Chapter Two

Writing Your First Portlet

"How do I write my own portlet?" This is almost always the first question asked by developers who are looking into WebSphere Portal for the first time. Thus we'll start by writing a portlet. The "hello world" program is commonly used in programming books as a first program. This tradition was popularized by the 1978 classic *The C Programming Language* by Brian Kernighan and Dennis Ritchie. Following in the footsteps of our ancestors, we will start with a "hello world" portlet.

We really don't know what type of experience our readers will have in Java and server-side programming. Therefore, in this chapter we'll build the simplest possible portlet from scratch by writing directly to the IBM Portlet API. We'll compose our Java code, Web application, and portlet descriptors; build and package our portlets for deployment; and test them using either a WebSphere Portal installation or the WebSphere Studio test environment.

Anatomy of a Portlet

Before we actually start building our first portlet, it is important to understand the various components of a portlet and how they fit together. The diagram in Figure 2.1 is a mishmash of several types of diagrams, but it helps to illustrate the different components that come together to form a portlet.

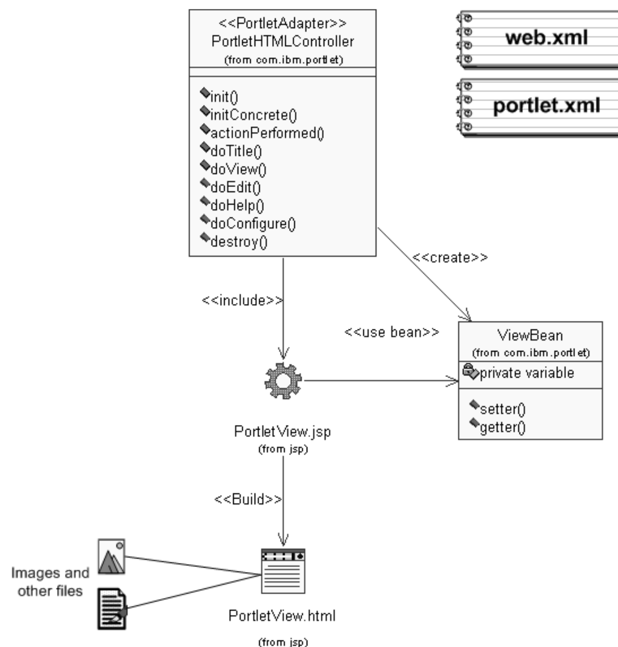


Figure 2.1: Portlet anatomy.

Generally when we think of a portlet, we imagine it as a WAR file. WAR stands for Web ARchive. This is a type of zip file that combines several other files into a format that the portal can understand. A basic directory structure is included in this file, and files are placed in specific locations within this structure. Figure 2.2 shows the basic directory structure contained in a portlet WAR file.

```
+Portlet
  +source
     +com
         +ibm
            +portlet
               +myportlet - portlet controller, util and bean source code
   +nls                   - resource bundles for internationalization
   +WEB-INF               - portlet.xml and web.xml files
      +lib                - jar from source dir, or external jar files
      +tld                - JSP tag library description files
      +classes            - individual classes if not put as jar in lib
      +jsp                - jsp files for portlet
      +images             - images that may be used in the portlet
```

Figure 2.2: Basic portlet WAR file directory structure.

This structure is simply an example, and many modifications can be made to the base structure depending upon the type of portlet you are creating. Additional directories can be created, and some directories, such as the jsp or images directory, can be moved around.

Now that you know where things go, let's provide an overview of the specific things to be included in a portlet and what they do:

- **Portlet controller:** This is a class that extends the portletAdapter. It is the main controller for your portlet and is called by the portal engine to render your portlet. The developer must code all control for your portlet to determine what a portlet does.

- **JSP files:** JSPs are used to render output for your portlet. They are usually called by the portlet controller, depending upon the state and view of your portlet.

- **View beans:** View beans are used to contain data that will be used by the JSP. This structure is used to ensure that the JSP does not perform any data logic itself. A bean is populated and passed between the controller and the JSP via the session or the request.

- **Deployment descriptors:** There are two deployment descriptors that are necessary for your portlet to be deployed and run correctly within the portal server: They are web.xml and portlet.xml

    - **Web.xml:** This is a standard J2EE deployment descriptor for WAR files. It determines the controller status and can contain a lot of information about your package that needs to be deployed.

    - **Portlet.xml:** This is a portal-specific deployment descriptor file. All the data within this file are used by the portal server upon installation to determine parameters for your portlet.

- **Graphics and other files:** A portlet may contain additional files or graphics that can be used by the portlet or included within the JSP or resulting HTML. These files can reside in their own directories within the portlet.

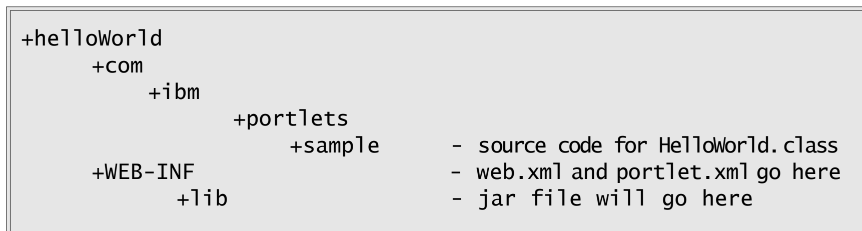
Starting Hello World

We'll start with some Java code, and then compile it and package it into a JAR file. Then we'll look at the deployment descriptors needed for telling the application server and the portal about the portlet. Finally we'll package it all together and deploy the new portlet into the portal.

For this first portlet we will need a much simpler structure with fewer components than what is shown above. The controller will do all of the work in this example so no JSPs will be called to display the portlet view. In later examples we will add JSPs and images as well as view beans and helper or utility classes.

Create the Directory Structure

To begin, you must create the directory structure in which you will create your portlet. Figure 2.3 shows a simple portlet directory structure that is serviceable for development and packaging. We'll use it for our first portlet.



```
+helloWorld
     +com
         +ibm
               +portlets
                   +sample      - source code for HelloWorld.class
     +WEB-INF                   - web.xml and portlet.xml go here
           +lib                 - jar file will go here
```

Figure 2.3: Basic portlet development directory structure.

We create these directories to enable us to easily create the JAR and WAR files. As we shall see there are other directories we can use for our portlets to enable assorted functionalities.

