Oracle JDeveloper 10*g*: Empowering J2EE Development

HARSHAD OAK

Apress™

ISBN (pbk): 1-59059-142-9

Printed and bound in the United States of America 12345678910

Technical Reviewer: Kenneth Cooper Jr.

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steven Rycroft, Dominic Shakeshaft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown Collins

Copy Editor: Kim Wimpsett, Brian MacDonald

Production Manager: Kari Brooks

Production Editor: Janet Vail

Proofreaders: Elizabeth Barry and Patrick Vincent

Compositor: Kinetic Publishing Services, LLC

Indexer: Valerie Perry

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 5

Servlet and JSP Development

This chapter introduces Servlets and Java Server Pages (JSPs) and then presents a simple example to display how to use JDeveloper in creating Servlets and JSPs.

JDeveloper provides excellent support for the Apache Struts framework, which I believe is the most popular Web framework around. I will delve into how you can easily create Struts-based JSPs and other files of relevance to Struts. Some of the most useful enhancements to JDeveloper introduced in version 10*g* are related to JSP and Struts development.

The thing about Servlets, and to a certain extent JSPs, that I like the most is that they represent an age when Java was meant to be simple. Before the dawn of Enterprise JavaBeans (EJBs) and the *n* number of application programming interfaces (APIs) that exist today, Servlets and JSP kept things nice and simple. You did not need to read and reread books to try and comprehend the subject and still be left wondering if you have really understood it.

There are many good things about EJBs, but if misused, they can be a real pain. A majority of projects do not need to venture into EJB territory. Servlets and JSPs used in tandem with ordinary Java classes can do a far cleaner and faster job for you.

Web Applications

Before you get into Servlets and JSPs, you need to understand the "why" and "what" of Web applications. Even Java 2 Enterprise Edition (J2EE) Web applications that do not delve into EJB territory are not always easy to comprehend. There are predefined directory structures, specific locations where libraries or code need to be kept, and key files that provide the container with deployment instructions. So although integrated development environments (IDEs) such as JDeveloper can make it a cinch to create and deploy Servlets and JSPs, if you are not aware of the intricacies of Web applications, it would be rather difficult to understand how the whole thing works and how to fiddle with it.

All containers that support deployment of J2EE applications need to adhere to certain rules and expect a Web application to be in a certain format. Unless the right things are in the right location, your application will not work.

I will show an example of a simple Web application named *simplewebapp* that you will create. If you presume that this application will have one Servlet and a Hypertext Markup Language (HTML) page, the most likely Web application structure would look like that depicted in Figure 5-1. Here the directory simplewebapp is the top-level directory and can hold any files that are meant to be publicly accessible, either directly under it or under a subdirectory other than WEB-INF. So if you create an HTML file named *index.html* and want the file to be accessible publicly, you can place the file either directly under the simplewebapp directory, or you could create a new directory by any name other than WEB-INF. The files placed here are generally HTML files, image files, and JSPs.

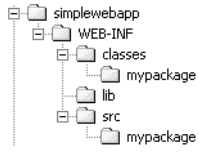

Figure 5-1. Web application structure for simplewebapp

The WEB-INF directory is the directory that holds the meat of the application, and the contents of this directory and its subdirectories are not publicly accessible when deploying the application. Public and private access does not mean that the files cannot be read or modified on the server. Access is only referring to a user trying to access the application after it has been deployed on a server. The WEB-INF directory has three elements, discussed next.

classes

The classes directory under the WEB-INF directory holds the compiled Java files. As shown in Figure 5-1, there exists a mypackage directory under the classes directory. This directory is where the class file for a servlet or Java class mypackageTest Servlet would have to be placed. The directory structure under the classes directory is strictly based on the package names used for the Java files involved. Normal usage is that during development, class files are maintained in the classes directory because the server can be configured to keep checking for changes to this directory and automatically update the reference if it finds that the file has changed. This can be a useful feature because you do not need to restart the server for any changes to be reflected.

lib

The lib directory holds packaged Java archive (JAR) files. The big advantage of the classes and the lib directory is that they eliminate the need to fiddle with the

classpath for the container to be able to locate any of the requisite classes. Unlike the classes directory, the container does not check for changes to this directory; and so if a new JAR file is placed, the container needs to be restarted for changes to be reflected. The lib directory is useful when you are using third-party code in your application such as an Extensible Markup Language (XML) parser or a Java database connectivity (JDBC) driver. In this case, you can just dump the JAR file into the lib directory and get going.

web.xml

The deployment descriptor for a Web application needs to be named *web.xml* and should be placed right under the WEB-INF directory. The web.xml file is the only place where you can provide the container with deployment instructions specific to a particular Web application. A web.xml file can specify the following:

- **Servlet declarations**: Declare names and classes for key Servlets in the application. All Servlets used in the application do not need to be declared. Servlet mappings use the names specified in the declaration.
- **Servlet mapping**: Specify the Servlet to invoke when encountering a specific uniform resource locator (URL) pattern.
- **Session configuration**: Specify session configuration details such as timeout period.
- **Listeners**: Specify classes meant to listen for the occurrence of specific events.
- **Tag libraries**: Declare the custom tag libraries being used in the application.
- **Welcome file list**: Welcome files are files that are to be used even if only the context root is specified in the URL and no file is mentioned.
- **MIME types**: Specify the MIME types that are to be used in the application.
- **Filters**: Specify any filters used in the application. Servlet API 2.3 introduced the concept of filters.

