to return back from the input page without saving. The following code segment shows the creation of the two URI strings.

```
// Create the URI with an action to save the form data
// Add the URI string to the request for use by the JSP
PortletURI saveURI = portletResponse.createURI();
saveURI.addAction("save");
portletRequest.setAttribute("saveURI", saveURI.toString());

// Create the URI to return to the previous (view) mode
// without saving. Add URI string for use by the JSP
PortletURI cancelURI = portletResponse.createReturnURI();
portletRequest.setAttribute("cancelURI", cancelURI.toString());
```

If the portlet class has implemented the `ActionListener` interface then that class's `actionPerformed(ActionEvent event)` method is invoked when the HTTP request associated with that URI is processed by the portal. The `ActionEvent` object that is passed when the `actionPerformed` method is invoked will contain the name of the action event, as is shown in the following code sample.

```
public void actionPerformed(ActionEvent event)
    throws PortletException {

  PortletRequest request = event.getRequest();
  String actionName = event.getActionString();

  if ("save".equals(actioName)) {
    // Retrieve the form data from the request
    // object and save it.
  }
}
```

This is the same name that was used in the `addAction(actionName)` method on the `PortletURI`. Using action names, you can distinguish the action event notifications that get processed by the portlet class and respond appropriately. The following `actionPerformed()` method shows how this is performed.

This same URI creation function is available through the portlet API JSP tag library. The following tags are available and behave as expected given the previous discussion. A required parameter "name" specifies the action name for the `URIAction` tag.

```
<portletAPI:createReturnURI />
<portletAPI:createURI />
<portletAPI:URIAction />
```

JSR 168 API Addressability

As we said, references to a specific portlet, portal page, or portal resource must be encoded in the portal Web application URI. The JSR 168 API creates a `PortletURL` to assist with this encoding. The `RenderResponse` object has two methods to create URLs, the `createActionURL()` method and the `createRenderURL()` method. The `createActionURL()` method will create an action URL while the `createRenderURL()` method creates a render URL.

There are two main differences to the portlet developer in the generation and handling of portlet URLs. The first is that with the JSR 168 API you do not have a single URL for the portlet to which you optionally add a named action event, which then triggers an event notification and listener. Instead, this API defines two different URLs, one for the action phase processing and one for the render phase.

The following code fragment shows the creation of two `PortletURLs` to handle our example to provide the links for a button to save the form data and a button to cancel without saving.

```
// Create the save URL for the page and add the
// URL string to the request for use by the JSP
PortletURL saveURL = renderResponse.createActionURL();
saveURL.setParameter("action", "save");
renderRequest.setAttribute("saveURL", saveURL.toString());

// Create the cancel URL for the page and add the
// URL string to the request for use by the JSP
PortletURL cancelURL = renderResponse.createActionURL();
renderRequest.setAttribute("cancelURL", cancelURL.toString());
```

The other difference between this approach and the IBM Portlet API is how the action is identified. Portlets can add parameters to the `PortletURL`. Typically you would add a parameter (name–value pair) to an action URL to indicate the action to be performed. This name is analogous to the action name you added to the `PortletURI` in the IBM Portlet API. It is a name that you can later interrogate in the action phase's `processAction()` method.

The JSP tag library with the JSR 168 API defines custom tags that provide the creation of `PortletURLs` in the JSP. The tags are `<portlet:actionURL/>` and `<portlet:renderURL/>`.

```
public void processAction(ActionRequest request,
    ActionResponse response) throws PortletException, IOException {

  String actionName = request.getParameter("action");
  if ("save".equals(actionName)) {
```

```
        // Retrieve the form data from the request
        // object and save it.
    }
}
```

Portlet Window

Both APIs provide maximize, minimize, and normal window states. The JSR 168 API also allows portlets to define custom window states. Custom window states are defined by the portal. Portlets, in their deployment descriptors, declare the window states that they support using the custom-window-state element. These can be defined in the portlet deployment descriptor. The portlet can query these state definitions using the `getSupportedWindowStates()` method on the `PortletContext` object. The portlets can change their window state when processing an action request.

A portlet window is defined by the JSR 168 API as the occurrence of a portlet and a `PortletPreferences` object on a portal page. There can be more than one portlet window for a given portlet on a page. Each portlet window associated with its own portlet mode, render parameters, and `PortletPreferences`.

Developing JSR 168 Portlets for WebSphere Portal

The Portal Toolkit V5.0.2 includes JSR 168 API support. The toolkit provides the ability to generate a portlet application project that uses the JSR 168 Portlet API and also includes the JSR 168 portlet tag library declaration file (`std-portlet.tld`).

The toolkit provides a JSR 168 project wizard for creates new portlet application projects which extend the `GenericPortlet` class defined in the JSR 168 "Portlet Specification."

Enabling the Portlet Container for JSR 168

The JSR 168 support is provided in WebSphere Portal 5.0 with fix pack 2 (version 5.0.2) and subsequent releases. In version 5.0.2 you must apply interim fix PQ77263 to the WebSphere Application Server.

The portlet container for JSR 168 portlets is not enabled by default with WebSphere Portal 5.0.2. You can enable this support by setting the `portal.enable.jsr168` property to true in the `ConfigService.properties` file. That properties file is in the `<wp-root>/shared/app/`

`config/services/` directory. The portal server will need to be restarted for this change to take effect.

Which API Should You Use for Your Portlet?

While both portlet APIs are supported in the current version of WebSphere Portal and will be supported for announced future product enhancements, the focus of IBM's development effort and IBM's commitment to open standards should be on the JSR 168 API being the API of choice.

In the near term, the JSR 168 API implementation is not as functionally rich as the IBM Portlet API as we have seen in this chapter. However, that gap is expected to close quickly as the API and its implementation matures.

There is no immediate need to migrate existing portlets to the JSR 168 API. For development of new portlets you should consider using the JSR 168 API when the functional support meets the needs of the portlet, and also if the portlet is going to be published as a Web service since JSR 168 portlets can be published as a Web Service for Remote Portlets (WSRP).

Summary

In this chapter you learned about the portlet APIs supported in WebSphere Portal. We discussed the differences between both the IBM Portlet API and the JSR 168 API in the contest of key portlet topic areas.

In the following chapters we will develop an example portlet using both ofthese APIs to understand how these APIs are applied and to highlight the differences by using a common example.

CHAPTER 15

WebSphere Portlet Development Environment

In this chapter you will learn about portlet development using WebSphere Studio and the IBM Portal Toolkit. Both of these components are part of the WebSphere Portal package. WebSphere Portal version 5.0 ships WebSphere Studio Site Developer (WSSD or WebSphere Studio) and the Portal Toolkit. We will introduce the WebSphere Studio development environment and the Portal Toolkit extension that assists in portlet development.

We will introduce a sample portlet application and then work through its design and development in the next chapter. We will focus on showing key aspects of the Portlet APIs and in the process show the Studio development environment. We will use WebSphere Studio version 5.1.0 and the Portal Toolkit version 5.0.2 throughout these examples.

This sample portlet will be implemented using both the IBM Portlet API and the JSR 168. We will then compare the results seeing the differences in the two implementations.

An Example Poll Portlet

In this example we will look at creating a portlet that handles an online poll. The portlet will present a question and a series of answer choices from which the user can make a selection and then register a vote.

Figure 15-1 shows a screen capture of a portal page that contains the Poll portlet. If the user who is logged in to the portal has not yet voted, the portlet displays the question and answer choices. When the user has voted, the view will change to display a graphical representation of the current voting totals for the answer options, as shown in Figure 15-2.

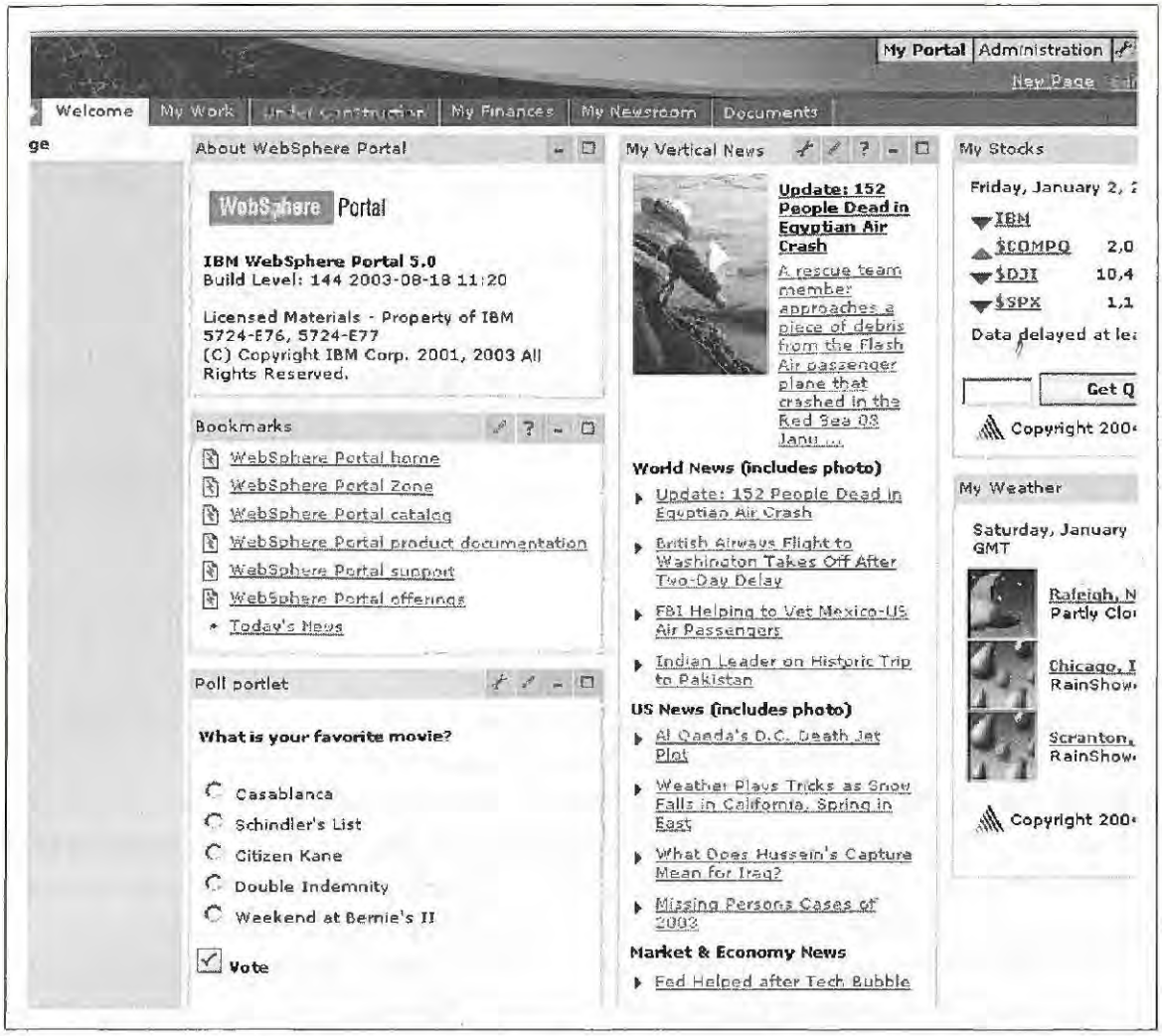


Figure 15-1 Portal page with poll portlet.

The portlet will also provide both an edit mode and a configure mode. In configure mode the administrator will (1) define the persistence store option for the portlet, (2) define the name of a new poll to be created or select an existing poll to be associated with this portlet instance, and (3) modify the poll question and answer selection choices. A portlet instance can be configured to display a poll that is in progress (votes have been recorded against the poll with that name), but in that case the poll question and answer choices cannot be modified.

Figures 15-3 to 15-5 show the configuration pages of the portlet. In the first page the persistence store is defined. The portlet will need to store the portlet question and answer choices as well as the number of votes cast. This poll information can be held in memory or in a database. If the data is held in memory then it is lost when the portal is restarted. This option is useful for demonstration and portlet testing purposes. The poll data is not actually persistent. If a database is used to store the poll information then the portlet

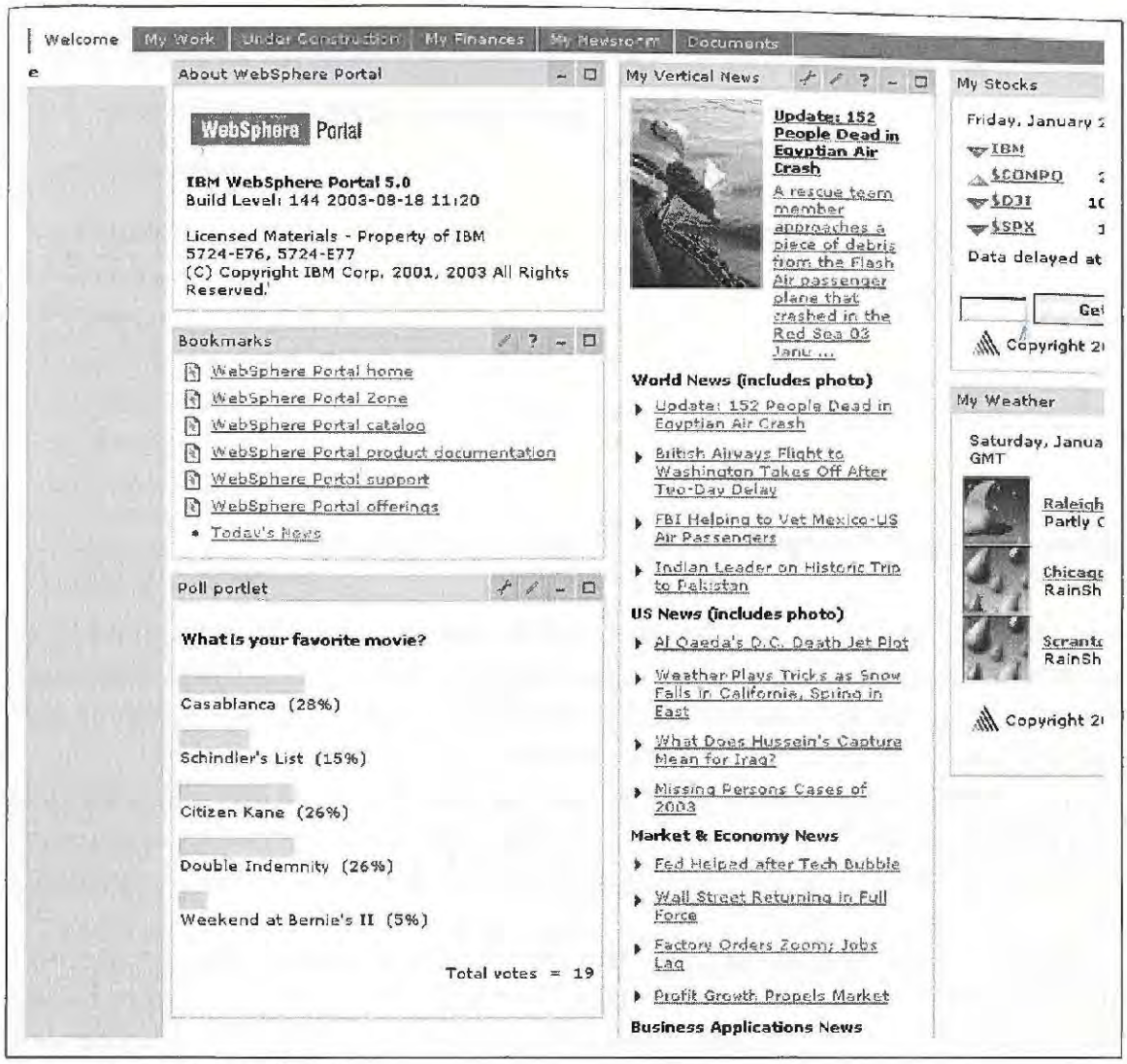


Figure 15-2 The Poll portlet.

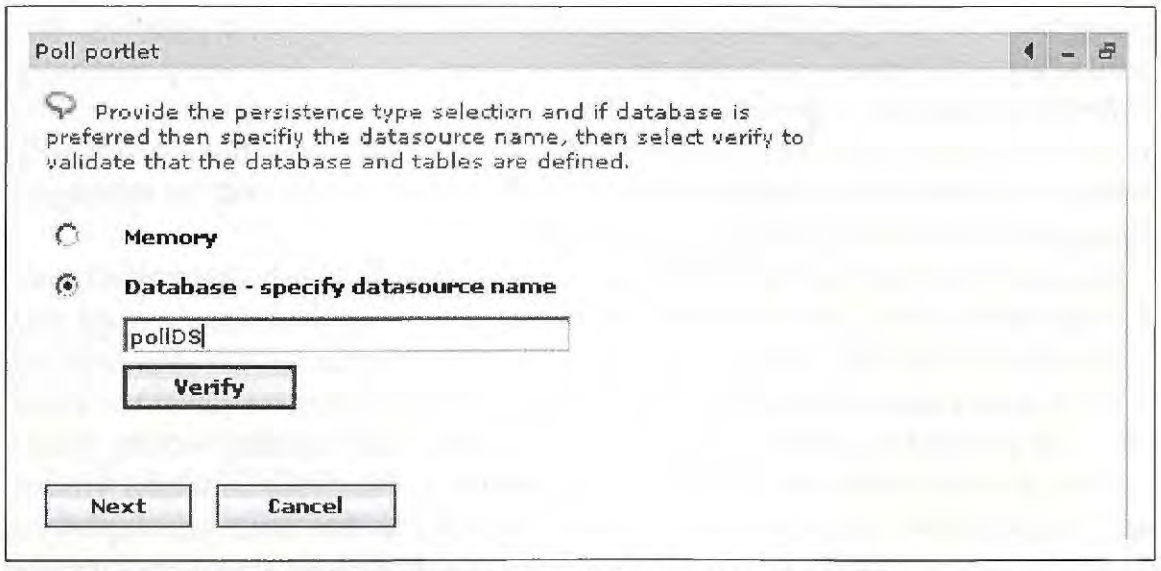


Figure 15-3 Portlet configuration page specifying persistence store.

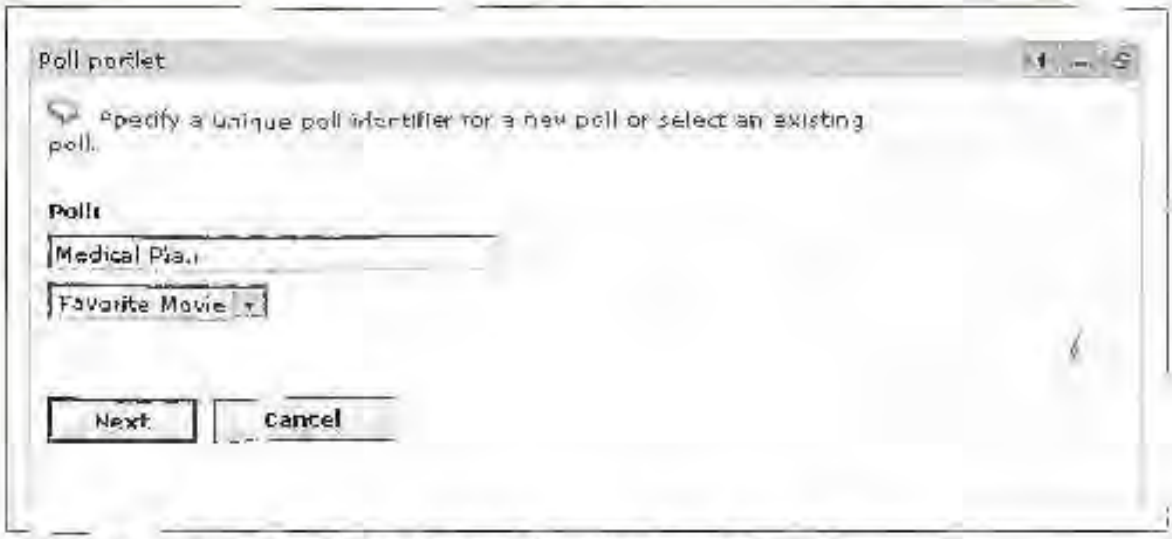


Figure 15-4 Portlet configuration page identifying poll by name.

must be configured to specify the database information. The database data source name must be specified here. The Verify button will test to ensure that the datasource name is defined correctly in the application server and that the tables have been correctly created to support the portlet.

In the next configuration page the portal administrator identifies the poll by a unique name (see Figure 15-4). A new poll can be created by specifying a unique name here or an existing poll can be chosen from the drop-down list.

In the last configuration page the portal administrator defines the poll question and the answer choices (see Figure 15-5). If the poll selected is an exiting poll that is in progress, then the question and answer choices cannot be modified.

These portlet configuration changes are applicable to the concrete portlet instance. You can copy the portlet multiple times and put the copied versions on portal pages each defining a different poll (also with potentially different persistent store options). These configuration parameters will be held in `PortletSettings` when implemented using the IBM Portlet API. When implemented using the JSR 168 API, these values will be managed through the `PortletPreferences` object.

Finally, there is an edit mode for the portlet. In edit mode the portlet user (as opposed to the portal administrator who configured the portlet) can customize the portlet (see Figure 15-6). For this simple portlet the level of user-specific customization is minimal. We allow the user to select the color that will be used on the bar graph, which displays the current voting totals for the poll. As with the configuration settings, the customization values can be specified uniquely for each poll portlet that the user has deployed on portal pages. Unlike the configuration portlet settings these options are

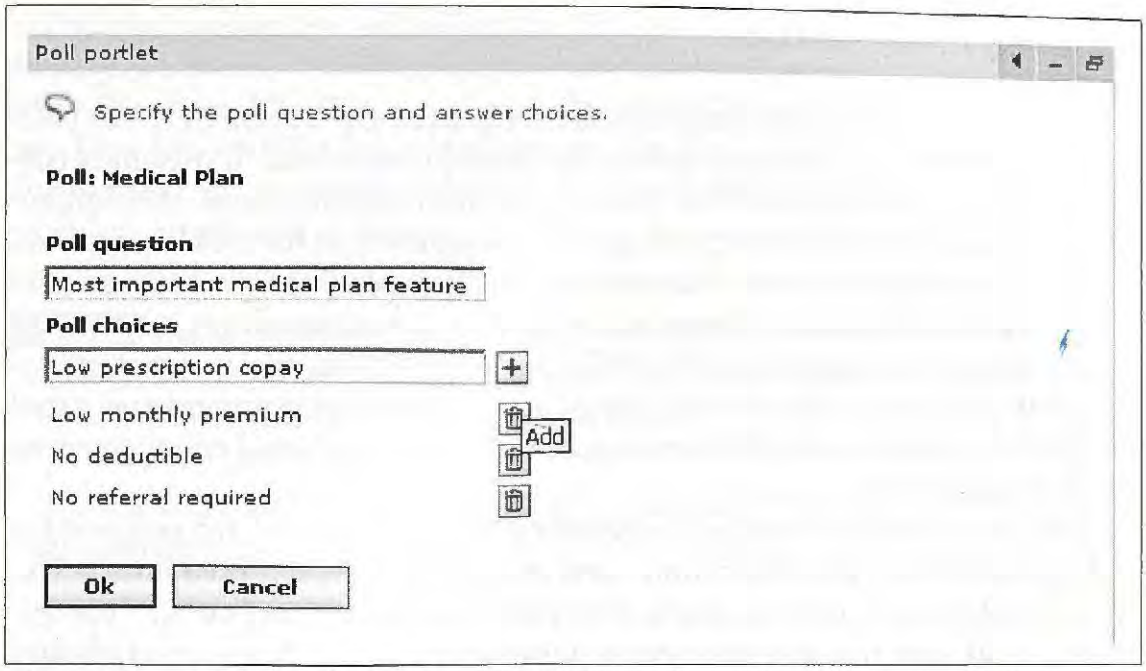


Figure 15-5 Portlet configuration page defining poll question and answer choices.

specific to a user. Using the IBM Portlet API these parameters are held in `PortletData`. When the portlet is implemented using the JSR 168 API these parameters are held in `PortletPreferences`.

In edit mode the user can select a specific color or select to have the color be the same as the color of the portlet toolbar as defined by the current portlet skin.

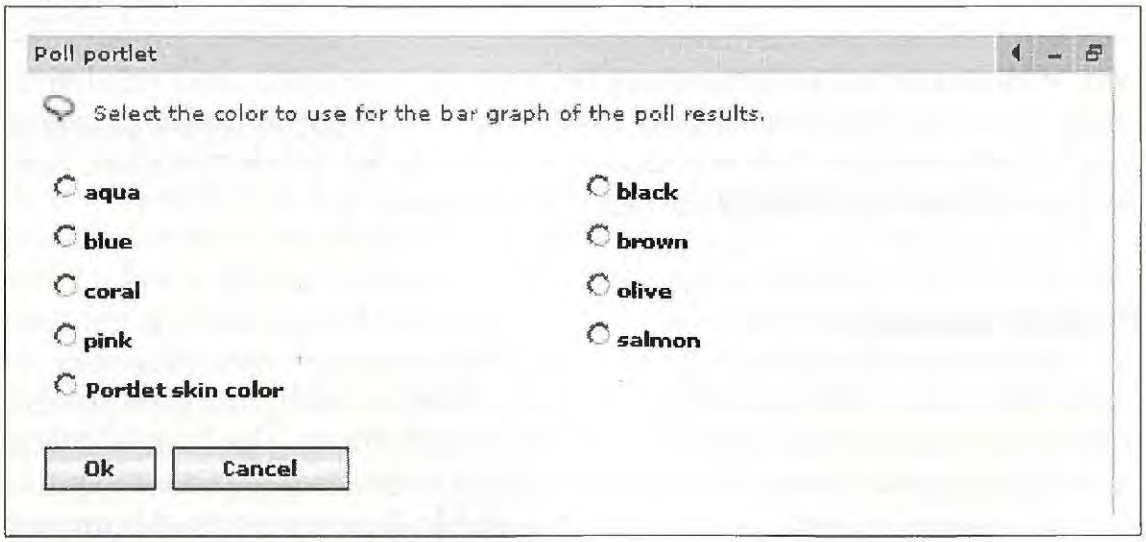


Figure 15-6 Edit mode: color selection.

WebSphere Studio

WebSphere Studio is the integrated development environment (IDE) platform of choice for building portlets for WebSphere Portal. It provides editors, builders, and wizards for Java application development. WebSphere Studio is built on Eclipse technology, an open platform for tool integration.

Eclipse provides common function on its open platform for tool vendors to provide advanced development environment functionality such as team programming, the user workbench and user interface, and interactive debugging. Eclipse is both an integrated development environment and a tool integration platform. WebSphere Studio provides functional enhancements to the Eclipse IDE.

WebSphere Studio version 5 supports JDK 1.3 and provides support for adding additional JRE definitions and switching between Java levels. The Java development environment provides automatic incremental compilation, local and remote interactive debugging, customizable and configurable workbench perspectives, views, and editors. The workbench also provides an extensive search capability and a scrapbook where Java code snippets can be evaluated easily and execution results inspected.

WebSphere Studio provides a rich set of tools to support Web application development. WebSphere Studio version 5 supports development of Web applications that meet the J2EE 1.3 specification, which includes the Sun Microsystems Java Servlet 2.3 specification and the Sun Microsystems JSP 1.2 specification. Web application support includes Web project creation. This creates an application project using the J2EE hierarchy and includes the creation of the Web application deployment descriptor file (`web.xml`).

The development environment supports Java Server Pages (JSP) creation, validation, editing, and debugging. JSP tooling also includes support for JavaScript editing and validation and custom JSP 1.2 tag development support. Web Archive (WAR) files can be imported, exported, and validated. There are a variety of additional tools for editing images and Cascading Style Sheets (CSS), a Web site designer, wizards for servlet creation, and integration with the WebSphere test environment.

Portal Toolkit

The IBM Portal Toolkit extends WebSphere Studio further by providing a complete set of tools for developing portlet applications. The Portal Toolkit provides support to create, debug, and deploy individual portlets and portlet applications. Wizards and templates assist in developing portlet project and portlets, while powerful interactive debugging capability dramatically reduces development and test time for portlet developers.

The mechanism for extending the Eclipse-based Studio environment is through plug-ins. The development environment is designed for extensibility. The Portal Toolkit is a plug-in for either WebSphere Studio Application Developer (WSAD) or WebSphere Studio Site Developer (WSSD). As mentioned before, WebSphere Portal ships with WSSD. We refer to either product generically as Studio.

NOTE The latest information, including documentation and downloads, about the Portal Toolkit is available on the WebSphere Portal Toolkit Web site at `www.ibm.com/websphere/portal/toolkit`.

The toolkit provides the following portlet development tools and IDE enhancements.

Portlet Project

A new project type is defined for portlet applications and a wizard tool is available to create portlet projects. A portlet project is similar to a J2EE Web application project and includes a Web deployment descriptor that defines the Web application. In addition, a portlet project contains a portlet deployment descriptor file (`portlet.xml`) that defines the portlet application and the portlets within that application.

The portlet application wizard assists in the creation of the project directory structure, the Web deployment and the portlet deployment descriptors, and a set of sample Java source files that make up the portlet application. The amount of generated code is dependent on the type of portlet application that was selected in the wizard: an empty portlet application or a basic portlet application.

Portlet Perspective

A new perspective is provided for portlet development. A perspective in Studio defines a set of views and the layout of those views along with a set of editors. Studio defines multiple perspectives each aimed at a specific task or a specific type of resource. For example, the Web perspective shows workbench views that are appropriate to Web application resources. The debug perspective shows views that apply to the interactive debug of Java applications.

The portlet perspective is similar to the Web perspective provided by Studio, showing views and editors for Web application resources. In addition, this perspective provides an editor specific for the portlet deployment descriptor file (`portlet.xml`). The `portlet.xml` editor also assists in

keeping the portlet and Web deployment descriptors in sync. As discussed in a previous chapter there are links between the Web deployment descriptor and the portlet deployment descriptor. The servlet ID defined in the `web.xml` file must link to the portlet href attribute in the portlet application definition in the `portlet.xml` file. The `portlet.xml` editor manages this and other associative descriptor attributes. The editor also provides a validator to check the deployment descriptor definitions.

Portal Server Configuration

The Portal Server Configuration defines a WebSphere Studio server configuration that allows you to publish your portlet application to WebSphere Portal. Published portlets are deployed on a debug page of WebSphere Portal. Interactive debugging is supported.

A portlet application can be published and debugged in a local portal test environment or a remote (attached) portal server. Local debugging requires that you install the WebSphere Portal for a test environment. You can use the Portal Toolkit installer to set up the Portal test environment. Once installed and started you can interactively make changes to your portlet in Studio and immediately test the change on the local portal without an additional package, publish, or deploy step.

You can use remote debugging for portlets running on a portal server outside the Studio environment. To enable remote debugging you need to configure the JVM for the WebSphere Portal Application Server instance. Debug mode needs to be enabled, with the debug arguments set to

```
-Djava.compiler=NONE -Xdebug -Xnoagent -
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777
```

We will look at portlet developing using Studio in more detail in Chapter 16 and explore interactive debugging using Studio in Chapter 17.

Portlet Preview

Portlet preview allows you to preview the portlet under development as deployed on a remote Portal system. This is different than executing the portlet in the test environment. The preview function works only with external Portal systems, not an installed WebSphere Portal for a test environment. It does not set up a debugging environment. Instead the portlet application is deployed on the remote Portal system using the XML configuration interface of Portal. The portlet is put on a preview page and Studio's internal Web browser opens that portal page.

Building the Portlet Application WAR File

A portlet application project is a J2EE-compliant Web application. You can easily build a deployable WAR file for your portlet application simply by using the Export utility in Studio. Similarly, you can import resources from a WAR file to your portlet application project using the Import function.

Creating the Poll Portlet Project

In this section you will create the portlet application project in Studio for our sample portlet application. This section assumes that you have already installed the Portlet Toolkit in Studio. If the Portal Toolkit is not already installed, follow the instructions that came with the toolkit (either downloaded from the Web site mentioned earlier or obtained from the Portal product CD) to run the toolkit.

The intention of this chapter is to introduce the Portlet development toolkit in the context of creating our example portlet application. This chapter is not intended as a tutorial on WebSphere Studio itself. The discussion here assumes you have a familiarity with Studio, its key concepts, and how to navigate the workbench. If you do not have experience using WebSphere Studio you may want to review the product help content, which has excellent information about the development environment and includes scenario-based tutorials.

The Portlet Perspective

First let's take a look at the portlet perspective in Studio (see Figure 15-7). Start WebSphere Studio and open the portlet perspective from the Window item on the menu bar.

The default perspective shows a number of views and an editor. The key views to consider are the Project Navigator view, the editor, and the tasks view. These will be used initially as you create the portlet project. The Palette view, Attributes view, and Quick Edit view are used primarily with the Page Designer editor when editing JSP or HTML files. Change to the Tasks view by selecting the Tasks tab on the bottom of the view with Quick Edit.

Create a portlet project for our Poll Portlet by selecting New ➪ Portlet Application Project from the File menu item in the Studio toolbar. This opens the Portlet Project creation wizard (see Figure 15-8).

Enter the name of your portal project. Choose **Poll** for this example. The project location specifies the directory where the project files will be created. You may change the directory using the Browse button, but the selected

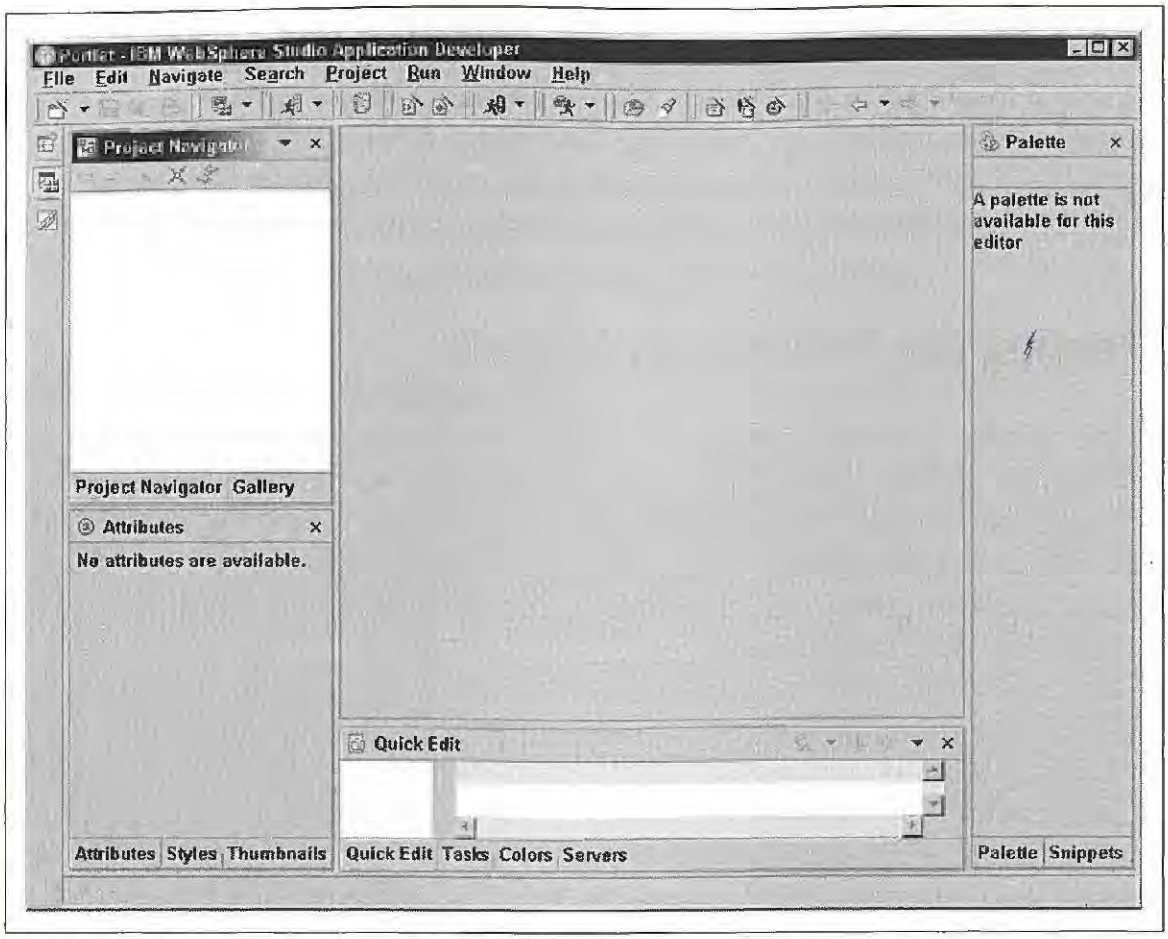


Figure 15-7 Portlet perspective in Studio.

directory must be empty. You can select to create either an empty portlet or a basic portlet.

An *Empty Portlet* creates a portlet project directory structure and the Web and portlet deployment descriptors. The wizard sets the Java build path for the project, including the required jar files for the portlet API. The portlet deployment descriptor that is created defines a portlet application and a single portlet in that application. You can add additional portlets to the portlet application by re-executing the wizard and specifying a *portlet* instead of a *portlet application*.

