("Output from the JSR286Portlet's render() method.");

    String path = "/view.jsp";

    PortletSession portletSession = renderRequest.getPortletSession();
    PortletContext portletContext = portletSession.getPortletContext();
    PortletRequestDispatcher portletRequestDispatcher = portletContext.getRequestDis
patcher(path);

    if (portletRequestDispatcher == null) {
        System.err.println(path + " is not a valid include");
    }
    else {
        portletRequestDispatcher.include(renderRequest, renderResponse);
    }
}
```

3. **Resolve** the imports and save the file.
4. **Run** the Gradle `deploy` task to redeploy the portlet.
5. **Refresh** the page where you have installed the *JSR-286-portlet* portlet.

The render method now includes and outputs the `view.jsp` generated by the wizard:



Bonus Exercise: Set a Render Parameter

In `view.jsp`, you can make use of the tag libraries, simplifying the implementation of the portlet view. We'll now use the standard portlet tag library to create a render url, set a render parameter, and explore the behavior of our portlet in the render phase:

1. **Implement** the file `webapp/view.jsp` as follows:

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

<%
    String cssStyle = "";

    String backgroundColor = renderRequest.getParameter("backgroundColor");

    if (backgroundColor != null && !backgroundColor.isEmpty()) {
        cssStyle = "background-color: " + backgroundColor + ";";
    }
%>
<div style="<%= cssStyle %>">

    <p class="caption">
        <liferay-ui:message key="jsr-286-portlet.caption" />
    </p>

</div>

<portlet:renderURL var="viewRedURL">
    <portlet:param name="backgroundColor" value="red"/>
</portlet:renderURL>
<portlet:renderURL var="viewYellowURL">
    <portlet:param name="backgroundColor" value="yellow"/>
</portlet:renderURL>

<div class="btn-group">
```

```
    <a class="btn btn-default" href="<%= viewRedURL %>">Set red</a>
    <a class="btn btn-default" href="<%= viewYellowURL %>">Set yellow</a>
</div>
```

> The taglib includes are defined in the `init.jsp`, which the `view.jsp` includes. Both files have been created by the wizard when you created the project. The Tag Library Descriptor files can be found in the `src/main/webapp/WEB-INF/tld` folder of your project.

You can now explore the portlet's behavior when it is deployed multiple times on the same page:

1. **Deploy** the refactored portlet to the Liferay server.
2. **Add** the portlet two or more times onto your page.
3. **Try** clicking buttons on different instances.

When you click the button in one instance, the background color of the respective portlet changes, while all other instances remain the same: