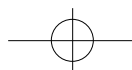


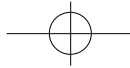
Bonus Chapter: Shedding Some Light on Pluto

Pluto is often called the “dark planet” because it is so distant from the sun that it resides in perpetual darkness. The remote orb regularly generates interest from observers because it has always been shrouded in secrecy, which some might contend is the same level of murkiness shared by the portal developers prior to the release of the long awaited Pluto reference implementation that resides in the Apache Incubator. Okay, that might be a stretch, but one cannot deny—especially portal architects and developers—that the JSR168 implementation with Pluto was shadowy for a long time, and this affected portal design decisions for quite some time over the past year. Today, many portal implementations, both commercial and open source, have implemented their own interpretations of the JSR168 standard and are working furiously to adapt their portal products to comply with the specification.

The portlet container in Pluto enables portlets to be instantiated and deployed on top of the Tomcat Web container that delivers dynamic content delivery through servlets and Java Server Pages. One should understand before diving into Pluto that the reference implementation’s primary purpose is to serve as a portlet container to enable developers to test their portlets against the JSR168 APIs that Pluto has furnished. Please remember that Pluto does not include the features that most portal frameworks possess, including portlet security realm integration, content and layout customization, search capabilities and elegant configuration management tools.

The first half of this chapter discusses important configuration files and setup procedures for Pluto using the Maven configuration tool. This discussion focuses on portlet.xml creation using XDoclet portal libraries, and the modification of XML files for visualization of a sample database query portlet. The second half of the chapter demonstrates how to render a portlet using a Java Server Page component along with the back-end portlet class to handle user invocations and request processing. It is hoped that this chapter reveals some of the language constructs and underlying concepts of the Pluto application so that you can develop JSR168-compliant portlets that can be easily migrated to other portlet frameworks for deployment, such as eXo, Jetspeed2, and Liferay.





Bonus Chapter

Creating, Installing, and Deploying Portlets with Maven

To test the Pluto Reference Implementation (RI), download the binary files from the Pluto Website at <http://jakarta.apache.org/pluto/index.html>. Two approaches can be employed to deploy the Pluto RI. The first technique involves running the ANT build script that is part of the download binaries. After the components are compiled and archived, an install script can be invoked that prompts the user for the location of your Tomcat directory for file migration and integration.

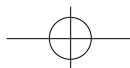
Alternatively, users can use Maven scripts as a conveyance tool to perform proper configuration management activities. The proceeding steps can be followed to deploy the test portlets for operation:

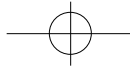
1. Download Tomcat and install.
2. Download Maven and configure (<http://maven.apache.org/>). Set the `MAVEN_HOME` environment variable to the directory in which Maven was installed.
3. Modify the `build.properties` file in the root directory. Specify the location of the Tomcat installation in which the Pluto libraries and portlets will be deployed.
4. Modify the `maven.xml` file to accommodate the portlets that will be appended to the default `testsuite` components.
5. Modify the `project.properties` file in the root directory.
6. For new portlet code, develop and append entries to the `maven.xml` script that closely resemble those of the default `testsuite` portlet. For portlets with form submission activities, develop `processAction()` methods that handle requests appropriately.
7. From the command line, run the following command: **maven fullDeployment**. Monitor your Java console for any errors.
8. Modify the `pageregistry.xml` and `portletentityregistry.xml` files for proper rendering.
9. Start the Tomcat server and type in the URL address: `http://<hostname>:<port>/pluto/portal`.

Building and Deploying Portlets with Maven

The Pluto download ships with a Maven script to facilitate build and deployment operations. Maven is an open-source build tool that incorporates a Project Object Model (POM) script, named `project.xml`, and a file named `maven.xml` to define program goals for configuration management activities. Plug-in files can also be used to enhance configuration activities. The following `maven.xml` listing defines two target goals, `allClean` and `fullDeployment`, which are invoked from the command line when the maven application is invoked:

```
001: <!-- maven.xml -->
002:
003: <project default="java:jar" xmlns:j="jelly:core"
004:           xmlns:maven="jelly:maven" xmlns:ant="jelly:ant">
005:
006:   <!-- Stuff here -->
007:
```





Shedding Some Light on Pluto

```

008: <goal name="allClean" description="Cleans a project targets">
009:   <attainGoal name="clean" />
010:   <maven:reactor
011:     basedir="${basedir}"
012:     includes="container/project.xml"
013:     goals="clean"
014:     banner="Cleaning Container"
015:     postProcessing="false"
016:     ignoreFailures="false"/>
017: </goal>
018:
019: <goal name="fullDeployment" >
020:
021:   includes="api/project.xml"
022:   includes="container/project.xml"
023:
024:   <maven:reactor
025:     basedir="${basedir}"
026:     includes="portal/project.xml"
027:     goals="jar:install,deployPlutoToTomcat"
028:     banner="Deploying Portal"
029:     postProcessing="false"
030:     ignoreFailures="false"/>
031:
032:   <maven:reactor
033:     basedir="${basedir}"
034:     includes="testsuite/project.xml"
035:     goals="deployTestsuite"
036:     banner="Deploying Test Suite to Pluto"
037:     postProcessing="false"
038:     ignoreFailures="false"/>
039:
040: </goal>
041:
042: </project>
043:

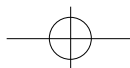
```