Context Root

The context root defines how the application will be accessed. So the simplewebapp as shown in Figure 5-1, although would be generally accessed using a URL such as http://localhost:8080/simplewebapp, can very well be

accessed using a URL such as http://localhost:8080/toughwebapp. Here toughwebapp is the context root that you have set to access simplewebapp using a different name. The context root configuration procedure differs between various J2EE servers.

Now that you have looked at concepts that are relevant across all J2EE Web applications, you will move into the specifics of Servlets and JSPs.

Servlets

Until a few years back, applets were the most popular feature of Java. So the first definition of Servlets that I came across was "Servlets are applets that run on the server." Although I did not understand it much, I thought that would be pretty cool. I would now say that a Servlet is Java code that runs on the server where the server does all the hard work of networking and other underlying tasks involved in processing client requests and sending an appropriate response. The Servlet code that you write sits at a higher level and deals with just application-specific tasks. The Servlet is concerned more with picking up input and generating output; there is no messing with protocols, sockets, and networking stuff.

Servlets are a great attempt to free application developers from having to worry about things they really should not be worried about. Like many of the other Java APIs, the Servlet API is also just a set of interfaces. The API provides no implementation, and it is solely up to the server vendor to actually code the implementation that will get the result as stated in the API. So the way an Oracle server internally provides Servlet support can be very different from, say, the Apache Tomcat server. However, the developer is just concerned with the Servlet API and does not need to worry about how the vendor complies with the API. As long as the developer is told that X server is Servlet API 2.3 compliant and Y Server is also compliant with that version of the API, he can rest assured that his code will work on both servers and get the same result. Because this same concept applies to all J2EE APIs, you ideally do not need to worry about vendors and implementations as long as you strictly adhere to the API.

A Servlet is pure Java, so everything that can be done with an ordinary Java class can be done using Servlets. A Servlet has to implement the javax.servlet.Servlet interface, either directly or indirectly by extending a class that implements the interface (see Table 5-1). The basic handling expected from a Servlet has to be specified in the methods of the interface that are implemented in the Servlet (see Table 5-2).

Table 5-1. Key Interfaces

Name	Description
javax.servlet.Servlet	This interface defines the life-cycle methods for a Servlet to initialize the Servlet, service requests, and then destroy the Servlet.
javax.servlet.ServletConfig	This interface defines methods meant to provide information about the Servlet such as its initialization parameters, name, and so on.
javax.servlet.ServletContext	This interface defines methods that can tell the Servlet more about the environment it is being run in. The Servlet can use this to get information such as initialization parameters for the application or the container being used.
javax.servlet.httpHttpServletRequest	This Hypertext Transfer Protocol (HTTP)–specific interface extends the javax.servlet.ServletRequest interface and defines methods to retrieve information from the request received.
javax.servlet.http.HttpServletResponse	This HTTP-specific interface extends the javax.servlet.ServletResponse interface and defines methods to create an appropriate response to a request received.
javax.servlet.http.HttpSession	A HTTP-only interface that provides ways to track a user across multiple requests and share information across a particular session.

Table 5-2. Key Classes

Name	Description
javax.servlet.GenericServlet	This class provides a generic Servlet that is independent of any particular protocol. The abstract method service needs to be implemented by any subclass of the GenericServlet to be able to process requests.
javax.servlet.http.HttpServlet	This is a class meant specifically for HTTP. It provides many method implementations that can make HTTP request handling a lot easier for any of its subclasses. HttpServlet extends GenericServlet.

For a Servlet to be independent of protocol, you can extend the javax.servlet.GenericServlet class and provide your implementation in the service method overridden in your Servlet. However, because Servlets are generally used to communicate over HTTP, the common usage is of extending the javax.servlet.http.HttpServlet class and, based on the nature of the request, overriding the appropriate methods. You will stick to extending HttpServlet for all these examples.

For the various types of HTTP methods, there are corresponding methods in the HttpServlet interface. The methods are doGet, doPost, doHead, doPut, doTrace, doOptions, and doDelete. Because GET and POST are the two most commonly used HTTP methods, the doGet and doPost methods are the most important ones provided by the HttpServlet class:

GET: A GET request is where the request information is sent as part of the URL. The request parameter names and their values appear as part of the URL. In the URL http://localhost:8080/TestServlet?paramOne=Oracle&paramTwo=JDeveloper, the two request parameters are paramOne and paramTwo and their values are Oracle and JDeveloper, respectively. Note the use of the ? and the & sign. The doGet method of the Servlet is automatically invoked when the Servlet encounters a GET request.

POST: In a POST request, the data is sent as part of the message body and not as part of the URL. POST requests are generally encountered in case of HTML form submissions. The values of the form fields are sent usin0g POST and do not appear as part of the URL. The doPost method of the Servlet is automatically invoked when the Servlet encounters a POST request.

Most Servlets do not go beyond the interfaces and classes stated in Table 5-1 and Table 5-2. A good understanding of these is essential to understand how Servlets work. This is especially true when you use advanced tools such as JDeveloper because the tool makes creation so simple that people often fail to understand the underlying fundamentals.

TIP *Spend some time understanding the key Servlet interfaces and classes before you leap into Servlet creation using JDeveloper. There are few classes involved, and the Javadocs can be very useful.*

New Web Module and Servlet

You will now dive into an example where you will create a Web application and a basic Servlet that will be a part of the application. The Servlet will accept a first name and last name as input and flash a welcome message.