Create the Java

The sample directory is where we will put our Java source file. Create a file named HelloWorld.java in the sample directory and open it in your favorite text editor. Figure 2.4 shows the class we will use for our very first portlet.

```
package com.ibm.portlets.sample;

//import portlet APIs
import org.apache.jetspeed.portlet.*;

//import Java io package
import java.io.*;

public class HelloWorld extends PortletAdapter {
```

Figure 2.4: Your very first portlet class, HelloWorld.java (part 1 of 2).

```
  public void service(PortletRequest request, PortletResponse response)
    throws PortletException, IOException {

    PrintWriter writer = response.getWriter();
    writer.println("<p>hello, world</p>");
  }
}
```

Figure 2.4: Your very first portlet class, HelloWorld.java (part 2 of 2).

Compile the Code

Once we have created the source file, we are ready to compile the Java code. The script below could be used in a batch file to compile the portlet. Start by defining some environment variables, such as where our Java home is. It's usually a good idea to compile this using the JDK provided with the WebSphere Application Server because that is the environment we will be running under. We also set our PATH so that we can get to the Java compiler. Set the variable LIBPATH to point to the directory where the WebSphere JAR files are located. Next, build a variable called CP for our compilation classpath. The CP variable could be built on a single line, but we broke it up to show the various JAR files required. We then invoke the Java compiler and compile our code. This assumes we are in the helloWorld directory. Adjust your directory paths as appropriate for your installation. Figure 2.5 shows the commands needed to compile your "hello world" portlet under the Windows operating system.

```
> set JAVAC=C:\WebSphere\AppServer\java\bin\javac
> set CP=.
> set CP=%CP%;C:\WebSphere\PortalServer\shared\app\portlet-api.jar
> set CP=%CP%;C:\WebSphere\AppServer\lib\j2ee.jar
> set CP=%CP%;C:\WebSphere\AppServer\lib\dynacache.jar
> %JAVAC% -classpath %CP% com\ibm\portlets\sample\HelloWorld.java
```

Figure 2.5: HelloWorld.java compilation steps for Windows.

If you are using Linux, use the commands in Figure 2.6 to compile your code.

```
$ JAVAC=/opt/WebSphere/AppServer/java/bin/javac
$ CP=.
$ CP=$CP:/opt/WebSphere/PortalServer/shared/app/portlet-api.jar
$ CP=$CP:/opt/WebSphere/AppServer/lib/j2ee.jar
$ CP=$CP:/opt/WebSphere/AppServer/lib/dynacache.jar
$ $JAVAC -classpath $CP com/ibm/portlets/sample/HelloWorld.java
```

Figure 2.6: HelloWorld.java compilation steps for Linux.

If for some reason this doesn't compile, you're going to have to fix the errors before moving on. Some common errors are

- **Typos:** Check for mistyped class names, paths, and variable names.
- **File name:** The file name and the class name must match. In our case, the file name is HelloWorld.java and the class name is HelloWorld. If you have changed one, change the other.
- **Editor woes:** Make sure your editor really saves the file as text. An editor like WordPad does not save as text by default.

Create the JAR File

Once the Java source is compiled, we need to create a JAR file in the WEB-INF\lib directory. Again, assume we are in the helloWorld directory. Figure 2.7 shows the commands used to package the "hello world" class files into a JAR file.

```
> set JAR= C:\WebSphere\AppServer\java\bin\jar
> %JAR% -cv0f WEB-INF\lib\HelloWorld.jar com\ibm\portlets\
    sample\*.class
```

Figure 2.7: JAR commands under Windows.

If you are using Linux, use the commands in Figure 2.8 to package your JAR file.

```
$ JAR=/opt/WebSphere/AppServer/java/bin/jar
$ $JAR –cv0f WEB-INF/lib/HelloWorld.jar com/ibm/portlets/
  sample/*.class
```

Figure 2.8: JAR commands under Linux.

Create the Deployment Descriptors

You will need to create two XML files:

- **helloWorld/WEB-INF/web.xml:** the Web deployment descriptor
- **helloWorld/WEB-INF/portlet.xml:** the portlet deployment descriptor

The Web deployment descriptor defines the class used for our “hello world” portlet. Figure 2.9 shows our “hello world” portlet’s Web deployment descriptor.

```
 <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="HelloWorldWebApp">
  <display-name>HelloWorldWebApp</display-name>
  <servlet id="HelloWorldPortlet">
    <servlet-name>HelloWorldPortlet</servlet-name>
    <servlet-class>com.ibm.portlets.sample.HelloWorld</servlet-class>
  </servlet>
  <servlet-mapping id="ServletMapping_com.ibm.portlets.sample.HelloWorld">
    <servlet-name>HelloWorldPortlet</servlet-name>
    <url-pattern>/HelloWorldPortlet/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Figure 2.9: “Hello world” web.xml file.

The portlet deployment descriptor defines the portlet to the portal. Each portlet is mapped to a servlet defined in the Web deployment descriptor by way of the href attribute on the portlet element. A portlet must define a unique ID that each concrete portlet can reference. Each concrete portlet references a portlet using the given ID in the href attribute of the concrete-portlet element. Figure 2.10 shows our “hello world” portlet deployment descriptor.

```
<?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.portlets.sample.HelloWorld.1">
  <portlet-app-name>Hello world portlet application</portlet-app-name>
    <portlet href="WEB-INF/web.xml#HelloWorldPortlet"
```

Figure 2.10: “Hello world” portlet.xml file (part 1 of 2).

```
               id="com.ibm.portlets.sample.HelloWorld"
               major-version="1" minor-version="0">
       <portlet-name>HelloWorldPortlet</portlet-name>
       <cache>
         <expires>0</expires>
         <shared>no</shared>
       </cache>
       <allows>
         <maximized/>
         <minimized/>
       </allows>
       <supports>
         <markup name="html">
           <view/>
         </markup>
       </supports>
     </portlet>
   </portlet-app>
   <concrete-portlet-app uid="com.ibm.portlets.sample.HelloWorld.1.2">
     <portlet-app-name>Hellow world portlet</portlet-app-name>
     <concrete-portlet href="com.ibm.portlet.sample.HelloWorld">
       <portlet-name>Hello world portlet</portlet-name>
       <default-locale>en</default-locale>
       <language locale="en">
         <title>Chapter 2 Hello world portlet</title>
       </language>
     </concrete-portlet>
   </concrete-portlet-app>
</portlet-app-def>
```

Figure 2.10: "Hello world" portlet.xml file (part 2 of 2).

Many things can go wrong when creating these XML files. You will not find out until you attempt to deploy the portlet into the portal. Here is a handful of things to double-check.

- Verify that for every XML element you have a closing element.
- Check for mistyped class names, especially (for this example) if you named your class something other than HelloWorld.
- Make sure your id and href attributes match.
- Make sure your editor saves the file as text.

Create the WAR File

Finally, we are ready to create the WAR file for distribution. We'll use the standard JAR command to build the WAR file. We run this from the helloWorld directory. Figure 2.11 shows the command for packaging the WAR file under the Windows operating system.