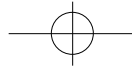
The preceding script includes the `project.xml` files on Lines 26 and 34. Each reactor element builds project components in an ordered fashion. Each reactor target in the `maven.xml` script executes individual maven scripts that reside in the independent project folders. Those project scripts run the `Deploy` class to package the portlet code for execution and to migrate components and libraries to their appropriate target directories:

```

<java classname="org.apache.pluto.portalImpl.Deploy" fork="yes">
  <classpath>
    <path refid="maven.dependency.classpath"/>
    <pathelement path="${maven.build.dest}"/>
  </classpath>
  <arg value="${tomcat.home.pluto}/Webapps" />
  <arg value="pluto" />
  <arg value="${basedir}/target/${maven.war.final.name}" />
  <arg value="../../container/target" />
</java>

```





Bonus Chapter

Four parameters are required to properly deploy portlets to your Web/portlet container:

- <TOMCAT-Webapps-directory>
- <TOMCAT-pluto-Webmodule-name>
- <Web-archive>
- <build-container-dir>

The test portlets that are packaged with the Pluto implementation pass the argument values specified in the preceding code snippet. Figure P.1 demonstrates visually what occurs when a user invokes the Pluto maven script with the `fullDeployment` target. Notice that additional library files, `pluto-1.0.jar` and `portlet-api-1.0.jar`, are migrated to the `/shared/lib` directory, as well as the `pluto.xml` file indicating that the Pluto Web application is cross-context, which enables objects to be shared across contexts with the portal application.

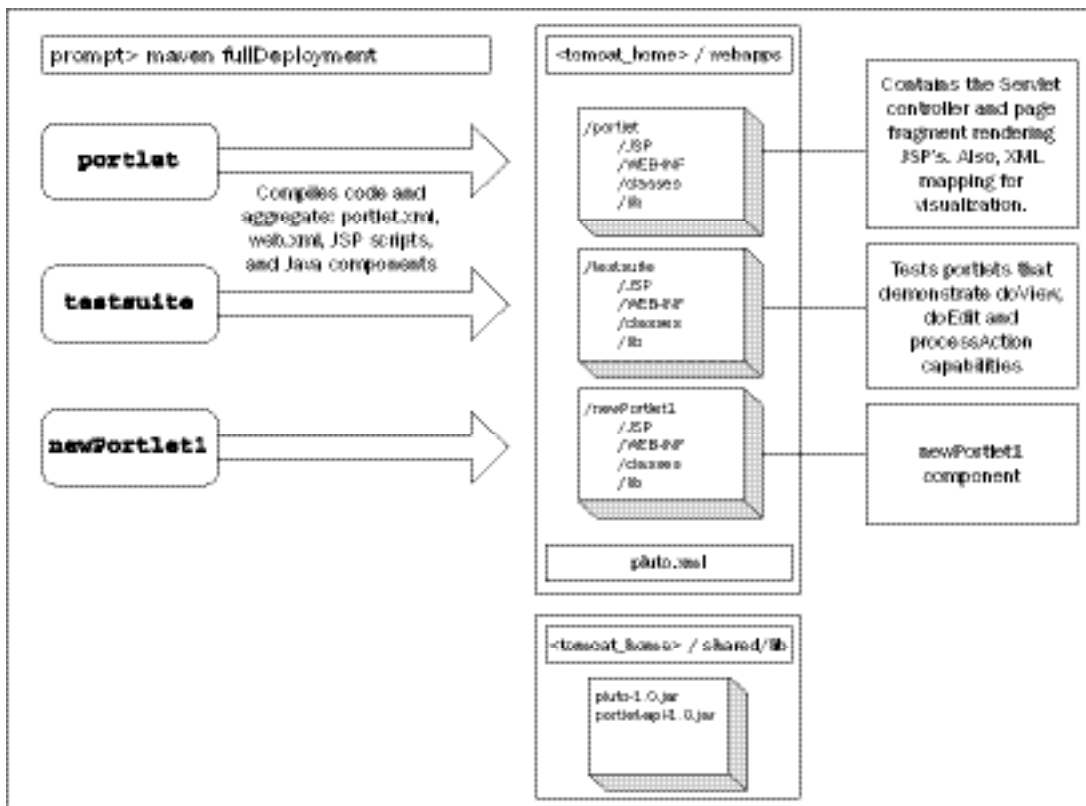
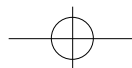
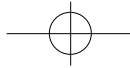


Figure P.1





Shedding Some Light on Pluto

Configuration Files for Portlet Presentation: pageregistry.xml

The `pageregistry.xml` file, which resides in the `/Webapps/Pluto/WEB-INF/data` directory, outlines all of the individual portlet components by fragment tags that are linked to metadata in the `portletentityregistry.xml` file. The default Pluto portal application contains two sample portlets that reside in the real estate indicated by the `p1` and `p2` tags, as shown in Figure P.2.