A *Basic Portlet* generation performs this function also but in addition generates source code for the portlet. The code generated will depend on the options selected in the wizard. At a minimum, a portlet class is created that extends `PortletAdapter` and implements a `doView()` method that invokes a sample JSP. Since the portlet class is created the deployment descriptors created can reference that class. This step is useful as both the Web descriptor references this as the servlet class for the Web application and the portlet class name is used in both descriptors. Create the Poll portlet using the *Basic Portlet* type.

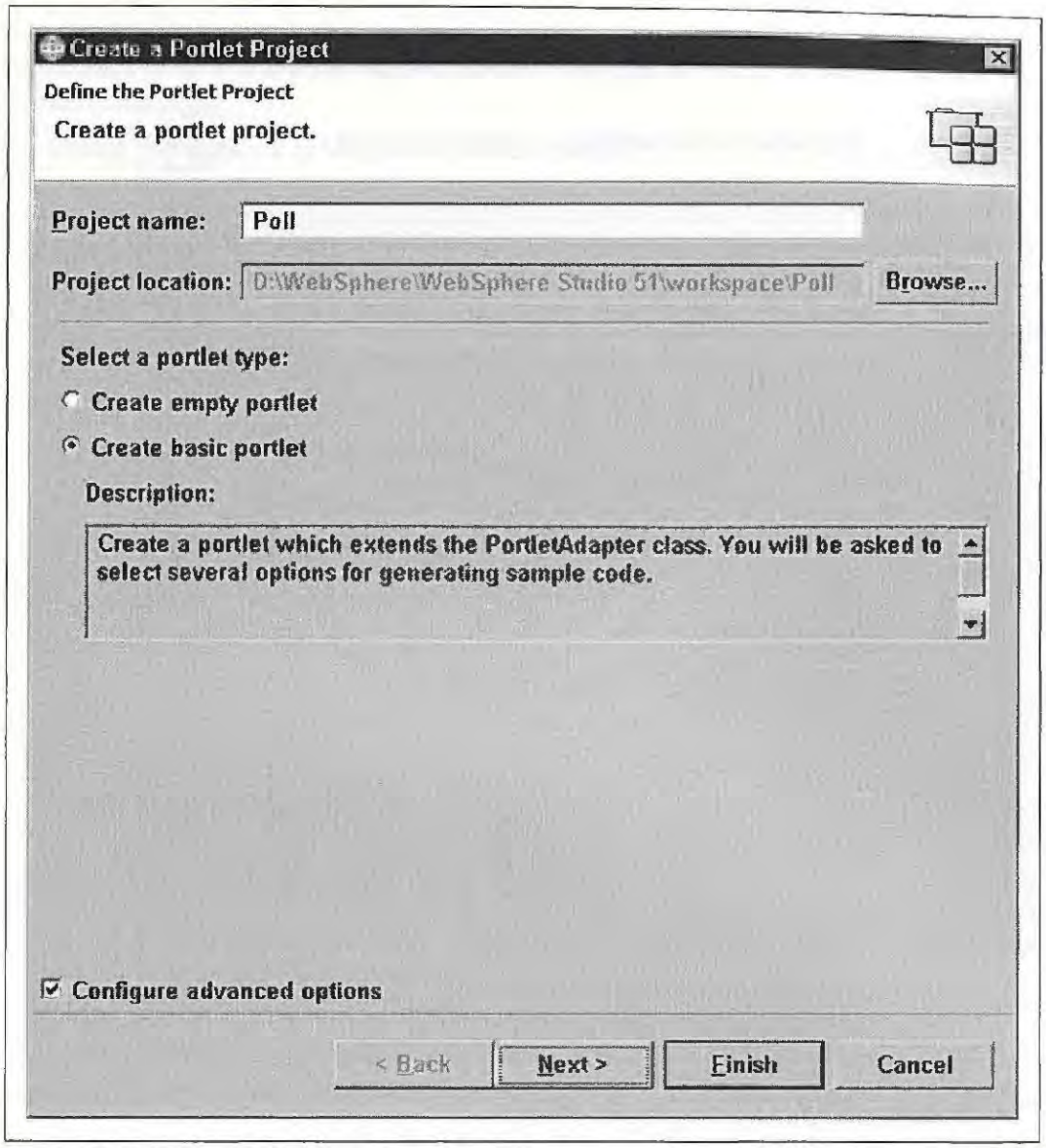


Figure 15-8 Portlet Project creation wizard.

After selecting Create Basic Portlet, select Configure Advanced Options and then Next. On the next page of the wizard you can specify the name of the Enterprise Application (EAR) project (see Figure 15-9). Since portlet applications are packaged and deployed to portal through a WAR file, the EAR file selected here will not affect that deployment. However, the EAR will be used for portlet deployment to the WebSphere Portal test environment. You may want to put portlet applications in separate EAR projects to be able to manage more easily the portlets that are executing in the test environment. This project being created is set, and can be modified later, in the EAR deployment descriptor. For this example portlet, leave the project set as the default EAR.

J2EE supports multiple Web applications running in the same server instance. The context root is a unique (within the server instance) identifier

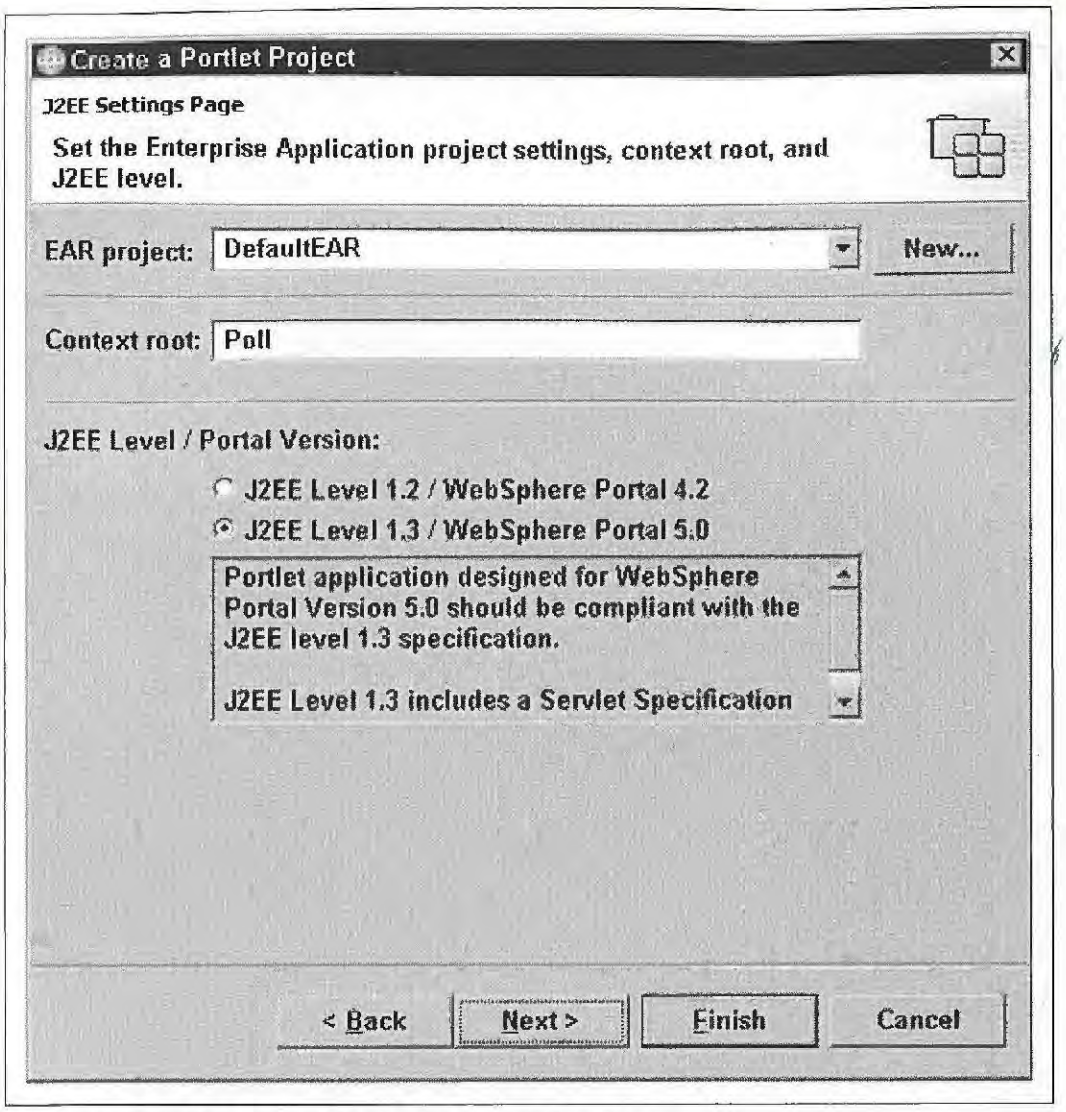


Figure 15-9 Enterprise Application project settings.

to ensure that an HTTP request reaches the correct Web application. The context root is a path identifier used in the URI immediately following the host name. The default value defined in the Studio wizard is the project name. This value is set, and can be modified later, in the EAR deployment descriptor with the Web module definition. For this example portlet use the provided default value.

Also on this page you can specify the J2EE level of the Portal into which you expect to deploy this portlet. WebSphere Portal 5.0 supports J2EE level 1.3. Select J2EE 1.3 for your example portlet and click Next.

The next page in the wizard gathers the final set of information needed for the portlet and Web deployment descriptors (see Figure 15-10). Here you specify the name of the portlet and the name of the portlet application. Also, you specify the default locale and the portlet title in the language of the default locale. Change these to values of your choice.

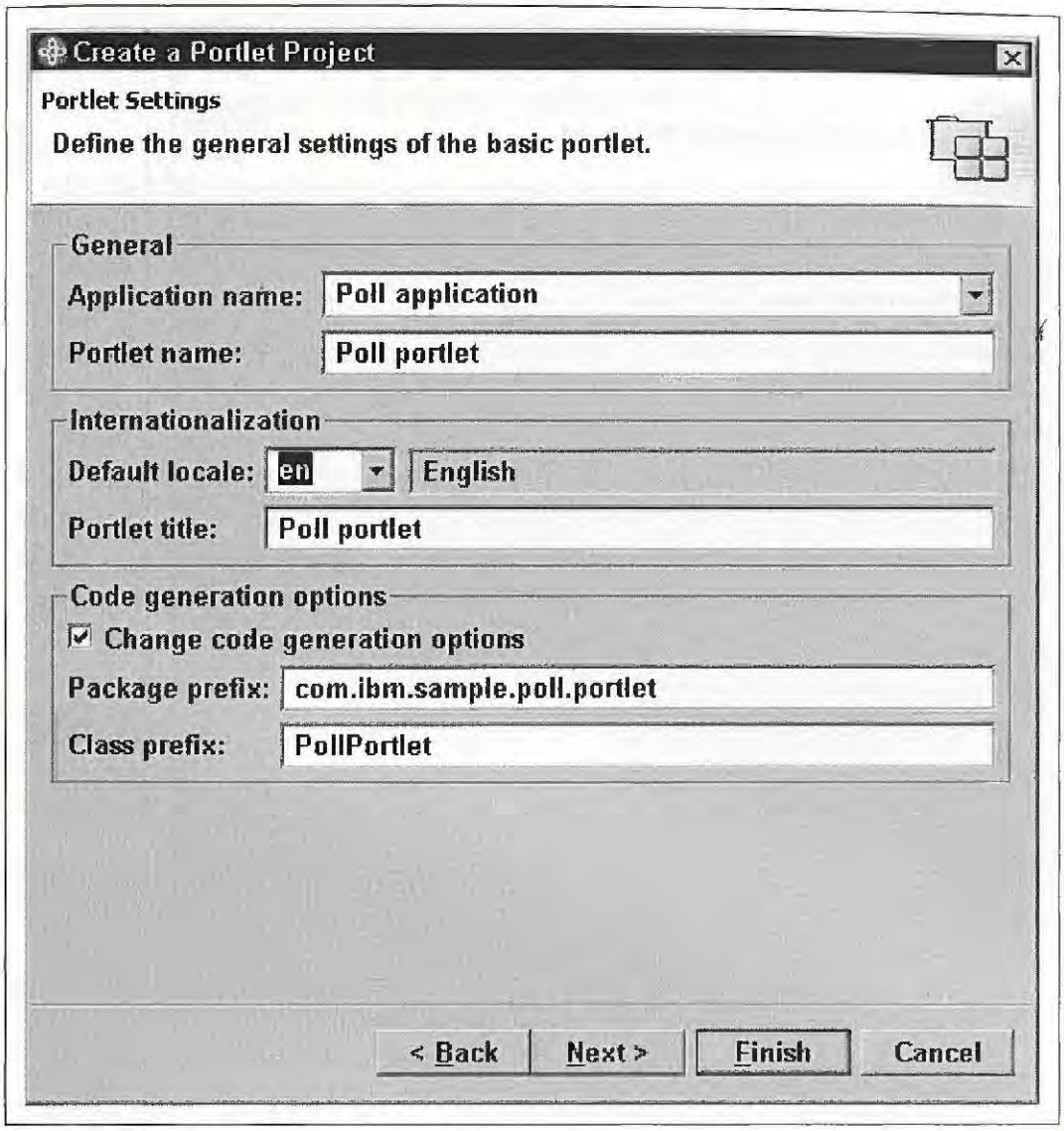


Figure 15-10 Specifying portlet settings.

Select Change Code Generation Options. The package prefix is the package name that will be created in this new project and it is the package that will be referenced in the Web deployment descriptor for the servlet class name. Change the package prefix to a package name of your choice. The class prefix will be used as the class name for the portlet class and also referenced in the Web deployment descriptor as the servlet class name (with the package prefix). A version of this package name will also be used to create a directory structure in the Poll Portlet project for other generated files such as JSP files. The default class prefix name is fine for our portlet, so click Next.

The final three pages in the wizard are used to determine what source code files to generate. This next page specifies portlet event handling options. Most portlets will have a requirement to implement an action event listener.

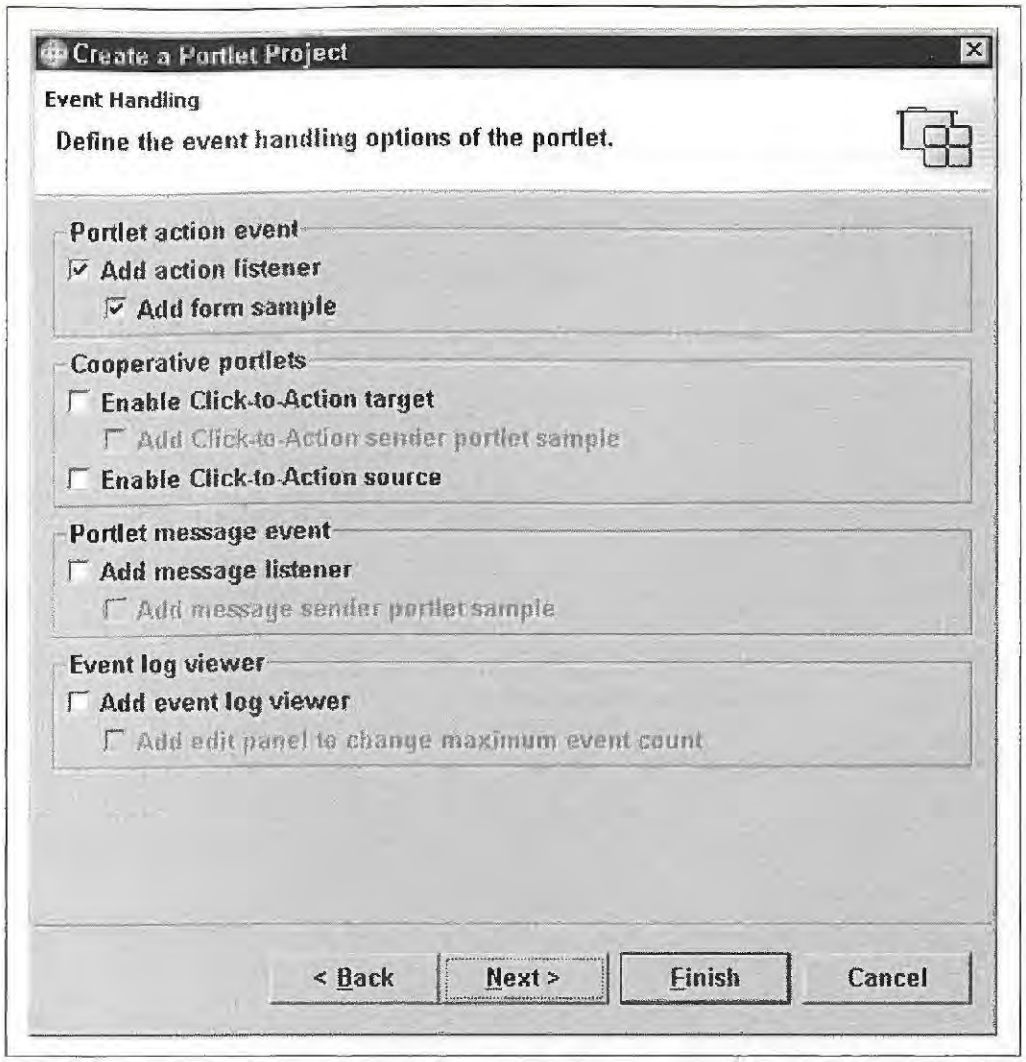


Figure 15-11 Event handling options.

This preselected choice will generate an `actionPerformed()` method in the portlet class and also modify that class definition to implement the `ActionListener` interface.

This selection will also generate a simple view bean and a simple session bean. The form bean has a getter and setter method for an instance variable to hold a portlet URI string. The session bean provides similar function for a text string. An example JSP using these beans will also be created.

Cooperative portlets are portlets that interact by sharing data elements through a broker mechanism that handles portlet requests to declare, publish, and share properties (see Figure 15-11). A portlet associates actions with properties that get invoked when the property change is received.

Selecting all three of these Cooperative Portlet options will generate a simple example of two cooperative portlets, their JSP files, and a control WSDL file. Selecting Enable Click-to-Action Target generates a control WSDL file so the target portlet can receive properties (select Add Form Sample to ensure

a complete working example). Selecting Add Click-to-Action Sender Portlet Sample creates a simple property sender portlet. Selecting Enable Click-to-Action source updates the source portlet JSP to use the Click-to-Action tag library to publish properties. These options are available only for J2EE 1.3 applications.

Portlets that reside on the same portal page can communicate during the action processing phase using message events. Selecting these options updates the portlet class definition to implement the `MessageListener` interface and the `messageReceived()` method. You can choose to generate a sample portlet to send a broadcast message.

Finally, on this page you can select to generate code in the sample portlet to record action events in session so that they can be displayed later. Leave this page with the default values and select Next.

The next page allows you to generate sample code to handle user credentials that are managed through the Credential Vault (see Figure 15-12).

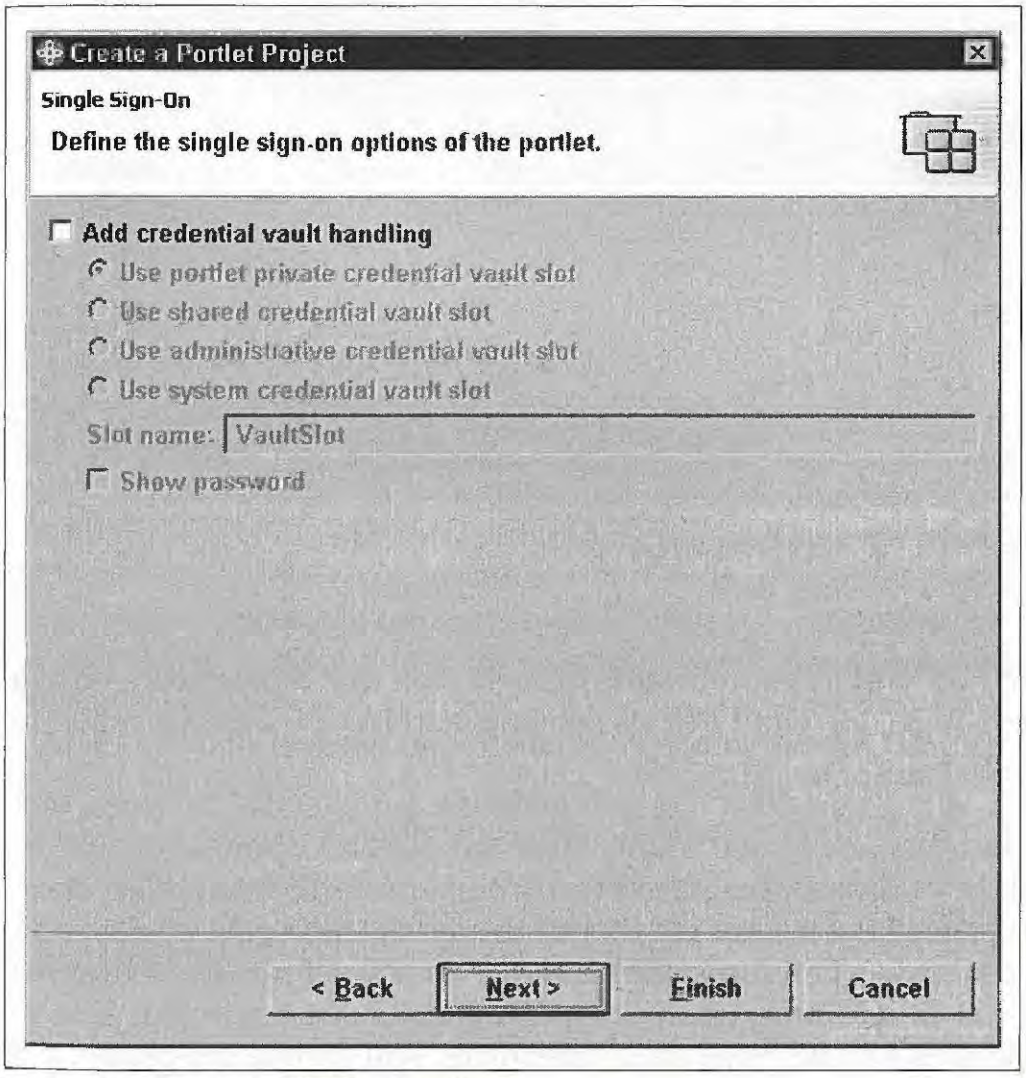


Figure 15-12 Credential Vault options.

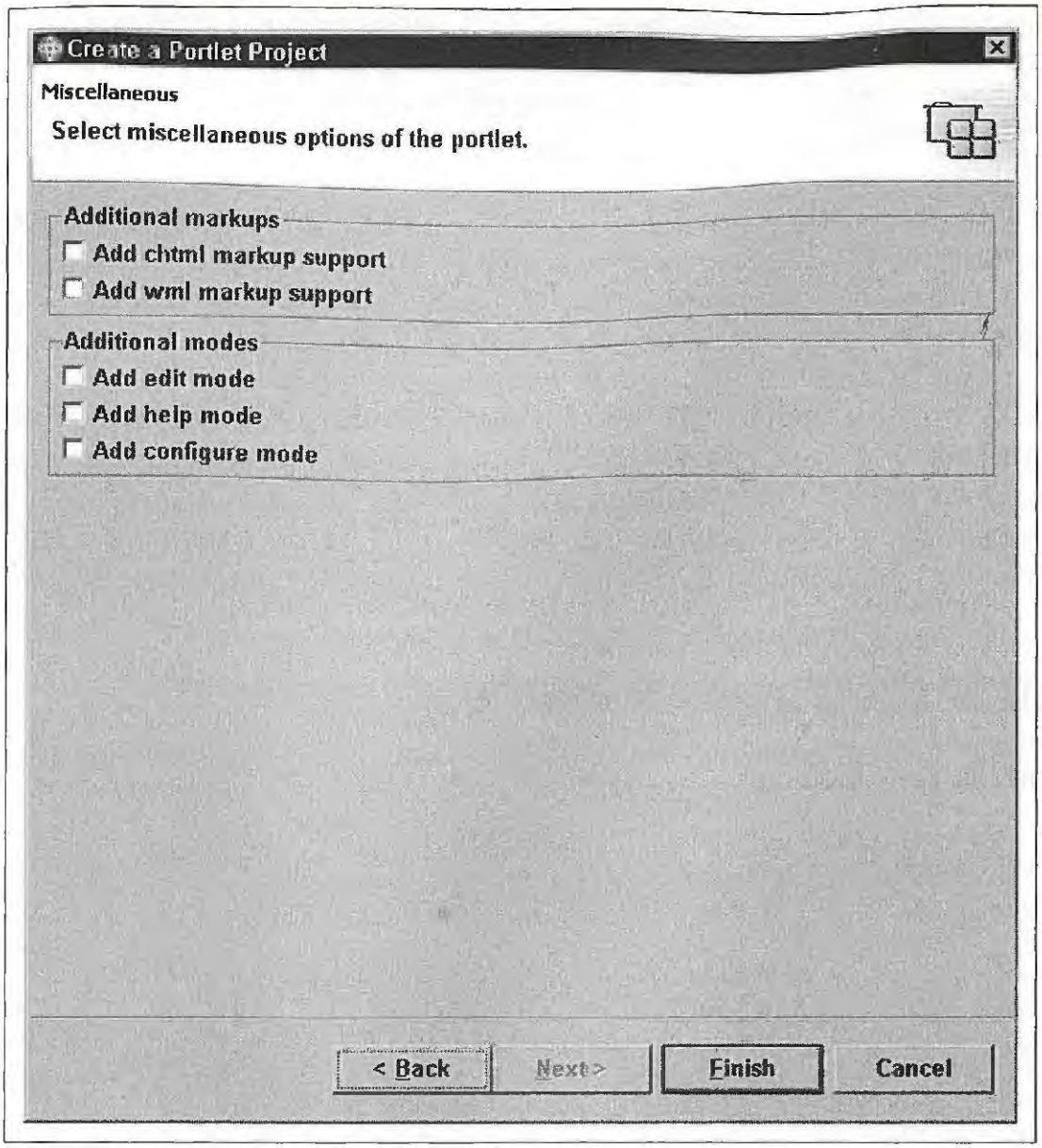


Figure 15-13 Markup and portlet mode options.

The Credential Vault is a secure store for user secrets and provides a portlet service to manage the vault and its contents and handle authentication to back-end applications.

Selecting options on this page will generate sample code to access credentials from the Credential Vault. There are several options to support credentials of various scopes. Leave this option unchecked and select Next.

The last page of the wizard allows you to modify the portlet deployment descriptor to support multiple markups and additional portlet modes (see Figure 15-13). In addition, when the markup options are selected, device-specific JSP files will be generated. The portlet will invoke the correct JSP depending on the client request type. Leave these options unchecked and select Finish.

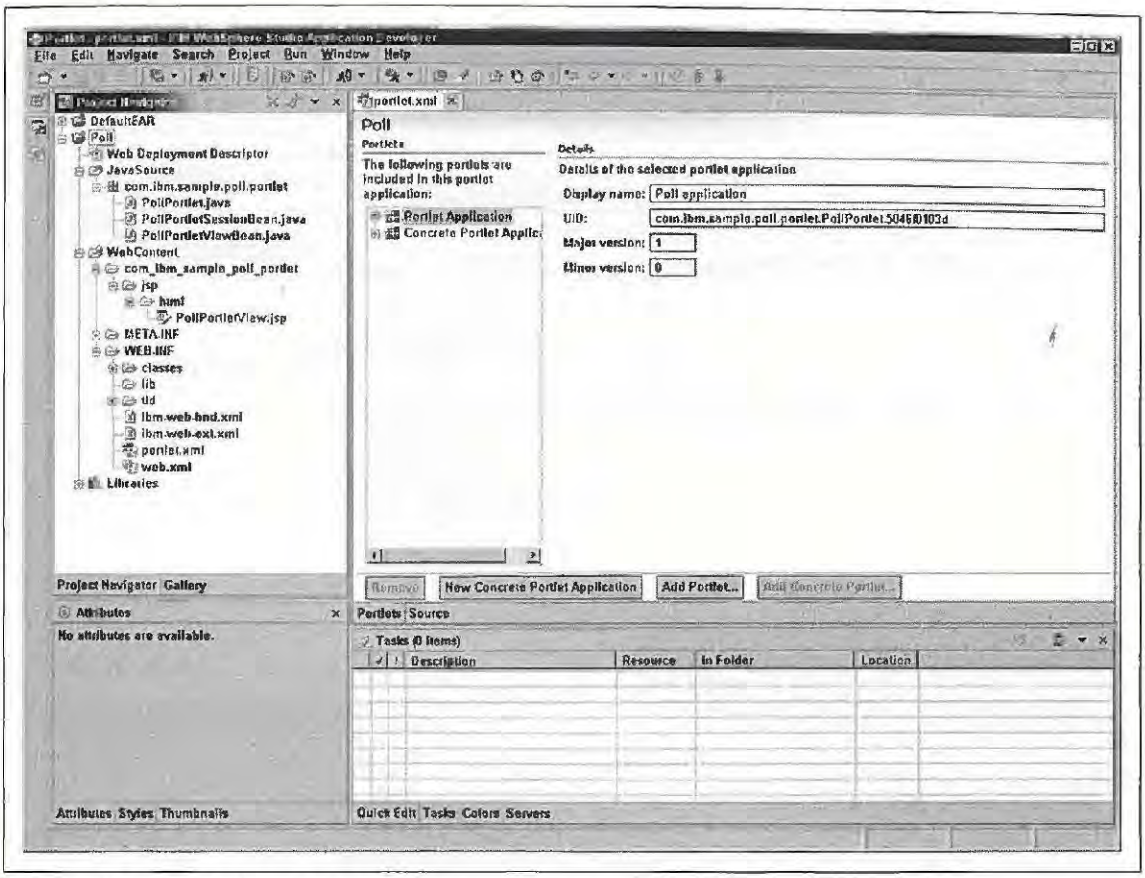


Figure 15-14 Poll project created.

As a result of completing the portlet application wizard the Poll project is created (see Figure 15-14). The project is shown in the Project Navigator view. It is created with the correct directory structure for a Web application, and so you can generate a deployable WAR file simply by selecting the Poll Portlet project and using the export function (File ⇨ Export ⇨ WAR File).

The appropriate portlet project build class path is also set when the project is created. The project properties view shows the Java build path along with other project metadata and preference settings. The properties view can be opened by selecting the project in the Project Navigator then selecting File ⇨ Properties.

In addition the portlet deployment descriptor and Web deployment descriptor are created and the deployment descriptor editor is opened. We will take a closer look at this editor and the generated values next.

Deployment Descriptor Editor

Click on Portlet Application in the editor to see the portlet application details (see Figure 15-15). The display name for the portlet application is the name you provided to the wizard. The application unique identifier (UID)

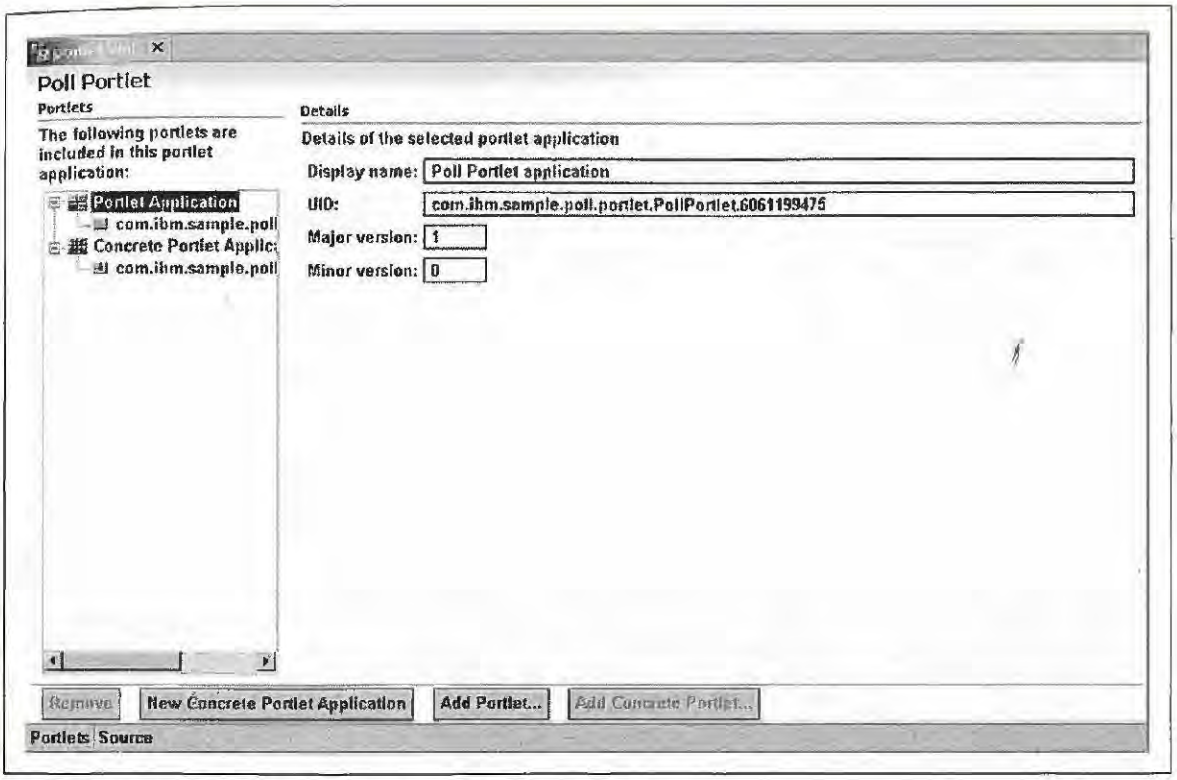


Figure 15-15 Portlet deployment descriptor editor.

is a generated value based on the portlet package and the portlet class name provided to the wizard. The major version and minor version default to 1 and 0 respectively. These values are used to track portlet application revisions.

Expand the Portlet Application and select the portlet. Its details are shown in Figure 15-16. Like the portlet application, the portlet name was provided to the wizard. The portlet ID is a generated value and the major and minor version values are the new portlet defaults.

The servlet name is set as the full class name of the portlet class. This value must map to the servlet name in the Web deployment descriptor. Using the wizard to create the deployment descriptors ensures that this mapping and the mapping between the abstract and concrete portlet definitions are set correctly.

The portlet cache defaults are to always expire and be nonshared. The portlet definition also includes the window states that are supported, the markup types supported, and the portlet modes supported. As we have seen, the Poll portlet will require configure and edit modes as well as its view mode. We could have specified this in the wizard but let's make the change here. Click None in the column labeled Configure on row labeled HTML, and from the drop-down list that appears select Fragment. Do the same for the column labeled Edit. This will enable our portlet to have icons for these three modes on the portlet menu bar.

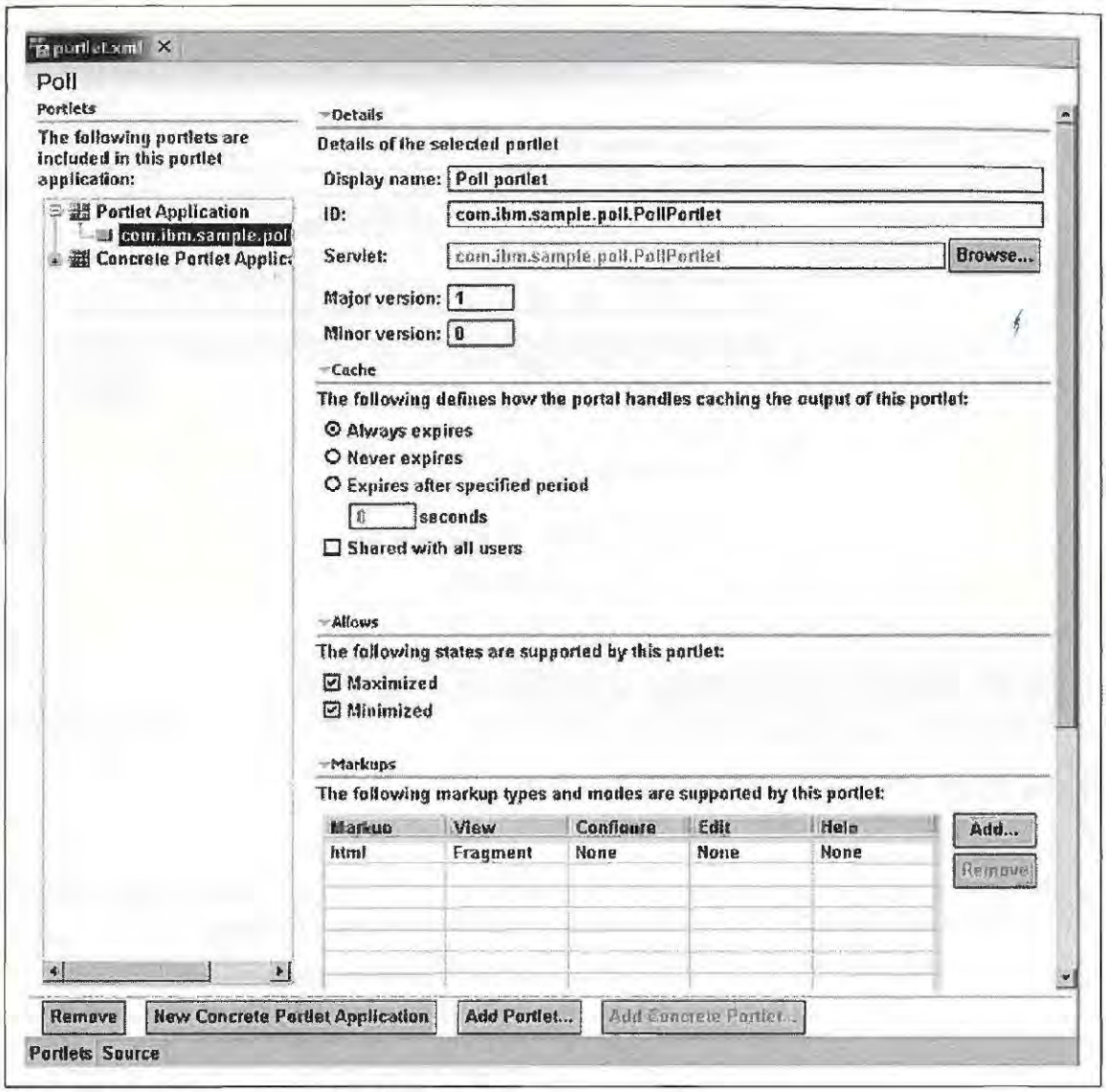


Figure 15-16 Portlet application details.

