JAVA PORTLETS

101

April 5, 2007

The following sample chapter is from *Java Portlets 101* by Sunil Patil, published by SourceBeat. Access more information about this book and our other titles from the SourceBeat Web site at www.sourcebeat.com.

The SourceBeat publishing model is different from that of traditional publishers: We are dedicated to open source media. Our books are published electronically, and our subscribers become members of the SourceBeat community with personal access to authors and their blogs and a variety of forums that address open source issues and solutions.

SourceBeat

Thank you for your interest in this title.

Java Portlets 101
by Sunil Patil

Published by SourceBeat, LLC, Highlands Ranch, Colorado.

Managing Editor: Sarah Hogan
Technical Editor: Bobby Walters
Copy Editor: Holly Lancaster
Layout Designer: Sarah Hogan
Cover Designer: Max Hays

ISBN: 0-9765534-2-2

5

PORTLET WEB APPLICATION

PortletRequestDispatcher, Portlet tag library, markup restrictions

In this chapter, you will learn how portlets can include markup generated by servlets or JSPs. You will also learn how to dispatch requests to JSPs for actual markup generation and how to dispatch requests to servlets. You will then learn how to use various tags defined in the portlet tag library. Next, you will learn how to use servlets to display images in a portlet. Last, you will learn about restrictions on markup generated by a portlet.

OVERVIEW

The main problem with the sample portlets developed so far is that the code for actual markup generation is embedded within the portlet class itself. This practice is not good because it makes your code hard to understand and difficult to maintain, and it does not allow you to distribute work efficiently between a Java developer and web page designer.

The Portlet Specification defines the **`PortletRequestDispatcher`** interface, which portlets can use to delegate control to a JSP or a servlet for actual markup generation. The **`PortletContext`** object defines two methods that you can use to get an instance of the **`PortletRequestDispatcher`** object:

- **`getRequestDispatcher()`**: This method takes a resource path name as an argument and returns the **`PortletRequestDispatcher`** object, which can be used to dispatch requests to the resource located at the path name.

- **`getNamedDispatcher()`**: This method takes the name of the servlet or JSP as defined in the web application deployment descriptor (web.xml) and returns the **`PortletRequestDispatcher`** object, which can be used to dispatch requests to that servlet or JSP.

You can put theory into practice by developing a sample application to demonstrate how you can use **`getRequestDispatcher()`** methods to dispatch requests to a JSP for actual markup generation.

DISPATCHING REQUESTS TO THE JSP

In this section, you will develop DispatchJSPPortlet, which provides the same functionality as FormHandling-Portlet developed in *Chapter 4: Request Processing*. DispatchJSPPortlet will display "Hello World" as a greeting to users in **`VIEW`** mode, and users can change this greeting in **`EDIT`** mode.

Create DispatchJSPPortlet.java by extending the **`GenericPortlet`** class, as shown in Listing 5.1.

Listing 5.1

```
public class DispatchJSPPortlet extends GenericPortlet{

      protected void doEdit(RenderRequest renderRequest,
      RenderResponse renderResponse) throws PortletException, IOException {
            System.out.println("Entering DispatchJSPPortlet.doEdit()");
            PortletRequestDispatcher requestDispatcher = getPortletContext().
            getRequestDispatcher("/WEB-INF/jsp/changegreeting.jsp");
            requestDispatcher.include(renderRequest,renderResponse );
            System.out.println("Exiting DispatchJSPPortlet.doEdit()");
      }

      protected void doView(RenderRequest renderRequest,
      RenderResponse renderResponse) throws PortletException, IOException {
            System.out.println("Entering DispatchJSPPortlet.doView()");
            PortletRequestDispatcher requestDispatcher = getPortletContext().
            getRequestDispatcher("/WEB-INF/jsp/displaygreeting.jsp");
            requestDispatcher.include(renderRequest,renderResponse);
            System.out.println("Exiting DispatchJSPPortlet.doView()");
     }

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

```
            System.out.println("Entering DispatchJSPPortlet.processAction()");

            String actionName = actionRequest.getParameter("actionName");
            if(actionName.equals("changeGreeting")){
                  String greeting = actionRequest.getParameter("greetingStr");
                  // Perform some business logic on the server side
                  actionResponse.setRenderParameter("greetingStr",greeting);
            }