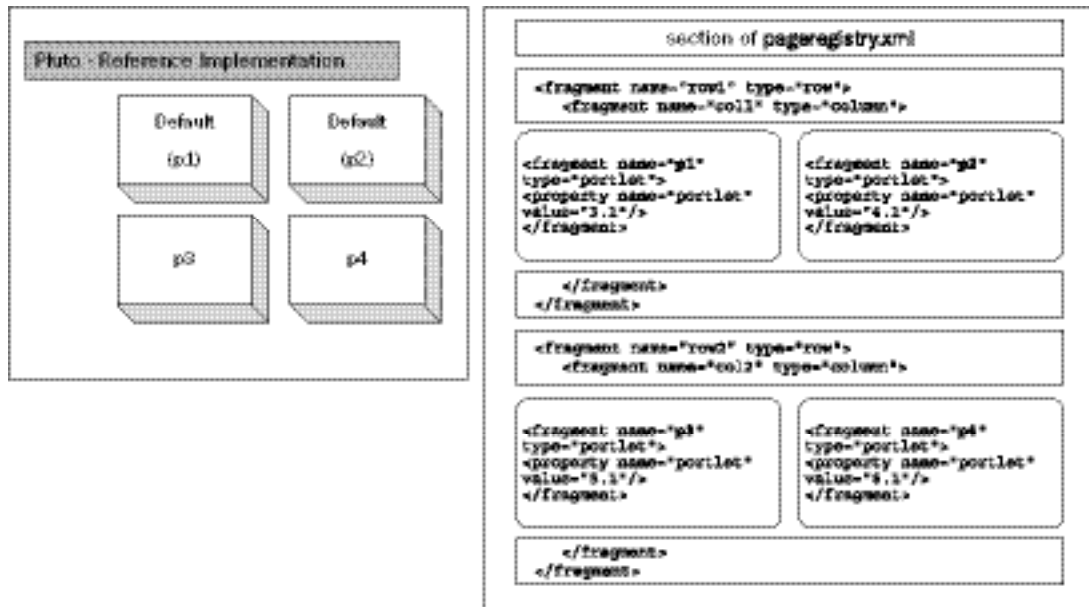
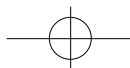


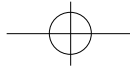
Figure P.2

The boxes labeled `p3` and `p4` are additional portlets that are affiliated with fragment names and values linked to the `portletentityregistry.xml` file.

Configuration Files for Portlet Presentation: portletentityregistry.xml

The `portletentityregistry.xml` file, which also resides in the `/Webapps/Pluto/WEB-INF/data` directory, describes the portlet components that will be rendered in the portal display. In the following code snippet, the application and portlet IDs indicate which portlet will be associated with those identification numbers:





Bonus Chapter

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-entity-registry>
  <application id="3">
    <definition-id>testsuite</definition-id>
    <portlet id="1">
      <definition-id>testsuite.TestPortlet1</definition-id>
      <preferences>
        <pref-name>TestName4</pref-name>
        <pref-value>TestValue4</pref-value>
        <read-only>true</read-only>
      </preferences>
    </portlet>
  </application>
  <!--additional portlet descriptors can be implemented below this line -->
</portlet-entity-registry>
```

Fragments

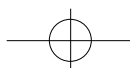
The content generated by a portlet is called a *fragment*, which is typically a piece of markup that adheres to a set of conventions. A fragment can be aggregated with other fragments to form a unified user view.

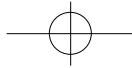
The file system structure described below is part of the `portal` Web application that renders the individual portlet components in the portlet container. The Java Server Page files in the aggregation folder shown in Figure P.3—`Banner.jsp`, `ColumnFragment.jsp`, `Head.jsp`, `PageFragment.jsp`, `RootFragment.jsp`, `RowFragment.jsp`, and `TabNavigation.jsp`—are invoked by the `Web/portlet` container to display the portlets specified in the `pageregistry.xml` file.

Pluto uses a servlet component named `Servlet.java` to control portal operations in a centralized fashion. This servlet performs presentation-tier request handling by controlling events that occur across all portlets in the `Web/portlet` container.

All of the portlet components in the `testsuite` Web application employ the use of the JSP bean `<jsp:useBean id="fragment" class="org.apache.pluto.portalImpl.aggregation.Fragment" scope="request" />` for pulling together the disparate portlet renderings.

Figure P.4 illustrates the `testsuite` Web application file system structure that the Maven scripts use to build the portlet component. Java Server Pages that generate dynamic Web content are housed in the `/Webapp` directory, while Java components that perform access operations on back-end business systems are stored in the `/java` directory.





Shedding Some Light on Pluto

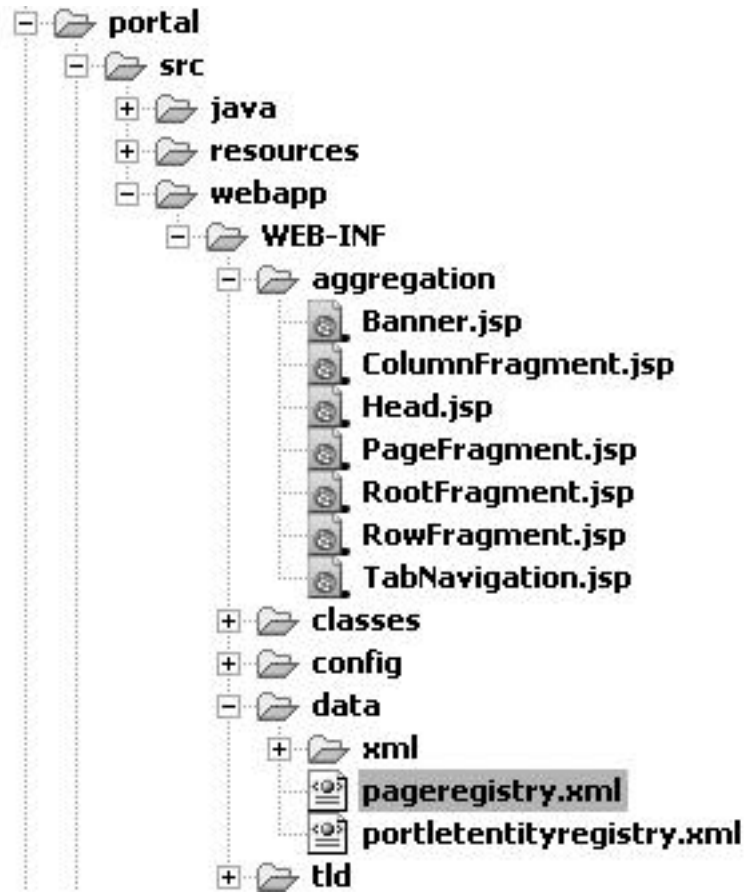


Figure P.3

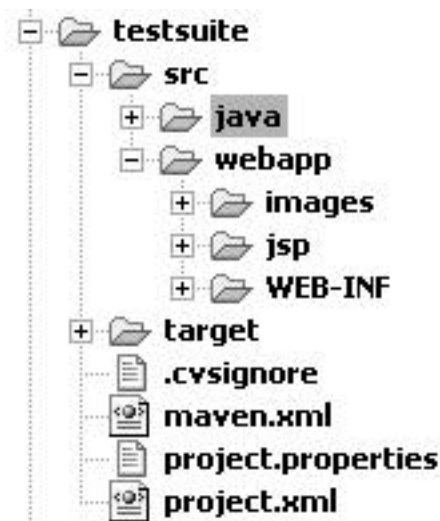
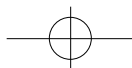
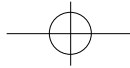


Figure P.4





Bonus Chapter

Portlet.xml Creation with XDoclet Libraries

The Deploy component that ships with the Pluto application creates and transfers the descriptor files, `portlet.xml` and `Web.xml`, to their target directories. Users not familiar with the class, or who would prefer another manner to create a new `portlet.xml` file might consider using an open-source code-generation application called XDoclet (<http://xdoclet.sourceforge.net/>).

XDoclet possesses `portletdoclet` libraries that enable developers to include JavaDoc tags that can be used to generate appropriate XML descriptors required by the `portlet.xml` file. In the following ANT code snippet, the `portletdoclet` tags would be stripped from a program header by invoking the `generatePortletXML` target in the script. The XDoclet portlet references would then be written to your designated `portlet.xml` artifact in the destination directory specified by `dest.dir`.

The following code snippet in is an ANT task that enables you to parse out the metatags from a program header using the `portletdoclet` libraries:

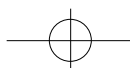
```