Before you create anything in JDeveloper, you need a workspace and project. In this case, you will use the workspace MyJavaApps that you created in Chapter 3. You are free to use any other workspace. No settings for the workspace are relevant to this project. You will next create a new project meant specifically for Web applications. Follow these steps:

1. Select the MyJavaApps workspace, and select the New option to take you to the New Gallery, as shown in Figure 5-2.

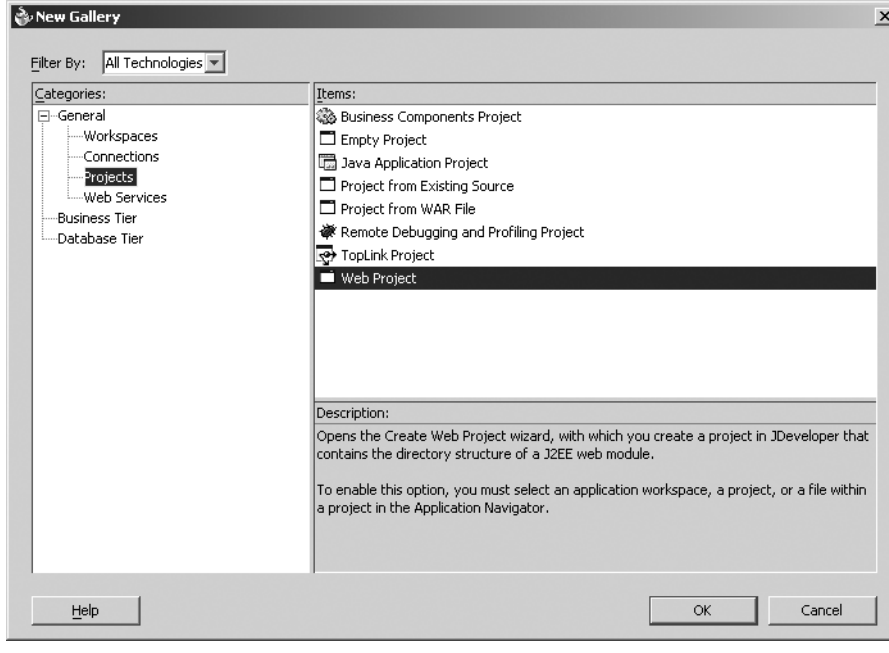

Figure 5-2. Creating a new Web project

2. In step 1 of the wizard that follows, name the project and the directory *BasicWebApp*.

3. In step 2, as shown in Figure 5-3, the wizard populates a default document root, Web application name, and context root. The document root directory specified will serve as the base reference that the container will use to access files in the Web application. Change the context root to *basicwebapp*. You need not change any of the other settings.

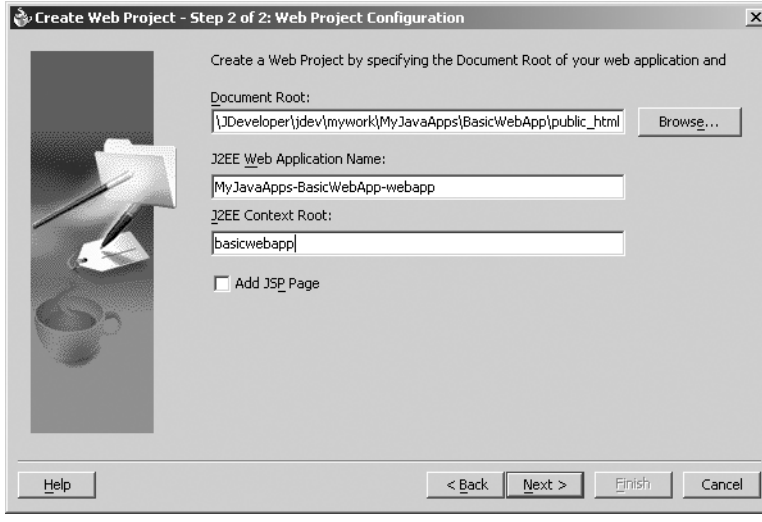

Figure 5-3. Configuring the Web project

4. Click Next.

5. Click the Finish button.

Web Module Project Settings

Now that you have created a new Web module, you need to look at the project properties relevant to a Web module. In the Project Properties dialog box, choose the Common ➤ Input Paths option. In the screen shown in Figure 5-4, change the default package to *com.jdevbook.chap5*. Note the directory where the Java source path is maintained and the HTML root directory. This directory forms the base for any links HTML, JSPs, and images. You do not need to change these settings.

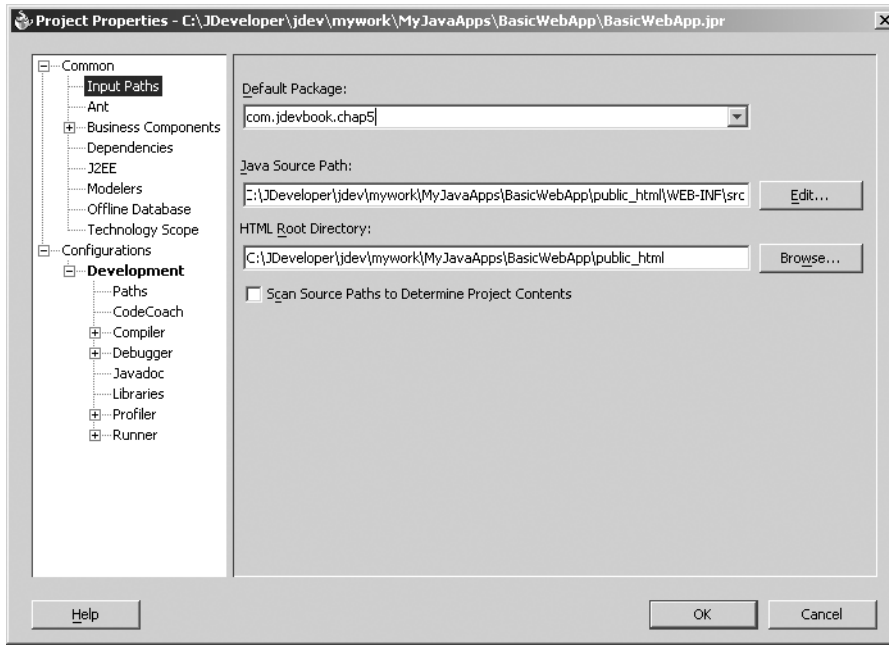

Figure 5-4. Setting the project input paths

NOTE *Naming the Java source directory* src *and having it under the WEB-INF directory is a matter of convention. The specifications do not force the name* src *or the location because after deployment, the location of the source code is immaterial to the container.*