There are no portlet configuration parameters set through the wizard but any of these values can be set here. Any configuration parameters set here are specified as init-param's in the Web deployment descriptor. This editor will modify both the portlet and web xml files.

Click on *Concrete Portlet Application* in the editor. The details view shows the concrete portlet application display name and its associated UID (see Figure 15-17). Both of these values are derived from input values to the wizard. The wizard does not capture portlet application context parameters but they can be set using this editor.

Expand the *Concrete Portlet Application* and select the concrete portlet definition (see Figure 15-18). Here the supported and default locales are defined. The portlet title, short title, description, and keywords (for search function) are specified by language. The concrete portlet settings are also defined here.

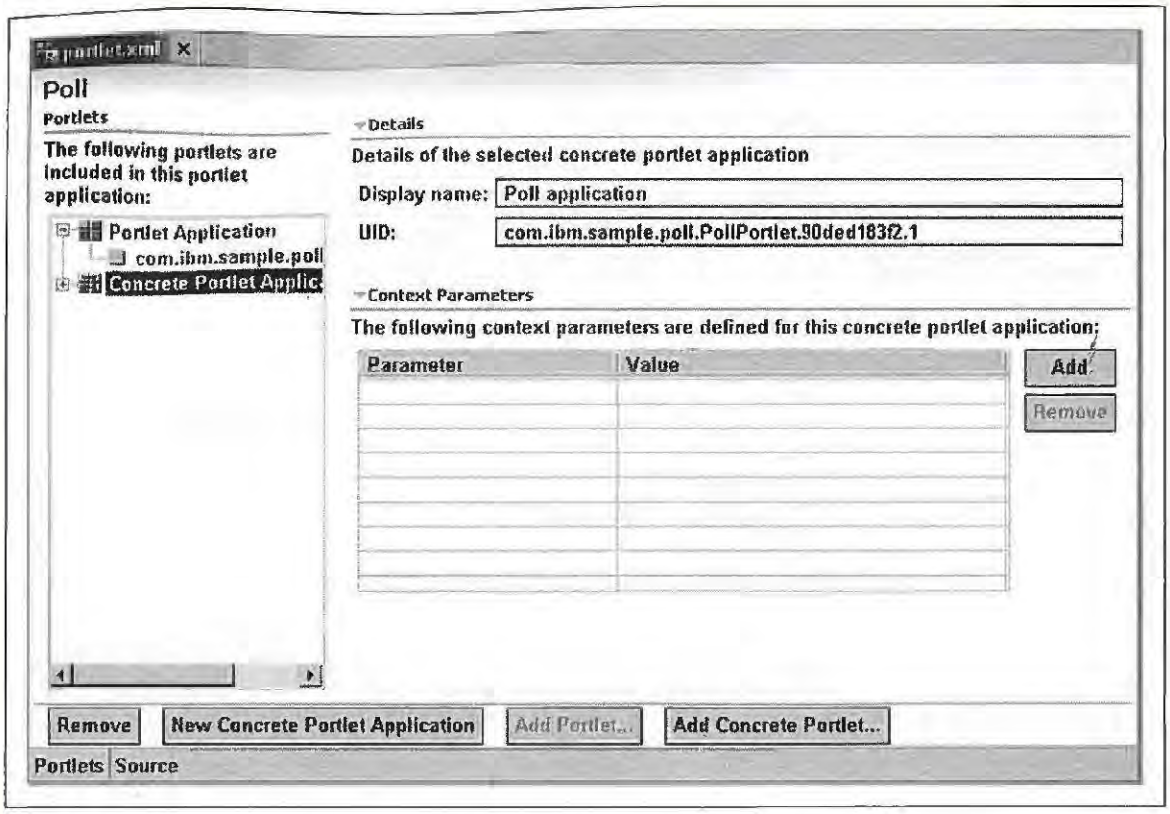


Figure 15-17 Concrete Portlet Application details.

The Poll portlet will require portlet settings but we will define them later. Close the deployment descriptor editor and save the changes.

Generated Source Code

Review the `PollPortletSessionBean` and the `PollPortletViewBean` classes that were generated. The Poll portlet will not use these classes so they can be deleted.

The `PollPortlet` class will now have a number of errors that are displayed in the task view as a result of deleting those bean classes. Open the source editor on `PollPortlet.java`. Let's start our Poll project development with a fairly empty portlet. Delete the `getSessionBean` method. We will put our portlet constants in a separate class so delete the static instance variables. Change the `doView()` method to remove the references to `PollPortletSessionBean` and `PollPortletViewBean`. We will use the JSP tags to generate the PortletURIs so delete them as well. Similarly clean up the `actionPerformed` method and remove the `getJSPExtenstion` method. Also, delete the JSP file and its directory structure. So our very basic `PollPortlet` looks something like Figure 15-19.



Figure 15-18 Concrete portlet definition.

Summary

In this chapter you reviewed the WebSphere Studio development environment. You learned about the capability of the Portal Toolkit and the value it brings to Studio for portlet development.

We introduced a sample application that we will develop using Studio. You used the portlet application generation wizard to help create our portlet project. You saw that the wizard was very helpful in creating the Portlet Project with the correct directory structure, setting the correct class path for the portlet project to build correctly, and creating the deployment descriptors needed for our portlet application. The wizard also generated portlet

```
package com.ibm.sample.poll.portlet;

import java.io.IOException;

import org.apache.jetspeed.portlet.*;
import org.apache.jetspeed.portlet.event.*;

/**
 * Poll Portlet
 */
public class PollPortlet extends PortletAdapter implements ActionListener {

    /**
     * Initialize the portlet
     */
    public void init(PortletConfig portletConfig) throws UnavailableException {
        super.init(portletConfig);
    }

    /**
     * Render the view mode
     */
    public void doView(PortletRequest request, PortletResponse response)
        throws PortletException, IOException {

        // Invoke the JSP to render
        getPortletConfig().getContext().include("jsp/view.jsp",request,response);
    }

    /**
     * Handle the action events
     */
    public void actionPerformed(ActionEvent event) throws PortletException {
        if (getPortletLog().isDebugEnabled())
            getPortletLog().debug("ActionListener - actionPerformed called");

        // ActionEvent handler
        String actionString = event.getActionString();
        // Add action string handler here
        PortletRequest request = event.getRequest();
    }
}
```

Figure 15-19 PollPortlet.

Java code as a starting point. Some of that we removed to leave a very minimal base on which to build.

In the next chapter we will develop our Poll portlet using this starting point.

CHAPTER 16

Portlet Development

This chapter explores in detail the implementation of the example Poll portlet. In the previous chapter we introduced the WebSphere Studio development environment. Using the Portal Toolkit in Studio, you used the portlet application wizard to create a portlet project and create the portlet deployment descriptors.

The complete source code for this portlet is available for download. Most of the key sections of the portlet code are shown in listings here and from the discussion and code shown, you should be able to create the portlet implementation in your Studio project. However, not all of the project code is shown here. You may want to import the completed project in Studio to have the total reference available while you follow the discussion.

This implementation is based on the IBM Portlet API. In the following chapter we will use this base portlet implementation and modify it to use the JSR 168 API. While doing that we will compare the differences in implementation and functionality.

Poll Portlet

Recall the description of the Poll portlet from the previous chapter and the associated screenshots. This is a relatively simple portlet but still has enough complexity to demonstrate some key aspects of the Portlet API.

- Action event processing
- Page navigation within the portlet
- Multiple portlet modes—view, edit, and configure

- Portlet configuration parameters in `PortletSettings`
- User customization parameters in `PortletData`
- Database access
- Trace logging
- Internationalization

While this example portlet is simple, we will still employ good principles to its design. We will structure the application to ensure good separation of components based on areas of responsibility. The portal architecture itself provides a good foundation for portlet designs. Consider how a Model–View–Controller (MVC) pattern would apply to this portlet.

We would implement the View components as JSPs, one for each of the portlet pages shown in the previous chapter. The Model or domain components are minimal for our portlet. These components represent the business objects and logic of the application domain and for our portlet that is a small set of objects such as a `Poll` object or a `User` object. The Controller component of a portlet is the portlet class. It is responsible to respond to user requests from the View components and invoke behavior in the Model components. Our portlet requires access to a persistent store and therefore has components for infrastructure components as well.

Implementing the Controller

Open the Poll project in WebSphere Studio that we created in the last chapter (see Figure 16-1). Use the portlet perspective. In this project you should have a `PollPortlet.java` file in the `JavaSource` directory in the package called `com.ibm.sample.poll.portlet`. This is the portlet class. The name of the portlet class was specified when you ran the portlet application wizard and is now defined in the Web deployment descriptor. Open the source editor for this file.

First consider the implementation for the `doView` method. This method gets executed when our portlet is asked to render itself in view mode. The portlet should display either the view with the question and answer choices or the view with the voting results depending if the user has already voted. You have not yet created our business objects or broker classes, so you have to fill in those pieces next. But the `doView` method can look something like the following:

```
/**
 * Render the view mode
 */
public void doView(PortletRequest request, PortletResponse response)
```

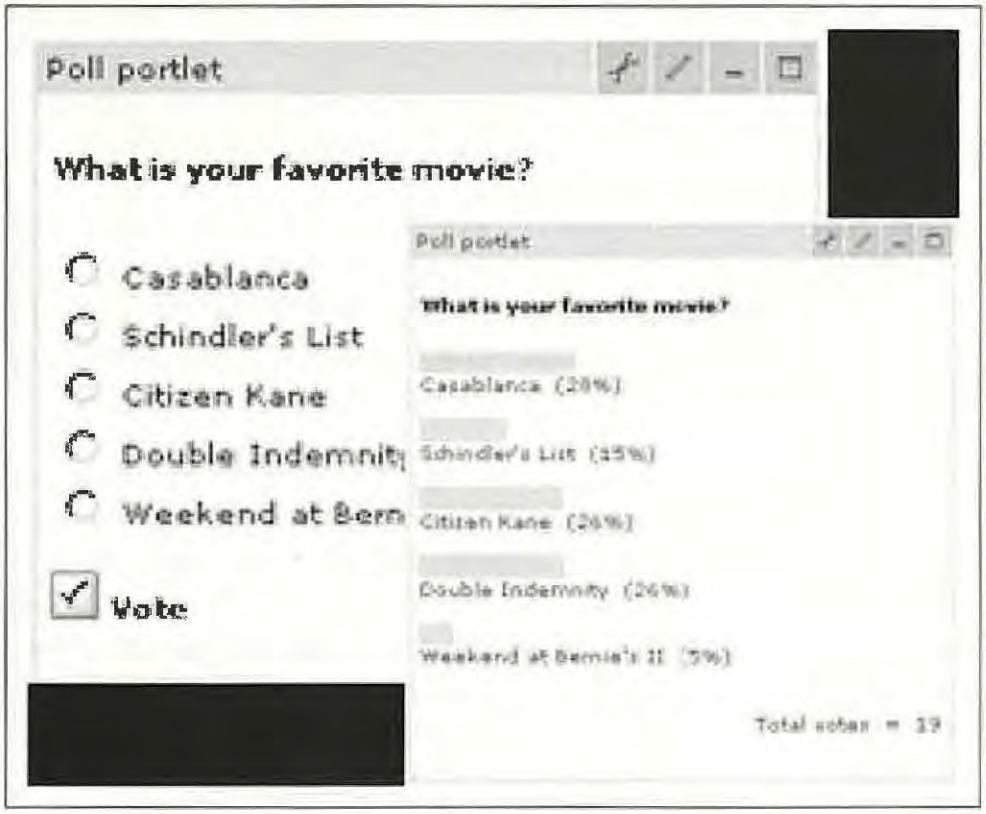


Figure 16-1 The Poll project in WebSphere Studio.

```
    throws PortletException, IOException {

    // Get the poll bean
    PortletSettings portletSettings = request.getPortletSettings();
    String persistType = portletSettings.getAttribute(PERSIST_TYPE);
    String datasource = portletSettings.getAttribute(DATASOURCE);
    String pollName = portletSettings.getAttribute(POLL_NAME);
    Broker broker = Broker.getInstance(persistType, datasource);
    PollBean pollBean = broker.getPoll(pollName);

    // If the user has not already voted show poll question
    // otherwise show poll results.
    String user = request.getUser().getUserID();
    boolean hasUserVoted = broker.hasUserVoted(user, pollName);

    if (hasUserVoted) {
      pollBean.initializeResults(broker);
      request.setAttribute("pollBean", pollBean);
        portletContext.include(VIEW_RESULTS_JSP, request, response);
    }
    else {
      request.setAttribute("pollBean", pollBean);
        portletContext.include(VIEW_QUESTION_JSP, request, response);
    }
}
```

You see from this code that the portlet configuration information that defines the name of the poll and the information that specifies the datastore are kept in the portlet settings. With that information, you can ask our datastore manager class (`Broker`) for an instantiated `Poll` object (`PollBean`) for the poll with the given name.

You can get the userid of the logged-in user from the `User` object. The `User` object is available from the `PortletRequest` object. Then determine if a vote for this poll for the current user has already been registered and if not invoke the JSP to show the poll question. If the user has voted, get the vote results from the `Broker` and invoke the JSP to show the results, as shown in Figure 16-2.

Next consider the configuration pages for this portlet. As you have just seen, the poll name and the persistent store information are retrieved from portlet settings. These can initially be configured in the portlet deployment descriptor but we want to allow the portal administrator to make modifications to these values after the portlet has been deployed. Also, the administrator will need to specify the poll question and answers, although that information will be kept in the persistent store, not the portlet settings. We want that information outside the scope of a single portlet instance since the information may be shared across multiple poll portlet instances. Also we don't want the data to be lost when the portlet gets reconfigured.

The implementation for configure mode will be very similar to what you just did for view mode, except now it gets a bit more complicated, as you have page navigation issues to consider and action event processing to handle.



Figure 16-2 The results of the poll.

```
/**
 * Render the configure mode view
 */
public void doConfigure(PortletRequest request,
  PortletResponse response)
  throws PortletException, IOException {

  // Create a form data object and put it on session
  PortletSession session = request.getPortletSession();
  SessionData sessionData = new SessionData();
  session.setAttribute(SESSION_DATA, sessionData);
  ConfigureFormData formData = sessionData.getConfigureFormData();

  // Get the portlet settings
  PortletSettings portletSettings = request.getPortletSettings();
  String persistType = portletSettings.getAttribute(PERSIST_TYPE);
  String datasource = portletSettings.getAttribute(DATASOURCE);

  // Set the portlet settings in the form data bean on session
  formData.setDatasource(datasource);
  formData.setPersistType(persistType);

  // Invoke the JSP to render
  portletContext.include(CONFIGURE_JSP, request, response);

}
```

This implementation is still simple. We introduce a `ConfigureFormData` object that will hold the input data as the user steps through the configuration pages. This object will be held in session and so we also create a `SessionData` object to hold the portlet data that will go on the actual session object. Putting the portlet session data in a single object makes it easier to clean the session data and also reduces the number of attribute names that need to be managed.

Now the processing gets a little more complicated as we have three actions that can occur on the user interface for this first page. The user can *verify* the datasource definition. The user can advance to the next configuration page or cancel from configure mode completely. The specifications for these action events are in the JSP through encoding in the link URI. The JSP uses a custom tag from the portlet tag library to create a URI that can be specified on links defined by the `Action` attribute or an `href` attribute. Two examples of using the tag are

```
<portletAPI:createURI>
  <portletAPI:URIAction name="verify"/>
</portletAPI:createURI>
```

and

```
<portletAPI:createReturnURI/>
```

The first tag creates a URI that encodes both the portlet reference and a trigger for an action event. The portlet container will interpret this URI and call the portlet's `actionPerformed` method, that is, if the portlet class implements that `ActionListener` interface. The action event is named so the `actionPerformed` method can interrogate the `Event` object, retrieve the name, and execute the correct behavior for this action event. The same `actionPerformed` method will get called for all action events for this portlet so the event name is needed to distinguish events.

The second tag does not associate an action event with the URI and simply asks the portal container to return the portlet to its previous mode. For example, if the user had been in view mode prior to configure mode, then this URI will return the portlet to view mode.

The following listing shows a complete JSP for the first page of configure mode. This is slightly different than the version that is available via download. It has been changed somewhat to reduce the size. If you are implementing this in Studio, this file is named `configure.jsp` and is in a JSP directory under `WebContent`.

```
<%@ page contentType="text/html"%>
<%@ page import="com.ibm.sample.portlet.poll.beans.ConfigureFormData"%>
<%@ page import="com.ibm.sample.portlet.poll.portlet.PollConstants"%>
<%@ taglib uri="/WEB-INF/tld/portlet.tld" prefix="portletAPI"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt"%>

<jsp:useBean id="sessionData"
class="com.ibm.sample.portlet.poll.beans.SessionData" scope="session"/>
<fmt:setBundle basename="nls.polling"/>
<portletAPI:init />

<%
  ConfigureFormData formData = sessionData.getConfigureFormData();
  String message = sessionData.getMessage();
  String dataSource = formData.getDatasource();
  String persistType = formData.getPersistType();
%>

<form method="post" name="<portletAPI:encodeNamespace value="form"/>">
<table align="left" width="400">

  <tr>
     <td class="wpsInlineHelpText" colspan="2">
       <img src='<%=response.encodeURL("/images/msg_inline_help.gif")%>'
```

```
        border="0"/> 
        <fmt:message key="configure.initinfo"/>
      </td>
    </tr>

    <tr>
      <td class="wpsPortletHead">
        <input type='radio' name='persist_type' value='memory'>
      </td>
      <td class="wpsPortletHead">
        Memory
      </td>
    </tr>

    <tr>
      <td class="wpsPortletHead">
        <input type='radio' name='persist_type' value='database'>
        >
      </td>
      <td class="wpsPortletHead">
        Database - specify datasource name
      </td>

    </tr>
      <td class="wpsPortletHead"></td>
      <td class="wpsPortletSmText">
        <input type='text' name='datasource' value='' size='40'>
      </td>
    </tr>

    </tr>
      <td class="wpsPortletHead"></td>
      <td align="left" valign="bottom" nowrap>
        <input class="wpsButtonText"
        type='submit' value='Verify' name='verify'
        onClick="document.<portletAPI:encodeNamespace
value='form'/>.action=
        '<portletAPI:createURI>
          <portletAPI:URIAction name="verify"/>
        </portletAPI:createURI>'">
      </td>
    </tr>

    <tr>
      <td class="wpsPortletSmText" colspan="2">
        <input class="wpsButtonText"
        type='submit' value='Next'
        onClick="document.<portletAPI:encodeNamespace
        value='form'/>.action=
        '<portletAPI:createURI>
```

```
        <portletAPI:URIAction name="config_pg2"/>
      </portletAPI:createURI>'">

      <input class="wpsButtonText"
      type='submit' value='Cancel'
      onClick="document.<portletAPI:encodeNamespace
      value="form"/>.action=
      '<portletAPI:createReturnURI>
        <portletAPI:URIAction name="Cancel"/>
      </portletAPI: createReturnURI>">
    </td>
  </tr>
</table>
</form>
```

Notice in the JSP that we namespace-encode the document object names. Instead of encoding each individually we encode the name of the form object. That way any references to objects within the form are namespace-protected as well. To do this we use the `<portletAPI:encodeNamespace/>` tag.

Also, as of Portal version 5 the preferred method to get text strings from resource bundles is through the Java Server Pages Standard Tag Libraries (JSTL) `fmt` tag library. You will need to get the open source libraries, `standard.jar` and `jstl.jar`, and add them to your portlet application libraries folder.

You will also notice a reference to an image file in the JSP. This file is packaged with the portlet in an images folder under `WebContent`. The URI reference to that resource must also be encoded to successfully resolve the reference. To do that we used the `encodeURL` method from the portlet response object:

```
<%=response.encodeURL("/images/msg_inline_help.gif")%>
```

The JSP also uses style definitions from the portal Cascading Style Sheet (CSS) file named `styles.css`. The JSP has access to this style sheet. The actual style sheet that is used for the portlet is dependent on the theme selected. Using these styles will allow the portlets to look and feel integrated well with other portlets and take on the selected theme's behavior. Look for the `styles.css` files in directories starting from

```
<was-root>/installedApps/<node>/wps.ear/eps.war/themes/...
```

Next, let's look at the `actionPerformed` method in the portlet class to handle the requests to verify the datasource and to move to the next page in configure mode. Don't add this code to your portlet just yet.

```
/**
  * Handle all Action requests
 */
public void actionPerformed(ActionEvent event)
  throws PortletException {

  PortletRequest request = event.getRequest();
  PortletContext portletContext = getPortletConfig().getContext();
  PortletLog log = portletContext.getLog();

  // Get the action name
  String actionName = event.getActionString();
  if (log.isDebugEnabled())
    log.debug("actionPerformed() processing action "+actionName);

  if ("verify".equals(actionName)) {
    verifyAction(request, portletContext);
  }

  if ("config_pg2".equals(actionName)) {
    page2Action(request, portletContext);
  }

}

/**
 * Verify datasource action
 */
protected void verifyAction(PortletRequest request,
  PortletContext portletContext)
  throws PollException {

  try {
    // Get configuration form data bean from session
    SessionData sessionData = SessionData.getSessionData(request);
    ConfigureFormData formData = sessionData.getConfigureFormData();

    // Update the persistence type and the datasource from request
    // parameters, validate the input.
    formData.updateDbParms(request);

    // Check the datasource name and the database tables
    // If an exception occurs put the exception message out
    DbBroker dbBroker = new DbBroker(formData.getDatasource());
    dbBroker.verifyTables();

    // Success, put a message out
    ResourceBundle bundle = ResourceBundle.getBundle("nls.polling",
      request.getLocale());
    String msg = bundle.getString("configure.verify_success");
```

```
        sessionData.setMessage(msg);

    }
    catch (PollException e) {
      SessionData sessionData = SessionData.getSessionData(request);
      sessionData.setMessage(e.getLocalizedMessage());
     }

    }

    /**
     * Prepare for configure mode page 2
     */
    protected void page2Action(PortletRequest request,
      PortletContext portletContext)
      throws PollException {

      // Get configuration form data bean from session
      SessionData sessionData = SessionData.getSessionData(request);
      ConfigureFormData formData = sessionData.getConfigureFormData();

      // Update the persistence type and the datasource from request
      // parameters, validate the input.
      formData.updateDbParms(request);

      // Get the list of current polls (by name) from the data store
      // Get a broker instance using the form data parameters
      Broker broker = Broker.getInstance(formData.getPersistType(),
        formData.getDatasource());
      List pollNames = broker.getPollNames();

      // Put it in form data object
      formData.setPollNames(pollNames);

    }
```

You see in this example code that the `actionPerformed` method is passed an `ActionEvent` object. From the `ActionEvent` you can get the `PortletRequest` object and you can also get the name of the action event. So, this single method becomes the dispatcher for all action event processing for the portlet. In this example we show two action events, `verify` and `config_pg2`, and the code in separate methods that needs to execute for each of those events.

You can see that the `actionPerformed` method can become very cluttered with "if" statements while matching event name strings to invoke appropriate application behavior. You can solve this problem easily by changing the action event names from arbitrary strings to class names and then have the `actionPerformed` method in your portlet class get an instance of that class and invoke its `actionPerformed` method. This allows you

to ensure better separation of responsibility in our code. You will see in the chapter titled *Struts Portlet Framework* that using the Struts framework to develop our portlets solves this problem.

Your JSPs will have to be changed to specify fully qualified class names in their `PortletURI` tags. We also introduce an abstract `Action` class as the superclass of our `Action` classes. The `ActionClassManager` is used to return `Action` subclass instances given a class name. Then our `actionPerformed` method is quite simple and looks like the following (this can be added to the `PollPortlet` in the Studio project):

```
/**
  * Handle all Action requests
 */
 public void actionPerformed(ActionEvent event)
  throws PortletException {

  PortletRequest request = event.getRequest();
  PortletContext portletContext = getPortletConfig().getContext();
  PortletLog log = portletContext.getLog();

  // Get the action handler class and dispatch
  String actionClassName = event.getActionString();
  Action action = ActionClassManager.getAction(actionClassName);
  if (log.isDebugEnabled())
        log.debug("Dispatching to action class "+actionClassName);

  // Dispatch to that class event handler
  action.actionPerformed(request, portletContext);
  action.setView(request);
}
```

Now the code that was in the `verifyAction` method can be moved to the `actionPerformed` method of the new class `ConfigVerifyDataSourceAction`. Likewise, the code that was in the `page2Action` method can be moved to the `actionPerformed` method of the new class `ConfigPg2Action`.

The last structural change you will make will allow a similar separation of concerns in our implementation for the render phase processing as you just did for the action phase. You would have a similar problem with the `doConfigure` method, as it would need to handle all render requests for all configure pages in our portlet. You would need to determine which page in configure mode you should render, and then create any necessary beans and invoke the appropriate JSP. Again, you can segment this work out by introducing the idea of `View` classes and use the `Action` event processing to set the `View` class name to be dispatched to it during the render phase. You see that in the `setView` method in the `Action` classes.

Action Classes

The following is the `ConfigVerifyDatasourceAction` class:

```
/**
 * Action - Handle the request to test the datasource
 */

public class ConfigVerifyDatasourceAction extends Action {
  public void actionPerformed(PortletRequest request,
  PortletContext portletContext) throws PollException {

  try {

    // Get configuration form data bean from session
    SessionData sessionData = SessionData.getSessionData(request);
    ConfigureFormData formData = sessionData.getConfigureFormData();

    // Update the persistence type and the datasource from request
    // parameters, validate the input.
    formData.updateDbParms(request);

    // Check the datasource name and the database tables
    // If an exception occurs put the exception message out
    DbBroker dbBroker = new DbBroker(formData.getDatasource());
    dbBroker.verifyTables();

    // Success, put a message out
    ResourceBundle bundle = ResourceBundle.getBundle("nls.polling",
      request.getLocale());
    String msg = bundle.getString("configure.verify_success");
    sessionData.setMessage(msg);

   }
  catch (PollException e) {
    SessionData sessionData = SessionData.getSessionData(request);
    sessionData.setMessage(e.getLocalizedMessage());
   }
 }

 /**
  * Set the next Portlet view... if this is a MODE change action
  * (createReturnURI) then set the next view to null to allow the
  * default view in the next mode
  */
 public void setView(PortletRequest request)
   throws PollException {
     setView(request, CONFIG_STATE);
 }
}
```

The `setView` method in the abstract `Action` class simply sets the `View` class name on the session object. During the render phase all you have to do is retrieve the `View` class name from the session object and dispatch to it. Realize that you can put the view class name on session using the portlet mode as the session attribute name. That is, you can have a view attribute on session, as well as configure, edit, and help attributes, each specifying a different `View` class name.

By implementing it this way, the user can leave configuration mode in page 2, for example, and when the user returns to configure mode he or she will return to that page instead of starting over from the initial page. Given this, you can dispatch not from the `doView`, `doEdit`, `doConfigure` methods in our `PollPortlet` class but from the `Service` method. So the `Service` method looks like the following:

```
/**
 * Service method
 */
public void service(PortletRequest request, PortletResponse response)
    throws PortletException, IOException {

  PortletContext portletContext = getPortletConfig().getContext();
  PortletLog log = portletContext.getLog();

  // Dispatch to the current View class for the current mode.
  // The View class name was set in the Action phase or is the
  // initial view.
  // View for the current portlet mode.
  View nextView = ViewClassManager.getView(request);
  if (log.isDebugEnabled())
    log.debug(Dispatching to view "+nextView.getClass().getName());
  nextView.performView(request, response,
    getPortletConfig().getContext());

}
```

The `ViewClassManager` simply retrieves the view class name from the session object using the current portlet mode as the attribute name and returns an instance of that class. If the attribute is not found on session then the initial view class for the current mode is returned.

Implementing the Action Classes

You have seen how the `Action` classes get invoked from the `actionPerformed` method in the Poll portlet class. You have also seen how a specific `Action` class is identified by name through the `PortletURI` tag in the

JSP and you have seen the implementation for the `ConfigVerifyDatasourceAction` class.

Configuration Mode

The configuration mode of this portlet has most of the action event processing. The view pages really have just one action and that is to register the user's vote. The edit mode we will see shortly also has only one action, to save the user's color preference for the results bar graph.

The configuration mode has the following actions to perform:

- Cancel the configuration (`CancelAction`)
- Add an answer choice (`ConfigAddChoiceAction`)
- Delete an answer choice (`ConfigDeleteChoiceAction`)
- Get user input and show next page (`ConfigPg2Action`)
- Get user input and show next page (`ConfigPg3Action`)
- Verify the datasource (`ConfigVerifyDatasourceAction`)
- Save the configuration updates (`ConfigSaveAction`)

The `ConfigAddChoiceAction` class and the `ConfigDeleteChoiceAction` class are very similar from a controller perspective. They simply get the current form data object from session, get the index of the select choice to be deleted or the choice string to be added, and call the appropriate method on the `PollBean` object. Here is the code for the action to add a choice.

```
/**
 * Action - Handle the request to add a poll choice
 */
public class ConfigAddChoiceAction extends Action {
  public void actionPerformed(PortletRequest request,
    PortletContext portletContext) throws PollException {

    // Get the form bean from session
    SessionData sessionData = SessionData.getSessionData(request);
    ConfigureFormData formData = sessionData.getConfigureFormData();

    // Get the updated choice and question values and
    // save them to the form data bean
    String pollQuestion = request.getParameter(POLL_QUESTION);
      formData.getPollBean().setQuestion(pollQuestion);
      String newChoice = request.getParameter(POLL_CHOICE);
      formData.getPollBean().addChoice(newChoice);
  }
```

```
    public void setView(PortletRequest request)
    throws PollException {
      setView(request,  CONFIG_P3_STATE);
  }
 }
```

The `ConfigPg2Action` class and the `ConfigPg3Action` class are also very similar. They get the current form data object from session, update the form data with the user settings from the request parameters, and update the form data object for the next page and invoke the JSP.

The following listing shows the `ConfigPg2Action` class. The datastore options from the first configuration page are retrieved and held in the form data and a list of existing polls (by name) is prepared for use by the JSP for the next configuration page. The `ConfigPg3Action` class is very similar to `ConfigPg2Action`, so we will not show it here.

```
/**
 * Action - Handle the request to move to the next
 * configuration page.
 */

public class ConfigPg2Action extends Action {

  public void actionPerformed(PortletRequest request,
    PortletContext portletContext) throws PollException {

    // Get configuration form data bean from session
    SessionData sessionData = SessionData.getSessionData(request);
    ConfigureFormData formData = sessionData.getConfigureFormData();

    // Update the persistence type and the datasource from request
    // parameters, validate the input.
    formData.updateDbParms(request);

    // Get the list of current polls (by name) from the data store
    // Get a broker instance using the form data parameters
    Broker broker = Broker.getInstance(formData.getPersistType(),
      formData.getDatasource());
    List pollNames = broker.getPollNames();

    // Put it in form data object
    formData.setPollNames(pollNames);

  }

  public void setView(PortletRequest request)
    throws PollException {
      setView(request, CONFIG_P2_STATE);
  }
}
```

The final action event is to save the configuration mode modifications. This is invoked from the last page and implemented in the `ConfigSaveAction` class. In the following listing you see that, like the other configuration action classes, the form data is retrieved from session and updated with the configuration values from the last page.

The poll name and the persistence options are saved to the portlet settings. Keeping those values there allows concrete portlet instances each to represent a different poll with potentially different persistent stores. The poll information itself is saved to the persistent store.

```
/**
 * Action - Handle the request to save the configuration
 * changes
 */
public class ConfigSaveAction extends Action {

  public void actionPerformed(PortletRequest request,
     PortletContext portletContext) throws PollException {

    try {

      // Get form data and poll bean from session
      SessionData sessionData = SessionData.getSessionData(request);
      ConfigureFormData formData = sessionData.getConfigureFormData();
      PollBean pollBean = formData.getPollBean();

      // Get the question and save it to the form data bean
      String pollQuestion = request.getParameter(POLL_QUESTION);
        pollBean.setQuestion(pollQuestion);

      // Update the portlet settings
      PortletSettings portletSettings = request.getPortletSettings();
        portletSettings.setAttribute(POLL_NAME, pollBean.getName());
      portletSettings.setAttribute(PERSIST_TYPE,
        formData.getPersistType());
      portletSettings.setAttribute(DATASOURCE,
        formData.getDatasource());

      // Save the portlet settings
      portletSettings.store();

      // Save the poll info
      Broker.getInstance(portletSettings).updatePoll(pollBean);

      // Clean up session data
      SessionData.removeFrom(request.getPortletSession());

    }
```

```
    catch (AccessDeniedException e) {
      ResourceBundle bundle = ResourceBundle.getBundle("nls.polling",
        request.getLocale());
      String msg = bundle.getString("exception.accessdenied");
      throw new PollWrapperException(msg, e);
    }
    catch (IOException e) {
      ResourceBundle bundle = ResourceBundle.getBundle("nls.polling",
        request.getLocale());
      String msg = bundle.getString("exception.io");
      throw new PollWrapperException(msg, e);
    }
  }

  public void setView(PortletRequest request)
    throws PollException {
      setView(request, null);
  }
}
```

View Mode

The view mode has only one action class and that is to register the user's vote. That class simply gets the `userid` from the `User` object for the logged-in user and retrieves the vote selection from the request parameter. It then calls the `Broker` class to register the vote. The following listing shows the `VoteAction` implementation.

```
/**
 * Action - Handle the request to cast a vote
 */
public class VoteAction extends Action {
  public void actionPerformed(PortletRequest request,
    PortletContext portletContext) throws PollException {

    // Get the userid and selected choice
    String userid = request.getUser().getUserID();
    String indexStr = request.getParameter(SELECTED_CHOICE);
    int index = new Integer(indexStr).intValue();
    // Register the vote
    PortletSettings portletSettings = request.getPortletSettings();
    String pollName = portletSettings.getAttribute(POLL_NAME);
    Broker.getInstance(portletSettings).vote(userid, pollName, index);
  }
  public void setView(PortletRequest request)
    throws PollException {
      setView(request, VIEW_STATE);
  }
}
```

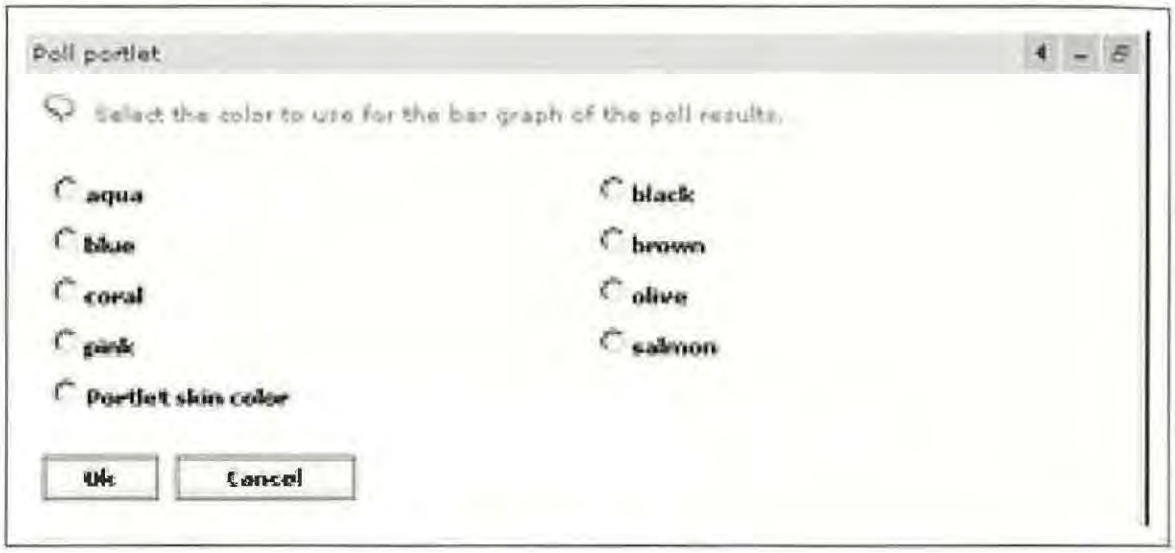


Figure 16-3 Edit mode.