```
> set JAR= C:\WebSphere\AppServer\java\bin\jar
> %JAR% -cf HelloWorld.war WEB-INF
```

Figure 2.11: Packaging the WAR file under Windows.

If you are using Linux, Figure 2.12 shows the commands for packaging the WAR file.

```
$ JAR=/opt/WebSphere/AppServer/java/bin/jar
$ $JAR -cf HelloWorld.war WEB-INF
```

Figure 2.12: Packaging the WAR file under Linux.

Congratulations, you've created your first portlet. The next step is to install and test your portlet. There are several ways of doing this. You can install your portlet and place it on a page using a standalone portal server. By updating your portlet when changes are made, you can continue to refine and debug the functionality contained within your code. If you are new to WebSphere Portal and don't understand the basic administration features like installing a portlet, refer to the WebSphere Portal's online InfoCenter installed with the Portal or on the Web at *http://publib.boulder.ibm.com/wcmid/.* Another resource is the book *IBM WebSphere Portal Primer* by Ashok Iyengar and Venkata Gadepalli.

Another approach is to use WebSphere Studio and the Portal Toolkit that we discussed installing in Chapter 1. It's important to understand that there are multiple approaches to developing portlets, and dependency on one tool can limit your understanding of the process. Our focus on using WebSphere Studio is limited within this book to get you started building portlets. Though these tools are available and fairly easy to use, remember that there are some organizations that have standardized upon their own development IDE or code on Linux or Unix workstations.

BUILDING HELLO WORLD WITH THE PORTAL TOOLKIT

In Chapter 1, we installed WebSphere Studio and the Portal Toolkit. These tools currently come bundled with the portal package so it's easy to incorporate them into your development environment and process. Appendix A discusses in further detail some processes that may offer some help in setting up your portal development shop.