            System.out.println("Exiting DispatchJSPPortlet. processAction ()");
      }
}
```

Inside the **doEdit()** method, you are calling the **getRequestDispatcher()** method of the **PortletContext** class, passing it an absolute path of the **changegreeting.jsp** file, which you want to use to generate markup for **EDIT** mode. Similarly, the **doView()** method gets the **PortletRequestDispatcher** object, which wraps **displaygreeting.jsp**. Please note that the path name must begin with a slash (**/**), and the portlet container interprets it as relative to the current context root. Inside the **getRequestDispatcher()** method, the portlet container creates an object of the **PortletRequestDispatcher** class, which wraps a resource at the JSP path name and returns it. You can call the **getRequestDispatcher()** method only during the render phase of the portlet (that is, only from one of the **doXXX()** methods). The **getRequestDispatcher()** method returns null if it is not able to return the **PortletRequestDispatcher** object.

Once you have the **PortletRequestDispatcher** object, you can call its **include()** method to include content of a resource in the response. Please note that you can use the **include()** method to include both static content (an HTML file) and dynamic content generated by the JSP file or servlet. The **include()** method takes the object of **RenderRequest** and **RenderResponse** as parameters, and you must pass the same **renderRequest** and **render-Response** parameters that are passed into the **render()** method of your portlet. Also note that you can get the object of the **PortletRequestDispatcher** class only during render phase, and you can call the **include()** method of the **PortletRequestDispatcher** class at any time and at multiple times during the render phase.

You can include query string information in the query path while creating the **PortletRequestDispatcher** object using the **getRequestDispatcher()** method. Parameters specified in the query string used to create the **PortletRequestDispatcher** object are aggregated with the portlet render parameters and take precedence over other portlet render parameters of the same name passed to the included servlet or JSP. The parameters associated with a **PortletRequestDispatcher** object are scoped to apply only for the duration of the include call.

Now that DispatchJSPPortlet.java is ready, the next step is to create displaygreeting.jsp, as shown in Listing 5.2.

Listing 5.2

```
<%@ page contentType="text/html"%>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet"%>
<portlet:defineObjects/>

<%
      String greetingStr = renderRequest.getParameter("greetingStr");
      if(greetingStr == null)
            greetingStr ="Hello World";
%>
<h3><%=greetingStr%></h3>
```

The Portlet Specification defines the portlet tag library (portlet.tld), which enables JSPs that are included in the portlet to have direct access to portlet-specific elements, such as **`renderRequest`**, **`renderResponse`**, and **`portletConfig`**. It also provides JSPs with access to portlet functionality such as the creation of portlet URLs. The first line in displaygreeting.jsp is used to include a portlet.tld in your JSP page.

You use the **`<portlet:defineObjects>`** tag to declare **`renderRequest`**, **`renderResponse`**, and **`portletConfig`** variables in your JSP file. This tag will be covered in detail in the *Portlet tag library* section. Once you get access to the **`renderRequest`** object, you can get the value of the **`greetingStr`** render parameter and display it to the user.

The next step is to develop the changegreeting.jsp file, as shown in Listing 5.3.

Listing 5.3

```
<%@ page contentType="text/html"%>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet"%>

<portlet:defineObjects/>
<%
      String greetingStr = renderRequest.getParameter("greetingStr");
      if(greetingStr == null)
            greetingStr ="Hello World";
%>

<form action="<portlet:actionURL>
                  <portlet:param name='actionName' value='changeGreeting'/>
              </portlet:actionURL>" method="POST" >
      <table>
            <tr>
                  <td>Greeting Name</td>
```

```
                <td><input type='text' name='greetingStr'
                value='<%=greetingStr %>'/></td>
                <td><input type='submit' value='Submit'/></td>
            </tr>
        </table>
</form>
```

After you declared dependency on portlet.tld and declared the **<portlet:defineObjects>** tag in your JSP, you got the value of the **greetingStr** render parameter from the **renderRequest** object. Next, you created the **FORM** element with the value of the **method** attribute equal to **POST**. Inside this form, you used the **<portlet:actionURL>** tag to create the **actionURL** object, which points to DispatchJSPPortlet. You used the **<portlet:param>** tag to set the **actionName** parameter in an action URL. The rest of the page is similar to your normal JSP page.

Now create the web.xml and portlet.xml files as usual. Then build your portlet and deploy it on the server to test dispatching user requests to the JSP page.

DISPATCHING REQUESTS TO A SERVLET

In this section, you will learn about how portlets can include servlet content for actual markup generation. You will develop a sample portlet application that will display "Hello World" as a greeting to the user in **VIEW** mode and will allow the user to change this message in **EDIT** mode, which is the same functionality as that of DispatchJSPPortlet developed in the *Dispatching requests to the JSP* section. The only difference is that you will include the content generated by ChangeGreetingServlet.java instead of **changegreeting.jsp** for **EDIT** mode markup generation.

Create DispatchServletPortlet.java, as shown in Listing 5.4.

Listing 5.4

```
public class DispatchServletPortlet extends GenericPortlet{
      protected void doEdit(RenderRequest renderRequest,
      RenderResponse renderResponse) throws PortletException, IOException {
            System.out.println("Entering DispatchServletPortlet.doEdit()");
            renderResponse.setContentType("text/html");
            PortletRequestDispatcher requestDispatcher = getPortletContext().
getNamedDispatcher("ChangeGreetingServlet");
            requestDispatcher.include(renderRequest,renderResponse );
            System.out.println("Exiting DispatchServletPortlet.doEdit()");
```