Edit Mode

The final action class is for edit mode and it is the code to save the edit customization information (see Figure 16-3). The customization data is user-specific and so it is stored in portlet data. The following listing shows the implementation for the `EditSaveAction` class:

```
/**
 * Action - Handle the request to save the edit customizations
 */
public class EditSaveAction extends Action {

  public void actionPerformed(PortletRequest request,
    PortletContext portletContext) throws PollException {

  try {
    // Save the selected color to portlet data
    String color = request.getParameter(COLOR);
    if (color != null) {
      PortletData portletData = request.getData();
      portletData.setAttribute(COLOR, color);
      portletData.store();
   }
  }
  catch (AccessDeniedException e) {
    ResourceBundle bundle = ResourceBundle.getBundle("nls.polling",
      request.getLocale());
    String msg = bundle.getString("exception.accessdenied");
    throw new PollWrapperException(msg, e);
  }
  catch (IOException e) {
    ResourceBundle bundle = ResourceBundle.getBundle("nls.polling",
```

```
            request.getLocale());
        String msg = bundle.getString("exception.io");
        throw new PollWrapperException(msg, e);
      }
    }
    public void setView(PortletRequest request)
      throws PollException {
        setView(request, null);
    }
}
```

Implementing the View Classes

The `View` classes are invoked during the portlet rendering phase. Typically the view classes simply invoke the JSP. Beans needed to render the JSP are most often created during the action phase processing and put in session data for use by the JSP. You might be inclined to instead put these objects in the request object during action processing; however, they would then not be available on subsequent portal page refreshes when the portlet is asked to render itself without explicit user interaction with the portlet and so no action processing occurs.

The following view classes are used in this portlet:

- Initial configure view (`ConfigureView`)
- Configure view—page 2 (`ConfigurePg2View`)
- Configure view—page 3 (`ConfigurePg3View`)
- Initial edit view (`InitialEditView`)
- Initial view (`MainView`)

Only in the initial views do we need to execute processing logic. The `ConfigurePg2View` and `ConfigurePg3View` simply invoke their respective JSPs. Similarly, the initial edit view and the initial configure view do not require any processing prior to invoking their JSPs. The edit page simply shows a static set of color choices from which the user can make a selection. The initial configuration page shows static persistent store choices.

The main view is different however. This view requires that we get the portlet settings to determine the poll name and the persistent store information. Then we ask the `Broker` for the poll details and check if the user has already voted. If so, we need to call the `Broker` again for the current voting results. Since the voting totals are dynamic we want to refresh this information when the portlet is asked to render itself not just when the user is interacting with this portlet.

We have looked at the code to generate the main view earlier. Here it is again in complete form in the `MainView` class:

```
/**
 * View - Poll view
 */

public class MainView implements View {

public void performView(PortletRequest request,
    PortletResponse response, PortletContext portletContext)
    throws IOException, PortletException, PollException {

  // Is the portlet configured
  validateConfiguration(request);

  // Get the poll bean
  PortletSettings portletSettings = request.getPortletSettings();
  String pollName = portletSettings.getAttribute(POLL_NAME);
  Broker broker = Broker.getInstance(portletSettings);
  PollBean pollBean = broker.getPoll(pollName);

  // If the poll was not found generate an error message
  if (pollBean == null) {
    ResourceBundle bundle = ResourceBundle.getBundle("nls.polling",
      request.getLocale());
    String msg = bundle.getString("exception.notfound");
    throw new PollMessageException(msg);
  }

  // If the user has not already voted show poll question
  // otherwise show poll results.
  String user = request.getUser().getUserID();
  boolean hasUserVoted = broker.hasUserVoted(user, pollName);

  if (hasUserVoted) {

    // Get customization color value from portlet data
    PortletData portletData = request.getData();
    String color = (String)portletData.getAttribute(COLOR);
    request.setAttribute(COLOR, color);

    pollBean.initializeResults(broker);
    request.setAttribute("pollBean", pollBean);
      portletContext.include(VIEW_RESULTS_JSP, request, response);
  }
  else {
    request.setAttribute("pollBean", pollBean);
      portletContext.include(VIEW_QUESTION_JSP, request, response);
  }
 }
}
```

Implementing the Model Classes

Let's start with the implementation of the model classes for the polling portlet. There are two model objects that we should consider for this portlet. The most obvious is the object to represent a poll (`PollBean`). In addition, we need access to user information. For this, the `User` object is provided to the portlet developer through the portlet API. If we ensure that our portlet executes only on authenticated portal pages then we have access to the `User` object that represents the user that is logged in to portal.

The `PollBean` object needs to implement the behavior that we have seen so far. It needs to know its poll name, the poll question, and the collection of answer choices. The `Poll` object should also know the total number of votes cast, the number of votes cast for each answer choice, and the percentage value of votes for each choice. We will also implement an `initializeResults` method to explicitly initialize the votes cast information for the poll.

The `PollBean` implementation is pretty straightforward. Create a package in the Poll Project under the `JavaSource` directory. In the package create a new class called `PollBean`. The following listing shows most of the class implementation. Some of the accessor methods have been deleted in the interest of space.

```
public class PollBean implements Serializable, PollConstants {

  // Poll attributes
  protected String name = "";
  protected String question = "";
  protected String choices = "";

  // Choice list
  protected int numberOfChoices = 0;
  protected List choiceList = null;

  // Votes cast on this poll
  protected int votesCast = 0;
  protected List voteCountList = new ArrayList();
  protected List votePercentageList = new ArrayList();

  /**
   * Set the current vote counts
   */
  public void initializeResults(Broker broker) throws PollException {

    votesCast = 0;
    for (int i = 0; i < numberOfChoices; i++) {
      int voteCount = broker.voteCount(name, i);
      voteCountList.add(i, new Integer(voteCount));
```

```
            votesCast = votesCast + voteCount;
          }
          calcVotePercentages();
        }

        /**
         * Calculate the voting percentages for each choice
         */
        protected void calcVotePercentages() throws PollException {

          int voteCount = 0;
          int voteLargestPercent = 0;
          int voteLargestIndex = 0;
          int votePercent = 0;
          int voteCumPercent = 0;

          for (int i = 0; i < numberOfChoices; i++) {
            voteCount = ((Integer) voteCountList.get(i)).intValue();
            votePercent = (voteCount * 100) / votesCast;
            votePercentageList.add(new Integer(votePercent));
            voteCumPercent = voteCumPercent + votePercent;
            if (votePercent > voteLargestPercent) {
              voteLargestIndex = i;
              voteLargestPercent = votePercent;
            }
          }

          voteLargestPercent = voteLargestPercent + (100 - voteCumPercent);
          votePercentageList.remove(voteLargestIndex);
          votePercentageList.add(
            voteLargestIndex,
            new Integer(voteLargestPercent));
      }

      /**
       * Sets the choices and parses the string
       *
       * @param choices The choices to set
       */
      public void setChoices(String choices) {
        this.choices = choices;
        parseChoices();
      }

      /**
       * Parse the choices string
       */
      public void parseChoices() {

        List choiceList = new ArrayList(3);
```

```
    int lastIndex = 0;
    int index = choices.indexOf(DELIMITER, lastIndex);
    while (index > 0) {
      choiceList.add(choices.substring(lastIndex, index));
      lastIndex = index + 1;
      index = choices.indexOf(DELIMITER, lastIndex);
    }

    // Set the number of poll choices
    this.numberOfChoices = choiceList.size();
    this.choiceList = choiceList;
  }

  /**
   * Add a poll choice
   * @param newChoice String
   */
  public void addChoice(String newChoice) {

    if (!newChoice.equals("")) {
      newChoice = stripDelimiter(newChoice) + DELIMITER;
      String newChoices = choices + newChoice;
      setChoices(newChoices);
    }
  }

  /**
   * Delete a poll choice
   * @param choiceIndex int
   */
  public void deleteChoice(int choiceIndex) {
    // Strip the choice from the choices string
    int index = -1;
    for (int i = 0; i < choiceIndex; i++) {
      index = choices.indexOf(DELIMITER, index + 1);
    }

    String prefix = choices.substring(0, index + 1);
    index = choices.indexOf(DELIMITER, index + 1);
    String suffix = choices.substring(index + 1);
    String newChoices = prefix + suffix;

    setChoices(newChoices);
  }

  /**
   * Strip the delimiter if it appears in the entered text.
   * No substitution or warning is provided.
   */
```

```
    protected String stripDelimiter(String choice) {

      int index = choice.indexOf(DELIMITER);
      if (index == -1)
        return choice;
      return stripDelimiter(
        choice.substring(0, index) + choice.substring(index + 1));
    }
}
```

A `Poll` instance will be created by a call to the persistent store access class. To simplify the persistence access for the answer choices they will be stored as a single string. When the choices field is updated, the setter method will also parse the string and update the list of choices that are held in the `choiceList` field as well as update the `numberOfChoices` field. This will be important when we define the persistence class definitions in the next section.

There will also be a method to initialize the poll results. This method will get the vote count for each answer choice from the persistent store and calculate the voting totals and the voting percentages.

Implementing Persistence Classes

There will be two implementations for persistence layers for this portlet. We will implement a database broker and a memory broker. Of course, the data "stored" in memory is lost when the server restarts. But, it is useful for demonstration purposes. Both broker types will extend the abstract broker that we will define.

The `Broker` class defines the following abstract methods:

- `getPoll(String pollName)`. Gets the poll information from the datastore for the poll of the given name. Creates and returns a Poll instance with that data.
- `getPollNames()`. Returns a list of the names of existing polls.
- `deletePoll(String pollName)`. Deletes the poll data with the given name.
- `updatePoll(PollBean poll)`. Updates the poll data.
- `vote(String userid, String pollName, int choiceIndex)`. Records a vote for the given user in the given poll for the choice index.
- `hasVotingStarted(String pollName)`. Returns true if a vote has been registered for a poll name; false otherwise.

- `hasUserVoted(String userid, String pollName)`. Returns true if the `userid` has a recorded vote in the poll with the given name.
- `voteCount(String pollName, int choiceIndex)`. Returns the number of votes for the given choice index for the poll with the given name.

The `Broker` class has static methods and static variables to return an instance of either a `DbBroker` or a `MemoryBroker`. The portlet configuration defines the persistent store type, memory, or database. If a type of database is selected, then a datasource name is specified as well. A call to the `Broker` `getInstance()` method will return either a `DbBroker` or `MemoyBroker` instance based on those values.

DatabaseBroker

The `DatabaseBroker` queries the poll and vote data from two tables in a database specified by the named datasource. The tables are POLL and VOTE and can be defined by the following SQL:

```
CREATE TABLE POLL (NAME VARCHAR(32) NOT NULL, QUESTION VARCHAR(256),
CHOICES VARCHAR(512), PRIMARY KEY(NAME))

CREATE TABLE VOTE (USERID VARCHAR(255) NOT NULL, POLL VARCHAR(32) NOT
NULL , VOTE INTEGER NOT NULL, PRIMARY KEY(USERID, POLL))
```

Implementing Utility Classes

Our portlet defines two exception classes. One is an exception to wrap another thrown exception and associate our own message with it. The other is a message exception that we can use to terminate processing and put a message out to the portlet page. These are `PollWrapperException` and `PollMessageException`. Both extend `PollException`.

Finally, the string constants needed for the portlet application are defined in the `PollConstants` interface. This completes the utility classes in use by the Poll portlet.

Summary

In this chapter you looked at the key elements of the code for the Poll Portlet implemented using the IBM Portlet API. We discussed the overall design of the portlet application and focused on a design that maintained good separation of concerns.

The portlet implementation should be in the use of `PortletSettings` and `PortletData` for configuration and customization data. We discussed issues of page navigation. The portlet demonstrated an implementation with multiple portlet modes. We accessed text strings from `ResourceBundles` in the java code as well as the JSP to enable the portlet for internationalization. We showed the use of the `PortletLog` to generate trace-logging messages. We implemented two `Broker` classes to manage a persistent store for our poll data.

In the next chapter you will fill in any missing pieces of this implementation from the download source code. We will execute the code using WebSphere Studio's test environment.

CHAPTER

17

Portlet Interactive Debug and JSR 168 Example

This chapter will examine the interactive test environment of WebSphere Studio. You will execute the example Poll portlet discussed in the last chapter. You will learn about configuring the test server and how to use the debugging environment using a local Portal test environment.

You will also modify the example Poll portlet to use the JSR 168 API instead of the IBM Portlet API used in its initial development.

Poll Portlet Project

If you did not work through the complete portlet implementation in the last chapter you may want to download the source code and import it into an empty portlet project, allowing the import to overwrite any existing files. You can also use the import utility to create a new project. After the import is complete and the project is created, you will see a number of errors in the project. That is because required portlet jar files are not on the project build path. As we have discussed, one of the benefits of using the portlet toolkit to create portlet application projects is that it will set up the correct class path definition.

To add the needed IBM Portlet API jar files to the project build class path, open the properties on the newly created project by right-clicking on the project in the Navigator pane and selecting Properties from the pop-up menu. Select Java Build Path and then the Libraries tab. You can add the required jar files with the following steps:

- Select Add Variable, then select WPS_V5_PLUGINDIR in the New Variable Classpath Entry and click Extend. Select `portlet-api.jar` and click OK.
- Select Add Variable, then select WPS_V5_PLUGINDIR in the New Variable Classpath Entry and click Extend. Select `wps.jar` and click OK.
- Select Add Variable, then select WPS_V5_PLUGINDIR in the New Variable Classpath Entry and click Extend. Select `wpsportlets.jar` and click OK.
- Select Add Variable, then select WAS_50_PLUGINDIR in the New Variable Classpath Entry and click Extend. Expand the lib directory and then select `dynacache.jar` and click OK.
- Click OK to exit properties and save changes.

You may need to rebuild the project to resolve some JSP errors. Right-click onthe project in the Navigator pane and select Rebuild Project.

Portal Server Configuration

You can use Studio to debug portlet applications that are running on WebSphere Portal running locally within the Studio environment or on a remote machine. In this chapter we will discuss local environment debugging, which is referred to as *WebSphere Portal 5.0 Test Environment*. You will need to ensure that you have installed WebSphere Portal in WebSphere Studio using the Portal Toolkit installer.

In order to run the portlet in the local portal test environment, you must create a server and portal server configuration in Studio (see Figure 17-1). After the portal server configuration is defined, you can publish and debug your portlet application.

In Studio, select File ➪ New ➪ Other from the menu bar. In the dialog box that opens, select Server in the left pane and then Server and Server Configuration on the right. Select Next.

In the next dialog box, enter a name of your choice for the server and select Test Environment under *WebSphere Portal version 5.0* for the Server type. Click Next.

The next dialog box allows you to change the HTTP port number that will be used in the server configuration (see Figure 17-2). Make a change if needed and then click Finish. Change to the Server perspective. You will see the server you just created in the Server view. Open the server to edit the configuration.

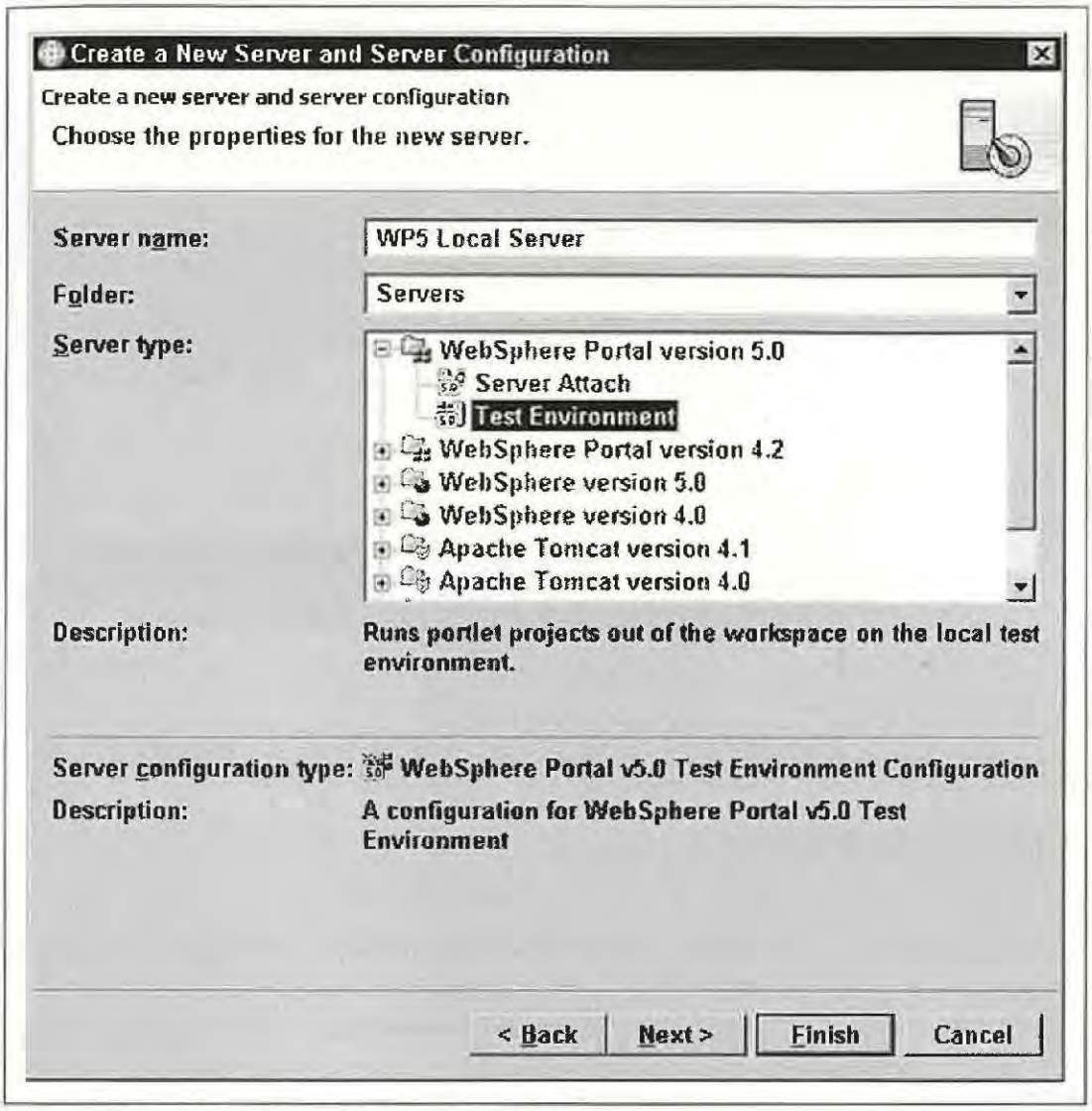


Figure 17-1 Creating a new server in Studio.

Here you can specify the data source for the poll application if you will use a database for storing the application data. First, select the Security tab to open the Security Options page. Create a JAAS Authentication Entry for the userid that will be used to authenticate to the database when a connection is requested. Then select the DataSource tab. From the Data Sources page create a JDBC Provider, if needed. For DB2, select IBM DB2 Database Type and DB2 Legacy CLI-Based Type 2 JDBC Driver for the JDBC provider type. On the second page make sure the class path is set to point to the correct JDBC driver library.

Select the JDBC Provider and click Add to add a data source. From the Create a Data Source dialog box, select the JDBC Provider and select Version 5.0 Data Source. Select Next. Provide a data source name of your choice. Enter a JNDI name of your choice, keeping the `jdbc/` prefix. The name you enter here (after `jdbc/`) is the data source name that you will use in the

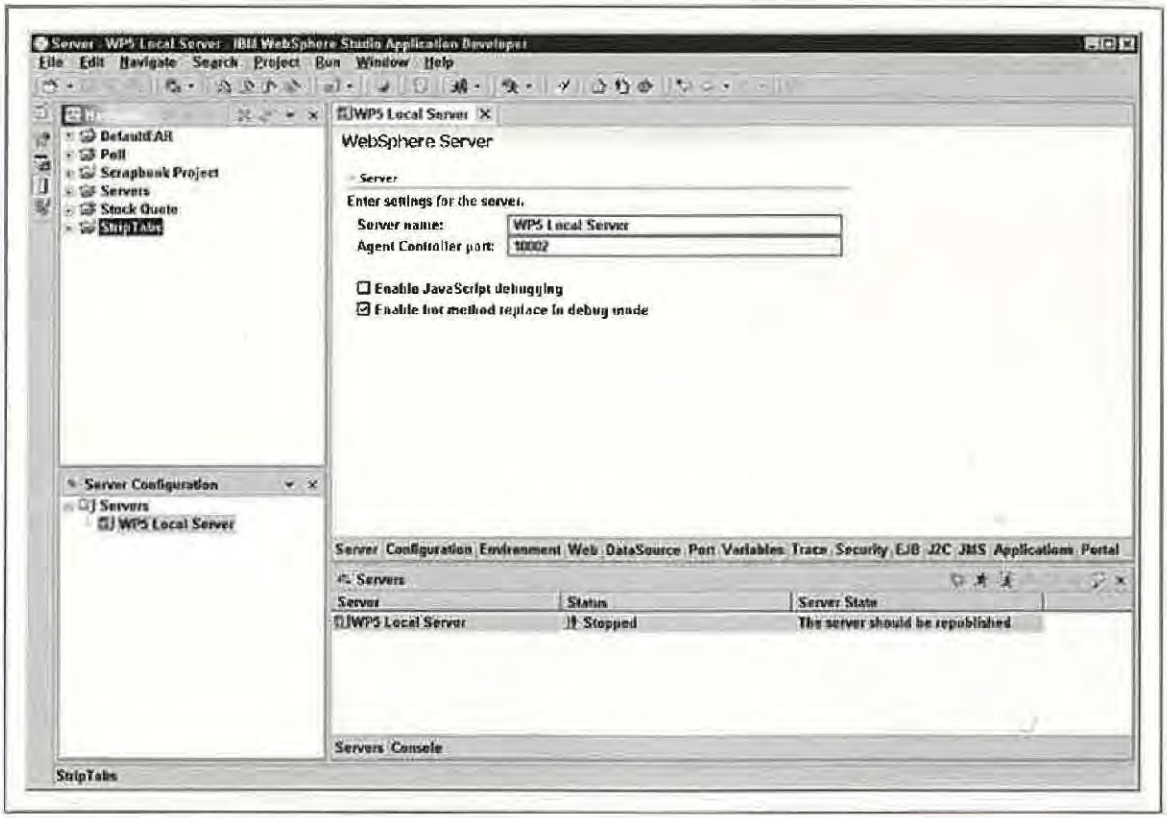


Figure 17-2 Server configuration.

portlet configuration. For Component-managed authentication alias select the JAAS Authentication Entry that you created. Uncheck the check box for Use this Data Source in Container Managed Persistence (CMP). This is the minimal set of changes you need to make. Make other changes as needed for your environment. Select Next.

Enter the database name on this page. This is the only required configuration on this page. Make other changes as needed for your environment. Select Finish. The data source has been defined. Close the server configuration edit and save the changes.

A portal server configuration is like a server configuration with the addition of portal-specific configuration information kept in the `wps-info.xml` file. You can view the configuration files through a filter change. Switch to the Web perspective and from the Project Navigator in the left pane click on the menu icon. From the drop-down list select Filters. From the Filters dialog box that appears, select Server in the project types list and select OK.

Back in the Server perspective, the server has been configured but we have not associated any portlet projects with it. Right-click on the server you just added from the Server view. Select Add and Remove Projects. You will add Enterprise Application (EAR) projects to the server configuration (see Figure 17-3). When you created the portlet project, it was associated



Figure 17-3 Adding and removing projects.

with an EAR, `DefaultEAR`, by default. If you did not change that then add the `DefaultEAR` project to the server configuration.

To change the EAR a portlet project is deployed into, open the EAR deployment descriptor (`application.xml`). Select the Module tab and use the Add and Remove buttons to change the Web applications associated with this EAR.

When the Portal test environment is started, the Poll portlet (and any other portlets in the `DefaultEAR` project) will be deployed on a debug portal page.

You can publish, start, and stop the server manually. Publishing copies the application project, resource, and configuration files to the correct server location. After the publishing is complete you can start the server. You can also start the server in debug mode. If not started in debug mode you will be

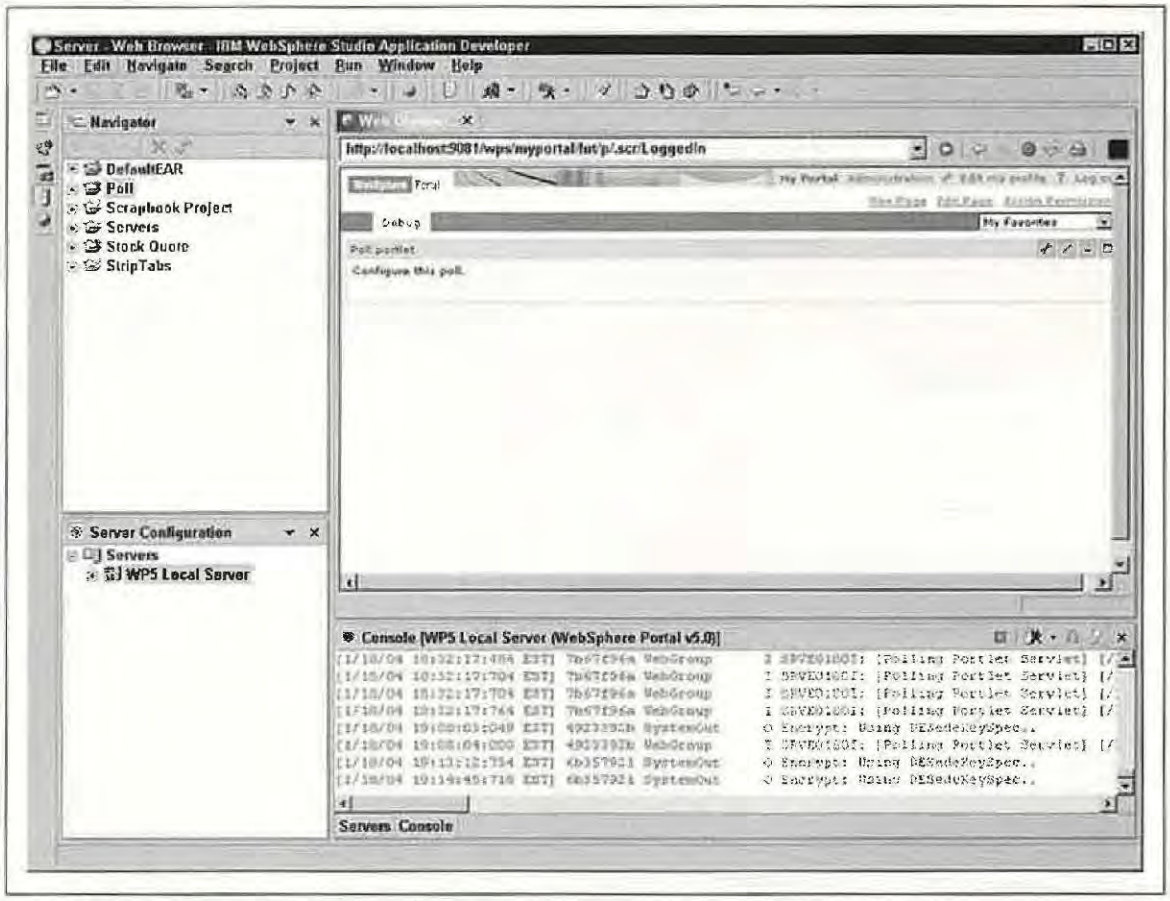


Figure 17-4 Portal page.

able to execute your portlets; in debug mode you can set breakpoints, interrogate code, and inspect variables. These actions can be performed from the Server view from the pop-up menu and the server. After the server starts you can open a browser to `http://hostname:9081/wps/myportal` where `hostname` is your server hostname and `9081` is the HTTP server port that was configurable when the server was created.

As an alternative you can start the server and launch your portlet in a Studio Web browser by right-clicking the portlet project and selecting either Debug on Server or Run on Server from the pop-up menu. After logging in with the default portal admin id (`wpsadmin`) the portlet is shown (see Figure 17-4).

Portal Debug Mode

When you start the server in debug mode you can set breakpoints in your Java source files or JSP files and then, using the Debug perspective, debug your code. You can step through the code, inspect variables, and evaluate expressions in context of the current thread's stack frame.

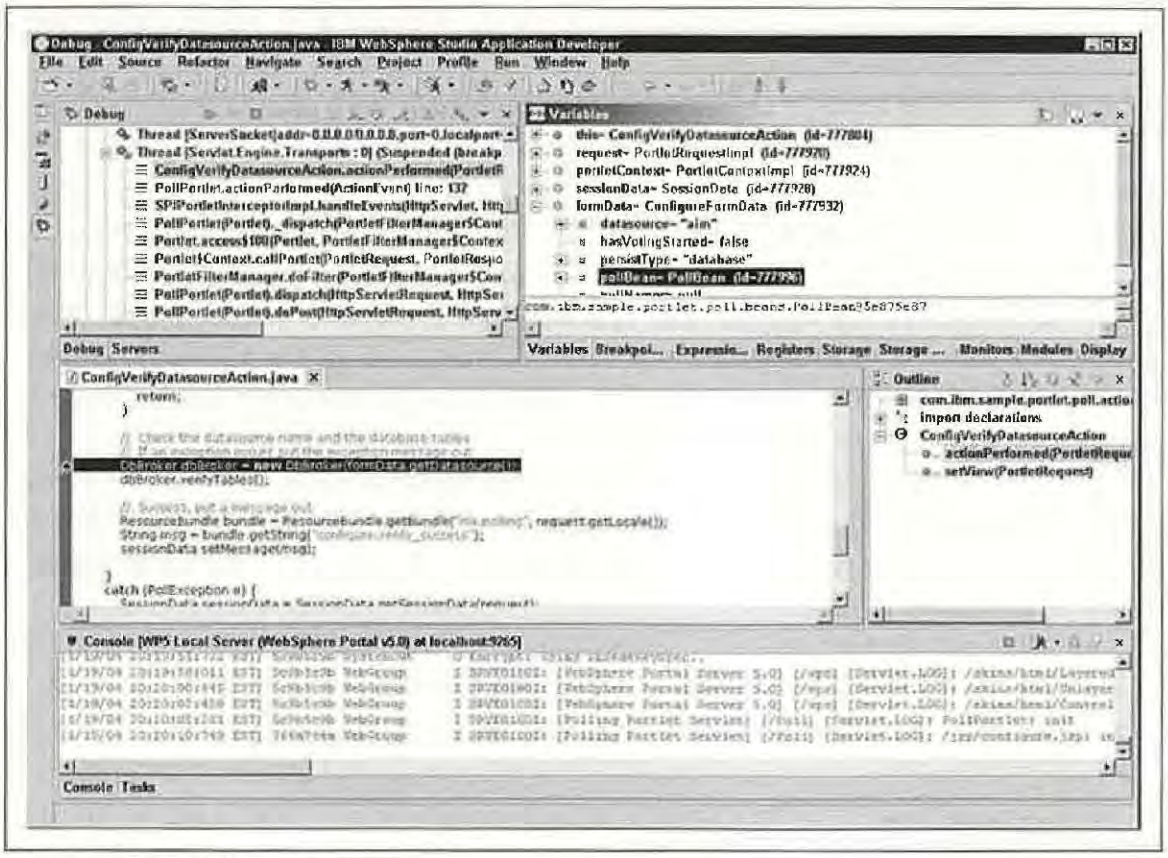


Figure 17-5 Debugging your code.

Breakpoints can be made conditional and they can be active based on a hit count. You can set exception breakpoints or breakpoints can be set on binary types where there is no source code available.

In our example application we added a breakpoint to the following line in the `ConfigVerifyDatasourceAction` `actionPerformed` method:

```
DbBroker dbBroker = new DbBroker(formData.getDatasource());
```

Then when we ran the portlet and clicked the Verify button in configure mode, the Debug perspective opened, with processing suspended at that breakpoint. Figure 17-5 shows the debugger opened, with execution suspended at that method. You see in the Variables view the current values of the method variables. The current stack is shown along with the current class outline. We can continue execution by stepping through the code to trace the application execution. We can also inspect code expressions in the current method or through the Expressions view.

We can make changes to our portlet code and have those changes immediately reflected in our executing test environment. There is generally no need to restart the server or republish the application in order to test code modifications. However, if changes are made to the deployment descriptors for example, those changes require a server restart. But, changes can

be made to Java classes or JSPs in the Portlet perspective, saved, and then that code immediately executed.

Studio's ability to interactively debug portlet code in a test environment with dynamic changes provides an extremely powerful development environment. The portlet toolkit provides assistance for getting the portlet application development started by creating the portlet project and starting the base implementations. The portal test environment significantly enhances that by providing a highly productive execution environment that assists in application code debugging while allowing developers to more quickly understand the portlet API through its interactive nature.

The step-by-step debug mode of WebSphere Application Server lets you debug WebSphere Portal internal methods. This is enabled by default. To disable the step-by-step debug mode so that you stop only at the breakpoints that you set, do the following:

1. Select Window ⇨ Preferences from the menu bar
2. Select WAS Debug mode to open the WAS Debug page.
3. Deselect the Use step-by-step Debug mode check box.
4. Click OK.

Portal Trace Logging

The IBM Portlet API provides a logging mechanism so that portlet developers can easily write trace messages to a common log. By default, the trace messages are written to a log file named `wps_<date>.log` in the `<wp-root>/log` directory, where `<date>` is the timestamp when the log was first created.

WebSphere Portal also supports the redirection of its trace logging to the WebSphere Application Server log. This function can be enabled in the `<wp_root>\shared\app\config\log.properties` file.

You can enable trace logging using an administrative portlet or through a properties file change. Using the Enable Tracing portlet will allow you to specify the trace settings temporarily. When the server is restarted, the trace settings return to the previous state. You can also update the same `log.properties` file and specify the trace setting there. In this case the change will get applied when the server is restarted.

In either case the trace specification is through a trace string. See the Portal Information Center in the section Using Logs for the definition of trace strings. For example, to enable tracing of portlets use the following

trace string:

```
com.ibm.wps.portlets.*=all=enabled:org.apache.jetspeed.portlet
.PortletLog=all=enabled
```

Enable tracing in our Portal test environment. The portal root (<wp-root>) in Studio is <studio-root>\runtimes\portal_v50. So the log file is at <studio-root>\runtimes\portal_v50\shared\app\log.properties. Uncomment the traceString property and set it to the string shown above.

You will recall that we had two trace messages in our portlet code. In the service method and in the actionPerformed method we had the following. Note it is good practice to test if logging is enabled before attempting to write the trace message and also before constructing the trace message.

```
if (log.isDebugEnabled())
 log.debug(this.getClass().getName()+".service() dispatching to "+
 nextState.getClass().getName());

if (log.isDebugEnabled())
 log.debug(this.getClass().getName()+".actionPerformed() dispatching to
"+
 actionClassName);
```

So after executing our portlet with tracing turned on we should see portlet trace messages in the <studio-root>\runtimes\portal_v50\log\wps_2004.01.19-23.51.33.log file (see Figure 17-6).

```
2004.01.19 23:56:59.405 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 1
  com.ibm.sample.portlet.poll.portlet.PollPortlet.service() dispatching to com.ibm.sample.portlet.poll.views.MainView

2004.01.19 23:57:01.579 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 0
  com.ibm.sample.portlet.poll.portlet.PollPortlet.service() dispatching to com.ibm.sample.portlet.poll.views.InitialConfigureView

2004.01.19 23:57:05.516 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 0
  com.ibm.sample.portlet.poll.portlet.PollPortlet.actionPerformed() dispatching to com.ibm.sample.portlet.poll.actions.ConfigPg2Action

2004.01.19 23:57:06.606 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 0
  com.ibm.sample.portlet.poll.portlet.PollPortlet.service() dispatching to com.ibm.sample.portlet.poll.views.ConfigurePg2View

2004.01.19 23:57:13.145 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 2
  com.ibm.sample.portlet.poll.portlet.PollPortlet.actionPerformed() dispatching to com.ibm.sample.portlet.poll.actions.ConfigPg3Action

2004.01.19 23:57:21.225 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 2
  com.ibm.sample.portlet.poll.portlet.PollPortlet.service() dispatching to com.ibm.sample.portlet.poll.views.ConfigurePg3View

2004.01.19 23:57:27.255 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 1
  com.ibm.sample.portlet.poll.portlet.PollPortlet.actionPerformed() dispatching to com.ibm.sample.portlet.poll.actions.ConfigAddChoiceAction

2004.01.19 23:57:27.295 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 1
  com.ibm.sample.portlet.poll.portlet.PollPortlet.service() dispatching to com.ibm.sample.portlet.poll.views.ConfigurePg3View

2004.01.19 23:57:31.972 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 1
  com.ibm.sample.portlet.poll.portlet.PollPortlet.actionPerformed() dispatching to com.ibm.sample.portlet.poll.actions.ConfigAddChoiceAction

2004.01.19 23:57:31.992 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 1
  com.ibm.sample.portlet.poll.portlet.PollPortlet.service() dispatching to com.ibm.sample.portlet.poll.views.ConfigurePg3View

2004.01.19 23:57:32.964 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 1
  com.ibm.sample.portlet.poll.portlet.PollPortlet.actionPerformed() dispatching to com.ibm.sample.portlet.poll.actions.ConfigSaveAction

2004.01.19 23:57:33.144 1 org.apache.jetspeed.portlet.PortletLog debug Servlet.Engine.Transports : 1
  com.ibm.sample.portlet.poll.portlet.PollPortlet.service() dispatching to com.ibm.sample.portlet.poll.views.MainView
```

Figure 17-6 Portlet trace messages.