After launching Studio we can begin to create a simple portlet using this environment. As we walk through the process, keep in mind that we are skipping much of the background information on using WebSphere Studio to its full advantage. There are several good books on using Studio that may fill some of those gaps. We recommend *WebSphere Studio Application Developer 5.0* by Igor Livshin and the IBM Redbook (SG246957) *WebSphere Studio Application Developer Version 5 Programming Guide* by Ian Brown, Fabio Ferraz, Maik Schumacher, and Henrik Sjostrand. See *http://www.redbooks.ibm.com* for the Redbook. Figure 2.13 shows WebSphere Studio started up and ready to go.

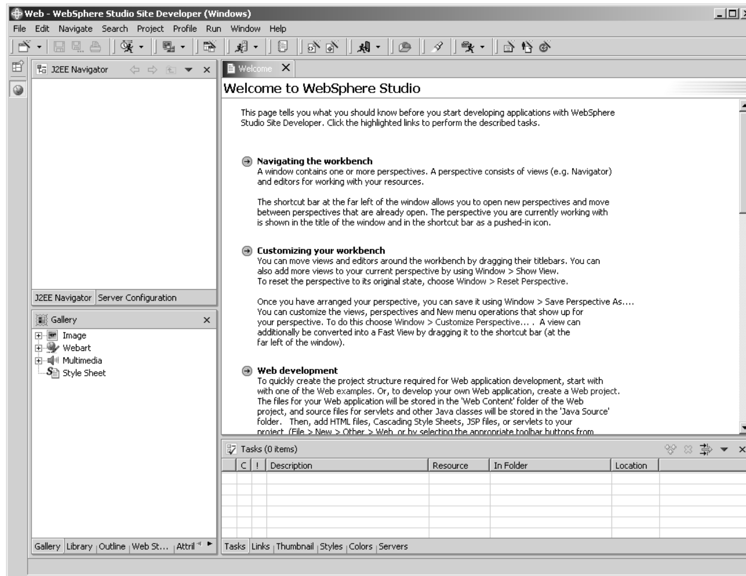


Figure 2.13: WebSphere Studio started up and ready to go.

Once Studio is up and running, we can create new portlet projects by clicking on File->New ->Project. Figure 2.14 shows how you create a project for your portlet in WebSphere Studio.

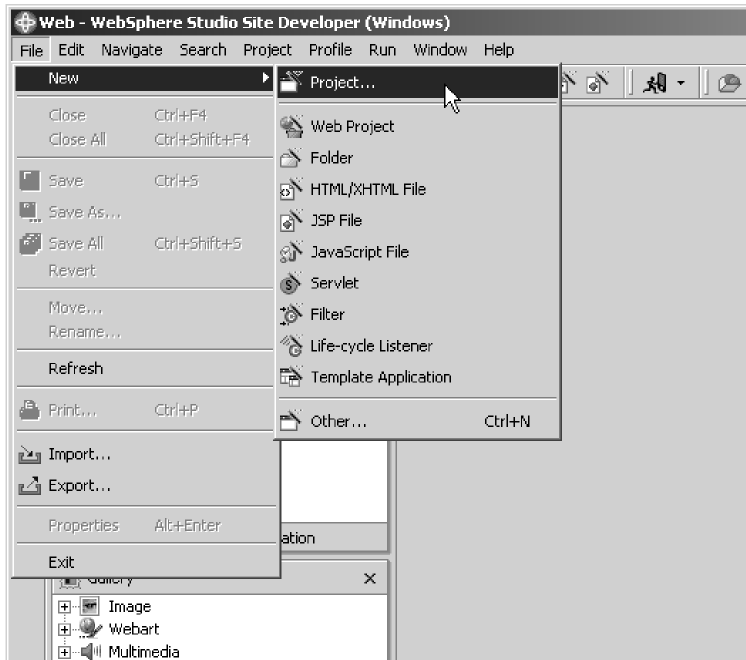


Figure 2.14: Creating a new project.

This will bring up the New Project dialog. The Toolkit provides two basic options for building sample portlets: the Portlet Application Project and the Web Services Portlet. Choose Portlet Development and Portlet Application Project and then click Next. Figure 2.15 shows the different project options for the toolkit.

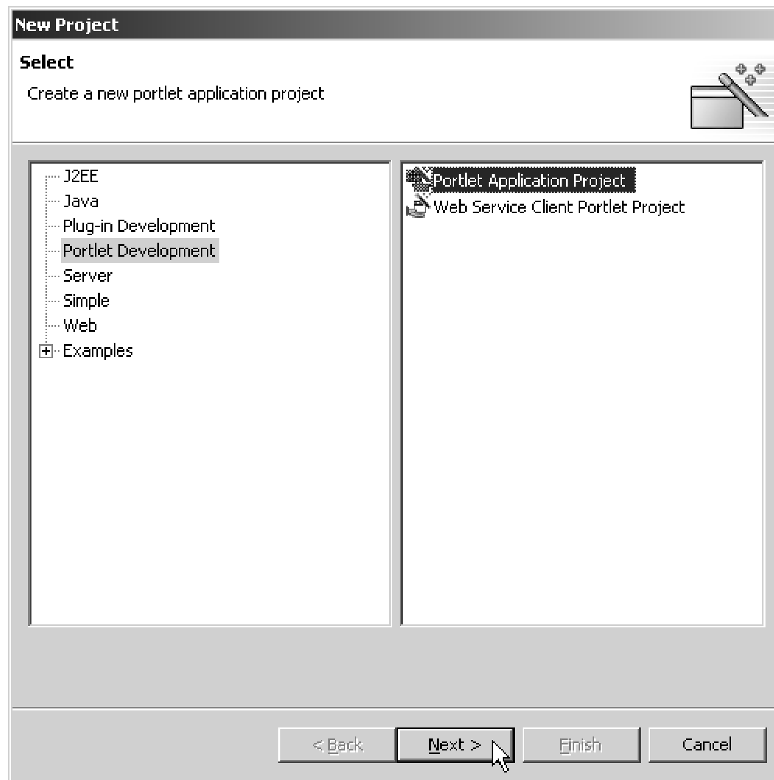


Figure 2.15: Different project options for the Toolkit.

Name your new portlet project "HelloPortlet" and choose the radio button for Create empty portlet. Though we don't cover them in this text, the Toolkit has a number of built-in examples if you choose the Create basic portlet option. As a follow-up, create a different portlet and examine some of the examples that can be built out of that option. Click Next to continue. Figure 2.16 shows the dialog for creating a new portlet project.

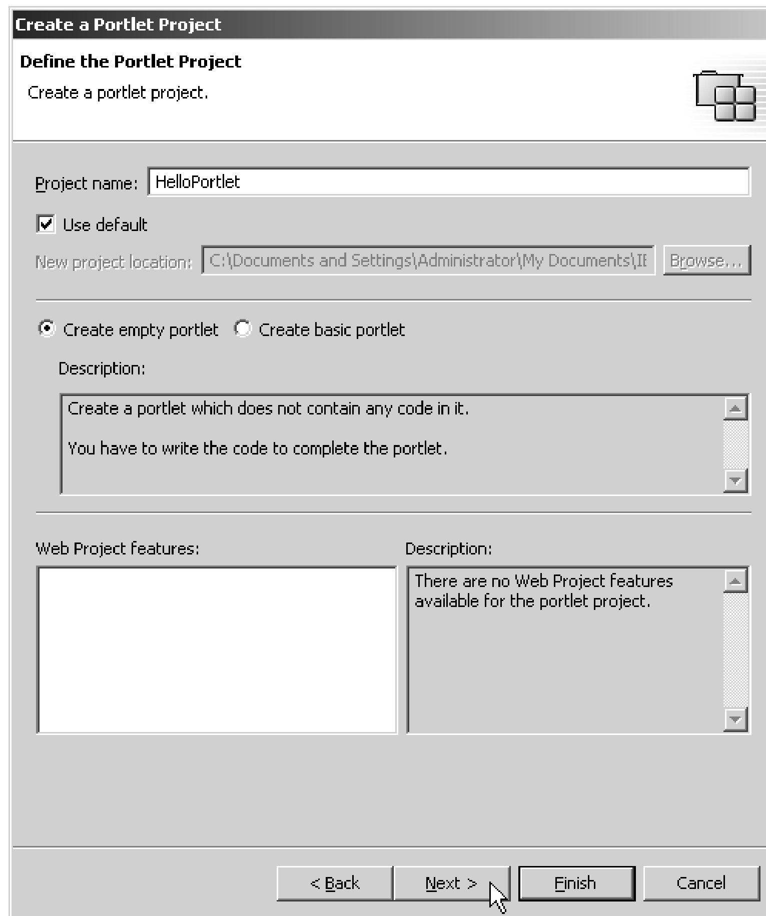


Figure 2.16: Creating the new portlet project dialog.

In the J2EE settings page choose a new Enterprise application project and enter a unique name. Leave the context root and ensure that J2EE Level 1.3 is selected. Click Next to continue. Figure 2.17 shows the J2EE settings page for creating a new portlet project.

Figure 2.17: J2EE settings page.