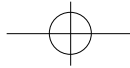
01:  <!-- define task definition -->
02:  <taskdef name="portletXML"
03:           classname="xdoclet.modules.portlet.PortletDocletTask"
04:           classpathref="xdoclet.lib"/>
05:
06:  <!-- set library classpaths -->
07:  <path id="classpath.build">
08:    <fileset dir="${lib.dir}"/>
09:    <fileset dir="${xdoclet.lib}"/>
10:  </path>
11:  <path id="xdoclet.lib">
12:    <path refid="classpath.build"/>
13:    <pathelement path="lib"/>
14:  </path>
15:
16:  <!-- generate portlet.xml file for distribution -->
17:  <target name="generatePortletXML">
18:    <portletXML destdir="${dest.dir}" mergedir="${temp.dir}">
19:      <fileset dir="${src.dir}">
20:        <include name="**/*Portlet.java" />
21:      </fileset>
22:      <portletxml/>
23:    </portletdoclet>
24:  </target>

```

The following table illustrates all of the XDoclet tags and the attributes they require for proper parsing. Again, this task is cited as an alternative to using the Deploy method of the Pluto library.

Tag	Attributes (M=mandatory) * All attributes are text
@portlet.portlet	display-name – The portlet's description expiration-cache – Time, in seconds, after which the portlet output expires name (M) – The name of the portlet





Shedding Some Light on Pluto

Tag	Attributes (M=mandatory) * All attributes are text
@portlet.portlet-info	title – Static title for the portlet keywords – Comma-separated keywords short-title – Short version of the portlet's static title
@portlet.portlet-init-param	description – The <code>init</code> parameter's description name (M) – The name of the initialization parameter value (M) – The value of the initialization parameter
@portlet.preference	read-only – If "true", the preference cannot be modified by the user name (M) – The name of the preference value (M) – The initial value of the preference
@portlet.preferences-validator	class (M) – The fully qualified name of the preferences validator
@portlet.security-role-ref	description – The description of the security role reference role-link (M) – The description of the security role reference role-name (M) – The description of the security role reference
@portlet.supports	mime-type (M) – The mime-type for which mode support is being defined modes (M) – A comma-separated list of modes supported by the specified mime-type

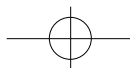
portlet.xml

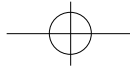
The `portlet.xml` file, which resides in your portlet application's `WEB-INF` directory, contains the portlet configuration settings associated with an individual portlet component. The following example shows a sample source code header of the XDoclet portlet tag references. Once the `portlet.xml` file is generated through the XDoclet mechanism, it should be migrated over to the Web container implementation of the portlet for which it was generated.

```

001: /**
002:  * @portlet.portlet
003:  *     name="DbqueryPortlet"
004:  *     description="Database Query Portlet"
005:  *     display-name="DBQuery"
006:  *     expiration-cache="1000"
007:  *
008:  * @portlet.supports
009:  *     mime-type="text/html"
010:  *     modes="EDIT, VIEW"
011:  *

```





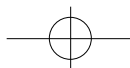
Bonus Chapter

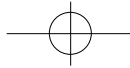
```
012: * @portlet.portlet-init-param
013: *     name="url"
014: *     value="jdbc:mysql://localhost/dbtags "
015: *
016: * @portlet.portlet-init-param
017: *     name="driver"
018: *     value="org.gjt.mm.mysql.Driver"
019: *
020: * @portlet.portlet-init-param
021: *     name="scot"
022: *     value=""
023: *
024: * @portlet.portlet-init-param
025: *     name="password"
026: *     value="schrager"
027: *
028: * @portlet.preference
029: *     name=""
030: *     value=""
031: *
032: * @portlet.preference
033: *     name=""
034: *     value=""
035: *
036: * @portlet.preference-validator
037: *     class="com.portalBook.DBQueryPreferenceValidator"
038: *
039: * @portlet.portlet-info
040: *     title="Database Query Portlet"
041: */
```

Sample Database Query Portlet Implementation

The Pluto implementation was not meant to demonstrate comprehensive features of a portal framework; its intent was to provide a JSR168-compliant container to ensure that portlets could be easily migrated across other portal platforms. The Database Query portlet shown in Figure P.5 is a fairly simple implementation of a JSR168-compliant portlet that displays the contents of the `test_books` table using the DBTags tags of the JSTL library (<http://jakarta.apache.org/taglibs/index.html>). If the portlet user clicks the Delete link associated with a table entry, then that particular entry will be deleted from the table. If the portlet user inputs a unique ID, author, and description in the add form of the portlet and clicks the Add button, the portlet will input the author information into the `test_books` table, which will be presented in the portlet display when the portlet is rendered again in the overall portal display.

The Database Query portlet shown here is fairly simplistic, and does have room for a validation object to ensure that users add suitable values for the form component.





Shedding Some Light on Pluto

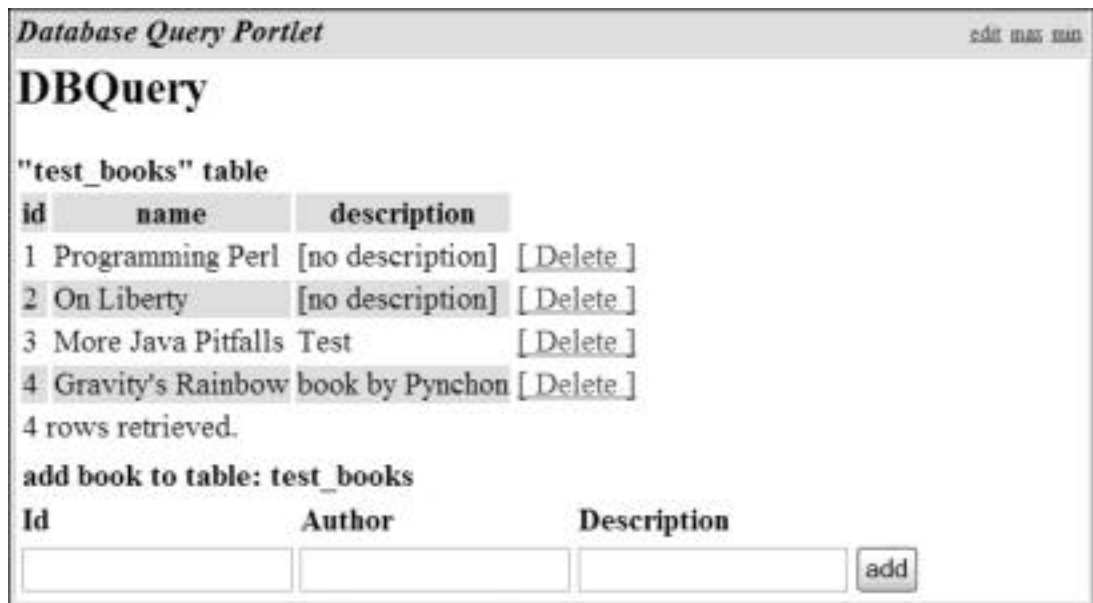


Figure P5

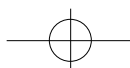
Lines 15–18 of the Java Server Page (`view.jsp`) define the URL and Driver class information needed by the DBTag libraries to connect to the `dbtags` table:

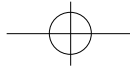
```