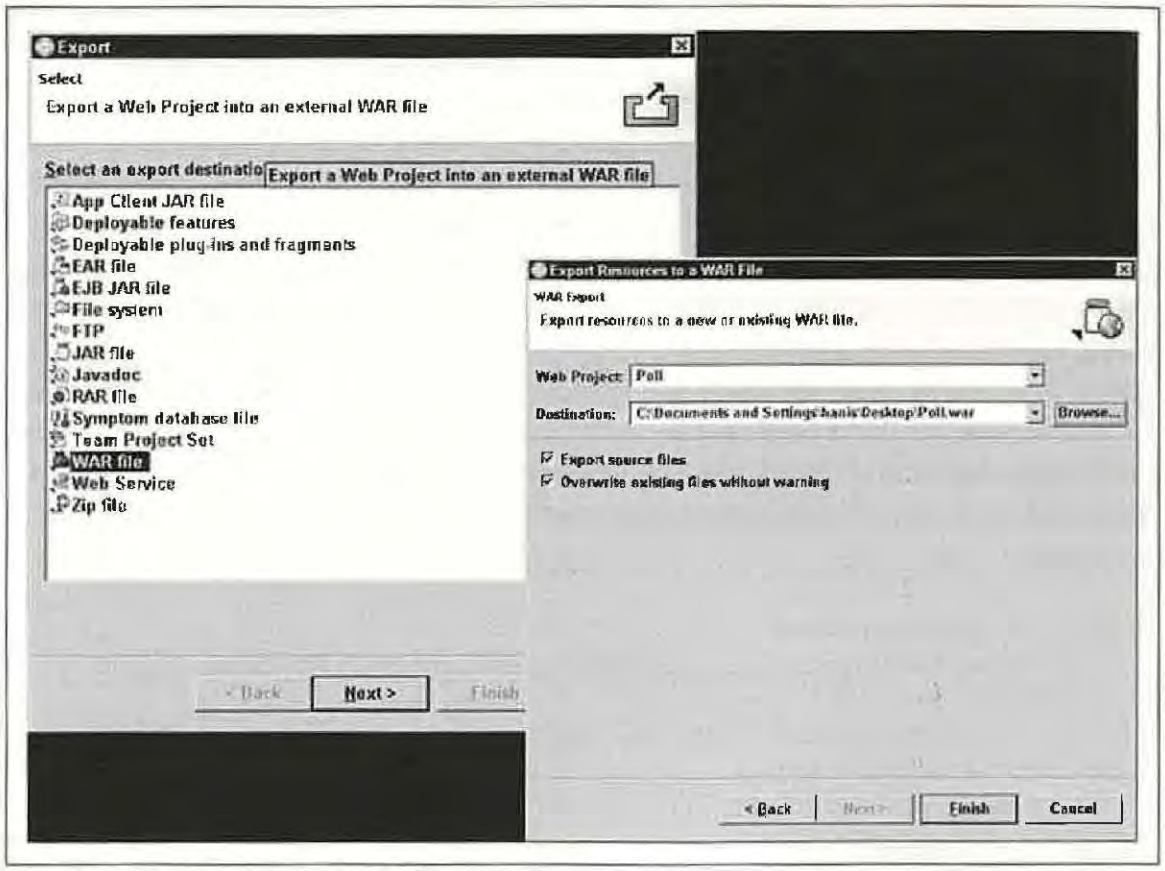


Figure 17-7 Exporting a WAR file.

Generating a Deployable WAR File

Generating a deployable WAR file in Studio using the portal toolkit is very easy. Since the portlet project is consistent with the expected contents of a Web archive file we simply need to export the project to a WAR file. That file can then be installed in WebSphere Portal.

For our Poll portlet simply right-click the `Poll` project in the Project Navigator view and select Export (see Figure 17-7). You can install this portlet application in Portal using the Install administrative portlet.

Deploying Using the Install Administrative Portlet

You can deploy the portlet application WAR file into Portal using the Install administrative portlet. Log in to portal as a user with administrative privilege (is a member of `wpsadmins`) and navigate to the Install pages by first selecting the Administration page, then the Portlets page, and finally the Install page.

The Install page contains a single portlet, the Install portlet. From this portlet you specify the location of the WAR file on your local file system and then execute the install function. The portlet application will be installed in portal. Typcially, the next administrative steps are to add portlet or portlets that were installed with that portlet application to a portlet page and assign resource permissions if required. Note that the Manage Portlet Applications portlet can be used to update an existing portlet application. This updates the portlets defined in the WAR file while preserving the portlet resource permissions and their page associations.

Deploying Using XML Configuration Interface

You can also deploy the portlet application using the XML configuration interface. The XML configuration interface is a command line portal configuration utility that uses an XML input file for configuration information and action requests. A stand-alone utility reads the XML source file and executes against a portal server through an HTTP connection.

Along with deploying portlet applications you can use this utility to back up and restore complete portal configurations, copy parts of a configuration to another portal, install additional resources on a portal, and perform recurring administrative tasks through a programmable interface.

See the current Portal Information Center for a definition of the structure of the XML file, the XML schema, and examples of common function (including deploying a portlet). If for any reason the administrative portlets to deploy portlets or manage access control become unavailable, it is very helpful to keep this utility in mind as it can provide critical function to resolve problems with portlets that you typically rely on to manage the portal environment.

Developing the Poll Portlet with JSR 168 API

In this next section we will discuss the changes needed to your Poll portlet implementation to use the JSR 168 API. A general discussion of the differences in the programming interface was described in Chapter 14. In the remaining section of this chapter we will apply that understanding to the portlet we just created using the IBM Portlet API.

Again we will start our portlet development using the Portal Toolkit in WebSphere Studio. We will be using the Studio wizard to create a new portlet application project to set up the correct class path defined for the JSR168 API, to create the base deployment descriptors, and to define our project directory structure.

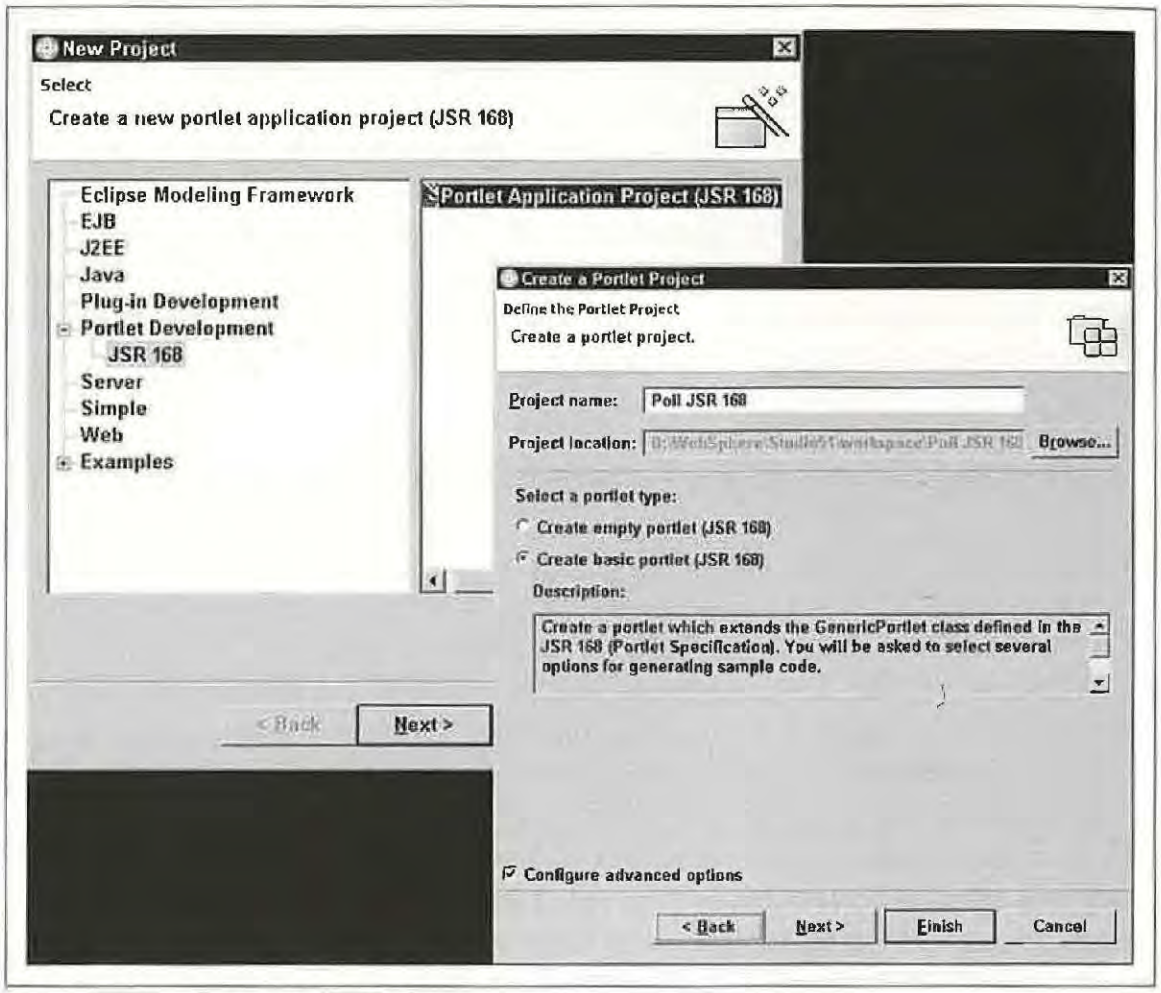


Figure 17-8 Create New Project.

Create the Poll Project

Using Studio 5.1 with the Portal Toolkit 5.02 (or later combination) select File ⇨ New ⇨ Project from the toolbar. From the New Project page that opens (see Figure 17-8), expand Portlet Development in the left pane and select JSR 168. In the right pane select Portlet Application Project (JSR 168) and select Next.

In the next page enter the project name **Poll JSR 168** and select Create Basic Portlet (JSR 168). Then select Configure Advanced Options and select Next.

In the J2EE Settings page (see Figure 17-9) accept the default settings and select Next.

In the Portlet Settings page select Change Code Generation Options to specify the poll portlet class name and the package name that is the same as the portlet you created using the IBM Portlet API. Then we can just copy that code into this project and make changes to the portlet classes without

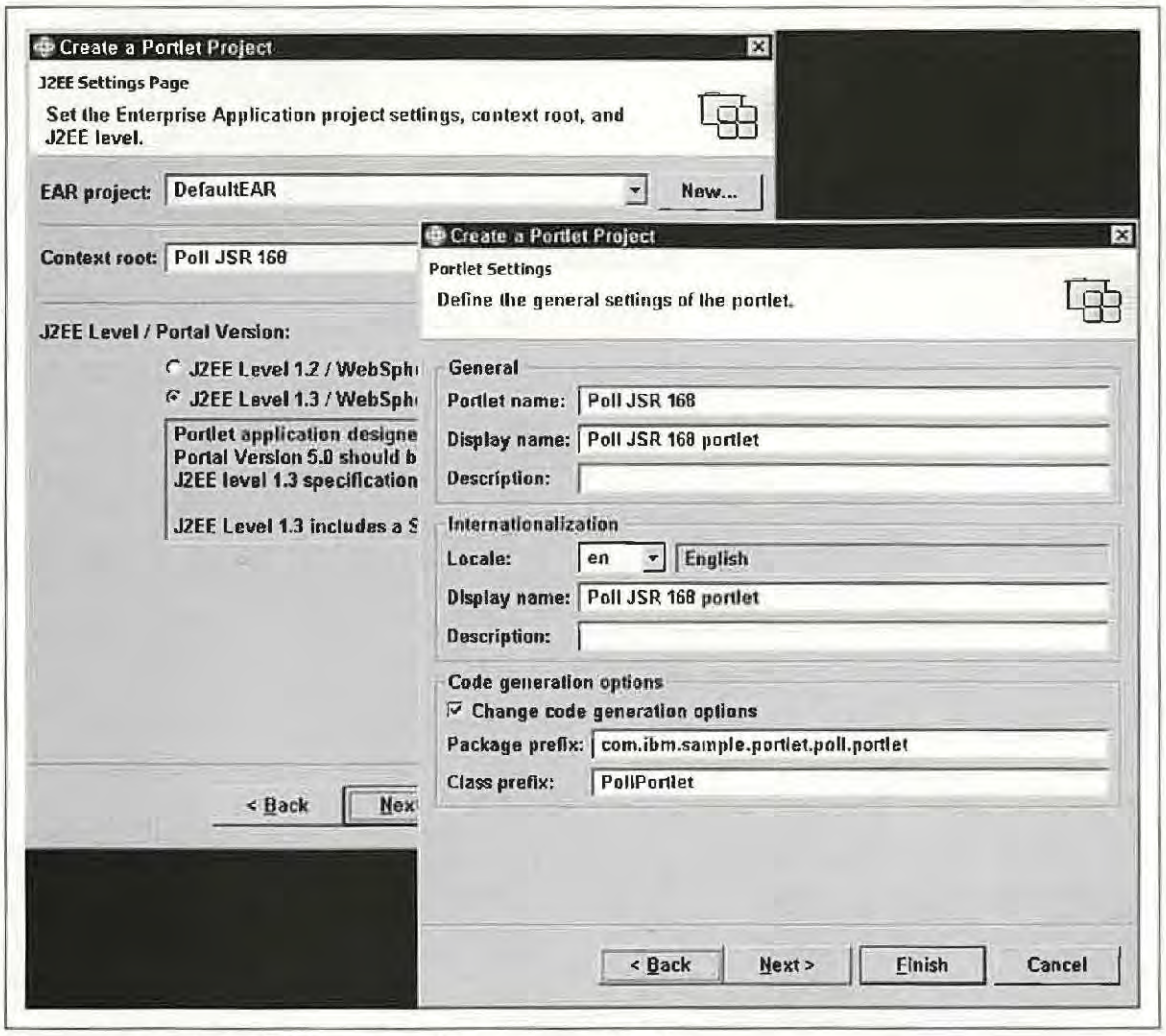


Figure 17-9 J2EE (left) and Portlet Settings (right) pages.

changing the deployment descriptor. The sample code provided specifies the *Package prefix* as `com.ibm.sample.portlet.poll.portlet` and the *Class prefix* as `PollPortlet`. Select Next.

In the Action and Preferences page (see Figure 17-10) deselect Add Form Sample and select Next. In the Miscellaneous page select Add Edit Mode and Add Configure Mode to see an example of the portlet implementation for processing of a standard mode (edit) and a custom mode (configure). Select Finish, completing the wizard and creating the project.

Generated Portlet Code

Next you will inspect the generated portlet code. The deployment descriptors are a key part of the generated portlet project.

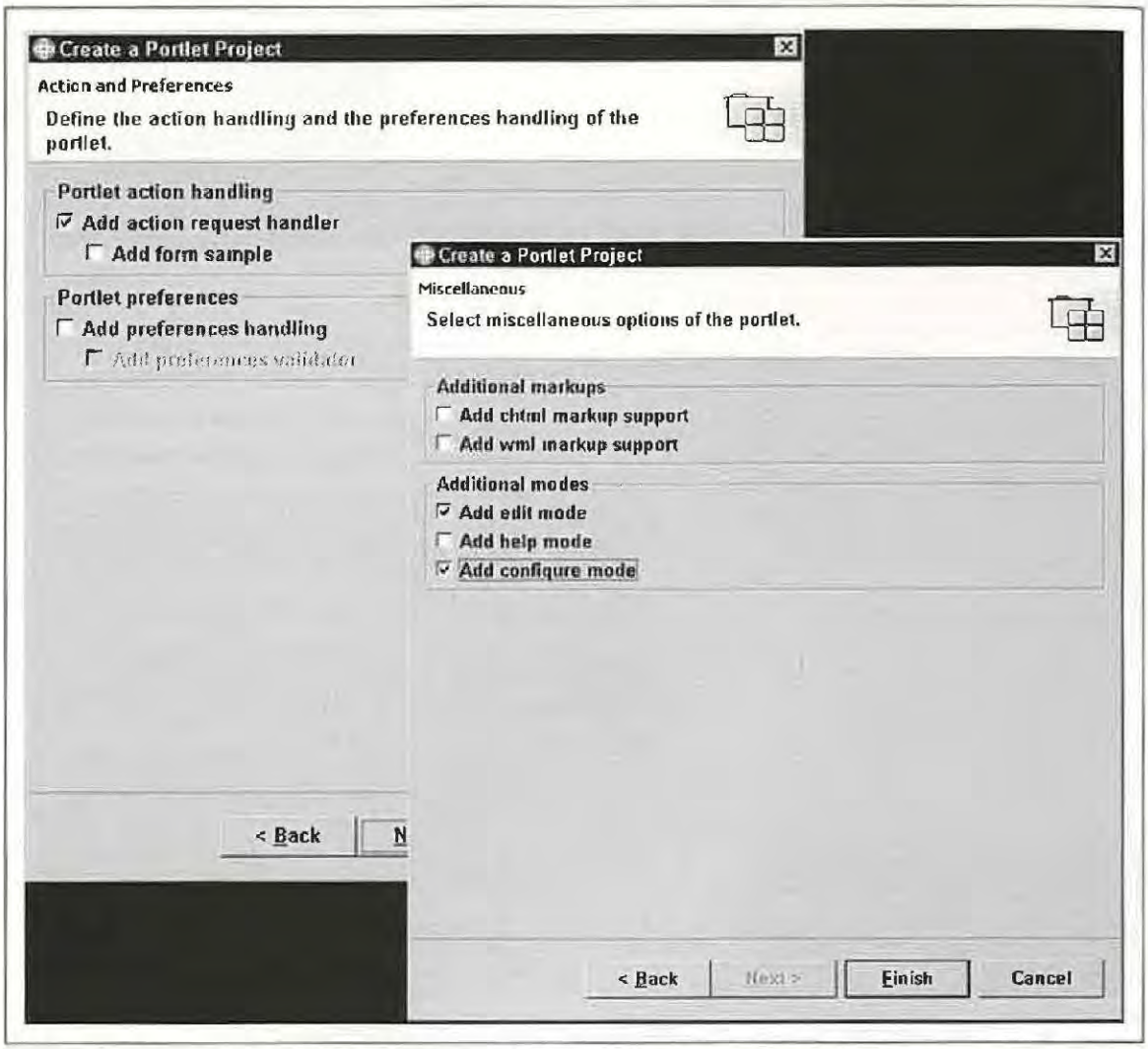


Figure 17-10 Action and Preferences (left) and Miscellaneous (right) pages.

Deployment Descriptors

The portlet and Web deployment descriptors are generated when the wizard completes. In a previous chapter we discussed the differences in the deployment descriptors between both APIs. Here we see the differences in these files.

Notice that with the JSR 168 API the portlet class is no longer defined in the Web descriptor `<servlet-class>` element. Instead it is defined in the portlet descriptor `<portlet-class>` element. The Web descriptor now simply has the welcome file list and a tag lib definition.

Also in the portlet descriptor you see the definition for *configure* custom portlet mode. You should update this file to include the portlet preferences (the portlet settings from the previous API) in the `<portlet-preferences>` element. The `<resource-bundle>` element should also

be updated to reference the correct path and filename of the bundle from the previous project that you will import.

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-
app_1_0.xsd"
  version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-
app_1_0.xsd
  http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
  <portlet>
    <portlet-name>Poll JSR</portlet-name>
    <display-name>Poll JSR portlet</display-name>
    <display-name xml:lang="en">Poll JSR portlet</display-name>
    <portlet-
class>com.ibm.sample.portlet.poll.portlet.PollPortlet</portlet-class>
    <init-param>
      <name>wps.markup</name>
      <value>html</value>
    </init-param>
    <expiration-cache>0</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
      <portlet-mode>edit</portlet-mode>
      <portlet-mode>config</portlet-mode>
    </supports>
    <supported-locale>en</supported-locale>
    <resource-bundle>nls.polling</resource-bundle>
    <portlet-info>
      <title>Poll JSR portlet</title>
    </portlet-info>
    <portlet-preferences>
      <preference>
        <name>persist_type</name>
        <value> </value>
      </preference>
      <preference>
        <name>datasource</name>
        <value> </value>
      </preference>
      <preference>
        <name>pollName</name>
        <value> </value>
      </preference>
    </portlet-preferences>
  </portlet>
  <custom-portlet-mode>
    <portlet-mode>config</portlet-mode>
  </custom-portlet-mode>
</portlet-app>
```

Poll Portlet Class

Review the generated `PollPortlet` class. The first obvious difference between this class and the IBM Portlet API version is that the portlet API objects are imported from the `javax.portlet.*` package. Those classes are resolved in the `portlet.jar` file that has been added to the build class path by the portlet application creation wizard. The portlet creation wizard also includes the `portlet-api.jar`, `wpsportlets.jar`, and `wps.jar` in the build path. You should remove the `portlet-api.jar` and `wpsportlets.jar` from the build path to help ensure you are not including any IBM Portlet API references. The `wps.jar` file includes the `org.apache.pluto.tags.*` classes which are needed for the custom JSP tags defined in the `std-portlet.tld` tag library.

The next noticeable difference is in the `PollPortlet` class definition: public class `PollPortlet` extends `GenericPortlet`. The portlet class extends `javax.portlet.GenericPortlet` and does not implement `ActionListener`. The JSR 168 API also implements two-phase processing, an action phase and a render phase. But the API defines two different URLs to reference the portlet, an `ActionURL` and a `RenderURL` instead of an event notification and listener mechanism. If the portlet is invoked using the `ActionURL` then the portlet's `processAction` method is called first. Also notice that this method is passed an `ActionRequest` object and an `ActionResponse` object. For render phase processing the `doView`, `doEdit`, `doHelp`, and `doConfigure` methods are passed a `RenderRequest` object and a `RenderResponse` object. The request and response objects are different between the processing phases.

In render phase processing notice that the `do` methods get invoked from the `doDispatch` method instead of the `service` method as in the previous API. The `GenericPortlet` implementation of the `doDispatch` method simply determines the current portlet mode and invokes the appropriate `do` method.

```
protected  void doDispatch(RenderRequest request,
    RenderResponse response)
    throws PortletException, IOException {
  WindowState state = request.getWindowState();
  if(!state.equals(WindowState.MINIMIZED)) {
    PortletMode mode = request.getPortletMode();
    if(mode.equals(PortletMode.VIEW))
      doView(request, response);
    else
    if(mode.equals(PortletMode.EDIT))
      doEdit(request, response);
    else
    if(mode.equals(PortletMode.HELP))
```

```
        doHelp(request, response);
      else
        throw  new PortletException("unknown portlet mode: " + mode);
    }
  }
```

However, the base implementation does not support a configure mode. The API allows portal vendors to support additional portlet modes as custom modes. The custom mode must be defined in the portlet deployment descriptor and then can be invoked by overriding the `doDispatch` method as shown in our generated `PollPortlet` code.

```
protected void doDispatch(RenderRequest request, RenderResponse
response) throws PortletException, IOException {
  if( !WindowState.MINIMIZED.equals(request.getWindowState()) ){
    PortletMode mode = request.getPortletMode();
    if( CUSTOM_CONFIG_MODE.equals(mode) ) {
      doCustomConfigure(request, response);
      return;
    }
  }
  super.doDispatch(request, response);
}
```

Modifying the Poll Project Source

We will work through the changes needed to convert our existing Poll Portlet to use the JSR 168 API. To do that we will copy the existing portlet code into this new project, preserving only the generated Web and portlet deployment descriptors. The generated source code, JSPs, and resource bundle along with their packages and folders should be deleted from the new project.

After deleting these files, use Copy and Paste in Studio to copy the Java Source packages and their contents and the WebContent images and jsp folders from the Poll project to the Poll JSR 168 project. You should have previously removed the `portlet-api.jar` and `wpsportlets.jar` from the project build path. As a result, you will see a number of errors in the new project after the copying completes.

PollPortlet

First change the `PollPortlet` class definition to extend `GenericPortlet` and remove "implements `ActionListener`." If you have an `init` method, then that method signature has changed and so it now throws a `PortletException` instead of an `UnavailableException`.

The actionPerformed method is replaced by a processAction method. The processAction method takes ActionRequest and ActionResponse parameters instead of an Event parameter. With no Event object available, the event processing is controlled by request parameters put on the ActionURL as opposed to an action name available from the Event object. All methods invoked from either processAction or actionPerformed must be updated to take an ActionRequest object instead of PortletRequest.

```
/**
 * Process an action request
 */
 public void processAction(ActionRequest request,
    ActionResponse response)
    throws PortletException, IOException {

  // Get the action handler class and dispatch
  String actionClassName = request.getParameter(ACTION);
  Action action = ActionClassManager.getAction(actionClassName);

  // Dispatch to that class event handler
  PortletContext
portletContext=getPortletConfig().getPortletContext();
  action.actionPerformed(request, portletContext);
  action.setView(request);
}
```

The PollPortlet service method must be replaced with a doDispatch method. The doDispatch method for our portlet simply gets the View class for the current portlet mode and dispatches to it.

```
/**
 * Dispatch to render View class for the current mode.
 * The View class name was set in the Action phase or
 * is the initial View for the current portlet mode.
 */
protected void doDispatch(RenderRequest request,
    RenderResponse response)
    throws PortletException, IOException {

  View nextView = ViewClassManager.getView(request);
  PortletContext portletContext=getPortletConfig().getPortletContext();
  nextView.performView(request, response, portletContext);
}
```

The ViewClassManager getInitialView method must then change to handle the custom configure mode. A new PortletMode instance is defined for the "config" mode defined by the portal vendor.

```
PortletMode CUSTOM_CONFIG_MODE = new PortletMode("config");
```

Then the current mode can be retrieved from the `RenderRequest` object and checked against this mode to determine the appropriate `View` class.

```
if (CUSTOM_CONFIG_MODE.equals(request.getPortletMode())) {}
```

Action Classes

In general, the only change needed for the `Action` classes is the `actionPerformed` method changes to the `ActionRequest` object. One difference we also need to consider is the portlet mode during action phase processing. With the IBM Portlet API a mode change occurs after the action processing. With the JSR 168 API the mode change occurs during the action processing. So, our portlet code needs to take this into consideration when the View is set for the current mode in the action phase. The only issue is during a mode change when we would normally remove the mode-specific View reference from session. With the new API, since the mode has already changed we remove View references for all modes from session.

The use of `PortletSettings` and `PortletData` must be changed to use `PortletPreferences`. Like the `PortletSettings` and `PortletData` objects the `PortletPreferences` object is available from the request object. You use the `setValue` and `getValue` methods to access the preference attributes and the store method to save them.

Model Objects

There are no additional changes needed for our model objects. Likewise, the persistence classes and utility classes require no changes.

View Classes

An additional change (beyond referencing `RenderRequest` and `RenderResponse` and changes to use `PortletPreferences`) is the invocation of the JSP.

Prior to rendering output to the response, either writing directly to the response `PrintWriter` or through a JSP, the response content type must be explicitly set. If it is not, you will get the following exception:

```
Nested Exception is java.lang.IllegalStateException: No content
type set.
```

JSPs are invoked from the `RequestDispatcher` instead of from the `portletContext`. So, an example JSP invocation is

```
response.setContentType(request.getResponseContentType());
portletContext.getRequestDispatcher(EDIT_JSP).include(request,response);
```

User Identifier

The API to retrieve the user profile information is quite a bit different between the APIs. The Poll portlet uses the id of the user to uniquely register a vote and to determine if the user has already voted. This function is implemented in the `VoteAction` action class and `MainView` state class, respectively. The IBM Portlet API returns the user identifier from the `User` object:

```
String user = request.getUser().getUserID();
```

The JSR 168 API returns the user identifier from the `java.security.Principle` object:

```
Principal principal = request.getUserPrincipal();
String user = principal.getName();
```

Note that WebSphere Applicaton Server security must be enabled for the `Principal` object to be created. If security is not enabled then the `getUserPrinciple` method will return null. By default for an LDAP configuration the `getName()` method will return the UID attribute.

Therefore, change the code in both classes to return the user name using the `Principle` object, making sure to check for a null object and return a meaningful error message if security is not enabled. The following code snippet assumes that the appropriate error message is defined in the resource bundle.

```
Principal principal = request. getUserPrincipal();
if (principal == null) {
     ResourceBundle bundle = ResourceBundle.getBundle("nls.polling",
        request.getLocale());
     throw new PollMessageException
        (bundle.getString("exception.security"));
}
String userName = principal.getName();
```

JSPs

The JSP must also change to use the new tag library. Instead of the `portlet.tld` tag library that was available with the IBM Portlet API, the `std-portlet.tld` is provided with the JSR 168 API implementation. This library contains the following tags:

```
- <portlet:defineObjects/> initialize RenderRequest,
RenderResponse, and PortletConfig objects support for use within the JSP
- <portlet:actionURL/> create a URL to invoke the processAction method
in the portlet.
```

```
- <portlet:renderURL/> create a URL to invoke the doDispatch method
in the portlet.
- <portlet:namespace/> provide the encoded namespace of the portlet.
```

For example, to generate a URL for the portlet specifying an action you would use the following tag:

```
<portlet:actionURL>
     <portlet:param name="action" value="anActionClassName"/>
</portlet:actionURL>
```

Summary

This chapter reviewed the interactive test environment of WebSphere Studio using the Poll portlet you developed in the previous chapter. You learned how to configure the test server and how to use the debugging environment using a local Portal test environment.

In the second part of the chapter you modified that Poll portlet to use the JSR 168 API instead of the IBM Portlet API used in its initial development. The needed changes were reviewed.

CHAPTER 18

Struts Portlet Framework

This chapter discusses the implementation of a portlet using Struts Portlet Framework. Struts is an Apache Jakarta project providing an open-source framework for building Web applications. WebSphere Portal provides Struts Portlet Framework that supports the deployment of Struts applications as portlets.

Like you did for the JSR 168 API portlet implementation, you will convert the Poll portlet developed using the IBM Portlet API for using Struts Portlet Framework.

After the implementation process has been discussed, various approaches to portlet development will be discussed, with recommendations given on when each approach is appropriate.

Creating the New Project

We initially designed and implemented the Poll portlet separating action event processing from our controller function and maintained clearly separated model and view classes. Because of this design our transition of this code to the struts framework will be very straightforward.

The first step in creating a portlet using Struts Portlet Framework is to import the `PortalStrutsBlank.war` file into your project. In WebSphere Portal V5.0 you will find the `PortalStrutsBlank.war` file in the `<portal_root>\dev\struts\StrutsPortlet` directory. Importing this war file adds the Struts jar files for both Jakarta Struts and IBM's Struts Portlet Framework to the project lib directory. Again, we will use WebSphere Studio for this development. The specific steps shown here are

based on WebSphere Studio 5.1.0 with Portal Toolkit version 5.0.2. Follow these steps to create the base project:

1. Import Struts Portlet Framework into a new project. To do so, follow these steps:
   a. Select File ⇨ Import ⇨ AR file from the Studio menu bar.
   b. In the Import page navigate to the `PortalStrutsBlank.war` file and select New for Project.
   c. In the New Web Project page, enter **Poll Struts** for Project Name and select Finish to create the project.
   d. Select Finish from the WAR file page to complete the import (see Figure 18-1).

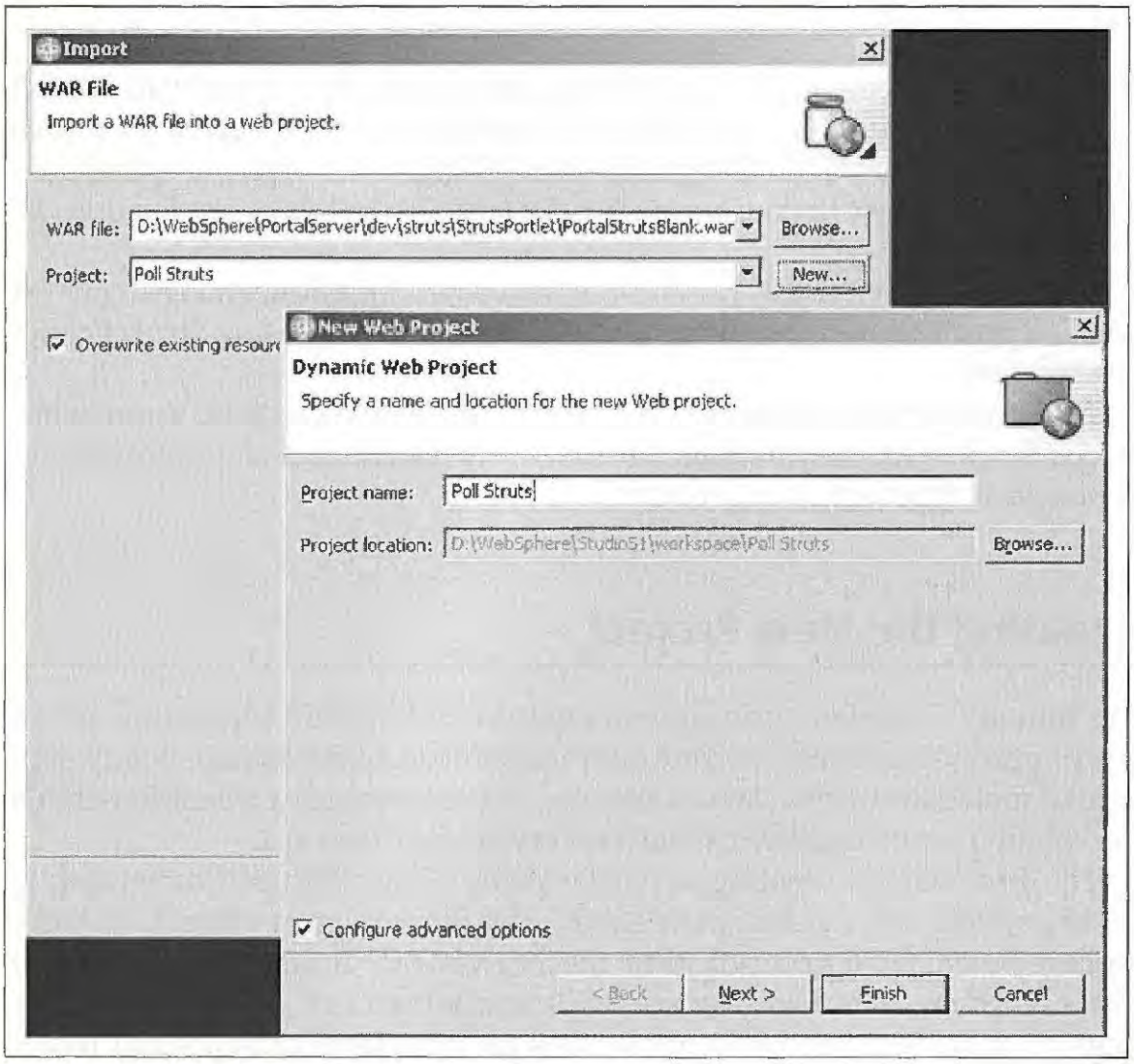


Figure 18-1 Importing the Struts Portlet Framework.

NOTE **After the import completes if you get the following error *Attribute "bundle" must be declared for element type "field,"* in the `validation.xml` file, then remove the attribute from this file.**

2. Add the portal jar files to the project build classpath. To do so, perform the following steps:
   a. Open the properties on the newly created project by right-clicking the project in the Navigator pane.
   b. Select Properties from the pop-up menu.
   c. Select Java Build Path and then the Libraries tab.
   d. Select Add Variable, then select WPS_V5_PLUGINDIR in the New Variable classpath entry, and click Extend. Select `portlet-api.jar` and select OK.
   e. Select Add Variable, then select WPS_V5_PLUGINDIR in the New Variable classpath entry, and click Extend. Select `wps.jar` and select OK.
   f. Select Add Variable, then select WPS_V5_PLUGINDIR in the New Variable classpath entry, and click Extend. Select `wpsportlets.jar` and select OK.
   g. Select Add Variable, then select WAS_50_PLUGINDIR in the New Variable classpath entry, and click Extend. Expand the lib directory, select `dynacache.jar` and select OK.
   h. Select OK to save the property changes.
3. Change the newly created project to J2EE 1.3. To do so, follow these steps:
   a. Open the properties on the newly created project by right-clicking the project in the Navigator pane.
   b. Select Properties from the pop-up menu.
   c. Select Web and then change J2EE Level to 1.3.
   d. Click OK to save the change.
4. Add `portlet.tld` to the tag lib folder in the new project. To do so, follow these steps:
   a. Right-click the `tld` folder in the project.
   b. Select Import and then File System.
   c. On the File System page browse to find the `<wp-root>\shared\app\WEB-INF\tld` directory and select OK. Highlight (do not click the check box of) tld, in the right pane.

d. Click the check box for `portlet.tld`.

e. Finally, click Finish.

5. If you see a warning that states "Web context root should start with slash(/) character for portlet application debugging" open the properties on the newly created project by right-clicking the project in the Navigator pane. Then select Properties from the pop-up menu. Select Web and then add a / to the Context Root.

6. Add the JavaServer Pages Standard Tag Libraries. Add `standard.jar` and `jstl.jar` to the portlet project's lib folder (`WebContent\WEB-INF\lib`). These tag libraries are used to retrieve language-specific text from resource bundles based on the user's locale. The tag libraries are open-source code available from Jakarta at (`http://jakarta.apache.org/`).

Web Deployment Descriptor

A default Web deployment descriptor is provided with the Portal Struts Framework. You need to update it for the Poll portlet. Edit the `web.xml` file and change the `<display-name>` element to a name for this portlet such as *Poll Struts*. The `ID` attribute of the servlet element must be unique in your deployed Web application. Change this value to a unique string such as the Poll Portlet class name, `com.ibm.sample.portlet.poll.portlet.PollPortlet`. Change the `<servlet-class>` element to the portlet class name, `com.ibm.sample.portlet.poll.portlet.PollPortlet`.

Struts Portlet Framework takes advantage of struts module support to provide both mode and device (markup) differentiation for the Struts configuration. When using modules, an application's configuration information and directory structure are separated by module. Therefore, instead of a single `struts-config.xml` file for the application, you have multiple control files within separate directory structure, identified by module-relative portions of the application URI. The search path is used to determine the module and `struts-config.xml` file to use. It also determines the base directory for locating JSPs.

Add the following initialization parameter to set the search path to take markup name and mode into consideration when searching for modules.