Edit the portlet settings as necessary. The wizard appends the word "portlet" after many of the portlet names so you may want to change the default "HelloPortletportlet" to just "HelloPortlet." This may also be the case with other names on the settings page. Click Finish when done. Figure 2.18 shows the Portlet Settings dialog.

Figure 2.18: Portlet Settings dialog.

Once the portlet is created you will see the directory structure, and the portlet.xml will be available for editing. With many of the sample portlets that can be created with the wizard the portlet would be ready to run at this point. Because we created an empty portlet, however, we need to code our own portlet controller and identify it in the deployment descriptor files. Figure 2.19 shows our portlet project after creation.

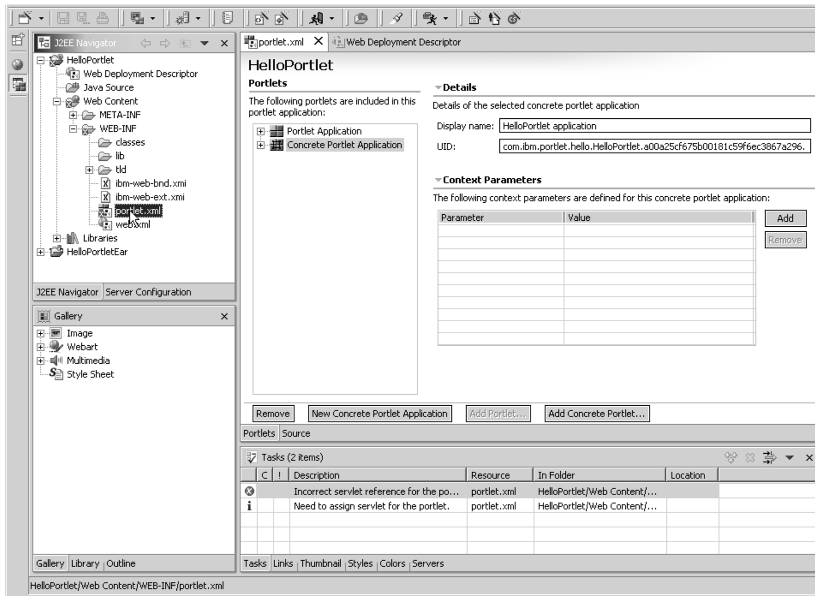


Figure 2.19: Our portlet after creation.

Right-click on the Java Source file and choose to create a new class file. Enter the package name prefix and click Finish. Figure 2.20 shows the Java Package creation dialog.

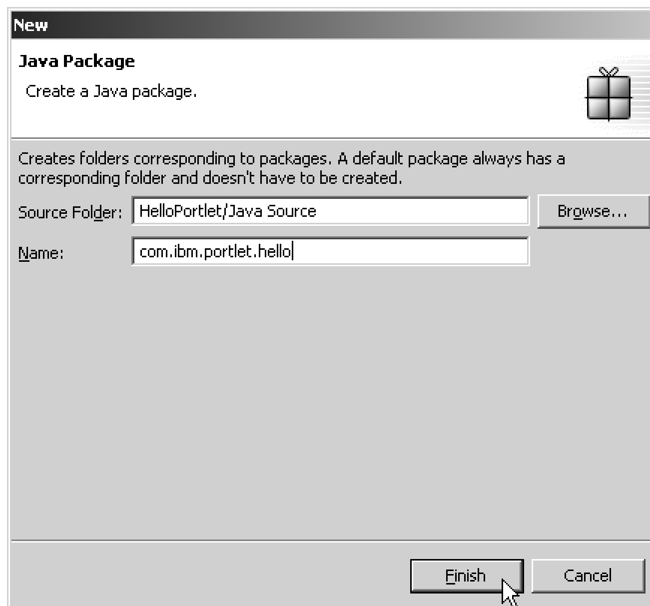


Figure 2.20: Java package creation dialog.

Once the package is created right-click on the package and choose to create a new class file. This will bring up the New Java Class dialog. Enter the name of the portlet controller in the name field and click Finish. Figure 2.21 shows the New Java Class dialog.

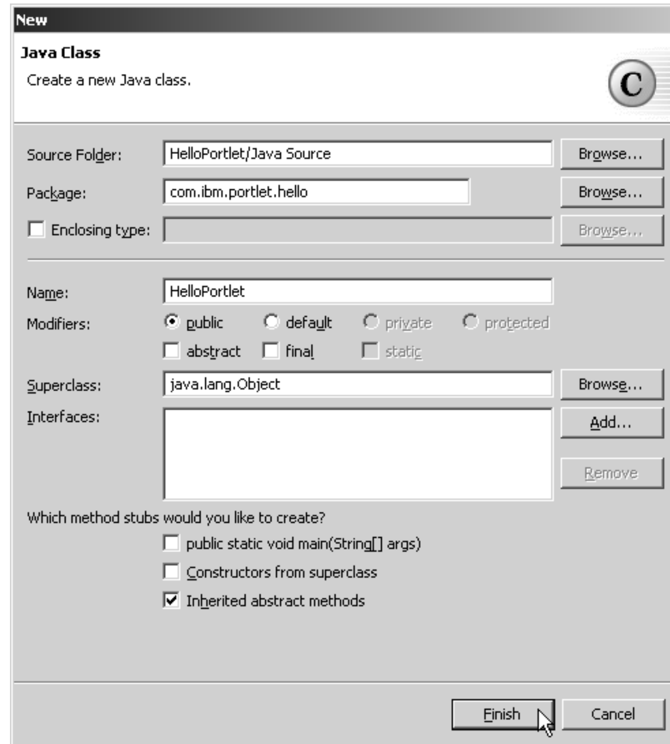


Figure 2.21: New Java Class dialog.

Open the class once it is created and enter the code from the previous example into the class file. Save the file and ensure that there are no errors. Figure 2.22 shows the code entered from the previous example.

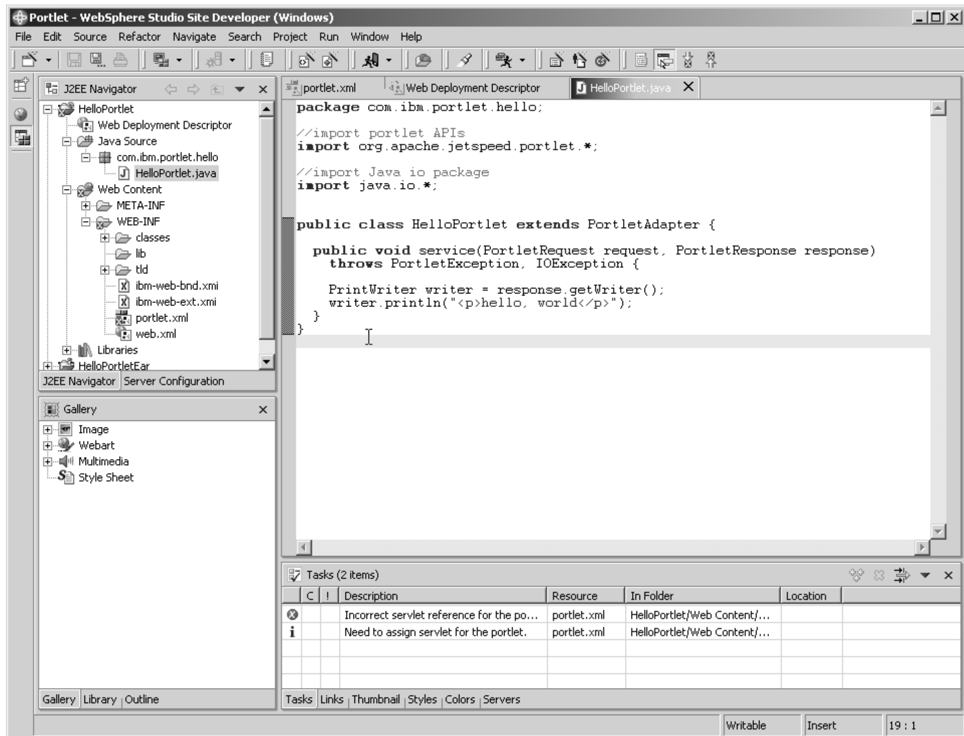


Figure 2.22: Coding the portlet.

Our portlet is almost complete. Because the portlet controller simply outputs a simple text string, the portlet itself is not very complicated right now. Later, as we add JSP files and more complex output, this will be more important.

Double-click on the web.xml file in the navigation window to bring up the editing interface for this file. On the Overview tab of our Web Deployment Descriptor we need to set up the controller class to be called when our portlet is run. Click on the Details button. (This can also be accomplished by clicking on the Servlets tab at the bottom of the screen.) Figure 2.23 shows the Web deployment descriptor editor.