```
        }

        protected void doView(RenderRequest renderRequest,
        RenderResponse renderResponse) throws PortletException, IOException {
              System.out.println("Entering DispatchServletPortlet.doView()");
              renderResponse.setContentType("text/html");
              PortletRequestDispatcher requestDispatcher =
              getPortletContext().getNamedDispatcher("displaygreeting");
              requestDispatcher.include(renderRequest,renderResponse );
              System.out.println("Exiting DispatchServletPortlet.doView()");
        }

        public void processAction(ActionRequest actionRequest,
        ActionResponse actionResponse) throws PortletException, IOException {
              System.out.println("Entering         DispatchServletPortlet.processAction()");

              String actionName = actionRequest.getParameter("actionName");
              if(actionName.equals("changeGreeting")){
                    String greeting = actionRequest.getParameter("greetingStr");
                    // Perform some business logic on the server side
                    actionResponse.setRenderParameter("greetingStr",greeting);
              }
              System.out.println("Exiting DispatchServletPortlet. processAction  ()");
        }
  }
```

You might have noticed that the DispatchServletPortlet.java code listing is similar to DispatchJSPPortlet.java. The only difference is in the **`doView()`** and **`doEdit()`** methods. In both of these methods, you used the **`PortletContext.getNamedDispatcher()`** method to get an instance of the **`PortletRequestDispatcher`** object. The **`getNamedDispatcher()`** method takes the name of the servlet or JSP as defined in web.xml and returns a **`PortletRequestDispatcher`** object that wraps that resource. Once you have the object of the **`PortletRequestDispatcher`** class, you can call its **`include()`** method to include markup generated by that resource.

Please note one important difference between the **`getRequestDispatcher()`** and **`getNamedDispatcher()`** methods. You cannot pass query string information to an underlying resource while using the **`getNamedDispatcher()`** method. Similarly, the servlet or JSP included using the **`getNamedDispatcher()`** method does not have access to the path used to obtain the request dispatcher.

The next step is to create **`ChangeGreetingServlet.java`**, as in Listing 5.5.

Listing 5.5

```
public class ChangeGreetingServlet extends HttpServlet{

      protected void doGet(HttpServletRequest httpServletRequest,
      HttpServletResponse httpServletResponse) throws ServletException, IOException {
            System.out.println("Entering ChangeGreetingServlet.doGet()");

            RenderRequest renderRequest =(RenderRequest)
            httpServletRequest.getAttribute("javax.portlet.request");
            RenderResponse renderResponse =(RenderResponse)
            httpServletRequest.getAttribute("javax.portlet.response");
            PortletURL actionURL = renderResponse.createActionURL();
            actionURL.setParameter("actionName","changeGreeting");
            try {
                  actionURL.setPortletMode(PortletMode.VIEW);
            } catch (PortletModeException e) {
                  throw new ServletException(e);
            }

            String greetingStr = renderRequest.getParameter("greetingStr");
            if(greetingStr == null)
                  greetingStr ="Hello World";

            PrintWriter out = httpServletResponse.getWriter();
            out.println("<form action='");
            out.println(actionURL.toString());
            out.println("' method='POST'>");
            out.println("<table><tr>");
            out.println("<td>Greeting Name</td>");
            out.println("<td><input type='text' name='greetingStr' value='");
            out.println(greetingStr);
            out.println("' /></td>");
            out.println("<td><input type='submit' value='Submit'/></td>");
            out.println("</tr></table>");
            out.println("</form>");
            System.out.println("Exiting ChangeGreetingServlet.doGet()");
      }

}
```

Creating **`ChangeGreetingServlet.java`** is similar to creating any other servlet. First you must create a class extending the **`HttpServlet`** class, and then depending on what **`HTTP`** methods that servlet will handle, you must

override the corresponding **doXXX()** method. The Portlet Specification states that whenever you include servlets or JSPs in your portlet, an **HTTP GET** request should be sent to that resource, which is why you wrote your **EDIT** mode markup generation logic in the **doGet()** method.

Before giving control to the actual JSP or servlet, the portlet container will set three request attributes:

- **javax.portlet.config**: The value of this attribute should be the **PortletConfig** object that is available to the portlet in the **render()** method.
- **javax.portlet.request**: The value of this attribute should be equal to the **renderRequest** object that is passed into your **render()** method.
- **javax.portlet.response**: The value of this attribute should be equal to the **renderResponse** object that is passed into your **render()** method.

Inside the **doGet()** method, you got the **renderRequest** and **renderResponse** request objects from the **httpServletRequest** object. Then you used the **renderResponse** object to create the **actionURL** object. You also used the **renderRequest** object to get the value of the **greetingStr** request parameter. The default value for the **greetingStr** parameter is **"Hello World"**.

Inside the **doGet()** method, you have two options for retrieving the **PrintWriter** object, which you can use to write markup. One is calling the **HttpServletResponse.getWriter()** method, and the other is calling the **RenderResponse.getWriter()** method. Both methods are equivalent to each other. In fact, the Portlet Specification states that you should get the same instance of the **PrintWriter** object from both the **HttpServletResponse.getWriter()** and the **RenderResponse.getWriter()** method.

If the servlet or JSP that is the target of a request dispatcher throws a **RuntimeException** or checked exception of **IOException** type, it must be propagated to the calling portlet. All other exceptions (including the **ServletException**) must be wrapped with a **PortletException**. The root cause of the exception must be set to the original exception.

Please note the restrictions on what you can do within included servlets or JSPs. First, you're not allowed to state the HTTP status code by calling the **HttpServletResponse** object's **setStatus()** or **setError()** methods. Similarly, you cannot set any HTTP headers; any attempt to set HTTP headers will be ignored. Last, if a servlet or JSP is included in the portlet, you should not try to forward control to some other resource; forwarding control will result in non-deterministic behavior.

The last step is to create web.xml, as in Listing 5.6.

Listing 5.6