Next, choose the option Common ➤ Technology Scope in the project's properties. In the screen shown in Figure 5-5, you can see the Technology Scope page for a Web project. Because you only intend to use a Servlet, you do not need to make any change to the technology scope of the project.

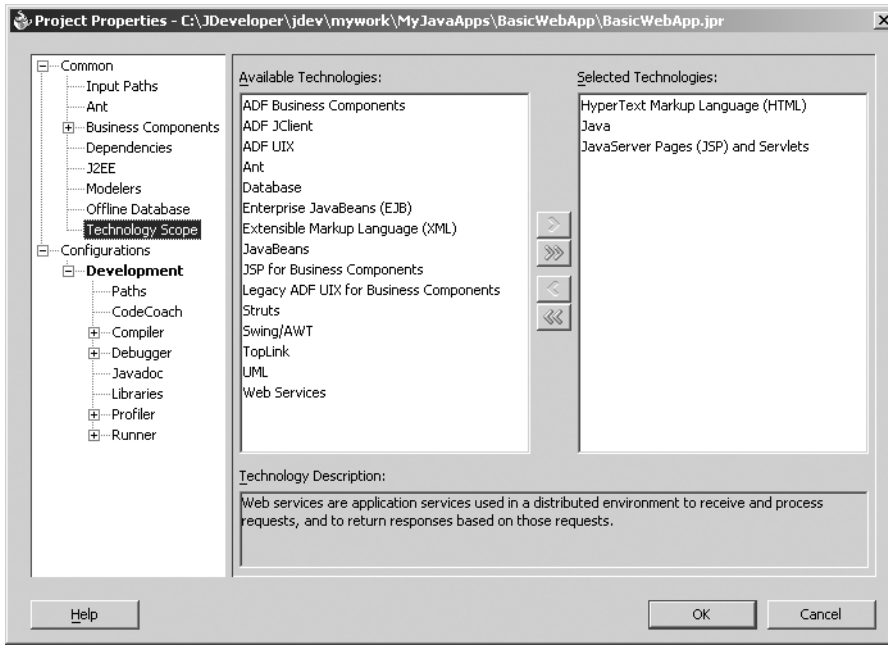

Figure 5-5. Setting the project's technology scope

You now have a proper Web module in place and can proceed to add Servlets and JSPs to the application. Select the project BasicWebApp.jpr in the Applications Navigator or System Navigator, and select the New option either by right-clicking or by using the File menu. In the New Gallery displayed, move to the Web Tier section. Here, select the Servlet option. You will find that JDeveloper is capable of creating a new HTTP Servlet, Servlet filter, and Servlet listener. Choose HTTP Servlet, and click OK.

Servlet Creation and Deployment

In step 1 of the wizard, enter the Servlet name as *WelcomeServlet*; the package name should be the same as the default package you stated in the project settings, *com.jdevbook.chap5*. As shown in Figure 5-6, check the Generate Header Comments option to have JDeveloper insert comments in the generated Servlet code. Choose doGet as the only method to be implemented.

NOTE *The Single Thread Model option will get the Servlet class to implement an empty marker interface named* javax.Servlet.SingleThreadModel. *Servlets by default use multi-threading so as to have one instance of the Servlet handling multiple requests. However, if you want one instance to process only one request at a time, implementing SingleThreadModel serves the purpose. Such usage is not advisable and is unnecessary in most cases.*

Note the various options displayed in the Generate Content Type drop-down list. Choosing WML or XHTML will only change the content type for the response sent and the default tags generated by JDeveloper. For this example, you will stick with HTML.

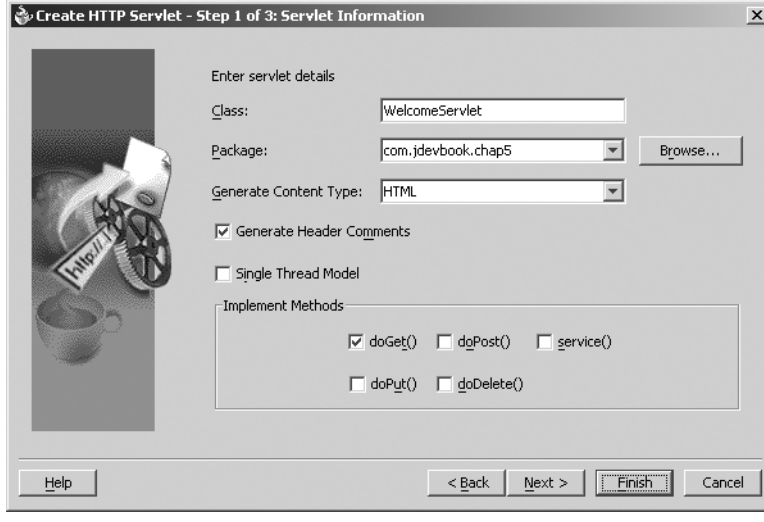

Figure 5-6. Create HTTP Servlet Wizard, step 1, setting the Servlet information

Step 2 of the wizard is what drastically cuts down the amount of code you need to write. The details specified in Figure 5-7 convey that you will be getting two parameters named *fname* and *lname* as part of the request. The value of these parameters will be picked from the request and stored in two String variables named *strFname* and *strLname,* respectively. Note the Type drop-down list; if you specify a type other than String, JDeveloper will also create code to convert the data type from the String received in the request to, say, an int or double. Enter these details, and click Next