```
<init-param>
<param-name>ModuleSearchPath</param-name>
<param-value>markupName, mode</param-value>
</init-param>
```

You will need to create two modules for this application, one for edit mode and one for configure mode. The view mode is configured through the base

struts configuration. The two modules are defined as init parameters in the form: `/config/<device>/<mode>`. The value associated with each of these parameters specifies the name and location of the struts configuration file.

```
<init-param>
  <param-name>config/html/configure</param-name>
  <param-value>/WEB-INF/html/configure/struts-config.xml</param-value>
</init-param>
<init-param>
  <param-name>config/html/edit</param-name>
  <param-value>/WEB-INF/html/edit/struts-config.xml</param-value>
</init-param>
```

Finally, update the welcome file list. The Struts Portlet Framework lets you specify the initial view for each device/mode combination. This is specified through the welcome file list. The default welcome file list only provides a welcome file specification for the view mode.

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>html/edit/index.jsp</welcome-file>
  <welcome-file>html/configure/index.jsp</welcome-file>
</welcome-file-list>
```

Portlet Deployment Descriptor

The portlet deployment descriptor must also be updated. The changes to both descriptors as well as the build path are necessary since we created a Web project during the WAR file import. Previously, we created a portlet application project using the Creation Wizard from the portal toolkit. Part of that process created complete deployment descriptors and set the correct build path.

Edit the `portlet.xml` file and make the following changes.

- Change the `uid` attribute of the `<portlet-app>` element to a unique identifier such as *"com.ibm.sample.portlet.poll.portlet.PollPortlet:1"*
- Change the `<portlet-app-name>` element to a meaningful name such as *Poll (Struts) Application.*
- Change the `href` attribute of the `<portlet>` element to reference the servlet id that you specified in the Web deployment descriptor appended after "`WEB-INF/web.xml`".
- Change the `<portlet-name>` element to a meaningful name such as *Poll (Struts).*

- Add markup elements for configure and edit modes as well as the default view mode.
- Change the `uid` attribute of the `<concrete-portlet-app>` element to a unique identifier such as `com.ibm.sample.portlet.poll.portlet.PollPortlet:1.1`.
- Change both the `<portlet-app-name>` element and the `<portlet-name>` element for the concrete portlet application to meaningful names such as *Poll (Struts) Applications* and *Poll (Struts)*.
- Finally, specify meaningful values for the language-specific attributes for title, short title, description, and key words.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE portlet-app-def PUBLIC "-//IBM//DTD Portlet Application
1.1//EN" "portlet_1.1.dtd">
<portlet-app-def>
  <portlet-app
uid="com.ibm.sample.portlet.poll.portlet.PollPortlet:1"
    major-version="1" minor-version="0">
    <portlet-app-name>Poll (Struts) Application</portlet-app-name>
    <portlet id="Portlet_1"
      href="WEB-
INF/web.xml#com.ibm.sample.portlet.poll.portlet.PollPortlet"
      major-version="1" minor-version="0">
      <portlet-name>Struts Blank</portlet-name>
      <cache>
        <expires>0</expires>
        <shared>NO</shared>
      </cache>
      <allows>
        <maximized/>
        <minimized/>
      </allows>
            <supports>
               <markup name="html">
                 <view />
                 <configure />
                 <edit />
               </markup>
            </supports>
    </portlet>
  </portlet-app>
  <concrete-portlet-app
uid="com.ibm.sample.portlet.poll.portlet.PollPortlet:1.1">
    <portlet-app-name>Poll (Struts) Application</portlet-app-name>
    <concrete-portlet href="#Portlet_1">
      <portlet-name>Poll (Struts)</portlet-name>
      <default-locale>en</default-locale>
```

```
        <language locale="en">
          <title>Poll (Struts)</title>
          <title-short>Poll (Struts)</title-short>
          <description>
             Poll portlet using the struts portlet framework
          </description>
          <keywords>Poll, Struts</keywords>
        </language>
             <config-param>
                <param-name>FilterChain</param-name>
                <param-value>StrutsTranscoding</param-value>
             </config-param>
    </concrete-portlet>
    </concrete-portlet-app>
</portlet-app-def>
```

Struts Configuration Files

The struts configuration file (`struts-config.xml`) contains the application definitions that the struts framework uses for control, navigation, and configuration settings, including form beans that are needed. The module initialization parameters defined in the deployment descriptor specified two module-specific struts configuration files. Along with the base struts configuration file that is located in the `WebContent/WEB-INF` directory, you need to create two new `struts-config.xml` files: one in the `WebContent /WEB-INF/html/configure` directory for configure mode and one in the `WebContent/WEB-INF/html/edit` directory for edit mode.

The struts configuration definitions map fairly well to our existing Poll portlet. Our implementation already has Action classes that provide the application function that is expected by the struts Action classes. The class definition and method signature will change somewhat but for the most part the implementation of these classes remains the same. With the exception of changes needed for portlet URL addressability, we do not need to change the existing JSPs. In general, the View classes were associated with Actions and simply invoked the appropriate JSP. In the struts configuration the JSPs are defined and in cases associated with actions. Our Poll portlet used form data beans for user input in configure mode and the struts implementation defines the form data beans as well.

Struts Configuration in Portlet View Mode

The portlet edit, view, and help modes are mapped to struts modules. The portlet view mode is controlled from the base struts configuration. You will

need to change the `struts-config.xml` file in the `WebContent/WEB-INF` directory to handle the view mode actions.

The view mode configuration requires two actions, one to display the poll and one to register a user's vote. Two JSPs are used by the portlet in view mode, one to render the poll question and answer choices (allowing the user to vote) and one to render the poll results. A poll bean is also used in this configuration.

We have also created a simple JSP to display information when an exception is encountered. The main view action, as well as the exception JSP, is globally defined since they could get invoked from other Actions in the portlet. For example, when you return from edit or configure mode you want to invoke the main view action to render the view page. The following listing shows the struts configuration.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <!-- ========== Form Bean Definitions ========== -->
  <form-beans>
    <form-bean name="pollBean"
      type="com.ibm.sample.portlet.poll.beans.PollBean" />
  </form-beans>
  <!-- ======== Global Forward Definitions ========== -->
  <global-forwards>
    <forward name="mainview" path="/mainview.do" />
    <forward name="exception" path="/exception.jsp" />
  </global-forwards>
  <!-- ======== Action Mapping Definitions ========== -->
  <action-mappings>
    <action path="/mainview"
      type="com.ibm.sample.portlet.poll.actions.ViewAction"
      name="pollBean" scope="request" validate="false">
      <forward name="question" path="/viewQuestion.jsp" />
      <forward name="results" path="/viewResults.jsp" />
    </action>
    <action path="/vote"
      type="com.ibm.sample.portlet.poll.actions.VoteAction"
      name="pollBean" scope="request" validate="false">
      <forward name="results" path="/viewResults.jsp" />
    </action>
  </action-mappings>
  <!--========== Controller Configuration ========== -->
  <controller
  processorClass="com.ibm.wps.portlets.struts.WpsRequestProcessor">
  </controller>
  <!-- ========= Message Resources Definitions ===== -->
  <message-resources parameter="nls.polling" />
```

```
  <!-- ============ Plug Ins Configuration === ==== -->
 <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
     <set-property
     property="pathnames"
     value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml" />
   </plug-in>
 </struts-config>
```

Struts Configuration in Portlet Edit Mode

The edit mode configuration is very simple. The portlet edit mode is a single page that allows the user to select a color scheme for the poll result graph. Therefore, your struts configuration will only need to contain an action to save the user preference and the JSP to render the selection page. There is no action needed to render the selection page since the creation of this page does not need any prior application logic processing or bean creation. An edit data form bean is also defined for the user input form.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <form-beans>
    <form-bean name="editFormData"
type="com.ibm.sample.portlet.poll.beans.EditFormData"/>
  </form-beans>
  <action-mappings>
    <action
      path="/editsave"
    type="com.ibm.sample.portlet.poll.actions.EditSaveAction"
    name="editFormData"
    scope="request"
    validate="false" >
    </action>
  </action-mappings>
  <controller
    processorClass="com.ibm.wps.portlets.struts.WpsRequestProcessor">
  </controller>
  <message-resources parameter="nls.polling"/>
</struts-config>
```

Struts Configuration in Portlet Configure Mode

The struts configuration for the portlet configure mode is a bit more complicated. Actions are needed for the creation of each of the three configuration pages, to verify the datasource, to add an answer choice, to delete an answer

choice, to save the configuration settings, and to cancel the configuration changes. The configuration data form bean is also defined.

The forward definitions for the JSPs for the first and third configuration pages are defined globally since they are used from multiple actions.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <form-beans>
    <form-bean
      name="configureForm"
      type="com.ibm.sample.portlet.poll.beans.ConfigureFormData"/>
  </form-beans>
  <global-forwards>
   <forward name="configure" path="/configure.do" />
   <forward name="configure_pg1" path="/configure_pg1.jsp" />
   <forward name="configure_pg3" path="/configure_pg3.jsp"/>
  </global-forwards>
  <action-mappings>
    <action path="/configure"
      type="com.ibm.sample.portlet.poll.actions.ConfigPg1Action"
      name="configureForm" scope="request"
      validate="false">
    </action>
    <action path="/configure2"
      type="com.ibm.sample.portlet.poll.actions.ConfigPg2Action"
      name="configureForm" scope="request"
      validate="false">
      <forward
        name="configure_pg2"
        path="/configure_pg2.jsp"/>
    </action>
    <action path="/configure3"
      type="com.ibm.sample.portlet.poll.actions.ConfigPg3Action"
      name="configureForm" scope="request"
      validate="false" >
    </action>
    <action path="/configure_verify"
type="com.ibm.sample.portlet.poll.actions.ConfigVerifyDatasourceAction"
      name="configureForm" scope="request"
      validate="false" >
    </action>
    <action path="/configure_cancel"
      type="com.ibm.sample.portlet.poll.actions.CancelAction"
      name="configureForm" scope="request"
      validate="false" >
    </action>
    <action path="/configure_save"
```

```
      type="com.ibm.sample.portlet.poll.actions.ConfigSaveAction"
      name="configureForm" scope="request"
      validate="false" >
    </action>
    <action path="/configure_delete"
      type="com.ibm.sample.portlet.poll.actions
      .ConfigDeleteChoiceAction"
       name="configureForm" scope="request"
       validate="false" >
    </action>
    <action path="/configure_add"
      type="com.ibm.sample.portlet.poll.actions.ConfigAddChoiceAction"
      name="configureForm" scope="request"
      validate="false" >
  </action>
  </action-mappings>
  <controller
    processorClass="com.ibm.wps.portlets.struts.WpsRequestProcessor">
  </controller>
  <message-resources parameter="nls.polling"/>
</struts-config>
```

Action Classes

Your struts action classes will extend `org.apache.struts.action.Action` and override the execute method. With a struts portlet implementation you no longer implement a central controller class such as the PollPortlet class from the previous implementation. This responsibility is managed by the struts request processor, which we have configured as WpsRequestProcessor.

You will define an abstract action class to include common functions that should occur prior to any action getting invoked. This is the function you had in the PollPortlet's actionPerformed or service methods.

The `AbstractAction` class then implements the *execute* method and performs any needed processing before dispatching to the subclass. The listing below shows the `AbstractAction` class for our struts portlet. Notice that the *execute* method is passed `HttpServletRequest` and `HttpServletResponse` objects. In order to have access to the portlet API, you need to get the `PortletRequest` and `PortletResponse` objects from these.

```
public abstract class AbstractAction extends Action
    implements PollConstants {
  public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
```

```
        try {
          // Check for existing session with logged-in user.
          PortletRequest portletRequest = (PortletRequest) request;
          PortletSession session = portletRequest.getPortletSession(false);
          User user = portletRequest.getUser();
          if (session == null || user == null) {
            ResourceBundle bundle = ResourceBundle.getBundle(BUNDLE,
               portletRequest.getLocale());
            String msg = bundle.getString("exception.login");
            throw new PollMessageException(msg);
          }

          // Invoke the action subclass performAction method
          return performAction(mapping, form, user, request, response);
        }

        catch (Exception e) {
          PortletRequest portletRequest = (PortletRequest) request;
          PortletSession session = portletRequest.getPortletSession();
          session.setAttribute(POLL_EXCEPTION,e);
          return (mapping.findForward("exception"));
        }
      }

      public abstract ActionForward performAction(ActionMapping mapping,
        ActionForm form, User user, HttpServletRequest request,
        HttpServletResponse response) throws Exception;

    }
```

Then the Action classes only need to implement the `performAction` method with the parameters shown previously. Copy the source files from the IBM Portlet API Poll portlet implementation to the Poll Struts project to make the appropriate changes.

First you should modify the abstract action class as shown. Then update the method signature of the Action classes and delete the `setView` methods. You will also need to return an `ActionForward` instance from the `performView` method indicating another action to call or a JSP to render using the definitions in the struts configuration.

For example, the following listing shows the action to generate the first page in the configuration mode. Notice that the session data bean is created and the JSP to render the page is identified through a mapping in the configuration file.

```
public class ConfigPg1Action extends AbstractAction {

  public ActionForward performAction(ActionMapping mapping,
      ActionForm form, User user, HttpServletRequest request,
```

```
      HttpServletResponse response) throws Exception {
    // Get the session data
    PortletRequest portletRequest = (PortletRequest) request;
    SessionData sessionData =
      SessionData.getSessionData(portletRequest);

    if (sessionData == null) {
      // Create a new sessionData instance and add it to session
      sessionData = SessionData.createSessionData(portletRequest);
      // Initialize form data from portlet settings
      ConfigureFormData formData = sessionData.getConfigureFormData();
      PortletSettings portletSettings =
        PortletRequest.getPortletSettings();
      String persistType = portletSettings.getAttribute(PERSIST_TYPE);
      String datasource = portletSettings.getAttribute(DATASOURCE);
      formData.setDatasource(datasource);
      formData.setPersistType(persistType);
    }
    return (mapping.findForward("configure_pg1"));
  }
}
```

The other action classes should be modified in a similar way. However, we will not discuss them in detail here. Note also that there are two additional action classes in the struts implementation, one just shown here to generate the first configuration page and one to generate the view page. The logic for this had been in the view classes in the previous implementation but with the Struts Portlet Framework the rendering phase is simply a JSP invocation without additional application logic.

Remaining Poll Portlet Implementation

Now that the struts configuration files are created and the action classes have been updated, much of the work of getting your Poll portlet to work in the Struts Portlet Framework is handled. Differences do exist in portlet URL addressability that will be examined in more detail shortly, but first you need to move the remaining code from your initial implementation into the Poll struts project and see what further changes are needed.

Data Beans

The only change for the data beans is to extend `ActionForm`. A new form data bean has also been created for edit mode that just contains the color field. This keeps the struts action processing consistent.

Persistence Classes

There are no changes needed for the persistence classes.

Portlet Controller Classes

Portlet Controller classes include the `PollPortlet` controller class and the helper classes for managing Action classes and View classes (`ActionClassManager` and `ViewClassManager`). These classes are not needed, since the controller function is handled by the request processor and therefore can be deleted. The `PollConstants` class that holds application constants will continue to be used.

Utilities Classes

There are no changes needed for the Utilities classes.

View Classes

View classes are not needed since the rendering phase is a direct invocation of a JSP. The classes should be deleted.

JSP Differences

Recall the changes to the Web deployment descriptor that you made. The first is the change to the welcome file list and the second is the initialization parameter that determines the search path for the module resources.

The following listing shows the welcome file list. This list is used to specify the initial JSP that is invoked. The list specifies JSP files by device and by portlet mode. The default welcome file is `index.jsp` and is located in the WebContent folder. The initial file for edit mode using HTML markup is also an index.jsp file but that is located in the `WebContent\html\edit` folder. Likewise for configure mode in HTML the `index.jsp` file is located in the `WebContent\hntl\configure` folder.

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
    <welcome-file>html/edit/index.jsp</welcome-file>
    <welcome-file>html/configure/index.jsp</welcome-file>
</welcome-file-list>
```

You will need to create these three files in the appropriate location. The initial page in configure mode and the page in view mode both require

action processing prior to rendering. Since we can't add an action in the welcome file list you need to create an `index.jsp` that does a forward such as (for configure mode)

```
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<logic:forward name="configure"/>
```

The edit mode page is a bit different in that it does not need additional processing. There is mostly static information that can be rendered from the JSP without requiring prepared beans or other actions. In this case `index.jsp` will contain that JSP code that we had in the `edit.jsp` prior.

Also recall the init parameter for the module resources search path. It was:

```
<init-param>
  <param-name>ModuleSearchPath</param-name>
  <param-value>markupName, mode</param-value>
</init-param>
```

As a result of this, you need to move the view JSPs (`viewQuestion.jsp` and `viewResults.jsp`) to the WebContent folder. Move the remaining JSPs associated with configure mode to the `WebContent\html\configure` folder.

Portlet URL Addressability

One significant difference in developing struts applications for deployment into the portal environment lies in the structure of portlet URIs. Struts action mappings as well as page object references such as JSPs are defined in terms of paths. However, these struts path references must be made in context of a portlet. Portlets have a specific and expected URI format. Therefore, the additional path information needed by the Struts application must be added to the portlet URI to ensure that the portlet gets properly invoked and that the struts path is also available. The struts tags have been modified in the Struts Portlet Framework to provide this function.

The struts tags have also been extended to support specific portal requirements. For example, consider the JSP that displays the last page in configuration mode. The action on the HTML form tag should be a reference to the action class to save the configuration settings. However, the behavior you want in a portal environment is to save the configuration settings *and* return to view mode. So, you need a form tag to do both, invoke the action class to save the configuration settings and also change modes to the previous mode. Since modes are unique to the portal environment, these standard struts tags do not provide that level of support.

The tag implementation in the Struts Portlet Framework supports this behavior and for this example our tag would be the following.

```
<html:form method="post" action="/configure_save" urlType="return">
```

The `urlType` attribute is introduced. This attribute can take either of the values *"return"* or *"standard."* The form tag generated using the above custom tag uses the portlet URI that handles the return and appends the struts path to reference the *configure'save* action.

Generating Portlet URIs

In this portlet, like many others, there is a need to generate URIs in Java code so the URI can be dynamically associated with a form submit. The following listing shows how a URI string can be created in a JSP scriplet. This URI string is used in the JSP that renders the final page in the portlet configuration mode. This portlet URI invokes the action to add an answer choice to the poll.

```
<%@ page import="com.ibm.wps.struts.common.PortletApiUtils" %>
<%
  PortletApiUtils portletUtils = PortletApiUtils.getInstance();
    if (portletUtils != null) {
      String url = "/configure_add.do";
      url = portletUtils.addModulePrefix(url, request);
      Object portletURI =
        portletUtils.createPortletURIWithStrutsURL(request,
        url);
      String addChoiceURI = portletURI.toString();
    }
%>
```

You also need to generate the URI as a string outside a form tag definition but with a portlet mode return action. The next example shows similar code to generate a portlet URI that returns to the previous mode and also invokes a struts action:

```
<%@ page import="com.ibm.wps.struts.common.PortletApiUtils" %>
<%@ page import="com.ibm.wps.struts.common.PortletURIAttributes" %>
<%
  PortletApiUtils portletUtils = PortletApiUtils.getInstance();
    if (portletUtils != null) {
      String url = "/configure_cancel.do";
      PortletURIAttributes uriAttributes = new PortletURIAttributes();
      uriAttributes.setUriType("return");
      url = portletUtils.addModulePrefix(url, request);
      Object portletURI =
        portletUtils.createPortletURIWithStrutsURL(request,
```

```
            url, uriAttributes);
        String cancelURI = portletURI.toString();
    }
%>
```

Which Implementation Should You Use?

The strategic direction for IBM support in portlet API is clearly JSR 168. However, the IBM portlet API will continue to be supported in future versions of WebSphere Portal. There is no immediate need to migrate existing portlets to the JSR 168 API unless this portlet is required to interoperate with other portals that support JSR 168.

For new portlet development the JSR 168 API should be a first consideration when the functionality it provides is sufficient for the portlets' needs or when the portlet is expected to be published as Web Service for Remote Portlets service. If the portlet needs more functionality than that provided by JSR 168, the IBM portlet API should be used. As of Portal 5.02 the JSR 168 support did not extend to portal services such as ContentAccessService and the CredentialVault service.

Struts is the MVC framework of choice for use in portlet development. As of WebSphere Portal 5.02 Struts Portlet Framework did not yet support JSR 168. As the JSR 168 API implementation progresses to meet all the functional capabilities of the IBM Portlet API and Struts Portlet Framework is extended to support the industry standard portlet API, the clear preference in portlet implementation is the MVC framework of struts. As technologies continue to develop Java Server Faces (JSF) plays a more active role as a key development framework. Integration work is underway to allow struts actions, ActionForms and struts configuration within the JSF environment.

As these technologies continue to develop, we might see support in Struts Portlet Framework in the JSF environment. A new request processor would be needed that could receive JSF events and send them to a struts action within Struts Portlet Framework. However, the direction of struts and JSF and the implementation support through Struts Portlet Framework was not known at the time of writing this chapter.

Summary

This chapter examined the implementation of the example portlet using Struts Portlet Framework. As you did for the JSR 168 API portlet implementation, you took the Poll portlet developed using the IBM Portlet API and converted it to use Struts in the portal environment.

PART

IV

WebSphere Portal within the Enterprise Environment

CHAPTER 19

Implementing Authentication for Large Enterprises

Authentication is the process where users are challenged to identify themselves to gain access to a system. The challenge can be immediate or upon trying to access a protected resource. It is usually encountered by the user as requesting for a user id and password but it can also be authenticating through a biometric device or a digital certificate. Information required for authentication is stored in user registries while information relating to the user's profile and preferences is stored in the user repository.

If you have a few users, WebSphere Portal authentication is pretty simple and easy to configure. However, if yours is a large enterprise with many users accessing the site from both your global intranet and extranet, then this subject gets a lot more complex. In this chapter we are going to discuss issues relating to enterprise identity management for WebSphere Portal.

Enterprise Identity Management

WebSphere Portal performs identity management though Member Manager. When a user authenticates, Member Manager validates the information in both the authentication registry and the member database.

WebSphere Portal supports four types of identity management configurations, as seen in Figure 19-1. The first is where the user registry and profile information for WebSphere Portal and WebSphere Application Server is stored on the same LDAP server, which is usually remote. Member Manager maintains and updates all user information in the LDAP server. The second configuration is where Member Manager uses a look-aside repository (that is, a database) to store user profile attributes not supported in the LDAP.

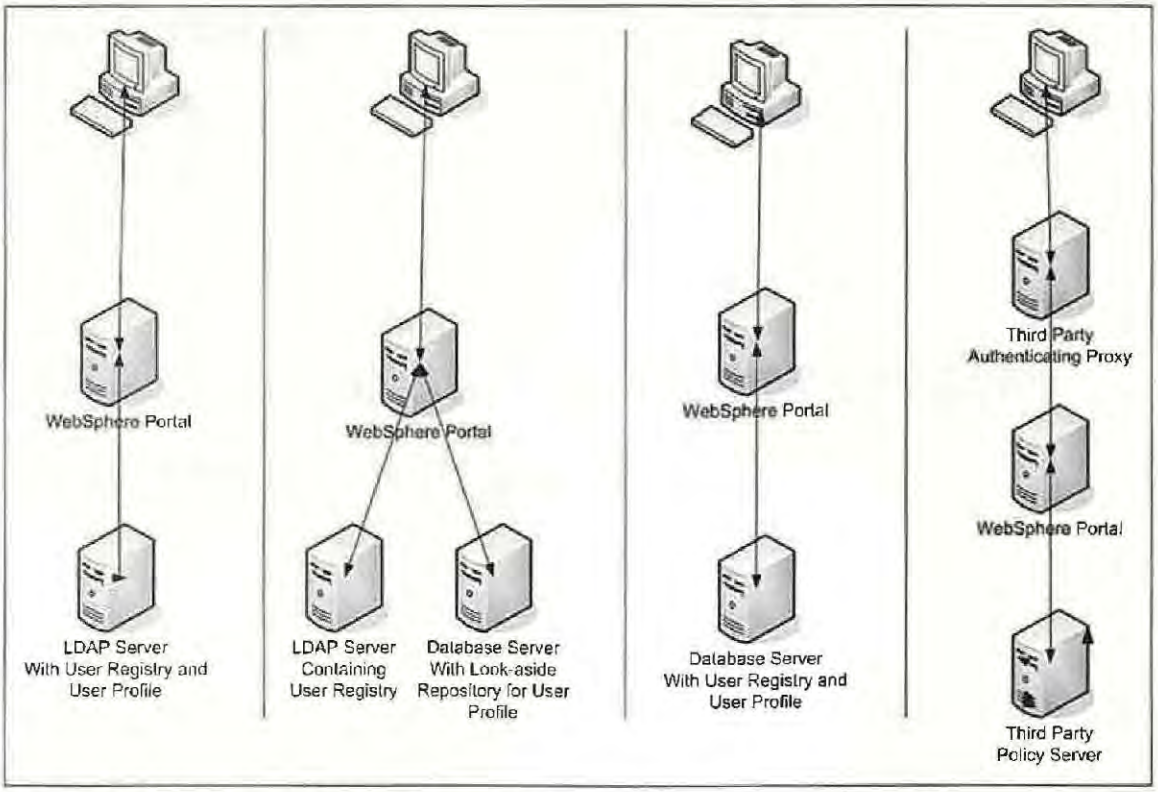


Figure 19-1 Different LDAP configurations.

The split is based on Member Manager configuration XML files. The third identity management configuration is where Member Manager and WebSphere Application Server create and maintain the user identity and profile information in the same database For the third identity management configuration, WebSphere Application Server poses the actual challenge using the data from WebSphere Portal Custom User Registry. For all the configurations, the user registry is shared by both WebSphere Portal and WebSphere Application Server.

The last identity management configuration is using third-party authentication using Trust Association Interceptors. Chapter 20 goes into great detail on this configuration.

LDAP or Database

So the first question you ask yourself when you design the identity management infrastructure is whether to use an LDAP server or an enterprise database. This, of course, leads to the next question: what is LDAP?

LDAP or Lightweight Directory Access Protocol is an open industry standard that defines a standard method for accessing and updating information in a directory. A directory is a database variety that stores typed and ordered information about objects. An LDAP directory is designed for read performance, which means it assumes that the user data will be read far more than it will be changed.

If your company already stores its user registries on an LDAP, then the answer is a "no-brainer." But if it doesn't, then you have to consider the following factors:

1. Does your enterprise have any skill set in configuring and running an LDAP server in production? While some people say that LDAP is simple to install and maintain, this is not true for large directories. It experiences the same operational issues as any database and you need to have the knowledge within your IT area to support it. If they have no experience with an LDAP server but have expertise in DB2 or Oracle, then go with the database.
2. How often is the user data being updated or created versus being read. If the ratio of read to updates for the user data is high, then an LDAP server is well suited. Usually the profile and authentication information such as user ID, password, name, title, address, e-mail, and demographical information does not change that often and is well suited for an LDAP while privilege information such as roles and access control groups can change more frequently and is better suited for a database.
3. Is the management of the users decentralized? LDAP servers usually have a more efficient replication process than databases.
4. Does any of the data have transactional dependencies? That is, if the name and address are being updated, they must both successfully update and if not, they should rollback to their original values. Most LDAP servers do not support transactionality.
5. Do you need to perform sophisticated queries against the directory? LDAP directories use a simple access protocol that cannot match the power of an enterprise database's Structured Query Language for complex update and query functions.

Rules of Thumb for Designing and Maintaining Your LDAP Server

After answering the above questions, you have probably decided that for a large enterprise it is better to go with an LDAP server. So the next steps are as follows:

- Determine Directory Tree by designing the schema and arranging the entries in the tree structure.
- Define the replication and partitioning strategies.
- Create an infrastructure plan.

Designing the Information Model

One of the biggest factors affecting the performance and maintainability of the LDAP server (and thus the Identity Management component of WebSphere Portal) is implementation of the information model correctly.

The information model describes how information is stored in an LDAP directory. Information is represented in an LDAP directory by an entry. An entry is equivalent to an object such as employees, servers, and so on. Entries consist of a collection of attributes that contain information about the object. Each attribute has a type and one or more values associated with it. Associated with a type is a syntax that specifies what values can be stored and how a value behaves during searches and other operations. Attributes can also have an alias name that can be used instead of the attribute name. For instance, the common LDAP attribute surname has an alias sn and an LDAP syntax cis associated with it. cis stands for case ignore string, which means case is not significant during comparisons. An attribute can also have a constraint associated with it to limit the number of values or the total size of the value.

The LDAP Schema defines the object classes and attribute types. Object class describes the types of objects each entry can represent and attribute type defines the attributes of each object type, and whether these attributes are optional or mandatory. In order to facilitate interoperability, the schemas have been standardized by Internet Society. Two standards that were adopted by manufacturers are RFC 2252, LDAP (V3); Attribute Syntax Definitions and RFC 2256, A summary of X.500(96) User Schema for use with LDAP V3. For instance in the RFC 2256 standard schema, sn is X.500 surname attribute, which contains the family name of a person. An example of an objectclass is person, which must have the objectclass value on top and either the attribute type surname or commonName and may have the attribute types `userPassword`, `telephoneNumber`, `seeAlso`, and description.

Most of the LDAP servers will have default schema and directory tree layout for the user registries. You should basically not change the structure or elements since this will obviously cause interoperability problems between other directory services and LDAP clients. Definitely do not delete standard schema elements. If you need to add an object class then extend the default schema. Objects added to the LDAP directory should be accessed by more than one user or client. If they are not, then they should be placed in a database.

You need to define who owns the identity management data and who is responsible for keeping it up-to-date. An operations manual should be created that identifies this information. Programs may also need to be created to allow the help desk to administer the data. Insure that there is no

inconsistency in the schema; that is, two different attributes with the same data.

If the identity management data is being imported from other sources, develop a strategy for bulk imports and incremental updates. Insure that the data is being imported from the main identity management database and not from another mirror site.

Designing the Directory Information Tree

Data is stored hierarchically in a directory information tree (DIT) over one or more LDAP server(s). The top level of the LDAP directory tree is called the base distinguished name (DN) or a suffix. Each directory record has a unique DN and is read backward through the tree from the individual entry to the top level. The DN is used as a key to the directory record.

For example in Figure 19-2, Ken Chow's entry would be accessed using cn=Ken Chow, ou=Marketing,dc=rigorconsultants,dc=com.

If the DIT is split over multiple servers, then referrals are used to link the servers. This is done by creating an entry of `objectClass` referral with an attribute `ref` and assigning the value an LDAP URL, which points to the entry on another LDAP server.

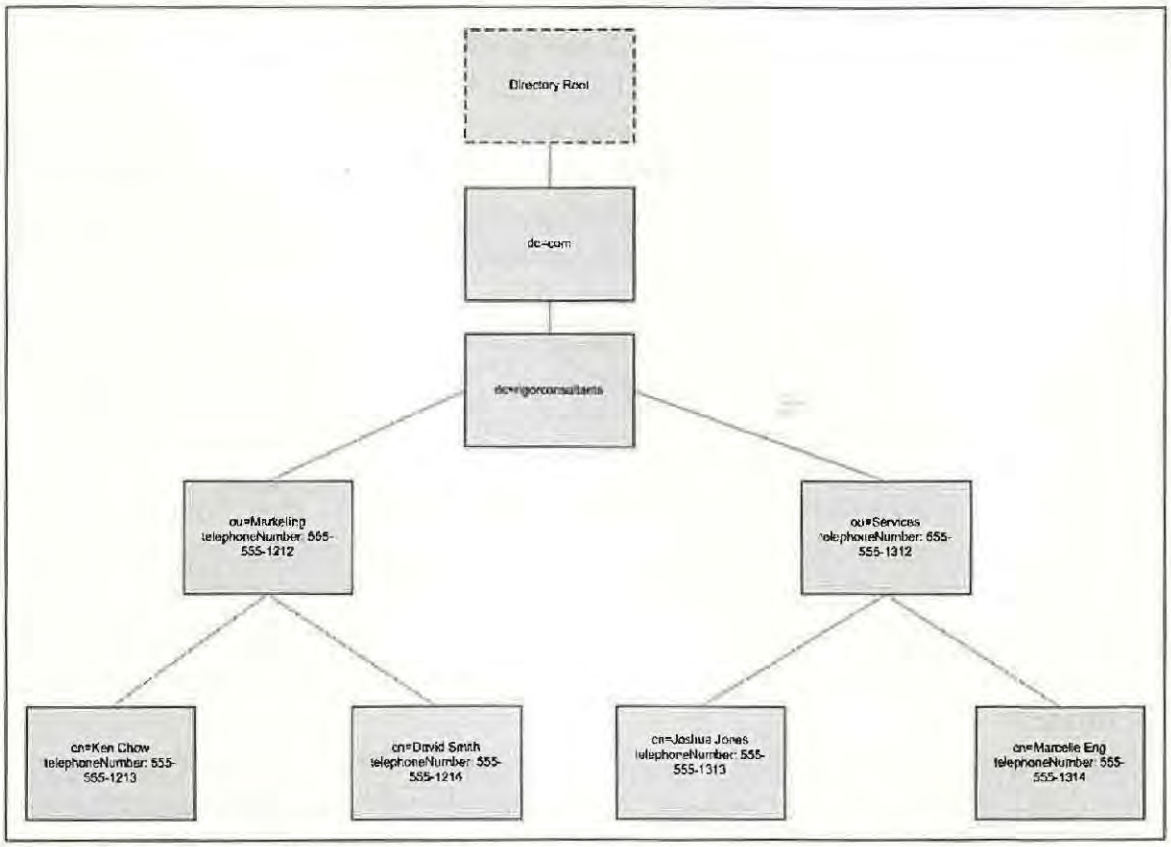


Figure 19-2 A sample directory naming structure.

Given that the LDAP directory is hierarchical, changing the tree is very difficult and should be avoided. Choosing the appropriate naming structure is very important and is usually dictated by the LDAP server. Most LDAP servers base their naming models on either the X.500 methodology or the DNS naming model. The X.500 methodology sets the root of the directory to an organization and has a suffix like o=rigorconsultants, c=us. The DNS model uses the domain name as the suffix like dc=rigorconsultants.com. Microsoft's Active Directory uses the DNS naming model.

When designing the DIT, you have to pay special attention to the branching methodology. If the root branch is based on departments, then a reorg can cause an administrative nightmare. Branching based on geography can restrict sharing organizational information. The branching methodology should be designed to minimize changes to the directory tree. Also make the tree as shallow as possible to maximize search performance. Lastly, consult your security administrator to determine if there are parts of the tree that compliance or organizational rules dictate and that must be physically separated.

Designing the Infrastructure

Identity Management systems for large enterprise must be highly available and able to handle large number of users. LDAP servers support these requirements using partitioning and replication.

Figure 19-3 shows a highly available LDAP server solution. In this solution we have one master server and three slave servers, each with dual network cards. Each server is connected to two separate networks and the connections are bridged together. Read queries are sent to any of the servers while updates, create, and deletes are performed against the master, which replicates the data to the slave servers. If the server fails, client redirection can be done by DNS switchover or using load balancing through the router or the LDAP server. This design eliminates any single point of failure and since writes are infrequent on an LDAP configuration, performance is not impacted.

Scalability issues can be addressed partitioning the directory by geography or some other methodology that accommodated the differences in various network bandwidths and server performances. This also spreads the risk by minimizing the area impacted by a server failure. You can also replicate the LDAP servers to servers closer to the client's geographical location.

Remember that the more complex you make the LDAP infrastructure, the more difficult it is to manage. Try to keep it simple and minimize the input data sources.

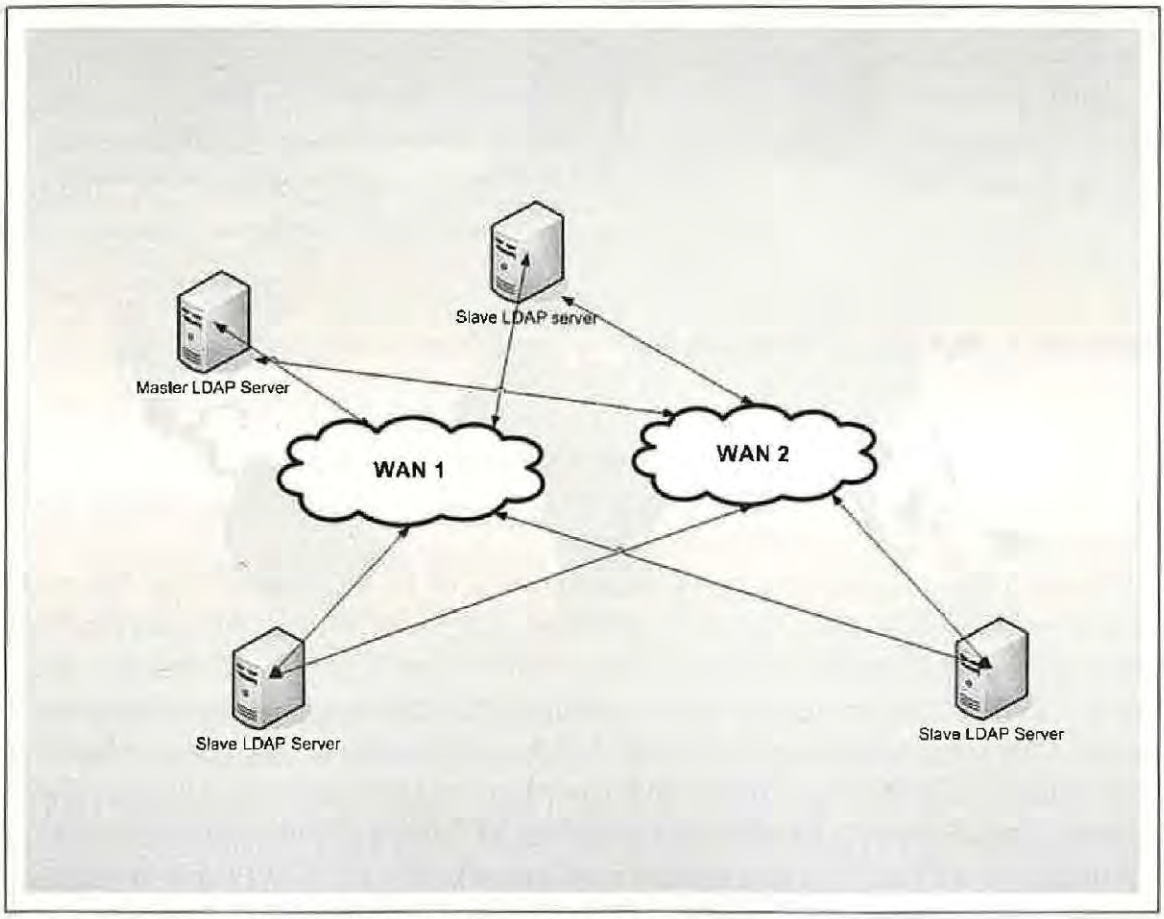


Figure 19-3 A highly available LDAP server configuration.

Implementing WebSphere Portal Enterprise Identity Management

In the previous section we explored the reasons for using an LDAP server for Enterprise Identity Management and discussed the general design issues. Now it is time to leave the theory and deal with the practical aspect. In this section we will explore how to implement Identity Management in WebSphere Portal using the more popular LDAPs:

1. IBM Directory Server V5.1
2. Sun One
3. Microsoft Active Directory

What is conspicuously missing is Domino 6.0. Domino is required if you want to implement collaboration. However, Domino is a lot more than an LDAP server and introducing it in this discussion will cause many

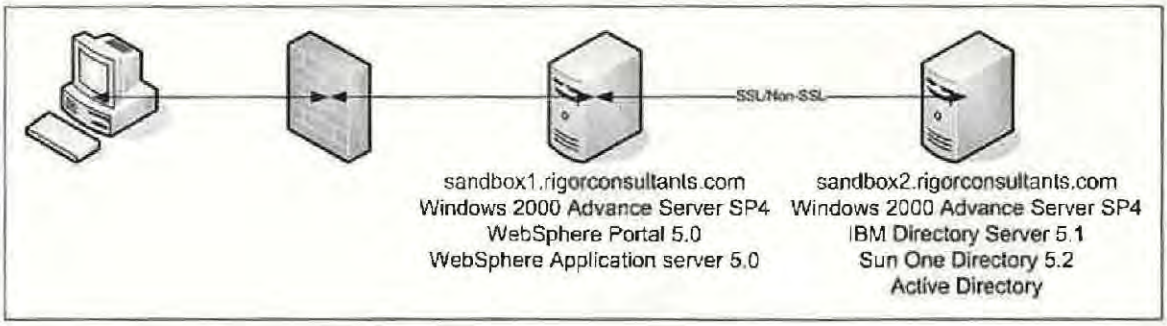


Figure 19-4 WebSphere Portal with LDAP server example configuration.

deviations that will cause us to lose focus on the topic. Instead, you will get more value to read the sections on Domino in WebSphere Portal InfoCenter after you have read this chapter.

Figure 19-4 shows the configuration we will be implementing for each LDAP server. The user registry is stored on a remote LDAP server. We assume that you will not use a look-aside repository because it makes the infrastructure unnecessarily complex and tends to have performance issues. We are assuming that each LDAP in question has been installed and tested on both the server and the client where the client is the WebSphere Portal server. Due to the amount of information that needs to be covered, we will only be discussing implementation of LDAPs in a Windows 2000 environment. The configuration differences between the Windows 2000 environment and Unix/Linux are minor and are documented in the InfoCenter.

Setting Up Your LDAP Servers

The first step to getting WebSphere Portal working with your LDAP server is to create the Portal Administrator User ID and WebSphere Administrator User ID.

The first directory server we will discuss is IBM Directory Server or Tivoli Directory Server, which comes with WebSphere Portal Extend or Experience. It is built on top of DB2 (tuned for read applications) and WebSphere Application Server. The directory is full function and robust but cannot be considered "light."

There are two methods to create the user ids. The first (recommended) is to modify the LDIF file `portalUsers.ldif` on the first WP CD to conform to your tree structure. LDIF or LDAP Data Interchange Format is a standard format that enables data to be imported and exported between LDAP-based directory servers. The second is to add it using LDAP Directory Web Administrator. If you are going to create the portal administrator user in the LDAP, use the Add Entry function under Directory Management instead of

defining it under a Realm using a template. You might find the templates too restrictive especially when you want to associate multiple object classes with the entry.

To create the administrators through an LDIF import, you need to modify the file with your suffix and conform it to your directory tree structure and the naming structure. For IBM Directory we used:

- LDAP suffix is set to dc=rigorconsultants,dc=com
- user prefix is uid
- group prefix is cn=users
- group suffix is cn=groups
- Portal administrator DN is uid=wpsadmin, cn=users, dc=rigorconsultants, dc=com
- Portal administrator group is cn=wpsadmins, cn=groups, dc=rigorconsultants, dc=com of which wpsadmin is a member
- WebSphere Security ID DN is uid=wpsbind, cn=users, dc=rigorconsultants, dc=com

This resulted in the following portalUsers.ldif:

```
version: 1

dn: dc=rigorconsultants,dc=com
objectclass: domain
objectclass: top
# Add lines according to this scheme that correspond to your suffix
dc: rigorconsultants,dc=com
dc: rigorconsultants

dn: cn=users,dc=rigorconsultants,dc=com
objectclass: container
objectclass: top
cn: users

dn: cn=groups,dc=rigorconsultants,dc=com
objectclass: top
objectclass: container
cn: groups

dn: uid=wpsadmin,cn=users,dc=rigorconsultants,dc=com
objectclass: organizationalPerson
objectclass: person
objectclass: top
objectclass: inetOrgPerson
uid: wpsadmin
userpassword: wpsadmin
```

```
sn: admin
givenName: wps
cn: wps admin

dn: uid=wpsbind,cn=users,dc=rigorconsultants,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: wpsbind
userpassword: wpsbind
sn: bind
givenName: wps
cn: wps bind

dn: cn=wpsadmins,cn=groups,dc=rigorconsultants,dc=com
objectclass: groupOfUniqueNames
objectclass: top
uniquemember: uid=wpsadmin,cn=users,dc=rigorconsultants,dc=com
cn: wpsadmins
```

To import into IBM Directory Server, stop the directory server

1. Click Start ⇨ Programs ⇨ IBM Directory Server 5.1 ⇨ Directory Configuration.
2. Click Import LDAP, enter the location of the file to import, and click OK. A message will appear that states that you have successfully imported six out of six items or 6 of 6 if your suffix is already defined.
3. Exit and restart your LDAP server.

The next directory server we will look at is Sun One Directory Server formally known as Netscape Directory Server. This is one of the oldest and most popular LDAP directory servers. For Sun One Directory Server 5.2, the process is pretty much similar. The only difference lies in the default naming structure

- group prefix is ou=People
- group suffix is ou=Groups
- Portal administrator DN is uid=wpsadmin, ou=People, dc=rigorconsultants, dc=com
- Portal administrator group is cn=wpsadmins, ou=Groups, dc=rigorconsultants, dc=com of which wpsadmin is a member
- WebSphere Security ID DN is uid=wpsbind, ou=Groups, dc=rigorconsultants, dc=com

Do the following to import into Sun One Directory Server:

1. Start the Sun One Server Console.
2. Under Servers and Application, expand the tree and double-click Directory Server.
3. Go to the Task tab and click Import LDIF.
4. Type **LDIF** file (or browse to it) and click OK.
5. After the data has been imported, go to the Directory tab and validate your directory tree.

The last directory server is the most popular directory server based on market penetration, Microsoft Active Directory. This LDAP server is designed to work very well within a Microsoft environment; however, due to some its propriety features and lack of support for many objectclasses it does not integrate easily to other environments.

To use Active Directory with WebSphere Portal, you must have the Windows 2000 High Encryption Pack, Internet Information Services, and Certificate Services using the standalone root CA option installed. Active Directory requires Secure Sockets Layer (SSL) enabled in order for WebSphere Portal to use it as a user registry (more on this later).

The naming structure that we used with Active Directory is as follows:

- LDAP suffix is set to dc=rigorconsultants,dc=com
- user prefix is cn
- group prefix is cn=users
- group suffix is cn=groups
- Portal administrator DN is cn=wpsadmin, cn=users, dc=rigorconsultants, dc=com
- Portal administrator group is cn=wpsadmins, cn=groups, dc=rigorconsultants, dc=com of which wpsadmin is a member
- WebSphere Security ID DN is cn=wpsbind, cn=users, dc=rigorconsultants, dc=com

Normally you would import the data as an LDIF file using LDIFDE utility. However, the changes to the LDIF are so great that it is easier to add the wpsadmin and wpsbind users and wpsadmins group using the Windows Active Directory Users and Groups administrative tool.

Configuring WebSphere Portal for Your LDAP Server

Now that your LDAP server is all set up, you need to configure WebSphere Portal server to work with it by using the following steps:

1. Ensure that WebSphere security is disabled. If it is not, do the following:
   a. Copy the template security_disable.properties file from <wps_root>\config\helpers to <wps_root>\config.
   b. Modify the properties based on your values.
   c. Open a command prompt and change the directory to <wps_root>\config.
   d. Enter.

```
WPSconfig -DparentProperties "<wps_ root>\config\security_disable" -
DSaveParentProperties=true
```

   e. Upon completion enter **WPSconfig disable-security.**
2. In <wps_root>\config\helpers, find the appropriate template and copy it to <wps_root>\config. For IBM Directory Server, use security_ibm_dir_server.properties, Sun One Directory; use security_ibm_dir_server.properties, and Microsoft Active Directory; and security_active_directory .properties.
3. In the template, assign the properties values based on Tables 19-1 to 19-3. The values assigned in Table 19-1 are those we used to access each LDAP.
4. Save the file as ldap.properties.
5. Ensure that WebSphere Application Server is running but WebSphere Portal has stopped functioning.

Table 19-1 WebSphere Application Server Properties

		PROPERTY VALUE		
PROPERTY	DESCRIPTION	IBM DIRECTORY SERVER 5.1	SUN ONE 5.2	WINDOWS 2000 ACTIVE DIRECTORY
WasUserid	The fully qualified name of the WebSphere Application Server security ID with no spaces	uid=wpsbind, cn=users, dc=rigorcon-sultants, dc=com	uid=wpsbind, ou=people, dc=rigorcon-sultants, dc=com	cn=wpsbind, cn=users, dc=rigor-consultants, dc=com
WasPassword	The password for WebSphere Application Server security ID			

Table 19-2 WebSphere Portal Properties Configuration

PROPERTY	DESCRIPTION	PROPERTY VALUE		
		IBM DIRECTORY SERVER 5.1	SUN ONE 5.2	WINDOWS 2000 ACTIVE DIRECTORY
PortalAdminId	The fully qualified name of the WebSphere Portal Administrators with no spaces	uid=wpsadmin, cn=users, dc=rigorconsultants, dc=com	uid=wpsadmin, ou=people, dc=rigorconsultants, dc=com	cn=wpsadmin, cn=users, dc=rigorconsultants, dc=com
PortalAdminShort	Short form of the WebSphere Portal Administrator ID	wpsadmin	wpsadmin	wpsadmin
PortalAdminGroupId	The fully qualified group ID from the group to which the WebSphere Portal Administrator belongs	uid=wpsadmins, cn=groups, dc=rigorconsultants, dc=com	uid=wpsadmins, ou=groups, dc=rigorconsultants, dc=com	cn=wpsadmins, cn=users, dc=rigorconsultants, dc=com
PortalAdminGroupIDShort	The short form of the WebSphere Portal Administrator group ID	wpsadmins	wpsadmins	wpsadmins

Table 19-3 LDAP Properties Configuration

PROPERTY	DESCRIPTION	PROPERTY VALUE		
		IBM DIRECTORY SERVER 5.1	SUN ONE 5.2	WINDOWS 2000 ACTIVE DIRECTORY
LookAside	States whether you are using a look-aside database, which stores attributes that cannot be stored in your LDAP server	false	false	false
LDAPHostName	LDAP server host Name	sandbox2.rigorconsultants.com	sandbox2.rigorconsultants.com	sandbox2.rigorconsultants.com
LDAPPort	The port number that the LDAP server uses	389	389	636
LDAPAdminUid	The LDAP administrator Id	cn=ldsadmin	cn=ldsadmin	cn=ldsadmin
LDAPAdminPwd	The LDAP administrator Id password			
LDAPServerType	Type of LDAP server	IBM_DIRECTORY_SERVER	IPLANET	ACTIVE_DIRECTORY

LDAPBindId	The user Id for LDAP Bind authentication	uid=wpsbind, cn=users, dc=rigorconsultants, dc=com	uid=wpsbind, ou=people, dc=rigorconsultants, dc=com	cn=wpsbind, cn=users, dc=rigorconsultants, dc=com
LDAPSuffix	The LDAP suffix	dc=rigorconsultants, dc=com	dc=rigorconsultants, dc=com	dc=rigorconsultants, dc=com
LdapUserPrefix	User prefix	uid	uid	cn
LDAUserSuffix	User suffix	cn=users	ou=People	cn=users
LDAPGroupPrefix	Group prefix	cn	cn	cn
LDAPGroupSuff Suffix	Group suffix	cn=groups	ou=groups	cn=users
LDAPUserObjectClass	User object class	inOrgPerson	inOrgPerson	user
LDAPGroupObjectClass	Group object class	groupOfUniqueNames	groupOfUniqueNames	group
LDAPGroupMember	The attribute name of the membership attribute of your group objectclass	uniqueMember	uniqueMember	member
LDAPsslEnabled	Is LDAP SSL enabled?	false	false	true

6. In the command prompt (you should still be in the directory `<wps_root>\config`), enter **WPSconfig - DparentProperties= "<wps root>\config \ldap.properties" - DSaveParentProperties=true.**
7. Test your configuration settings by entering **WPSconfig validate-ldap**. If the test is unsuccessful, recheck your configuration settings.
8. If it is successful, run WPSconfig secure-portal-ldap. If you are using Active Directory or you have enabled SSL, there are a few more steps that will be discussed in the next section. Do not try to implement an SSL configuration (unless you are implementing Active Directory) without getting a non-SSL configuration to work.
9. Validate that your LDAP is properly configured by logging into WebSphere Portal and adding a user under Portal Administration functions. Go to the LDAP Web Administrator and check the new user entry.

Enabling WebSphere Portal to Access Your LDAP Server over SSL

At times you may want information traveling between your WebSphere Portal server and LDAP to be encrypted. For instance if your LDAP servers are spread over multiple geographical locations and are accessed through the Internet, you would want to ensure that sensitive information such as user IDs and passwords is not hijacked, manipulated, or viewed by unauthorized persons. A method to ensure privacy is to enable WebSphere Application Server and WebSphere Portal to access the LDAP server with SSL enabled. Of course, this assumes that you have also enabled SSL between the Web Browser and your Web server.

SSL is a security protocol that supports data encryption, authentication, and data integrity verification. An SSL operation consists of an initiation and a data transfer. At the initiation stage, the client/server and the server try to set up the session and connection state by coming to an agreement on encryption and authentication. Agreement is made once the protocol version is chosen, the encryption algorithm chosen, and both parties have authenticated each other (optional). Upon successful initiation stage, data will be transferred using public-key encryption. Data required for the public-key encryption (including keys) is done through certificates. Certificates need to be created for each component and stored on each counterparts' key database. Certificates can be obtained from a certificate authority (CA) such as Verisign or they can be self-signed. Usually you use self-sign if you have two internal servers communicating with each other as in this instance. Thus to get WAS and WP to access LDAP over SSL, you need the LDAP

server certificates in the WAS and WP key storage files. The certificate trust chain can consist of one self-signed certificate or a CA where the CA has confirmed the identity and validity of the certificate.

Setting Up the Server Certificates

Do the following to set up an LDAP server (except Active Directory) over SSL:

1. This step applies to users who need to generate a self-signed certificate. If you are importing a CA certificate (which is in base64-encoded ASCII data and has a .arm extension), skip to step 2.
   a. Execute your security key management utility such as gsk6ikm.
   b. Create a CMS Key Database file or open an existing file and create a new self-signed certificate using X.509 version 3 format and 1024-bit key size. Assign the certificate a label and remember it.
   c. Extract the new self-signed certificate as a certificate file using base64-encoded ASCII data as the data type and save it. The filename can be a filename of your choice but the extension must be .arm.
   d. Configure your LDAP directory with SSL enabled using the CMNS Key Database file contained in the self-signed certificate.
2. Import the certificates to a WAS keystore by doing the following:
   a. Open a command prompt and execute `<was_root>\ikeyman.exe`.
   b. Determine the keystore file by going into the WebSphere Application Server Admin Console, and click Security ➪ SSL.
   c. Open the keystore file. If it already exists, you will be asked for a password, otherwise you will need to supply it.
   d. Select Signer Certificates from the top drop-down list and click Add.
   e. Select base64-encoded ASCII data as the data type and browse to the certificate file that you exported from your LDAP server.
   f. Enter the certificate label that you assigned in step 1b, and click Save.
3. Next you must import the certificate into WebSphere Portal keystore
   a. Open a command prompt and execute `<was_root>\ikeyman.exe`.
   b. Open the default keystore file `<was_root>\java\jre\lib\security\cacerts`. The default password is changeit.

c. Select Signer Certificates in the top drop-down list and click Add.

d. Select base64-encoded ASCII data as the data type and browse to the certificate file that you exported from your LDAP server.

e. Enter the certificate label that you assigned in step 1b and click save.

4. Shut down the non-SSL port (389) if you want to ensure that data exchanged between WAS and WP is confidential.

To configure Active Directory over SSL, you need to first install Active Directory, Internet Information Services, and then WebSphere Application Server. Then perform the following tasks:

1. Export the root CA certificate:
   a. Open your Web browser and connect to `http://localhost/certsrv`.
   b. Select the task Retrieve the CA Certificate or certificate revocation list and click Next.
   c. Choose the certificate you created and set the format to base 64 encoded. Click Download CA Certificate.
   d. Save the certificate in a file called `certnew.cer`.
   e. Open a command prompt, enter **mmc.exe**, and click File ⇨ Add/Remove Snap-in.
   f. Click Add, choose CA Snap-in, and click OK. Find the root certificate public key and save to a file.
2. Import the certificate to the WebSphere Application Server keystore:
   a. Open a command prompt and execute `<was_root>\ikeyman.exe`.
   b. Open the default keystore file `<was_root>\java\jre\lib\security\cacerts`. The default password is changeit.
   c. Select Signer Certificates in the top drop-down list and click Add.
   d. Select base64-encoded ASCII data as the data type and browse to the `certnew.cer` file.
   e. Type a name for the certificate and click OK.

Configure WebSphere Portal to Support SSL

To configure WebSphere Portal to use LDAP over SSL (excluding Active Directory) do the following:

1. Ensure that WebSphere Application Server has been configured and is operating with SSL support by selecting Security ➪ User Registries ➪ LDAP and ensuring SSL is enabled and the sslconfiguration property file has been correctly entered.
2. Configure WebSphere Portal to use LDAP over SSL by modifying the `<ldapRepository...>` stanza in `<wp_root>/shared/app/wmm/wmm.xml`:
   a. Change the LDAP port number to the SSL port number.
   b. Add the following key/value pairs: java.naming.security.protocol="ssl".
3. Restart WebSphere Portal.

For Active Directory do the following:

1. Ensure that WebSphere Application Server has been configured and is operating with SSL support by selecting Security ➪ User Registries ➪ LDAP and ensuring SSL is enabled and the sslconfiguration property file has been correctly entered.
2. Configure WebSphere Portal to use LDAP over SSL by modifying the `<ldapRepository...>` stanza in `<wp_root>/shared/app/wmm/wmm.xml`:
   a. Change the LDAP port number to the SSL port number.
   b. Set wmmmGenerateExtId to false.
   c. Add the following key/value pairs: java.naming.security.protocol="ssl".
3. Remove all occurrences of ibm-appUUIDAux from `wmm.xml`.
4. Enable Active Directory to recognize a short name (as opposed to a fully qualified DN) by selecting Security ➪ User Registries ➪ LDAP ➪ Advanced LDAP Settings in the WebSphere Application Server console and change the user filter from (&(cn=%v)(objectclass=user)) to (&(sAMAccountName=%v)(objectclass=user)).
5. Restart WebSphere Portal.

Mapping of Member Manager Attributes to Your LDAP Attributes

The configuration parameters that are generated by WebSphere Portal installation and WPSconfig are stored in `<wp_root>\shared\app\wmm\wmm.xml`. When you configure WP for LDAP, there is an entry configurationFile

="<wp_root>/wmm/wmmLDAPServerAttributes_XXX.xml under the LDAPRepository tag in the wmm.xml file. This entry refers to an xml that maps the profile attributes to the LDAP object classes and attributes. The last three letters refer to the LDAP directory being mapped. For instance:

- wmmLDAPServerAttributes_IDS contains the maps for IBM Directory Server
- wmmLDAPServerAttributes_SO contains the maps for Sun One Directory
- wmmLDAPServerAttributes_IDS contains the maps for Active Directory

In each file, the names of the various repositories, their Member Managers adapter implementation classes, and the mapping between the Member Manager attribute and the repository are defined. Mapping of the user attributes to the LDAP directory is based on the inetOrgPerson schema. By modifying the wmmLDAPServerAttributes_XXX file, you can expose new LDAP attributes to Member Manager.

For instance look at this entry in wmmLDAPServerAttributes_IDS:

```
<attributeMap wmmAttributeName="uid"
                        pluginAttributeName="uid"
                        applicableMemberTypes="Person"
                        requiredMemberTypes="Person"
                        dataType="String"
                        valueLength="256"
                        multiValued="false" />
```

You see that the LDAP attribute (wmmAttrributeName) uid is mapped to the Member Manager attribute (pluginAttributeName) uid where the member type is Person and it is required. The data type is string of length 256 bytes and it cannot have more than one value assigned.

What is more important is that this is the process to define which attributes will be stored in a look-aside database.

Summary

In this chapter we focused in great detail on implementing WebSphere Portal Identity Manager. We specifically focused on using LDAP servers due to their suitability in handling large user registries that are dispersed over multiple locations. In the next chapter we will discuss the logical next topic: implementing WebSphere Portal with Single Sign-on.

CHAPTER 20

Integrating Security and Identity Management Tools with WebSphere Portal

In this chapter you will learn to integrate security and identity management tools with WebSphere Portal. Because of the complexity involved with security features such as authentication and authorization in environments including many applications and information sources, a new category of products has emerged in the past few years. These products manage repositories of users and their profiles and implement security policies for authenticating and authorizing access based on identifying users and mapping them to static or dynamic roles. These tools allow you to manage a complex entitlement model that spans multiple applications and sources.

Because portals are in many ways the ultimate integrated environment, knowing how to use these tools in the context of WP is increasingly important and many portal initiatives succeed or fail depending on identity management and security. Perhaps the most well-known issue that must be addressed in the context of a portal is that of single sign-on (SSO). A good SSO environment means that once a user has authenticated with the system (and normally this is during the initial log-on to the portal), he or she will not be asked to authenticate again even when they traverse application boundaries. A bad SSO implementation (or no SSO implementation) will constantly ask the user for a user name and a password—every time the user accesses a separate application. Because portals often integrate a large number of back-end applications as portlets, a bad SSO implementation will create a horrendous user experience and can potentially create a usability issue that will cause limited user adoption. This is the basic reason that security and identity management tools have been highly successful in the past few years and the reason that a new category of products has emerged.

Two approaches to SSO can be used. In Chapter 23 we discuss back-end SSO within WP using the Credential Vault. This approach makes use of built-in WP functionality that allows you to store back-end application credentials within WP and associate them with WP users. In this chapter, the focus is the use of external security managers to perform SSO.

Many products address this need for using external security managers to perform SSO. Among them are market-share leaders such as Netegrity SiteMinder and Tivoli Access Manager (TAM). At a conceptual level most of these products provide similar functionality. In this chapter you will learn how to integrate both products into a WP environment. We will not review all functions in both products—from a functional perspective all features described are available in both products. Our choice to provide a description using TAM or SiteMinder is somewhat arbitrary and we will keep this discussion somewhat generic so as not to have to duplicate this description for both products (or others in this category).

Isn't J2EE Security Enough?

The first questions you should be asking yourself are: Why you even need to consider additional security products? and Why are WP and its underlying WAS platform not enough? The answers depend on your portal implementation. If you have a simple portal focused on a single function and are not integrating many applications within the portal or are integrating only J2EE applications developed for WAS, then you might be able to get away with using WAS and WP security. You can use the WAS J2EE security model to manage both authentication and authorization so long as the elements you manage are servlets, EJBs, JSPs, Java clients, and the like. When you also have portlets, the same model can be applied using WP security. These models include the following:

- Declarative security expressed in various deployment descriptors managed by application assemblers or administrators, and
- Programmatic security coded into the application by developers.

In addition, if you need to implement highly specialized security features (such as dynamic roles) you need to delve deep into the underlying JAAS layers. All of these options do not provide you with a full solution when the protected entities are more complex, when the resources are not all running within WP/WAS, and in some cases even when everything is completely within WP/WAS. As an example, WAS has a different model for protecting EJBs than Web applications making the administration process harder. Access control is limited in terms of granularity—forcing you to use either

fine-grained method level access control or course-grained package level access control.

These issues have created a category of products focused on identity and security management. With these tools you implement a better and more comprehensive security model that is separate from the underlying WAS/WP platform. Benefits for doing this include the following:

- Support for heterogeneous environments and servers within a single and consistent security model
- Ability to manage virtually any resource
- Central management of security information
- Configurable session management (for example, session time-outs)
- Full support for user provisioning
- Definition of security and access control rules based on users, roles, dynamic roles, and even through rules that match data in a user context with conditions that determine whether the user should have access to a particular resource
- Support for personalized Web and portal content using a consistent rule set regardless of the underlying provider
- Policies and personalization based on IP addresses
- Enhanced security attributes (from an attack perspective)
- Multi-grained security (that is, the ability to define fine-grained access control on some resources and coarse-grained access at the same time)
- Support for SSO (see next section)

Third-Party Authentication and SSO Architectures

SSO Architectures are primarily based on a third-party authentication server that is inserted in your infrastructure as a reverse proxy. Because it serves as a reverse proxy, it intercepts all requests regardless of which server the request is ultimately dispatched to. Because you will have so many applications in a portal environment, the concept of a third-party authentication server says that it makes no sense for any one of these systems and applications to be the main authentication engine; instead, a dedicated authentication server is used to authenticate the user. Once authenticated, the authentication server places the user information in a token that is passed to all systems and applications. These systems look for the information within the token and trust this information. This means that the systems and

applications rely on the authentication server and trust the authentication server rather then needing to reauthenticate the user.

In a WP environment, a third-party authentication server is also trusted by WP itself. The mechanism by which this occurs is actually a WAS feature that WP implicitly inherits. Regardless of whether you use SiteMinder or TAM, SSO works through a Trust Association Interceptor (TAI). A TAI is an interface that is part of the WAS security model. It is implemented by a Java class that is installed on WAS (or WP in your case); its task is to validate that the request is legitimate by inspecting agreed upon information that is stored by the authentication server in the request. Once a user is authenticated, the authentication server creates a token; in an HTTP request this information may be placed as specific headers in the request. When the request reaches WP the TAI is invoked. The TAI knows which headers need to be checked and retrieves the correct information from the headers. Because it trusts the authentication server, it applies the appropriate credentials within WP and WP continues as though it is the one who performed the authentication. The TAI should return either a Distinguished Name (DN) or a short name. WP then performs a lookup within the user registry to verify the DN (potentially first converting the short name into a DN if the TAI returns a short name). If the lookup fails, WP will refuse to trust the information and will refuse the request; otherwise, an LTPS token is created and stored as a cookie for subsequent requests within the session.

In order to correctly support user identification and authorization in addition to authentication, both WP and the authentication server must share the same user registry. Specifically, an LDAP server maintaining user information and profiles needs to be registered with both WP and the authentication server.

Once both WP and the security server share the same user registry, you can go one level further than authentication (hence we use the term *security server* here for the first time). Access rights within WP are usually administered using access control lists and are stored within WP's administration database. However, you can also configure an external security manager such as TAM and SiteMinder to protect WP resources. Both TAM and SiteMinder can be used to manage authorization policies for WP resources such as portlets and pages in addition to WAS constructs such as servlets and EJBs. This is mostly based on URL paths and the mapping of URLs to users and roles.

Integrating TAM with WP

In addition to supporting all the features described in the previous section, TAM also supports some WAS-specific security features. WAS supports an IBM proprietary SSO token called Lightweight Third Party Authentication

(LTPA). This is the preferred SSO token method in WAS and therefore in WP. LTPS is naturally also supported by TAM and hence the TAM/WP security integration is very comprehensive.

You should consider using TAM security rather than straightforward WP security for a variety of reasons. First, TAM implements a comprehensive SSO architecture that will cover most if not all of your back-end systems very naturally. In addition, TAM will provide you with a higher level of security for the following reasons:

- TAM includes a secure proxy server (WebSEAL) that has been deployed in very large installations and has proven to be very effective. The WebSEAL proxy acts as an additional security layer within your DMZ, breaking direct access between browsers (and potential attackers) and your core application servers.
- TAM supports multiple authentication schemes including user name/passwords, various forms of certificates, tokens, and biometrics.
- TAM supports cross-domain SSO.
- TAM supports not only WP but also other WebSphere products such as WebSphere Everyplace (see Chapter 24).
- WebSEAL is a layer that can be hardened to a great extent without affecting important platforms such as WP (which would be much harder to lock down and harden).
- TAM can easily be integrated with existing user registries.
- TAM supports rule-based decisions.
- TAM supports full audit capabilities.

Installing TAM Support on WP

The first thing you need to do in order to enable TAM-to-WP integration is to set up the TAM TAI. This TAI is not unique to WP and the installation steps are identical to those you would follow when integrating WAS and TAM. Log in to the WAS Administration Console and click Security ➪ Authentication Mechanisms ➪ LTPA. Scroll down, as shown in Figure 20-1, and click Trust Association.

Make sure that trust association is enabled and click Interceptors. Also make sure that your interceptor is defined as `com.ibm.ws.security.web.WebSealTrustAssociationInterceptor`; if it is not, click New and select this class.

Next you need to set up the LDAP settings, including the security server ID, password, host, port, base DN, bind DN, and bind password. Click

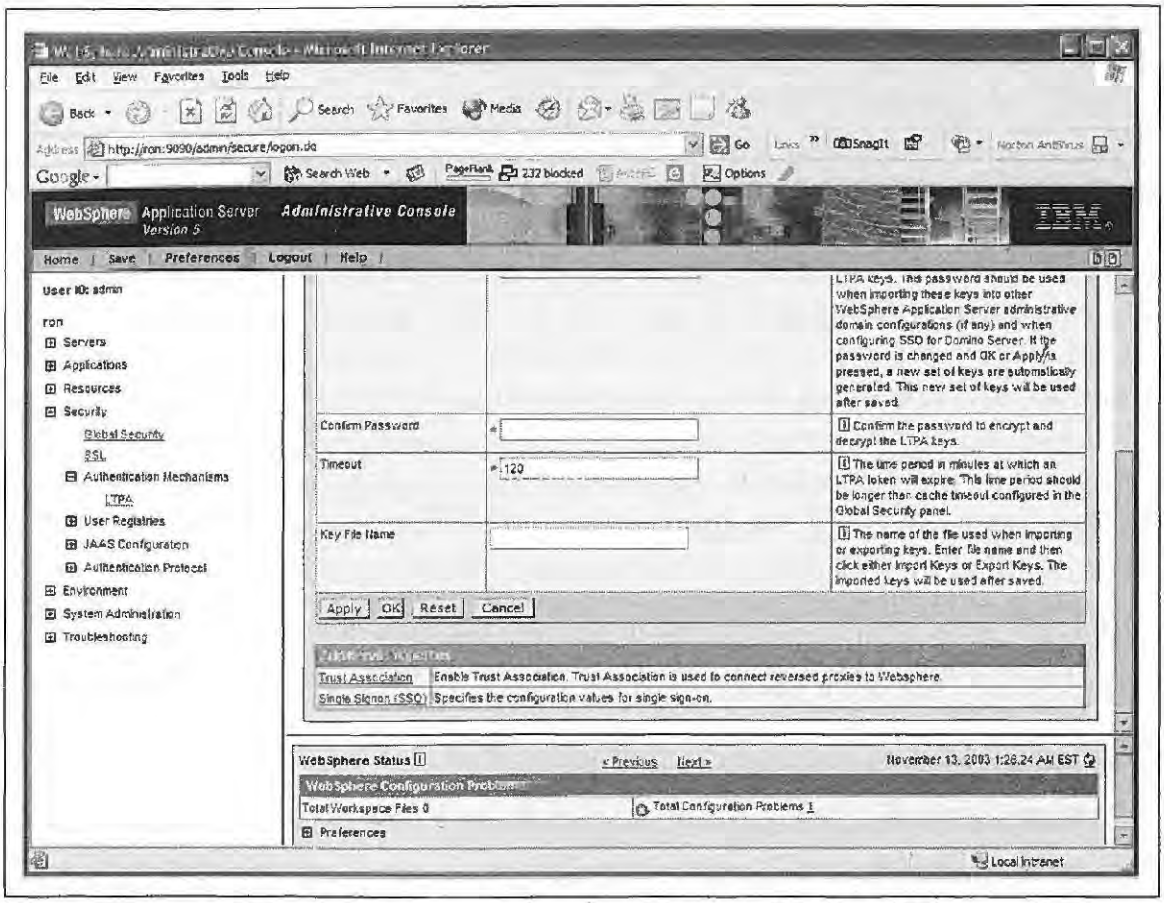


Figure 20-1 Opening the Trust Association Screen within the WAS Administration Console.

Security ⇨ User Registries ⇨ LDAP and enter these values as shown in Figure 20-2. Finally, you will need to restart WAS.

While authentication using TAM is really set up within WAS, authorization via TAM is a WP-specific process and involves WP resources such as portlets and pages. This requires you to configure the connection between WP and TAM by running the following routine from the TAM installation:

```
<TAM_ROOT>/sbin/pdjrtecfg -action config -java_home <WAS_ROOT>/Java/jre
```

Then configure an SSL connection between WAS and TAM, create a key file and a properties file, as well as an agreed-upon user name and groups. To do so, run the following command from your command line:

```
<WAS_ROOT>/Java/jre/bin/java com.tivoli.mts.SvrSslCfg <USER NAME>
<SECURITY MASTER PASSWORD> <PDMGRD HOST> <PDACLD HOST>
```

Next, add the following line to `services.properties`:

```
com.ibm.wps.services.authorization.ExternalAccessControlService =
com.ibm.wps.services.authorization.PDExternalAccessControlServiceImpl
```

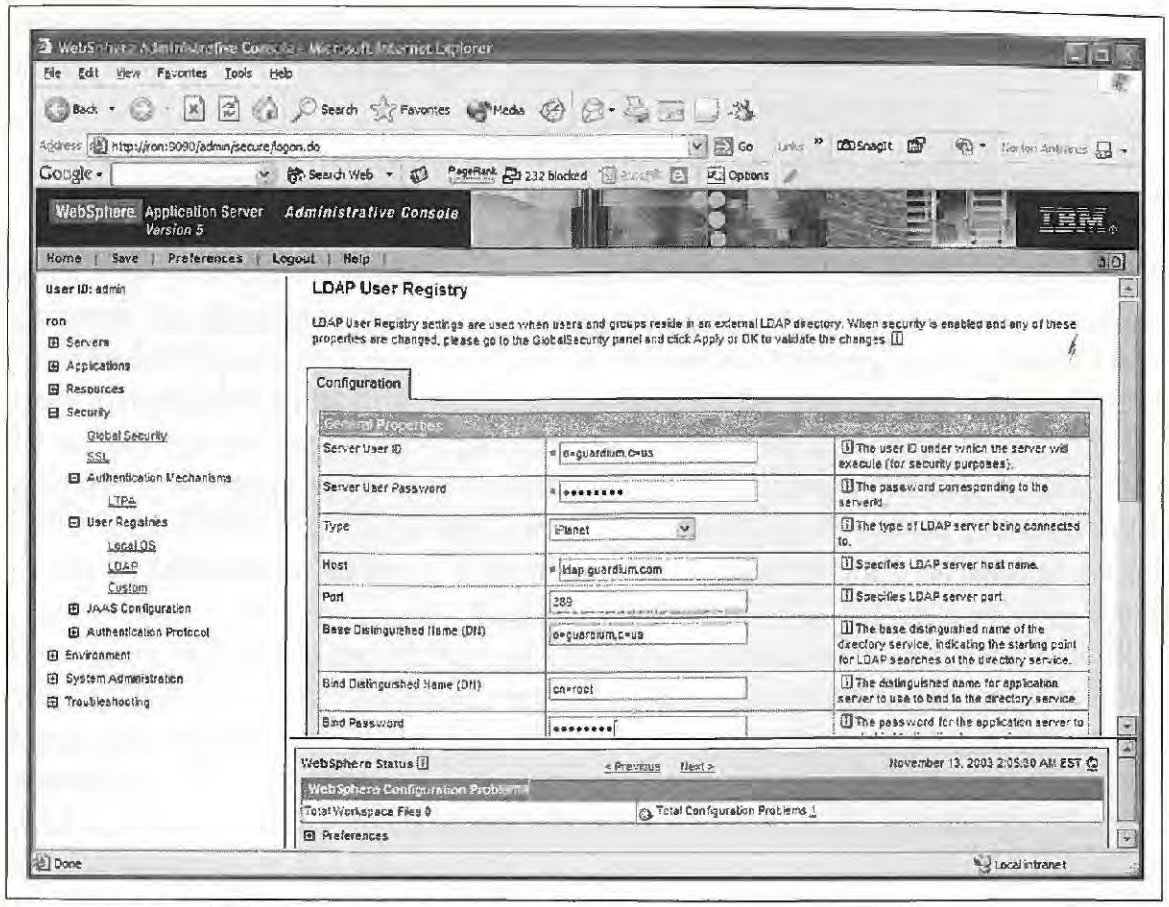


Figure 20-2 Configuring the LDAP User Registry.

Next, edit `externalAccessControlService.properties` within your TAM installation and change the value of `accessControl` to equal `true`.

Next, edit `portallogin.config`. There are two sections called `WpsNewSubject` and `WpsSubjectExists`. In each section you need to add the following line at the end:

```
Com.ibm.websphere.security.auth.module.proxy.WSLoginModuleProxy required
delegate=com.tivoli.mts.PDLoginModule;
```

Restart the portal server and you are done.

Disabling WP User Provisioning and Manipulation

When using an external security manager such as TAM you should also ensure that all user provisioning occurs through TAM. Because TAM is the main security manager all new users must be registered through TAM, and user provisioning directly within WP will create situations in which users have registered and received acknowledgment by WP but when they try to log in they will be denied. Therefore, you must disable WP's user creation

capabilities. In addition, you should disable the User Manager portlet. You can do both of these tasks by removing the user management portlets when logged in as an administrator.

Setting Up SSO

Before proceeding on to discuss Netegrity SiteMinder with WP, let's look at some specific (and common) scenarios in which you would use security and identity management tools. You would use a tool such as TAM or SiteMinder when deploying numerous applications and data sources through WP. In this section we will walk you through some specific examples of such a deployment. Specifically, we will walk you through a few scenarios involving the integration of SAP systems with TAM and WP. The discussion is similar in concept (although different in details) when SiteMinder is used.

SAP is one of the largest software vendors in the world and is most well known for its Enterprise Resource Planning (ERP) solution. It maintains the largest market share of packaged solutions and most large corporations run SAP software. In this section we show you two scenarios that are common when integrating SAP solutions in an existing WP environment using TAM. Because such integration may involve presenting SAP functionality through WP and because SAP systems manage their own security, you will need to set up an SSO environment that includes WP and the SAP software—through TAM. This means that in addition to what you already know in terms of WP and TAM, you will have to set up a trust relationship between SAP servers and TAM.

Because the SAP solution set is so broad, an integrated solution can be implemented in many ways and you can choose to use or omit many SAP components. While we cannot cover all options in this section, we will describe two different scenarios that are fairly typical—both in the SAP world as well as in cases in which other suites are being used (for example, PeopleSoft and Siebel) The first scenario describes the integration of the `mySAP.com` Internet-enabled infrastructure and the second describes the integration of the SAP Enterprise Portal. In both cases the assumption is that you are using a back-end component of SAP such as various SAP R/3 modules (for example, finance and warehousing) or SAP BW (the business information warehouse).

mySAP.com Integration

Before we can show you the steps involved in integrating `mySAP.com` or the `mySAP` Workplace, let's review the elements involved in the `mySAP.com` architecture, and specifically SAP's Internet Transaction Server (ITS).

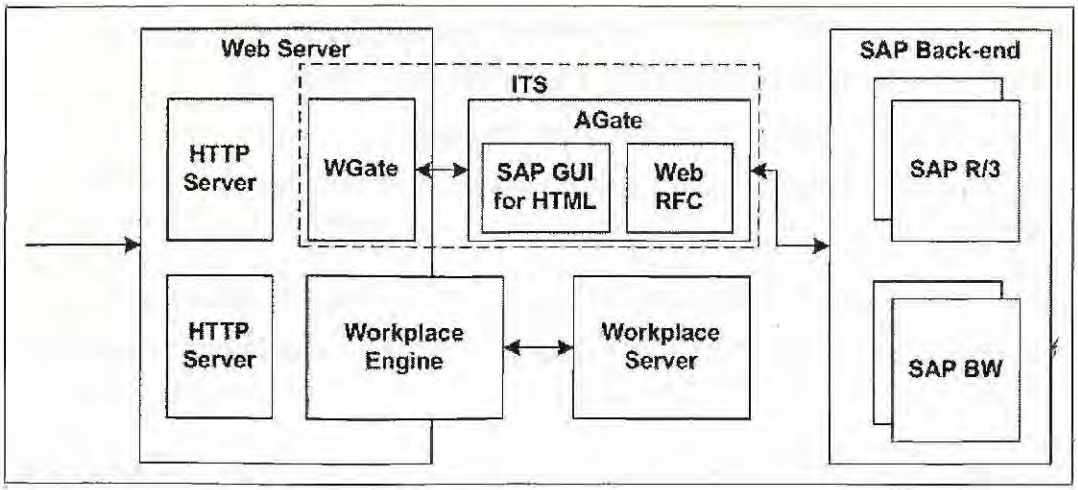


Figure 20-3 mySAP.com Architecture.

Figure 20-3 shows a typical layered scheme of `mySAP.com`. The back end consists of various application modules within SAP R/3 and SAP BW. In addition to these servers the Workplace Server is a component built on the same infrastructure as the SAP applications and is responsible for SAP application authentication, role administrations, and remote invocations. Because it is built using the same infrastructure (called R/3 Basis), it is able to implement security on behalf of the applications and is the key to SSO.

The infrastructure responsible for creating and delivering Web screens based on the back-end function is ITS. ITS comprises of the `WGate` and the `AGate`, the `WGate` being the interface to the Web server and the `AGate` being the interface to the SAP back end. Of the two components the `AGate` does most of the work and is responsible for session management, generating HTML screens from R/3 screens, invoking functional modules from the HTML, managing connections and sessions to the back end, and so on.

From a security perspective, ITS can make use of the internal security services provided within `mySAP.com` Workplace or can delegate to an external security manager using the Pluggable Authentication Service (PAS)—much like WAS/WP can use its own authentication or use a TAI with a third-party authentication server.

In order to use PAS, you need to establish a trust model between your authentication server (for example, TAM) and `mySAP.com`. PAS is deployed as a shared library used by the ITS gateway interface. ITS uses PAS to verify the user authentication information generated by TAM. PAS queries TAM and then sends the validated information to the SAP Workplace, which returns a login ticket. This login ticket is then used as the SSO ticket when accessing SAP modules. At a more detailed level the sequence is as follows:

1. User provides credentials, which are forwarded by the Web server to `WGate`

2. WGate sends the credentials to AGate
3. AGate forwards the credentials to TAM using PAS
4. TAM verifies credentials and replies to PAS
5. PAS passes the user information to the SAP Workplace
6. SAP user information is returned to PAS
7. PAS creates a login ticket and sends it back to the Web browser
8. WGate creates a cookie from the login ticket and sends it in the HTTP response

Integrating TAM/WP with mySAP.com is done by defining the mySAP.com infrastructure within a TAM WebSEAL junction. Follow these steps to configure both TAM and SAP (for more details refer to the mySAP.com and the TAM documentation):

1. Create a WebSEAL junction to the mySAP.com environment.
2. Add the line-META=CONTENT to webseald.conf.
3. Set script-filer to yes in webseald.conf.
4. Add entries for filtering URLs to the JMT.
5. Add the line header=Accept-Encoding to the filter request headers section in webseald.conf.
6. Restart WebSEAL.
7. Copy sappdssocgi.exe from the mySAP.com installation into the www/docs/cgi-bin directory in WebSEAL.
8. Create a secret trusted key named-sappdsso.key.
9. Install SAP Secure Network Communications (SNC) on your ITS instance.
10. Stop the ITS Manager if it is running.
11. Copy the sappdssoauth shared library to the ITS programs directory.
12. Copy sappdssoauth.srvc into the ITS services directory.
13. Copy your mySAP templates into the ITS templates directory.
14. Under TAM's Policy Director directory, create an SAP directory with subdirectories for etc, var, and bin.
15. Copy the secret key file generated in step 8 to the etc directory.
16. Modify the login.html template copied in step 13; replace WEBSEALSERVERNAME with the appropriate junction.

SAP Enterprise Portal Integration

Because SAP has so many modules and offerings, it also develops and sells portal solutions. In fact, in many environments SAP Portal is a direct alternative to WP and there are many companies which deploy SAP Portals as its portal even when the content delivered through the portal is not exclusively sourced in SAP software. Still, SAP Portal is primarily used in environments where SAP applications cover a majority of the business functionality deployed in the organization. In environments where this is true and yet portal functionality is still important, it is common to see two portal infrastructures—WP as the overall portal and SAP Portal as a subportal. This eliminates some of the pain involved with integrating many SAP applications into WP. More on this topic in Chapter 23.

The SAP Enterprise Portal architecture is shown in Figure 20-4. Much like WP Extend, Enable, and Experience, SAP Enterprise Portals are available as three different offerings:

- SAP Enterprise Information Portal (SEIP)—creating a unified Web interface
- SAP Enterprise Collaboration Portal (SECP)—including all functionality in SEIP and adding personalization, collaboration, categorization, and search capabilities
- SAP Enterprise Unification Portal (SEUP)—including all functionality in SECP with enhanced coordination capabilities and virtual desktop capabilities

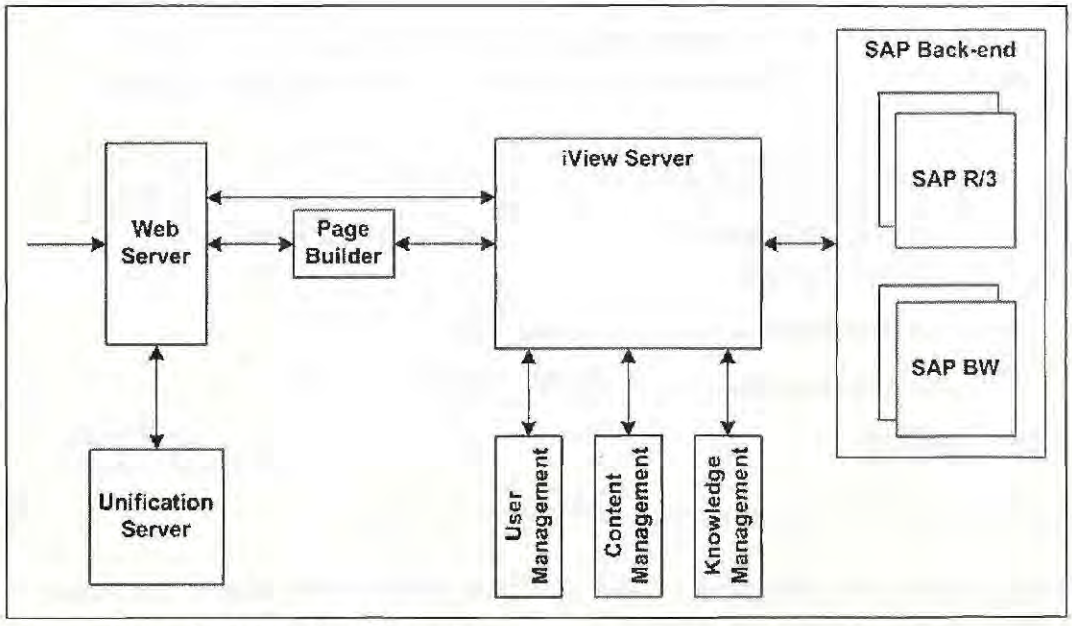


Figure 20-4 SAP Portal architecture.

As any portal, an SAP Enterprise Portal has its own implementation of authentication, authorization, and administration of users. It also has its own implementation of an SSO environment based on a login ticket. However, SAP Portals can externalize authentication and SSO to a third-party product such as TAM and this is the preferred method when an SAP Portal needs to be a part of a larger WP deployment. You can do this in one of two ways. The first involves configuring the SAP Portal to use external authentication by specifying the name of an HTTP header that contains the authenticated user name and creating the appropriate WebSEAL junction. The second is to use a WebSEAL Global Sign-On (GSO) junction, which is configured to pass the user name and password in the form of a Basic Authentication header and configuring the SAP Portal to do normal LDAP lookup. To implement the first option, follow these steps:

1. Ensure that both WebSEAL/WP and the SAP Portal share the same LDAP user registry.
2. Set up a WebSEAL junction for the SAP Portal using the `-c iv_user` option, allowing WebSEAL to pass the user id in the HTTP header.
3. Create a WebSEAL Junction Mapping Table (JMT).
4. Add `IFRAME=REFRESHURL` and `PARAM=VALUE` to `webseald.conf` in the `filter-url` section.
5. Add `TD=ONCLICK` to the `filter-events` section in `webseald.conf`.
6. Add `scheme=hrnp` to the `filter-schemes` section in `webseald.conf`.
7. Add `script-filter=yes` and `rewrite-absolute-with-absolute=yes` to the `script-filtering` section in `webseald.conf`.
8. Log on to the SAP Portal as an administrator.
9. Select System Configuration ➪ User Management Config ➪ Authentication Server.
10. Set User Authentication Type to External.
11. Set User Name Header to HTTP_IV_USER.
12. Click Apply.

To implement the second option, follow these steps:

1. Ensure that both WebSEAL/WP and the SAP Portal share the same LDAP user registry.
2. Create a WebSEAL junction for the SAP Portal with the `-v` option.

3. Follow steps 3–9 as described in option 1.
4. Set User Authentication Type to LDAP.
5. Enter the LDAP server information.
6. Click Apply.

Installing SiteMinder Support on WP

When using SiteMinder you have three options:

1. Using SiteMinder as a third-party authentication server
2. Using the SiteMinder Policy Server as an external security manager, or
3. Both

Figure 20-5 shows the different Netegrity components that you may choose to use and how they interact with WAS/WP. You may choose to use Netegrity's authentication server or another authentication server and in either case can still use the Policy Server. The SiteMinder Policy Server allows you to better define and control the security policy and, more specifically, access control on your portal. At a WAS level, the Policy Server interacts with three components (with portlets being managed by the Servlet Agent Controller):

1. The EJB Agent Controller
2. The Servlet Agent Controller
3. The Web Agent Controller

When a protected resource is requested, the appropriate agent validates the request with the SiteMinder Policy Server. As an example, when the

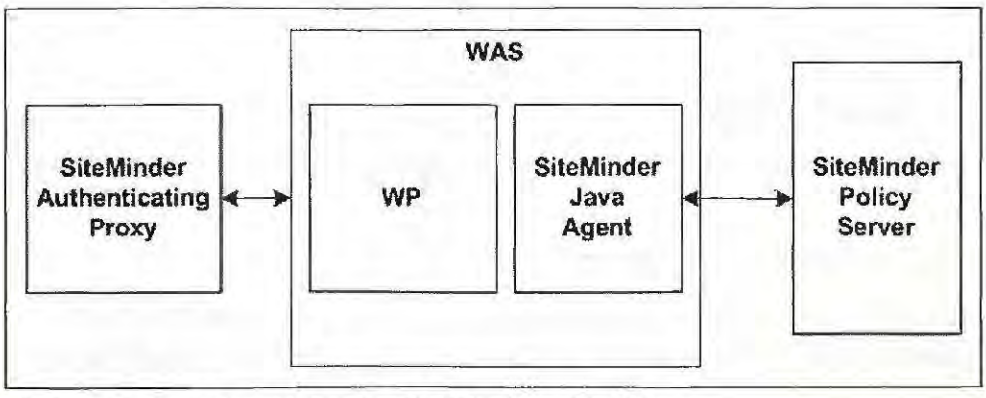


Figure 20-5 Using Netegrity SiteMinder with WP.

resource is an EJB or an invocation of an EJB method, the EJB Agent (which is itself an EJB) communicates with the Policy Server using a set of Java APIs and offers resource protection at the following granularity levels:

- An entire package
- Beans
- Bean methods
- Method signatures

The last granularity type—method signatures—allows you to define protection elements that are based on a combination of method and its parameters, that is, the signature a method may have rather than a unique bean method.

Using SiteMinder as a Third-Party Authentication Server

In order to install SiteMinder as a third-party authentication server you need to install the SiteMinder TAI:

1. Install SiteMinder as a trusted authentication server:
   a. Log in to the WAS Administration Console and click Security ➪ Authentication Mechanisms ➪ LTPA.
   b. Click Trust Association.
   c. Check the Trust Association Enabled check box.
   d. Click Interceptors in the Additional Properties section.
   e. Because SiteMinder is not one of the default authentication servers, you will have to add it. Click New.
   f. Enter the interceptor class name:

      ```
      com.ibm.wps.sso.SiteMinderTrustAssociation Interceptor
      ```

   g. Click Apply.
   h. Click Custom Properties.
   i. Enter custom properties as defined in the SiteMinder installation guide.
   j. Click OK and restart the server.
2. Now that you have completed setting up SiteMinder as a trusted reverse proxy, you can continue to Netegrity-specific configuration:
   a. Install the SiteMinder Java Agent API by adding the JAR files in `<SITEMINDER_ROOT>/sdk/java` to `<WAS_ROOT>/lib` and by

adding the appropriate JNI support library to the PATH variable of the Java VM (for example, in Windows the support library is called `smjavaagentapi.dll`). The primary Java class needed is `netegrity.siteminder.javaagent.AgentAPI`, which can be used to enforce access control by connecting to the Policy Server. The Java Agent API insulates you from the implementation details within the Policy Server and supports full session management, automatic encryption key rollover, and real-time policy updates. All Java-based SiteMinder agents make use of the Java Agent API, and custom agents built using the Java Agent API can participate with standard SiteMinder agents within an SSO environment.

b. Set the `validateSessions` parameter to `true` within `siteminder.properties`.

c. Follow the instructions in the Netegrity Policy Server Management manual to define a Web Agent in the SiteMinder Administration Console. This agent is required to establish a trust association between WAS and SiteMinder and allows the TAI to validate the sessions with the Policy Server.

d. Set the `com.ibm.wps.sso.SiteMinderTrustAssociation` as the class name for the interceptor TAI.

3. Finally, set the following configuration parameters:

```
com.ibm.websphere.security.trustassociation.enabled=true
com.ibm.websphere.security.trustassociation.types=Netegrity
com.ibm.websphere.security.trustassociation.interceptor=
com.ibm.wps.sso.SiteMinderTrustAssociationInterceptor
com.ibm.websphere.security.trustassociation.config=siteminder
```

Using SiteMinder as an External Security Manager

Installing SiteMinder as an external security manager is a little more involved. There are two primary steps involved: installing SiteMinder and modifying WP to delegate policy decisions to SiteMinder. Installing SiteMinder as an external security manager involves the following steps (these steps are for Windows; refer to the Netegrity Policy Server Installation Guide for other platforms):

1. Run the Policy Server setup program `nete-ps-5.5-win32.exe`.
2. Click Next twice and accept the license agreement. Click Next.
3. When asked for the Web servers to configure for use with Policy Server, select the Web server being used by WP.

4. When asked for a JRE for the Policy Server, make sure you give it a JRE 1.3.1—if your currently installed JRE is not version 1.3.1 download a new one and install it.
5. Choose a destination for the installation and click Next.
6. Enter an encryption key and reconfirm it. Remember to record this key for future use. Click Next.
7. If you do not want to configure hardware keys, click Next. If you do, refer to the Netegrity Policy Server Installation Guide.
8. Enter and confirm an administrator password and record this password for future use. Click Next.
9. If you want to configure an SNMP agent at this time refer to the Netegrity Policy Server Installation Guide.
10. Review the settings and click Next if all is correct. When copying is done click Finish to reboot your system.

Modifying WP to use SiteMinder as an external security manager involves setting the following configuration properties:

In `externalacecsscontrolservice.properties`:

```
accesscontrol.domainname=WPS Portal Server
accesscontrol.accountingport=44441
accesscontrol.authport=44442
accesscontrol.aznport=44443
accesscontrol.maxtimeout=5400
accesscontrol.idletimeout=3600
accesscontrol.syncaudit=false
accesscontrol.ipaddress=<your IP>
accesscontrol.scheme=Basic
accesscontrol.agentname=agent
accesscontrol.agentsecret=<Your secret/password>
accesscontrol.admin=siteminder
accesscontrol.password=<Your password>
accesscontrol.userdir=<Your LDAP server>
accesscontrol.public_access_mode=1
accesscontrol.anonymousid=anonymous
accesscontrol.anyauthuser=anyauth
```

In `services.properties`:

```
com.ibm.wps.services.authorization.ExternalAccessControlService=com.ibm.
wps.services.authorization.SiteminderExternlAccessControlImpl
```

In `portlalogin.cfg` add the following to `WpsNewSubject`:

```
com.ibm.websphere.security.auth.module.proxy.WSLoginModuleProxy required
delegate=com.tivoli.mts.SiteMinderLoginModule;
```

In `portlalogin.cfg` add the following to `WpsSubjectExists`:

```
com.ibm.websphere.security.auth.module.proxy.WSLoginModuleProxy required
delegate=com.tivoli.mts.SiteMinderLoginModule;
```

Add a `callbackheaderslist.properties` file to the portal `app config` directory with the following contents:

```
header.1=sm-serversessionspec
header.2=sm-serversessionid
cookie.1=SMSESSION
```

Summary

Setting up a portal is not a trivial thing, but creating a good portal experience is even harder. In this chapter we introduced you to identity management tools and SSO architectures. We explained when you would choose to use these products and how these tools help you in complex portal environments. We then walked you through a brief overview of how to integrate TAM and SiteMinder with WP; a full account of such integration cannot be accomplished in such a short space and you should refer to the documentation of these respective products. Most importantly, we hope you understand the importance of SSO and the importance of creating a good user experience. The main reason of failed portal projects is low user adoption and the main reason for low user adoption is annoying requests to re-login every time a new application is accessed.

While integrating WP within an SSO environment is crucial for large portal projects that serve a large population of users, setting up scaleable and fault-tolerant portal environments is just as important. In fact, you should view this as a prerequisite—after all, if the server is down it really doesn't matter how good of a user experience it could have provided if it were up. In the next chapter we will show you what it takes to build a WP environment with high-availability attributes and will cover topics such as high-availability strategies, portal clusters, horizontal vs. vertical scaling, and how to deploy portlets into a clustered environment.

CHAPTER

21

Designing High Availability into Your Portal Server

In today's large corporations, one of the most common requirements is that the portal should be available 24 hours and 7 days. When users want 24 by 7, they are not referring to uptime but rather a lot more. The requirement is that the system should be perceived to be working 100 percent of the time, with response times consistently meeting expectations (usually under 3 seconds), and all transactions executing accurately, completely, timely, and reliably. While much of the IT industry considers this definition a highly available system, most users consider it a basic requirement of a system design.

In this chapter we will discuss the issues in creating a highly available WebSphere Portal and show you how to manage and configure WebSphere Portal within a cluster.

The Challenges of High Availability

At first glance, most people do not understand the complexity of high availability. To understand it, examine a simple WebSphere Portal configuration such as the one shown in Figure 21-1. Typical users would feel happy if you told them that each component would have an uptime of 99 percent. At first glance, this seems pretty good but translates to a downtime of 3.65 days for one component. The problem is that a WebSphere Portal system is made up of many components. In Figure 21-1, a high level of 33 possible points of failure are identified. If each point had a 99 percent uptime, then the cumulative uptime would be .99**33 or 71.77 percent. This means that for approximately 103 days of a year, WebSphere Portal

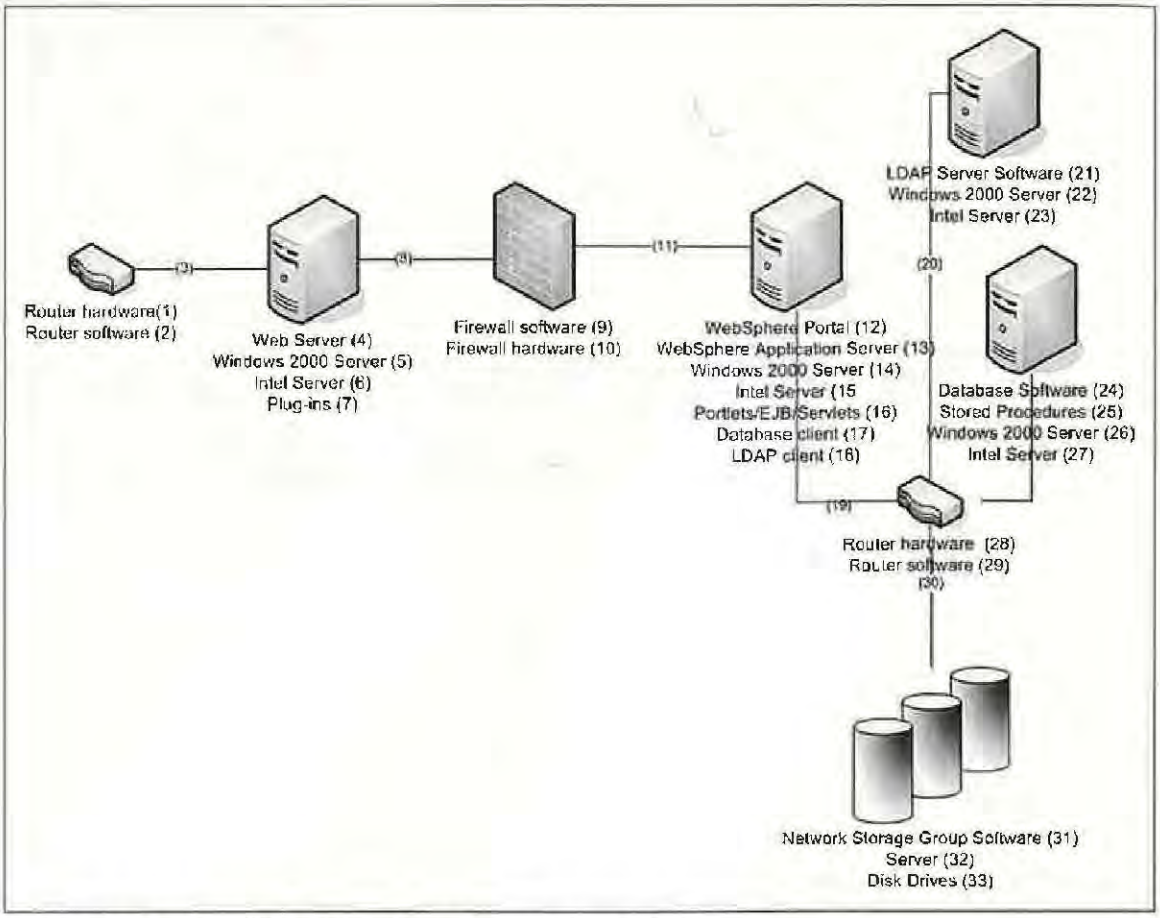


Figure 21-1 Possible WebSphere Portal points of failure.

would be down. It goes without saying that your users would not be happy.

Availability is a metric defined by a user's perception of a system's performance and reliability. It can be calculated as a percentage by the following equation:

```
MTBF/(MTBF+MTTR)
```

where NTGB is the mean time between failures and MTTR is the maximum time to repair or resolve the failure. A WebSphere Portal failure is not just a hardware or software failure but rather anything that prevents the user from performing his or her required function using WebSphere Portal. This includes slow performance, process, and environmental factors. One thing interesting about the equation, which relates to user perception, is that the faster you correct or are perceived to have corrected the problem, the higher the availability. So if problems are shifted to a different system without any interruption in their session, no failure is perceived.

Most IT people believe that high availability can be obtained through hardware redundancy. However, as seen in Figure 21-2, 30 percent of

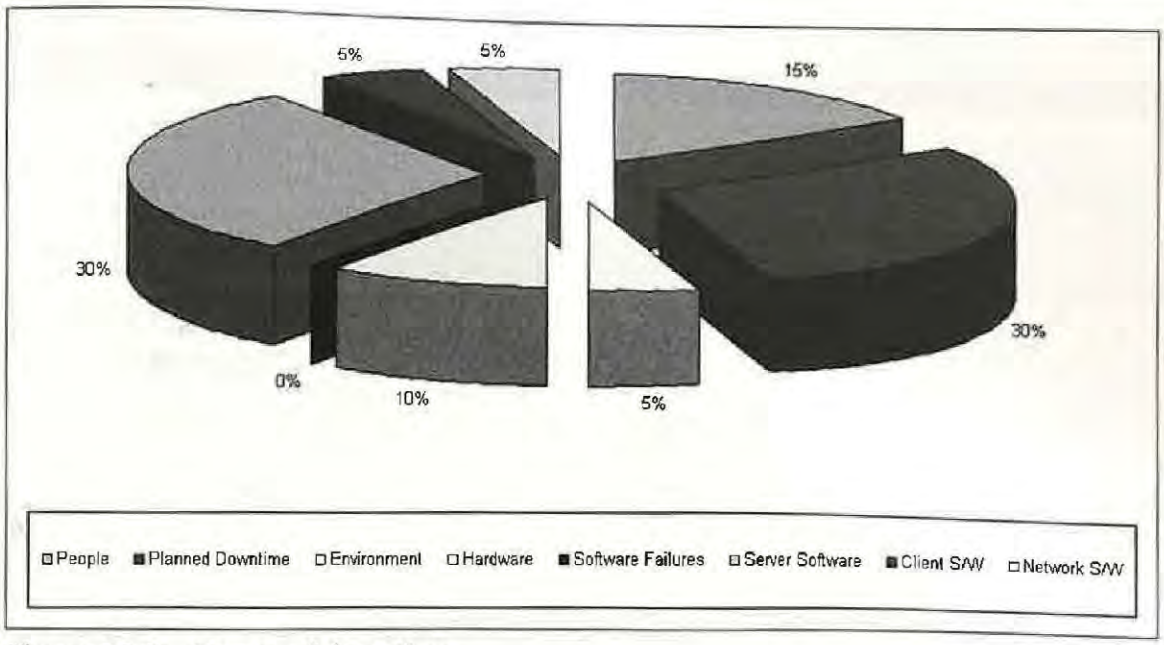


Figure 21-2 Causes of downtime.
Source: IEEE Computer April 1995.

downtime is planned, 10 percent is unplanned hardware downtime, and 40 percent is related to software. This was in 1995. Today with more reliable hardware that is designed to hot pluggable, software failures are increasingly dominating downtime.

Determining a High-Availability Strategy for WebSphere Portal

The most critical decision for you to make is how highly available do you want your WebSphere Portal system to be. The decision should be based on a cost–benefit analysis.

The first step is to determine the impact on your business in dollars and minutes if the system is down. Focus on lost revenue, both in the immediate and distant future. Revenue may be lost in the future due to the impact on customer loyalty and your company's reputation. Also, quantify the impact on employee productivity due to downtime. Look at lost hours, increase in cost due to overtime, and reduced employee efficiency due to poor system performance.

Next calculate the cost of availability. Table 21-1 lists the different levels of cumulative availability. For the first level, 98 percent, determine the cost of implementing availability and multiply it by 10 for each level to get a rough estimate. In other words, level 7 is 60 times more expensive than level 1.

Table 21-1 Availability Levels

LEVEL	PERCENTAGE UPTIME	DOWNTIME/YEAR
1	98%	7.3 days
2	99%	3.65 days
3	99.8%	17 hours, 30 minutes
4	99.9%	8 hours, 45 minutes
5	99.99%	52.5 minutes
6	99.999%	5.25 minutes
7	99.9999%	31.5 seconds

To identify the components that impact availability, you need to look at your process, organization, and technology. Given how little impact hardware has on availability, the largest focus is on the implementation of a well-defined process and the integration of an organization designed to support high availability.

High availability cannot be provided to your WebSphere Portal systems in a vacuum due to the interdependencies on the organization processes, environmental factors, the network, and other systems.

The first step is to get a level set of the state of affairs. A team must be established with representations from development, project management, and infrastructure support to look at system management processes, operations, and the application/infrastructure architecture. The team must also review the system outages and perform a root-cause analysis.

After performing the analysis, the data is used to identify weaknesses in the processes. Processes are examined to reduce the frequency of occurrence of a problem, the duration to fix the problem, and the impact on the complete IT environment. This is done by implementing the following steps:

- Proactive problem prevention that ensures when a problem occurs the root cause is identified and procedures are put in place to ensure it does not recur.
- Effective change management to ensure that only authorized and tested changes are implemented.
- Systems and Applications design is made reliable, scalable, complete, and accurate with features that integrate with the system management infrastructures. All exceptions are detected and are designed to be self-healing or provide automatic recovery. Proper logging is done with well written messages that clearly identify the problem, the module, time, and data that caused the exception.

- Monitoring is proactive so that solutions can be automatically implemented to prevent a problem from occurring.
- Recovery procedures including disaster recovery procedures are automated as much as possible and are documented and regularly tested.
- Situation management procedures are put in place and regularly followed through.
- WebSphere Portal is tested using a formal methodology that covers exception testing, user testing, boundary testing, and load/stress testing. Parallel testing must be done that simulates exactly a worst-case scenario for a production environment. This environment needs to include firewalls configured as they are for production, a similar network (including routers), load/boundary and/or stress test to simulate production, and simulated exceptions to test operation procedure and recovery process.
- The data design is reviewed by the data architects to isolate any corruption or performance issues. Security must also be reviewed to ensure that only authorized applications/users are accessing the data.
- WebSphere Portal system is isolated from other application servers and has sufficiently allocated bandwidth.
- Everything is documented and it is ensured that a process is in place to keep up-to-date. A librarian should be appointed and the documentation should be made public and reviewed regularly.
- Standards need to be in place for each component including operating system (including fix level), WebSphere Portal, hardware, firewalls, application build, routers, and router software. The version and release level should be consistent in each test and production environment and especially if it is a fail-over component. Each component version and release needs to be tested and well mature in the marketplace before being accepted as a standard. They should also be obtained from well-established vendors, thus minimizing the impact of having a nonsupported component.

Processes by themselves do not improve availability. They need to be integrated into the organization to ensure that they are being correctly implemented and those issues are receiving appropriate visibility. Service level agreements need to be established with each department and vendor and metric gathered to ensure that the departments/vendors are meeting their commitments. Regular reporting to upper management of system outages and service level deviation are required. The result of weekly meetings is