```
<web-app>
      <display-name>Dispatch request to Servlet</display-name>
      <servlet>
            <servlet-name>ChangeGreetingServlet</servlet-name>
            <servlet-class>
                  com.sourcebeat.portlet101.chapter5.ChangeGreetingServlet
            </servlet-class>
      </servlet>
      <servlet>
            <servlet-name>displaygreeting</servlet-name>
            <jsp-file>/jsp/displaygreeting.jsp</jsp-file>>
      </servlet>
      <servlet-mapping>
            <servlet-name>ChangeGreetingServlet</servlet-name>
            <url-pattern>/changegreeting</url-pattern>
      </servlet-mapping>
</web-app>
```

In the web.xml file, you can use the **<servlet>** element to give a name to a servlet or JSP file. In the sample web.xml file, you used the **<servlet>** element to map the **ChangeGreetingServlet** name to the com.sourcebeat.portlet101.chapter5.ChangeGreetingServlet.java file. Next, you used the **<servlet>** element to map the **displaygreeting** name to the /jsp/displaygreeting.jsp file. In DispatchServletPortlet.java, you used the value of the **<servlet-name>** element as the name of the underlying resource to include.

PORTLET TAG LIBRARY

In the *Dispatching requests to the JSP* section, you were introduced to the portlet tag library. Now it's time to take a closer look. The portlet tag library enables JSPs that are included in portlets to have direct access to portlet-specific elements, such as **renderRequest** and **renderResponse**, which are required in the render phase. (JSPs are used in the View part of the MVC pattern.) The portlet tag library also provides tags that implement portlet functionality, such as creating an action or render URL.

The Portlet Specification defines tags, tag parameters, and the functionality that these tags are supposed to provide. Every portlet container must provide an implementation for this tag library. As a developer, you can use this tag

library without worrying about the actual implementation. JSP pages using the tag library must include the following **taglib** instruction:

```
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
```

After you include this instruction, your JSPs can use the following four tags: **defineObject**, **actionURL**, **renderURL**, and **namespace**.

DEFINEOBJECT

The first task that you should do in your portlet JSP page is include the **<portlet:defineObject>** tag. This tag declares the following three variables in your JSP page.

- **RenderRequest renderRequest**: The value of this variable is equal to the **javax.portlet.request** attribute stored in the **HttpServletRequest** object.
- **RenderResponse renderResponse**: The value of this variable is equal to the **javax.portlet.response** attribute stored in the **HttpServletRequest** object.
- **PortletConfig portletConfig**: The value of this variable is equal to the **javax.portlet.config** attribute stored in the **HttpServletRequest** object.

A JSP using the **defineObject** tag can use these variables from scriplets throughout the page. Please note that the **defineObject** tag cannot have an attribute and it must not contain code in its body. You can use the **renderResponse** object declared by the **<portlet:defineObject>** tag to set a new title for the portlet, as in Listing 5.7.

Listing 5.7

```
<portlet:defineObjects/>
<%
      renderResponse.setTitle("This is new title");
%>
```

ACTIONURL

You can use the **actionURL** tag to create an action URL pointing to the current portlet. The following non-required attributes are defined for this tag:

- **windowState**: This attribute takes the string value that indicates the window state that the portlet should have when this link is executed. The value of this tag could be one of the predefined window states, such as **MINIMIZED**, **MAXIMIZED**, or **NORMAL**. If the specified window state is illegal for the current request, a

JSPException with an original **PortletException** as the root cause is thrown. The value of the **windowState** attribute is not case sensitive.

- **portletMode**: This attribute takes the string value that indicates the portlet mode that the portlet should have when this link is executed. The value of this attribute could be one of the standard portlet modes, such as **VIEW**, **EDIT**, **HELP**, or custom portlet mode. If the specified portlet mode is illegal for the current request, a **JSPException** with an original **PortletException** as the root cause is thrown. The value of the **portletMode** attribute is not case sensitive.
- **var**: By default, the value of the **<portlet:actionURL>** tag is written in the JSP page where this tag is declared. You can override this behavior by adding a **var** attribute to the **actionURL** tag. When you do that, the portlet container creates a new page-level variable of the **String** type and assigns the value of the **actionURL** tag to that variable.
- **secure**: The value of the **secure** attribute indicates whether the resulting URL should be a secure or insecure connection. If this attribute is not set explicitly, it must stay the same as the security setting of the current request. If the current request is secure, every **actionURL** tag you create will be secure. If the current request is insecure, every **actionURL** tag you create will be insecure.

Change changegreeting.jsp in DispatchJSPPortlet as shown in bold in Listing 5.8 to use various attributes of the **actionURL** tag.

Listing 5.8