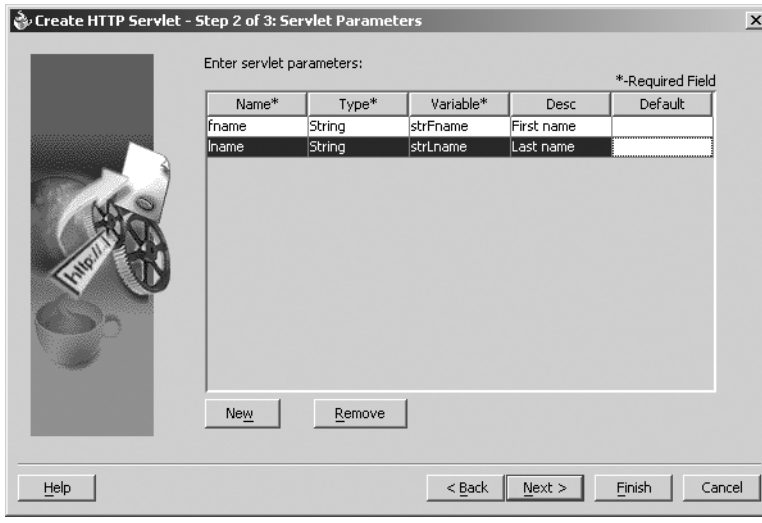

Figure 5-7. Create HTTP Servlet Wizard, step 2, setting the Servlet parameters

In step 3, shown in Figure 5-8, you can provide the mapping information. By specifying a mapping, you will be able to access the WelcomeServlet using the pattern you specify. Here, specify the URL pattern as **.welcome*. In this case, it means that any request that matches the pattern **.welcome* will be directed to the WelcomeServlet. This setting will be reflected in the Web deployment descriptor. So requests in the form of <context root>/abcd.welcome or <context root>/.welcome will be directed to the WelcomeServlet. Click the Finish button, and voilà! You have your Servlet code as well as the Web deployment descriptor prepared for you.

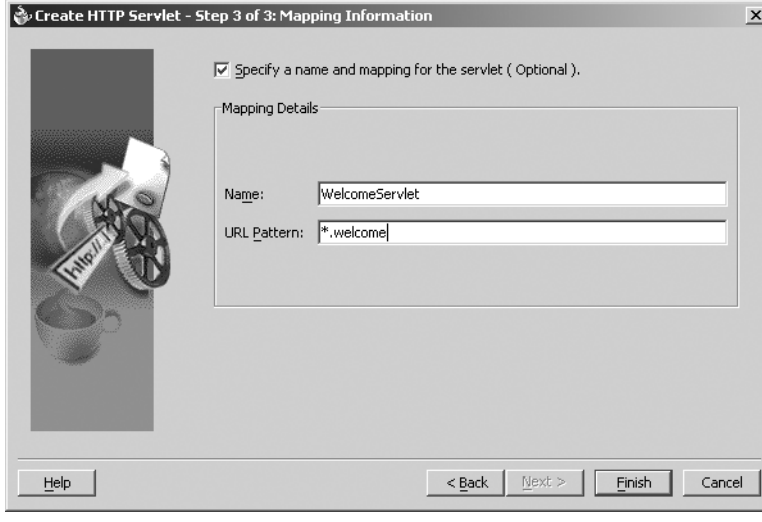

Figure 5-8. Create HTTP Servlet Wizard, step 3, providing mapping information

The files web.xml and WelcomeServlet.java will be listed in the System Navigator. Open the web.xml file to check how the various steps in the Create HTTP Servlet Wizard have led to specific tags being inserted in the file. The file WelcomeServlet.java is almost in its final form. Open the file WelcomeServlet.java in the Code Editor. Apart from the code to get request parameters, even the basic HTML generation code is in place. The only change you will make is to insert the following line:

```
out.println("<p>Welcome "+strFname+" "+strLname+"</p>");
```

This line will create a new paragraph in the HTML page displayed and flash a welcome message using the first and last name fetched from the request. So the HTML generation code looks like this:

```
out.println("<html>");
out.println("<head><title>WelcomeServlet</title></head>");
out.println("<body>");
out.println("<p>The Servlet has received a GET. This is the reply.</p>");
//Use the First and Last name received as part of the request.
out.println("<p>Welcome "+strFname+" "+strLname+"</p>");
out.println("</body></html>");
out.close();
```

Next, choose the Run WelcomeServlet.java option, either by using the Run menu or by right-clicking the file listing in Applications Navigator or System Navigator and choosing that option in the pop-up list. JDeveloper will compile and deploy the code to the embedded OC4J server that I will talk about in the next section. Your default browser will also be executed and the URL that will be accessed is http://127.0.0.1:8988/basicwebapp/*.welcome. As you probably have figured out, this URL is a result of the context root and the mapping you specified during Servlet creation. Replace the * with anything from *abcd* to *onomatopoeia,* and you will still get to the same page, courtesy of the mapping you specified.