001: <!-- view.jsp -->
002:
003: <%@ page session="false" %>
004: <%@ page import="javax.portlet.*"%>
005: <%@ page import="java.util.*"%>
006:
007: <%@taglib uri='/WEB-INF/tld/portlet.tld' prefix='portlet'%>
008: <%@taglib uri='/WEB-INF/tld/taglibs-dbtags.tld' prefix='sql' %>
009:
010: <jsp:useBean id="addUrl" scope="request" class="java.lang.String" />
011: <portlet:defineObjects/>
012:
013: <h2>DBQuery</h2>
014:
015: <sql:connection id="conn1">
016:   <sql:url>jdbc:mysql://localhost/dbtags</sql:url>
017:   <sql:driver>org.gjt.mm.mysql.Driver</sql:driver>
018: </sql:connection>
019:

```

The following code snippet reads all of the entries in the `test_books` table for presentation in a portlet. An action URL is created on Line 42–44 so that ID numbers associated with the individual database items can be passed to the `DbqueryPortlet` class:





Bonus Chapter

```

020: <B>"test_books" table</B>
021:
022: <table border="0">
023: <tr><th bgcolor="#dcdcdc">id</th>
024:   <th bgcolor="#dcdcdc">name</th>
025:   <th bgcolor="#dcdcdc">description</th>
026: </tr>
027: <sql:preparedStatement id="stmt2" conn="conn1">
028:   <sql:query>
029:     select id, name, description from test_books
030:     order by 1
031:   </sql:query>
032:   <% int i=0; %>
033:   <sql:resultSet id="rset1">
034:     <tr>
035:       <%
036:         String bgColor = (i%2==0) ? "#ffffff" : "#dcdcdc";
037:         %>
038:       <td bgcolor="<%=bgColor%>"><sql:getColumn position="1"/></td>
039:       <td bgcolor="<%=bgColor%>"><sql:getColumn position="2"/></td>
040:       <td bgcolor="<%=bgColor%>"><sql:getColumn position="3"/><sql:wasNull>[no
description]</sql:wasNull></td>
041:       <td>
042:         <portlet:actionURL var="removeUrl">
043:           <portlet:param name="dbItem" value="<%=rset1.getString(1)%>" />
044:         </portlet:actionURL>
045:         <a href="<%=removeUrl.toString()%>">[ Delete ]</a>
046:       </td>
047:     </tr>
048:     <% i++; %>
049:   </sql:resultSet>
050:   <tr>
051:     <td colspan="4">
052:       <sql:wasEmpty>No rows retrieved.</sql:wasEmpty>
053:       <sql:wasNotEmpty><sql:rowCount/> rows retrieved.</sql:wasNotEmpty>
054:     </td>
055:   </tr>
056: </sql:preparedStatement>
057: </table>
058:

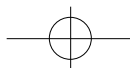
```

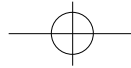
Lines 70–75 of the following code snippet encompass the `addUrl` form, which passes ID, author, and description data to the `DBQueryPortlet` for SQL construction, which is used to add data to the `test_books` database table:

```

059: <table>
060: <tr><td colspan="4">
061: <B>add book to table: test_books</B>
062: </td></tr>
063: <tr>
064: <td><b>Id</b></td>
065: <td><b>Author</b></td>
066: <td><b>Description</b></td>

```





Shedding Some Light on Pluto