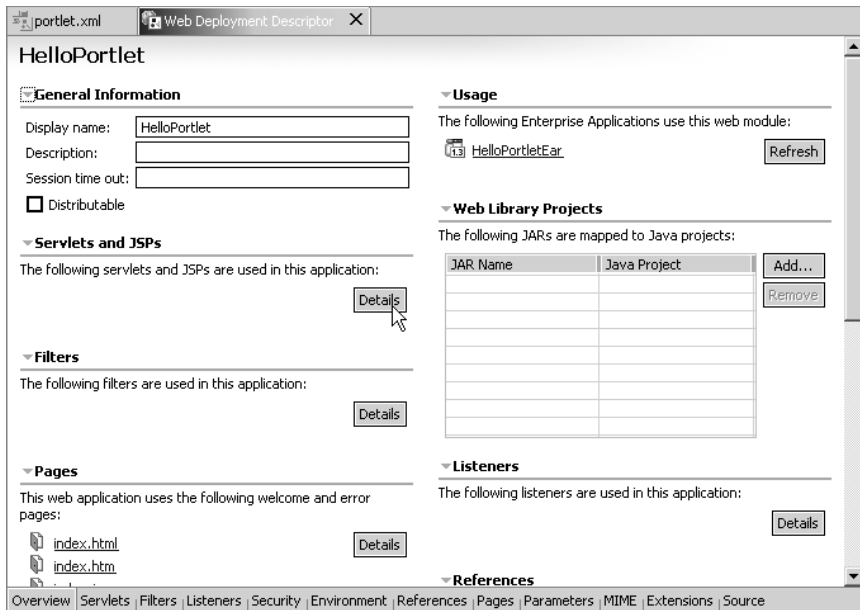


Figure 2.23: Web deployment descriptor editor.

In the Servlets detail screen click on the Add button at the bottom of the Servlets and JSPs section. Figure 2.24 shows the Servlets and JSPs section of the Web deployment descriptor editor.

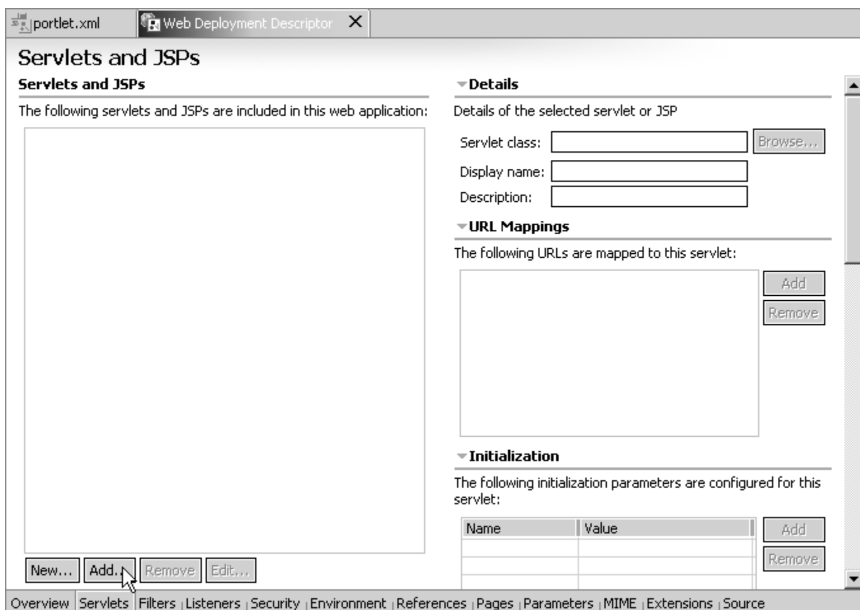


Figure 2.24: Servlets and JSPs section of the Web deployment descriptor editor.

To add the class that we created earlier in this chapter, choose the HelloPortlet class and then click OK. Double-check the qualifier to ensure that the package name is the same as the one you defined for your portlet. Figure 2.25 shows the Add Servlet or JSP dialog.

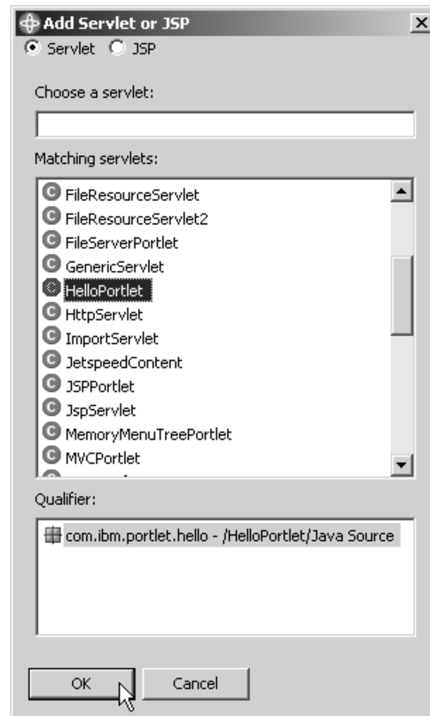


Figure 2.25: Add Servlet or JSP dialog.

Now that you have added the portlet class, highlight the HelloPortlet class in the list and add a URL mapping for this portlet. Under URL Mappings click Add. The mapping should be provided automatically. Double-check that the mapping is similar to /HelloPortlet/*. Once this is complete you can save and close the web.xml file. Figure 2.26 shows the completed Web deployment descriptor, as seen in the editor.

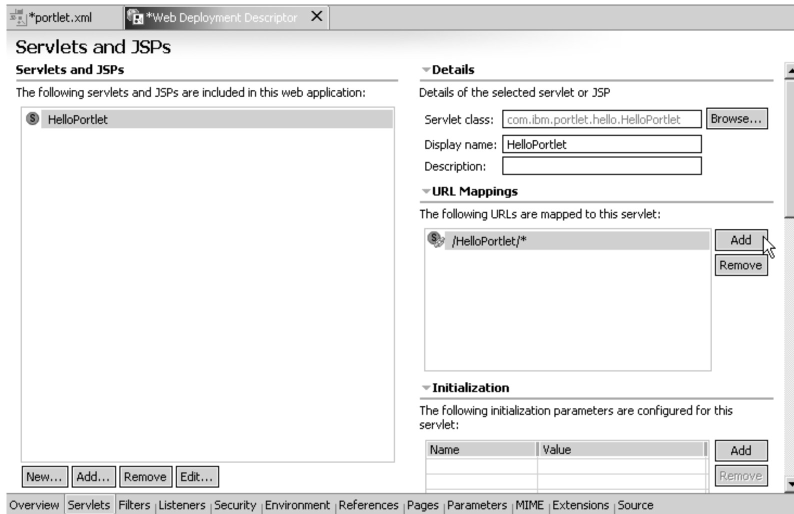


Figure 2.26: Web deployment descriptor editor.

Once the portlet controller has been set up within the Web deployment descriptor, it is necessary to set up the same controller within the portlet.xml or portlet descriptor file. Open the portlet.xml by double-clicking on the file in the navigator window.

Under the portlets section click the browse button next to the Servlet field. Figure 2.27 shows the portlet deployment descriptor editor.

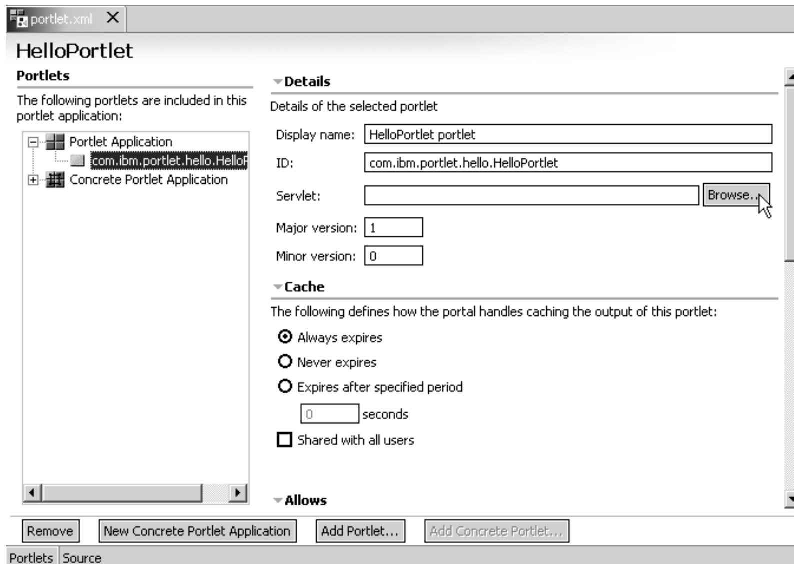


Figure 2.27: Portlet deployment descriptor editor.

The Select Servlet window dialog will appear with a list of the currently available servlets. Choose HelloPortlet and click OK. Figure 2.28 shows the Select Servlet dialog.

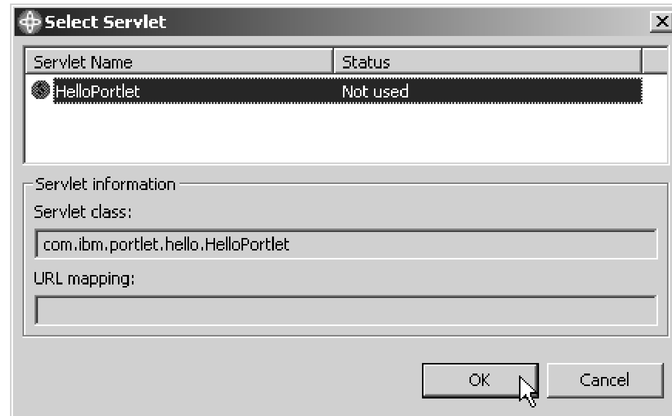


Figure 2.28: Select Servlet dialog.