NOTE *If required, you can configure the Web browser and proxy settings from the Web Browser/Proxy section in the Preferences dialog.*

Note that the message displayed on the page is “Welcome null null.” This is because the parameters are not part of the request, so the call to request.getParameter() returns null.

NOTE *The default HTTP method is GET. So even if no request parameters are part of the URL, the doGet method gets invoked.*

To display the expected message, you need to change the URL to pass the request parameters. Try changing the URL to http://127.0.0.1:8988/basicwebapp/tennis.welcome?fname=Boris&lname=Becker. You should now have a message welcoming Boris Becker on the HTML page.

Embedded OC4J

An important feature of JDeveloper is the Oracle Application Server Containers for J2EE (OC4J) server that is embedded in JDeveloper. The server, being embedded and tightly integrated into JDeveloper, eliminates the setup and configuration time involved in setting up the server. Rapid deployment and ease of use are also benefits that cannot be ignored. The OC4J server that is part of JDeveloper comes with EJBs and a Java Servlet container along with a JSP translator and a JSP runtime.

Running Web applications, debugging them, profiling them, and using code enhancement tools such as CodeCoach on them is all made possible using embedded OC4J. Because production-level deployment is not the purpose of the embedded OC4J container, it does not provide too many configuration possibilities within JDeveloper.

The embedded OC4J server is automatically started when running, debugging, and profiling or while using CodeCoach on a J2EE component such as an EJB, Servlet, or a JSP. To stop OC4J, you can terminate the process from the Run manager using the View ➤ Run Manager option.

NOTE *Only one instance of the embedded OC4J server can be run at a time. If you try to start another instance, the earlier instance gets terminated.*

OC4J configuration until version 9.0.3 was quite straightforward; however, with version 9.0.5, things have changed. OC4J configuration now has a separate tool listed under setting when choosing Tools ➤ Embedded OC4J Server Settings, as shown in Figure 5-9. OC4J configuration provided through this interface is stored in a file named *server.xml* that can even be edited manually.

In most scenarios, you will not have to tamper with any of the default settings of the server. However, if you want to use an OC4J installation other than the one bundled with JDeveloper, you could specify the directory for that installation. The change of port options can prevent port conflicts with other running instances of OC4J or a different server. The Load All System JARs and Lookup All EJBs During Startup options could slow down server startup and are not recommended.

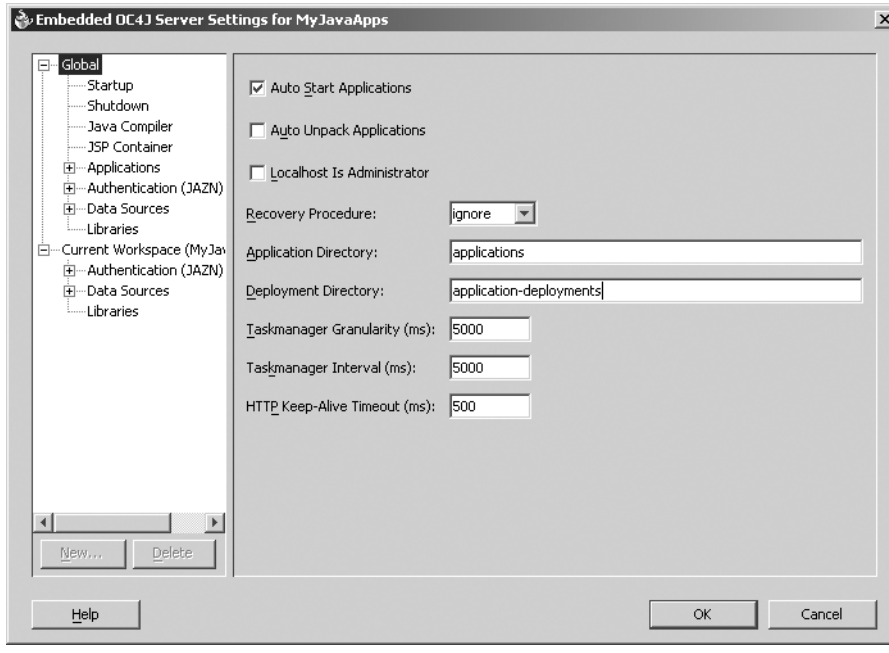

Figure 5-9. Configuring the OC4J server

When choosing the Shutdown option in the Embedded OC4J configuration, you can specify the shutdown methodology to use. A graceful shutdown is the best option; however, in some circumstances, a forced shutdown becomes essential.

You are provided configuration options for OC4J at two levels, Global and Current Workspace. I will explain some of the configuration elements when you encounter them in the example. Now that you have had a feel of J2EE applications and OC4J, you will move on to what is the most widely used element of J2EE, JSPs.

Java Server Pages

JSPs came a lot after Servlets had emerged; however, today JSPs are all over the place, but not many new Servlets are being developed. The primary reason for the decline of Servlets and for the emergence of JSPs was that Servlets were just not good enough to generate a lot of HTML output.

In the "Servlet Creation and Deployment" section earlier, you created a basic Servlet that flashed a welcome message. Note that in this Servlet, apart from the logic of displaying the message, there is a lot of HTML output being generated that is not directly relevant to the purpose of the Servlet. Imagine if your Servlet had to generate HTML for hundreds of complex pages having images and lots of JavaScript. Quite a scary thought.