```
<%@ page contentType="text/html"%>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet"%>
<portlet:defineObjects/>
<%
      String greetingStr = renderRequest.getParameter("greetingStr");
      if(greetingStr == null)
            greetingStr ="Hello World";
%>
<portlet:actionURL portletMode='VIEW' windowState='MAXIMIZED' var='actionURL' >
      <portlet:param name='actionName' value='changeGreeting'/>
</portlet:actionURL>


<form method="POST" action="<%=actionURL %>">
      <table>
```

```
            <tr>
                  <td>Greeting Name</td>
                  <td><input type='text' name='greetingStr' value='<%=greetingStr %>'/></td>
                  <td><input type='submit' value='Submit'/></td>
                  </tr>
      </table>
</form>
```

One of the problems changegreeting.jsp has in Listing 5.3 is that the **<portlet:actionURL>** element is embedded directly in the value of the form's **action** attribute. This embedding makes it hard to read, so in the revised changegreeting.jsp (as shown in Listing 5.8), you separated the **<portlet:actionURL>** element, which defines the **var** attribute with the value equal to **actionURL**. The portlet container creates a string-variable **actionURL** at this point and assigns the value of the **<portlet:actionURL>** tag to it. Then you wrote the value of the **actionURL** variable inside the form's **action** attribute value. You also made two more changes in the **<portlet:actionURL>** tag. First, you added the **portletMode** attribute with a value equal to **VIEW** Next, you added the **windowState** attribute with a value equal to **MAXIMIZED**. As a result, when the user submits this form from **EDIT** mode, the portlet will be redirected to **VIEW** mode in the **MAXIMIZED** window state.

Please note that if you try to set an illegal value for **portletMode**, **windowState**, or **secure**, a **JSPException** will be thrown. You can get access to the original exception by accessing the root cause of the **JSPException.**

RENDERURL

You can use the **renderURL** tag to create a render URL pointing to the current portlet. The **renderURL** tag has the same non-required attributes as the **actionURL** tag, and those attributes serve the same function.

Change displaygreeting.jsp as shown in bold in Listing 5.9 to use the **renderURL** tag.

Listing 5.9

```
<%@ page contentType="text/html"%>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet"%>
<portlet:defineObjects/>

<%
      String greetingStr = renderRequest.getParameter("greetingStr");

      if(greetingStr == null)
            greetingStr ="Hello World";
%>
```

```
<h3><%=greetingStr%></h3>
<a href='<portlet:renderURL portletMode='EDIT'></portlet:renderURL>' >Go to Edit mode</a>
```

The revised displaygreeting.jsp will display one link named "Go to Edit mode" at the end of the portlet. The **href** attribute of this link contains the **<portlet:renderURL>** element with the **portletMode** attribute equal to **EDIT**. When you click this link, it takes you to **EDIT** mode.

NAMESPACE

The namespace tag produces a unique value for the current portlet. Use this tag for named elements in the portlet output, such as JavaScript functions and variables. You might be wondering why you need the namespace tag. As you learned earlier, every portal page has one or more portlets. Sometimes, the same portlet is added twice on the same page. As a developer, you don't have control over what other portlets will be added to the same portal page. For example, it's possible to have a text input name that is used twice on the same page — for example, **greeting**. A JavaScript function name such as **validate()** could also be used twice. You can use the **<portlet:namespace>** tag to avoid naming duplication.

Modify changegreeting.jsp as shown in bold in Listing 5.10 to use the **namespace** tag.

Listing 5.10

```
<%@ page contentType="text/html"%>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet"%>
<portlet:defineObjects/>
<%
      String greetingStr = renderRequest.getParameter("greetingStr");
      if(greetingStr == null)
            greetingStr ="Hello World";
%>
<portlet:actionURL portletMode='VIEW' windowState='maximized' var='actionURL' secure='true'>
      <portlet:param name='actionName' value='changeGreeting'/>
</portlet:actionURL>