Once the servlet is mapped within the portlet.xml, you can save and close the portlet descriptor as well. You should not see any more errors within your task window for this portlet. Rebuild your portlet using the Project menu at the top of the screen. This will revalidate your recent changes and ensure that the portlet is ready to run.

RUNNING YOUR FIRST PORTLET IN WEBSPHERE STUDIO

The pleasure of working and developing within WebSphere Studio is that it provides a development and test environment from a single-user interface. In Chapter 1 we installed the portal toolkit with the Portal runtime functionality. This provides a complete ready-to-run environment for testing your portlets. Once you have a portlet running within the environment, you can make changes and retest without restarting or redeploying the code. Studio will pick up your saved changes and refresh the portlet in real time. To run the "hello world" portlet, right-click on the portlet and choose the Run on Server… option. Figure 2.29 shows the menu.

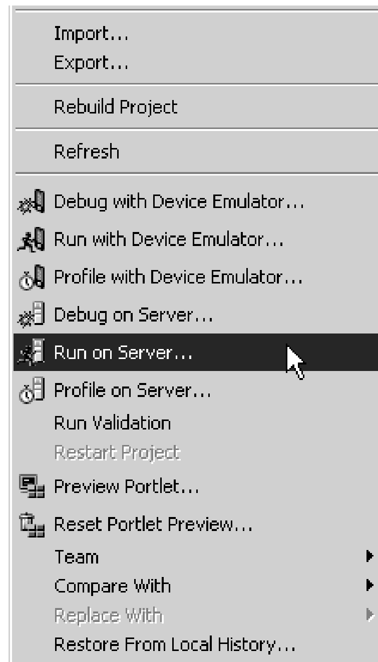


Figure 2.29: Menu to run the portlet.

Because this is your first time running a portlet in the test environment, Studio will prompt you to create a new test server to run your code in. Choose a new server of type WebSphere Portal v5.0 Test Environment and click OK. Figure 2.30 shows the Server Selection dialog.

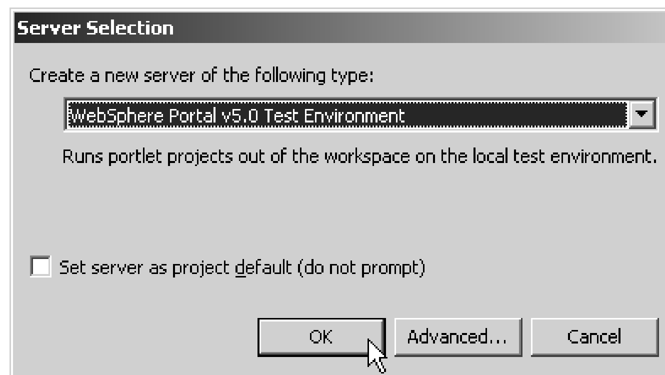


Figure 2.30: Server Selection dialog.

Studio will create and start your server for you. Once the server is running the portlet will be shown in the built-in browser window. You can use an external browser by pointing to the URL with the correct port number (:9081) as shown in Figure 2.31.

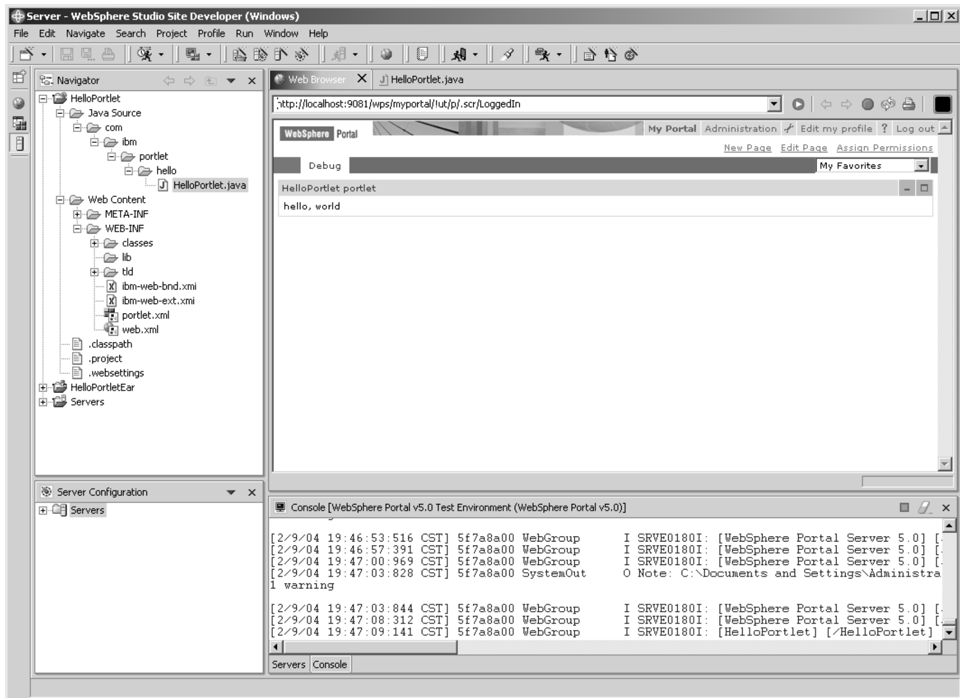


Figure 2.31: “Hello world” portlet running.

Below the browser window is the server console, where error and output messages from the server are displayed. This is a useful tool when looking for bugs in your code or for outputting messages to track the progress of a portlet.

EXAMINING HELLO WORLD

We just built a very simple portlet using a couple of different methods. Now let’s go back and take a closer look at the portlet we’ve written and what some of the code actually does. The controller starts off by importing the org.apache.jetspeed.portlet package and the Java IO package.