```

067: <td>&nbsp;</td>
068: </tr>
069: <tr>
070: <FORM ACTION="<%=addUrl%" METHOD="POST">
071: <td><INPUT NAME="id" TYPE="text"></td>
072: <td><INPUT NAME="author" TYPE="text"></td>
073: <td><INPUT NAME="description" TYPE="text"></td>
074: <td><INPUT NAME="add" TYPE="submit" VALUE="add"></td>
075: </FORM>
076: </tr>
077: </table>
078:

```

Figure P.6 outlines the two different portlet operations that the portal user can perform on the Database Query portlet. When the portlet user clicks the [Delete] link adjacent to the individual database entries, the IDs affiliated with those items are passed to the `DbqueryPortlet` so that a SQL Delete operation can be crafted to drop that ID from the `test_books` table.

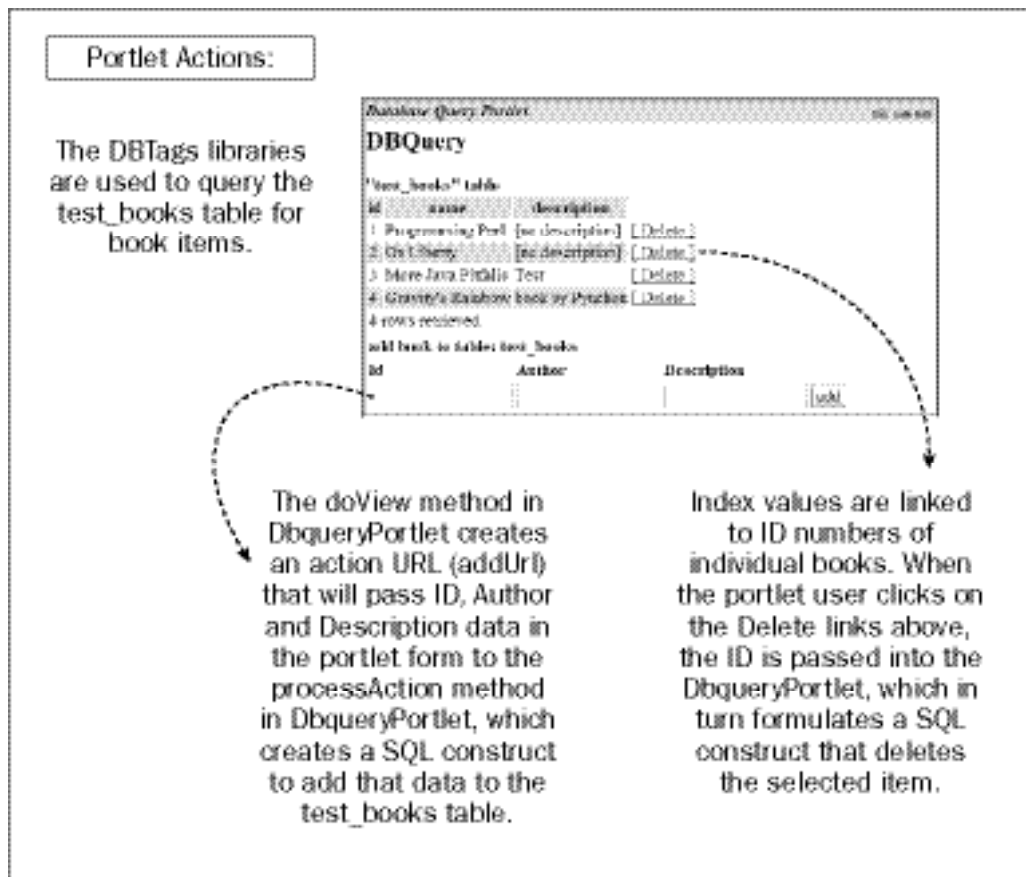
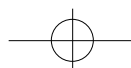
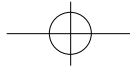


Figure P.6





Bonus Chapter

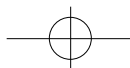
Lines 11–14 of the `DbqueryPortlet` class establish the parameters needed by the JDBC driver to connect to the `dbtags` database so that SQL operations can be performed on that table. The `logger` API is implemented on Line 15 so that the user may look at the Java console to better understand what operations are being executed during portlet operations.

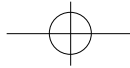
```
001: <!-- DbqueryPortlet.java -->
002:
003: package com.portalBook;
004: import java.sql.*;
005: import javax.portlet.*;
006: import java.io.IOException;
007: import java.util.logging.*;
008:
009: public class DbqueryPortlet extends GenericPortlet {
010:
011:     private static final String DRIVER_NAME="org.gjt.mm.mysql.Driver";
012:     private static final String DB_URL="jdbc:mysql://localhost/dbtags";
013:     private static final String USERNAME="";
014:     private static final String PASSWORD="";
015:     private static Logger logger = Logger.getLogger("portal");
016:
017:     public void init() {
018:         logger.info("[DbqueryPortlet:init]");
019:     }
020:
021:     public void render() {
022:         logger.info("[DbqueryPortlet:render]");
023:     }
```

The `processAction` method shown in Lines 24–67 is called when a portal user submits a form component in a portlet. An action URL activates the `processAction` method, which completes its operations and then invokes the `render` method for content visualization. Two parameters are passed into the `processAction` method, `ActionRequest` and `ActionResponse`. The `ActionRequest` object exposes request data that includes session, preference, mode, portal context, and window state information. During an action request, the `ActionResponse` object can be used to change window state and portlet mode data.

The `dbItem` request parameter on Line 31 is affiliated with each individual book ID number in the `test_books` table. If a non-null value is passed into the method, the method will form a SQL construct that deletes the database entry associated with that ID. If the `addOperation` is not null, a SQL construct will be created that aggregates ID, author, and description data for entry into the `test_books` table.