JSPs changed things such that Java is used only where it is absolutely needed. If that same Servlet functionality were to be implemented as a JSP, the JSP would look something like this:

```
<%
  String strFname = request.getParameter("fname");
  String strLname = request.getParameter("lname");
%>
<html>
  <head>
      <title>Hello</title>
  </head>
  <body>
      <p>Welcome <%=strFname%> <%=strLname%> </p>
  </body>
</html>
```

If you are wondering when you have to write the Java class and do all the extending classes and implementing methods, then surprise! This is all it takes. There is no need to create and compile any Java class. Just embed whatever Java you need right into the HTML, and you are on your way. This embedding is done using special tags; you can even create your own tags.

How JSP Works

The nature of Java demands that to execute anything written in Java, it needs to undergo a conversion to bytecodes, which can then be executed by the Java virtual machine (JVM). You are not writing and compiling the Java code in your JSP; it is your container that does that for you. In the translation phase, the container

generates Java code for all content in a JSP and also generates a corresponding *.java file. The file is compiled to generate a Servlet class file, and it is this Servlet that actually does the job for you.

Although a JSP developer does not really need to be aware of the methods generated in the Java code or the interfaces implemented, it always helps to understand them. The package javax.servlet.jsp is what holds the Servlet API's JSP-related classes and interfaces. The interface javax.servlet.jsp.JspPage needs to be implemented, and the _jspService method is the method that corresponds to the body of the JSP page. The JSP developer should not define the _jspService method in the JSP because the container defines this method.

NOTE *The Java files created for JSPs, although a somewhat scary sight, can be extremely useful for debugging purposes. Strangely enough, all container vendors seem to fancy creating these files in the most obscure of locations. Searching for the pattern *<filename>*.java is the easiest way to locate these Java files. OC4J creates a new directory named* .jsps *in the WEB-INF/classes directory of the Web application to which the JSP belongs.*

Most containers, unless instructed otherwise, compile the JSP into a Servlet only on the first invocation of the JSP. So you will notice a slow response the first time you invoke a JSP. However, because your next request will be serviced by the compiled Servlet already in place, the response will be much faster. Once the Servlet generates the appropriate HTML, it is only the HTML that goes across to the client. So any JSP tags you use are not visible at the client side; what appears is just HTML.

JSPs files are treated more like ordinary HTML files than like Java code. For the JSP to be publicly accessible, it can be placed in any directory other than the WEB-INF directory or its subdirectories.

JSP Elements

For the container to be able to convert a JSP that includes HTML, embedded Java code, and special tags, it is imperative that the JSP has to follow a definite syntax for the container to be able to convert it into a Java file and compile it.

Under the JSP specification 1.2, many tags underwent a change to make them compliant with XML standards. Using these new XML-like tags, it is now possible to write a JSP adhering to the rules for well-formed XML. You will first quickly look at the various JSP elements and then create a new JSP in JDeveloper that uses all of these elements.

Directives

Directives are instructions for the container that are to be processed while translating the JSP into a Servlet. The directives play a part in the Servlet code that gets generated. The types of directives are as follows:

- **page**: The page directive is meant to specify the properties of the JSP. Things such as the classes to import, the error page to use, and the content type to be generated are specified in the page directive.

- **include**: JSPs have a capability to include text and/or code right when the page is being translated; the content in the JSP along with the included content is translated into a single Servlet and then compiled. The include directive can be used to specify the resource to be included.

- **taglib**: To use custom tag libraries in a JSP page, you first need to specify the tag libraries to be used in the page directive.

This is the normal syntax:

```
<%@ page import="java.util.ArrayList" language="java" %>
```

This is the XML syntax:

```
<jsp:directive.page import="java.util.ArrayList" language="java"/>
```

Declarations

Declarations are used to declare new variables and methods that can be used throughout the JSP. Variables declared in a declaration are converted into instance members of the Servlet created, and methods defined are reflected as new instance methods in the Servlet.

If you look at the Java file created for the JSP page, you will notice that the methods and variables declared in a declaration appear independent of the _jspService method that has most of the other request processing and response generation logic in it. The contents of a declaration have to just follow normal Java rules and syntax.

This is the normal syntax:

```
<%!
int i=0;
%>
```

This is the XML syntax:

```
<jsp:declaration>
int i=0;
</jsp:declaration>
```

Scriptlets

Scriptlets are perhaps the most used as well as misused feature of JSP. Although the usage of scriptlets is declining because of the growing popularity of custom tag libraries, scriptlets are still ubiquitous in the JSP world. Scriptlets are Java code snippets that are embedded right into the JSP. The way to go about it is to believe that you are writing Java code within a method and have a few predefined objects such as session, page, and application to work with.

NOTE *The easiest way to distinguish between scriptlets and declarations is that scriptlet code ends up in the _jspService method of the Servlet generated for the JSP, and the code for declarations appears outside this method.*

Scriptlets were quite popular in the early days of JSP; however, these days, any developer who codes using scriptlets is given what I believe is the Microsoft treatment. You know the thing will work well initially but will cause headaches later. And even if it does not, you hate it anyway. So to stay popular and appear smart, stay away from scriptlets and use custom tags wherever possible. Using tags created to accomplish a certain task instead of writing the Java code for it keeps the JSP easy to understand and maintain. Standardized and well-tested tags are also unlikely to throw unexpected errors at runtime.

This is the normal syntax:

```
<%
     System.out.println("In a Scriptlet");
%>
```

This is the XML syntax:

```
 <jsp:scriptlet>
     System.out.println("In a Scriptlet");
</jsp:scriptlet>
```

Expressions

An *expression* is just an out.println statement. In the Servlet code generated, the contents of the JSP expression are parameters to an out.println method call,

where *out* is the implicit JspWriter object provided. So the expression `<%= new java.util.Date() %>` would be represented in the _jspService method of the Servlet code as `out.print( new java.util.Date() );`.