```
//import portlet APIs
import org.apache.jetspeed.portlet.*;

//import Java io package
import java.io.*;
```

We actually are going to use four different classes from the org.apache.jetspeed.portlet package.

The first class we see in our portlet is the PortletAdapter class. This is the class our portlets should extend and use to override the service() method or the doView() method to produce output.

```
public class HelloWorld extends PortletAdapter {
```

Our HelloWorld portlet extends the PortletAdapter, which in turn extends the abstract Portlet class. Every portlet must implement the abstract Portlet class either by extending it directly or indirectly through a subclass of Portlet. We suggest that you build your portlets not to extend the Portlet class directly, but rather to extend one of its subclasses, such as PortletAdapter. PortletAdapter provides a few convenient methods that will help you as you develop more complex portlets. We override one method of the PortletAdapter class, the service method. The service method is called by the portlet container with a PortletRequest and a PortletResponse when the portlet needs to generate output. In our case, we are generating a fragment of HTML, which we write to a PrintWriter.

PortletRequest, PortletResponse, and Portlet Exception are used in the service or doView() methods of our portlet controller.

```
  public void service(PortletRequest request, PortletResponse response)
   throws PortletException, IOException {
```

The PortletRequest extends the HttpServletRequest and represents the client's request to the portlet. We use the PortletRequest in very much the same way that we would use the HttpServletRequest. We can get and set attributes, get parameters, get the details of the client device, and so on and so forth. For this simple portlet we are not using any of the information provided to us by the PortletRequest.

The PortletResponse extends the HttpServletResponse and represents the response sent to the client. We use this object to get the PrintWriter with the getWriter method. Our portlet uses this PrintWriter to write markup into the response for our client. The PortletResponse gives us other methods for determining what sort of markup we should be generating, the character encoding we should be using, the methods to create URLs that point back to our portlet, and a handful of other helper methods.

The service method can throw a PortletException. The PortletException class is a general exception that can be thrown by the portlet when it experiences problems executing. For our simple portlet we leave it to our super classes to throw this exception if needed.

Those are the base classes one needs to know about in order to begin writing portlets. In this chapter we've learned how to write our first portlet using a handful of classes. With

these classes we can write a wide range of portlets. We've also learned which JAR files are needed to compile our portlets, what deployment descriptors are needed, and how to package our portlets for deployment. In the next chapter we'll expand on these skills to create a more sophisticated portlet.

CONCLUSION

Reading through this chapter we have basically created a simple portlet twice. In reality most readers will probably focus on one path, depending upon their skill and environment. There is benefit, however, in looking at how it might be done in a different environment. If you use only WebSphere Studio to do your development, you might not fully understand the process of using commands on the OS to compile and package your components. On the other hand seeing how easily some tasks are accomplished in an IDE such as Studio could persuade some to switch to this type of environment.

One area that we haven't described in detail is packaging a portlet that has been successfully tested for deployment into another environment. WebSphere Studio makes this step very easy by providing a WAR creation wizard that can be used on your project. The following steps can assist in this process:

1. Right-click on your project in the navigation window and choose Export.
2. In the Export window choose WAR File and click Next.
3. On the WAR Export screen, verify the name of the project and type in the name and folder location where you want the WAR file exported to. Click Finish.

Your WAR file will be exported to this location and will be ready for deployment into any portlet server. Because you now understand how to build and package your portlets, the following chapters will guide you deeper into aspects of building WebSphere Portal portlets that can fully take advantage of the environment and create successful portals. Have fun!