<script language='javascript'>
      function <portlet:namespace/>validateForm(){
            var greetingStrInput = document.getElementById(
            '<portlet:namespace/>greetingStr');
           var greetingValue = greetingStrInput.value;
           //Check that greeting name is not null
           if(greetingValue.length == 0 ){
                 alert('Value of Greeting Name cannot be empty');
```

```
                return false;
            }
            return true;
        }
    </script>
    <form method="POST" action="<%=actionURL %>" onsubmit='return <portlet:namespace/>validateForm()'>
        <table>
            <tr>
                <td>Greeting Name</td>
                <td><input type='text' name='greetingStr' value='<%=greetingStr %>'
                    id='<portlet:namespace/>greetingStr'/></td>
                <td><input type='submit' value='Submit' /></td>
            </tr>
        </table>
    </form>
```

In Listing 5.10, you modified changegreeting.jsp so that it checks whether the value of the **greetingStr** input submitted by the user is empty; if it is empty, users see the following prompt: "Value of Greeting Name cannot be empty." To create this prompt, you made three changes. First you added the **id** attribute to the **greetingStr** input with the value of the **id** attribute equal to **<portlet:namespace/>greetingStr**. Second you added the **onsubmit** event handler for the **form** element with the value equal to **"return <portlet:namespace/>validateForm()"**. Third, you defined the **<portlet:namespace/>validateForm()** JavaScript function. Now when the user tries to submit a form, control goes to the **validateForm()** function. In this function, the value of **greetingStr** is checked. If it's null, an error message displays.

Now build DispatchJSPPortlet and install it on the server and then go to the portal page containing DispatchJSP-Portlet. When you view the HTML source code in **EDIT** mode, you will see that the value of the **greetingStr** input's **id** attribute is similar to **<Pluto__DispatchJSPPortlet_DispatchJSPPortlet_>greetingStr** (the first part might be different in your case). You will also notice that the same string is added to the **validateForm()** function name and the value of the **onsubmit** attribute of the **form** element. This unique string is generated by the **namespace** tag.

ADDITIONAL WAYS OF INCLUDING SERVLET/JSP CONTENT

In this section, you will develop a sample portlet, which will allow you to upload image files in **EDIT** mode and see them in **VIEW** mode. You will use the Apache Commons FileUpload library to read files uploaded by users.

Before you develop FileUploadPortlet, please note that portlets are not allowed to generate binary content because markup generated by a portlet is aggregated with markup generated by other portlets on the same page. FileUploadPortlet will give control to the servlet to display the image to the user. Create FileUploadPortlet.java as shown in Listing 5.11.

Listing 5.11

```
public class FileUploadPortlet extends GenericPortlet {

      protected void doView(RenderRequest renderRequest,
      RenderResponse renderResponse) throws PortletException, IOException {
            System.out.println("Entering FileUploadPortlet.doView()");
            renderResponse.setContentType("text/html");
            if (renderRequest.getPortletSession().
                  getAttribute("photo" ,PortletSession.APPLICATION_SCOPE) == null) {
                  renderResponse.getWriter().println("Image not found.
                     Please upload image using EDIT mode");
            } else {
                  System.out.println("Image found redirecting user to display.jsp");
                  PortletRequestDispatcher requestDispacther = getPortletContext()
       .getRequestDispatcher("/jsp/display.jsp");
                  requestDispacther.include(renderRequest, renderResponse);
            }
            System.out.println("Exiting FileUploadPortlet.doView()");
      }

      protected void doEdit(RenderRequest renderRequest,
      RenderResponse renderResponse) throws PortletException, IOException {
            System.out.println("Entering FileUploadPortlet.doEdit()");
            renderResponse.setContentType("text/html");

            PortletRequestDispatcher requestDispacther = getPortletContext()
            .getRequestDispatcher("/jsp/upload.jsp");
            requestDispacther.include(renderRequest, renderResponse);

            System.out.println("Entering FileUploadPortlet.doEdit()");
      }

      public void processAction(ActionRequest actionRequest,
      ActionResponse actionResponse) throws PortletException, IOException {
            System.out.println("Entering FileUploadPortlet.processAction()");
            DiskFileItemFactory diskFileItemFactory = new DiskFileItemFactory();
            PortletFileUpload portletFileUpload = new PortletFileUpload(
```