```
024:
025:     public void processAction (ActionRequest request,
026:                               ActionResponse actionResponse) throws
PortletException, java.io.IOException {
027:
028:         logger.info("[DbqueryPortlet:processAction]");
029:
030:         String addOperation = request.getParameter("add");
031:         String dbItem = request.getParameter("dbItem");
```

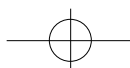


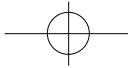


Shedding Some Light on Pluto

```
032:         String id = request.getParameter("id");
033:         String count = request.getParameter("count");
034:         String author = request.getParameter("author");
035:         String description = request.getParameter("description");
036:
037:         String sqlStatement = "";
038:
039:         if (dbItem!=null) {
040:             sqlStatement = "delete from test_books where id = '" + dbItem +
041:                 "'";
042:             logger.info("[DbqueryPortlet:processAction] REMOVE being
043:                 performed.");
044:         }
045:         if (addOperation!=null) {
046:             sqlStatement = "insert into test books (id, name, description)
047:                 values (" +
048:                     new Integer(id) + "','" +
049:                     author + "','" +
050:                     description + "')";
051:             logger.info("[DbqueryPortlet:processAction] ADD being
052:                 performed.");
053:         }
054:         if ((addOperation!=null) || (dbItem!=null)) {
055:             try {
056:                 Connection conn = DriverManager.getConnection(DB_URL,
057:                     USERNAME, PASSWORD);
058:                 Statement stmt = conn.createStatement();
059:                 logger.info("[DbqueryPortlet:processAction] sqlStatement= " +
060:                     sqlStatement);
061:                 ResultSet rslt = stmt.executeQuery(sqlStatement);
062:                 rslt.close();
063:                 stmt.close();
064:                 conn.close();
065:             }
066:             catch(SQLException se) {
067:                 logger.info("[DbqueryPortlet:processAction] SQLException." +
068:                     se.toString());
069:             }
070:         }
071:     }
072: }
```

Lines 69–87 of the following code snippet outline the `doView` method, which handles the `VIEW` requests of a targeted portlet component. This method generates an action URL for the `add` operation in the Database Query portlet in Figure P.1. The `PortletRequestDispatcher` object is used by a portlet to execute the `render` method of the Portlet interface. In the following code, the `jspView` parameter is culled from the `<init-param>` tag in the `portlet.xml` file. This token is associated with a Java Server page (`view.jsp`) for content aggregation and presentation.





Bonus Chapter

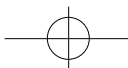
```
068:
069:     public void doView (RenderRequest request,
070:                         RenderResponse response) throws PortletException,
IOException {
071:
072:         response.setContentType("text/html");
073:
074:         String jspName = getPortletConfig().getInitParameter("jspView");
075:
076:         logger.info("[DbqueryPortlet:doView] generating action urls");
077:
078:         // generate action urls
079:         PortletURL addUrl = response.createActionURL();
080:         addUrl.setPortletMode(PortletMode.VIEW);
081:         addUrl.setParameter("add", "add");
082:         request.setAttribute("addUrl", addUrl.toString());
083:
084:         PortletRequestDispatcher rd =
getPortletContext().getRequestDispatcher(jspName);
085:
086:         rd.include(request, response);
087:     }
088:
```

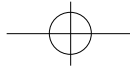
Lines 89–99 of the following code show the `doEdit` and `doHelp` methods that are summoned by the portlet container when the portal user clicks the Edit and Help links/buttons on the portlet title bar. Portlets extend the abstract `GenericPortlet` class on Line 8 to handle render requests by the portal user. The `doDispatch` method of the `GenericPortlet` class enables the processing of requests for the different modes, such as Edit and View.

```
089:     public void doEdit (RenderRequest request,
090:                         RenderResponse response) throws PortletException,
IOException {
091:
092:         logger.info("[DbqueryPortlet:doEdit] ");
093:     }
094:
095:     public void doHelp (RenderRequest request,
096:                        RenderResponse response) throws PortletException,
IOException {
097:
098:         logger.info("[DbqueryPortlet:doHelp] ");
099:     }
100:
101: }
```

Summary

The JSR168 specification implemented in the Pluto application establishes a standard API for creating portlets that can migrate across different portal implementations that claim to adhere to the specification. Granted, this demonstration has only scratched the surface of the implementation itself; it might





Shedding Some Light on Pluto

be desirable to build portlets and deploy them in a comprehensive portal framework to gain a better appreciation of how an individual portlet is viewed spatially among other portlets, and to perform benchmark tests to envision performance measures when content is aggregated for visualization.

Pluto can be an important development tool for your portlet development operations, but remember that it is just a JSR168-portlet container implementation for portlet development, and does not address complicated portal features that include portlet personalization, security, customization, and collaboration services. However, Pluto enables you to develop portlets without the overhead of a complicated portal framework, which enables users to better appreciate the properties of the JSR168 specification and the request handling activities that occur during portlet operations that ought to propagate across compliant server applications.