```
                diskFileItemFactory);
            try {
                List fileItemList =    portletFileUpload.parseRequest(actionRequest);
                Iterator fileIt = fileItemList.iterator();
                while (fileIt.hasNext()) {
                    FileItem fileItem = (FileItem) fileIt.next();
                    if (fileItem.getFieldName().equals("imageFile")) {
                        System.out.println("Saving image as
                        portletSession attribute. size of image is" +fileItem.getSize());
                        actionRequest.getPortletSession().
                      setAttribute("photo",fileItem.get(),PortletSession.APPLICATION_SCOPE);
                        }
                 }
            } catch (FileUploadException e) {
                e.printStackTrace(System.out);
            }
            System.out.println("Entering FileUploadPortlet.processAction()");
        }
    }
```

The **FileUploadPortlet** class overrides the following three methods of the **GenericPortlet** class:

- **doEdit()**: The **doEdit()** method gives control to upload.jsp for actual markup generation.
- **processAction()**: Inside the **processAction()** method, you are using the **org.apache.commons.fileupload.portlet.PortletFileUpload** class to read the file uploaded by the user. This class reads this file, calling the **actionRequest.getPortletInputStream()** method, and stores it in the object of the **FileItem** type. Once request parsing is complete, the code can iterate over the **fileItems** list and read the file item named **imageFile**. The **fileItem.get()** method will return the file uploaded by the user as a byte array. The **processAction()** method stores this byte array as the value of the **photo** attribute in the **PortletSession** object in the application scope so that this value is accessible to other components of the web application. Portlet session scope will be covered in detail in *Chapter 6: Portlet State*.
- **doView()**: Inside the **doView()** method, the value of the **photo** attribute stored in the application scope of the **PortletSession** object is checked. If it is null, the following message displays to the user: "Image not found. Please upload image using EDIT mode." If the value of the **photo** attribute is not null, the user is redirected to display.jsp, which will display the image uploaded by the user.

Next create the upload.jsp file, as shown in bold in Listing 5.12.

Listing 5.12

```
<%@ taglib uri='http://java.sun.com/portlet' prefix='portlet'%>
<portlet:defineObjects/>
<form action="<portlet:actionURL/>" method="POST" enctype="multipart/form-data">
<h3>Upload image form </h3>
      <table>
            <tr>
                  <td>Image file</td>
                  <td><input type="file" name="imageFile"></td>
            </tr>
            <tr>
                  <td><input type="submit" name="submit" Value="Submit"></td>
                  <td><input type="reset" name="Cancel" title="Cancel"></td>
            </tr>
      </table>
</form>
```

The upload.jsp file creates an HTML form asking the user for the image file. When the user selects a file here and clicks **Submit**, an action request is submitted to FileUploadPortlet.java. Please note that when you're submitting a form to a portlet, the value of your form **`method`** attribute must be **`POST`**. Also, when you want to upload a file to the server, the value of the **`enctype`** attribute must be **`multipart/form-data`**.

Create display.jsp, as shown in Listing 5.13.

Listing 5.13

```
<%@ taglib uri='http://java.sun.com/portlet' prefix='portlet'%>
<portlet:defineObjects/>

<%
      String imagePath = renderRequest.getContextPath() + "/imageServlet/photo.gif";
%>

<img src='<%= renderResponse.encodeURL(imagePath) %>' title="Image"/>
```

FileUploadPortlet.java will pass control to display.jsp, which will display the image to the user. The display.jsp file uses the **`<img>`** HTML tag to give control to the **`"/imageServlet/photo.gif"`** URL for actual image generation.

Some portlet container implementations may require that the URL for resources (such as servlets, JSPs, images, and other static files) be encoded to contain a **sessionID** string or some other container-specific data, so you should pass the absolute path or full path of the resource to the **RenderResponse.encodeURL()** method, which adds that container-specific information to the URL. Once the URL for ImageServlet is encoded, you can set it as a value of the **src** attribute for the **img** tag.

Now create the web.xml file, as shown in Listing 5.14.

Listing 5.14

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<servlet>
      <servlet-name>imageservlet</servlet-name>
      <servlet-class>
            com.sourcebeat.portlet101.chapter5.ImageServlet
      </servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>imageservlet</servlet-name>
      <url-pattern>/imageServlet/*</url-pattern>
</servlet-mapping>
</web-app>
```

You use this web.xml file to map all requests to the **/imageServlet/*** path to ImageServlet, which displays the actual image. This step brings you to the last step, which is to create ImageServlet.java, as shown in Listing 5.15.

Listing 5.15

```
public class ImageServlet extends HttpServlet{

      protected void doGet(HttpServletRequest servletRequest,
          HttpServletResponse servletResponse) throws ServletException, IOException {
            System.out.println("Entering ImageServlet.doGet()");
            try {
                  servletResponse.setContentType("image/gif");
                  HttpSession httpSession = servletRequest.getSession();
                  byte[] image= (byte[])httpSession.getAttribute("photo");
                  servletResponse.getOutputStream().write(image);
```

```
                servletResponse.getOutputStream().close();
            } catch (Exception e) {
                e.printStackTrace();
            }
                System.out.println("Exiting ImageServlet.doGet()");
        }

    }
```

ImageServlet.java is responsible for displaying images uploaded by the user. Inside the **`doGet()`** method, you set the content type to **`image/gif`** and then you wrote the image uploaded by the user as a byte array in the output stream. If you remember, the FileUploadPortlet.java stores images uploaded by users as a value of the **`photo`** attribute in the **`PortletSession`** object in the application scope. ImageServlet.java needs to read that image.

The Portlet Specification states that you can access attributes stored in the application scope of the **`PortletSession`** object through the **`HttpSession`** object and vice versa. In your sample code, the image is bound in the application scope of the **`PortletSession`** object with the name equal to **`photo`**, so you can access that image by calling the **`HttpSession.getAttribute("photo")`** method, which is what you did in ImageServlet.java. Once you have a byte array for images uploaded by users, simply write it back in an output stream.

With FileUploadPortlet ready, follow these steps to test it:

1. Build FileUploadPortlet.java and deploy it on the Pluto server by running the **`mvn -o integration-test`** command.
2. Create the FileUploadPortlet test page and add FileUploadPortlet.java to it.
3. Restart the Pluto server and go to the FileUploadPortlet test page. In **`VIEW`** mode, it will tell you to upload images in **`EDIT`** mode.
4. Go to **`EDIT`** mode, select a .gif file (/j2sdk1.4.2_11/jre/javaws/ javalogo52x88.gif), and click the **Submit** button.
5. Go back to **`VIEW`** mode to see the Java logo on that page, as shown in Figure 5.1.

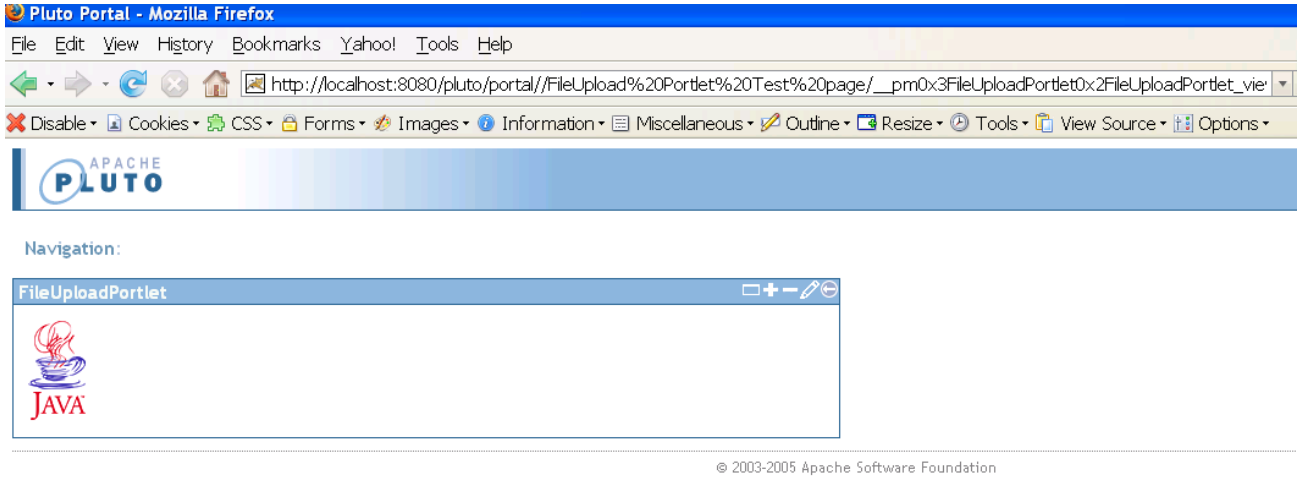

Figure 5.1: FileUploadPortlet in **`VIEW`** mode

RESTRICTIONS ON MARKUP GENERATED BY PORTLETS

Markup generated by portlets is called a *markup fragment*. Portal servers aggregate markup fragments from all the portlets on a particular portal page to build a final portal page. Therefore, some rules and limitations exist in the markup portlets generate. Basically, portlets are not allowed to generate tags that impact content generated by other portlets or tags that might even break the entire portal page. Portlets should conform to the following rules and limitations when generating content.

- Portlets generating HTML fragments must not use the following tags: **`base`**, **`body`**, **`iframe`**, **`frame`**, **`frameset`**, **`head`**, **`html`**, and **`title`**.
- Portlets generating XHTML and XHTML basic fragments must not use the following tags: **`base`**, **`body`**, **`iframe`**, **`head`**, **`html`**, and **`title`**.
- Portlet developers should decide carefully before including **`link`**, **`meta`**, and **`style`** tags in their markup. Please note that HTML, XHTML, and XHTML-basic specifications do not allow these tags to be used

outside the **head** tag, although some browser implementations support some of these tags in other sections of the document.

In addition to following these markup content restrictions, the Portlet Specification defines common CSS styles that portlets should use when generating content to achieve a common look and feel throughout the portal page. Please take a look at the Portlet Specification for more information.

In the next chapter, you will learn about portlet state and how to enable portlet intercommunication.

CAUTION: You won't get a compile-time or runtime error if you use one of the illegal tags such as **head**, **body**, or **html** in markup generated by your portlet. However, using illegal tags will break the final portal page. It's up to you as a developer to make sure that these tags are not used in markup generated by your portlet.