A variable you declare in a declaration or a scriptlet can be used in an expression to get the value out to the display. A common mistake is to end the expression with a semicolon. This will give you a compilation error because putting semicolons in a println call is just incorrect Java.

This is the normal syntax:

```
<%= new java.util.Date() %>
```

This is the XML syntax:

```
<jsp:expression> new java.util.Date() </jsp:expression>
```

Actions

Actions are some of the smartest tags that form a part of the core JSP specification. Based on attribute values, actions accomplish specific tasks. The standard JSP actions are as follows:

- **jsp:include**: Include another JSP in a JSP. This is different from the include directive because this include is processed at request time, unlike the directive include, where the two JSPs form a single translation unit.
- **jsp:forward**: This forwards a request to another JSP.
- **jsp:useBean**: This declares a JavaBean to be used in the JSP.
- **jsp:setProperty**: This sets property values of the Java bean being used.
- **jsp:getProperty**: This gets property values of the Java bean being used.

JSP Implicit Objects

For anything to be accomplished in the JSP, you need to be able to access and use the user's session, fetch request details, set response information, and so on. However, writing the code to get these things in every JSP does not make sense. So the container declares and initializes variables that could provide the requisite functionality.

Again, if you refer to the Servlet code, you will find that the first few lines of the _jspService method have nothing to do with the JSP's output or logic. These lines of code only create these variables for use by code specific to the JSP.

So within the JSP, the developer can just presume that these variables are there and use them. For a JSP named *MyTestJSP.jsp*, OC4J generates a file named *_MyTestJSP.java*. The _jspService method in this Servlet code would begin with these lines (the variables are in bold):

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {

        response.setContentType( "text/html;charset=windows-1252");
        /* set up the intrinsic variables using the pageContext goober:
        ** session = HttpSession
        ** application = ServletContext
        ** out = JspWriter
        ** page = this
        ** config = ServletConfig
        ** all session/app beans declared in globals.jsa
        */
        PageContext pageContext = JspFactory.getDefaultFactory().getPageContext( this
                , request, response, null, true, JspWriter.DEFAULT_BUFFER, true);

        // Note: this is not emitted if the session directive == false
        HttpSession session = pageContext.getSession();
        if (pageContext.getAttribute(OracleJspRuntime.JSP_REQUEST_REDIRECTED
                    , PageContext.REQUEST_SCOPE) != null) {

             pageContext.setAttribute(OracleJspRuntime.JSP_PAGE_DONTNOTIFY, "true"
                    , PageContext.PAGE_SCOPE);
             JspFactory.getDefaultFactory().releasePageContext(pageContext);
             return;
        }

        int __jsp_tag_starteval;
        ServletContext application = pageContext.getServletContext();
        JspWriter out = pageContext.getOut();
        _MyTestJsp page = this;
        ServletConfig config = pageContext.getServletConfig();
```

The following is the list of variables:

- **request**: This holds the details for the current request.
- **response**: The response details are to be set into this object.
- **pageContext**: As shown in the previous code, the other variables in question are derived out of the pageContext. The pageContext is what stores references to all other implicit variables.
- **session**: This object refers to the user's session and can be used to store many details about the user and the current session.
- **application**: The application object holds details about the environment in which the JSP is being run.
- **out**: This is a reference to JspWriter that is used for all generation of output.
- **page**: This refers to the current object of the Servlet generated for the JSP in question.
- **config**: This object is used to fetch configuration parameters for the JSP.

Now that you have gone through most JSP concepts, you will move on to creating a JSP using JDeveloper.

A JSP Example

With JSP being the most extensively used component of J2EE, it becomes a critical element for the success of any J2EE development tool to provide good JSP features. JDeveloper does a good job at creating and editing JSPs. An important addition to version 10*g* is the introduction of visual editors for HTML and JSPs. This is a significant step because these tools are quite competent and work toward making a separate What You See Is What You Get (WYSIWYG) editor usage unnecessary.

Earlier in this chapter you created a new Web module named *BasicWebApp* (refer to Figure 5-2). You will use the same Web module to create a new JSP. Considering that all J2EE developers willingly or unwillingly have to work with HTML and JavaScript as a part of Web development projects, JDeveloper provides good HTML editing features as well. You will quickly look at the HTML editing features of JDeveloper while creating your JSP.

Open the New Gallery for the BasicWebApp project, and choose Web Tier ➤ JavaServer Pages. Of the options listed, JSP Document will create a JSP adhering to XML standards having a <jsp:root>, and the JSP Page option will create a free-flowing JSP that does not need to adhere to XML standards.

NOTE *If you intend to transform the JSP using Extensible Stylesheet Language (XSL) or parse it using an XML parser, you can go for the JSP Document option. If not, go for a JSP Page because it has less clutter and is easier to work with.*

Choose the JSP Page option, and name the new JSP *Welcome.jsp*. JDeveloper creates a bare-bones JSP that has a title and displays the current time. The JSP appears in two views, Design and Code. The Design view is the WYSIWYG editor, but you always have the option of directly editing the code. The WYSIWYG features are similar to using any other word editor. The toolbar provided makes changing text formatting, using lists, colors, and so on a simple task. What you intend to do in your JSP example is take user input of first name and last name and call the WelcomeServlet with these values as part of the request.

For this you need to create an HTML form. As per normal HTML usage, the form has to be enclosed between the body tags of the HTML. With JDeveloper you hardly need to write any HTML. While in the Design view, choose the HTML option in the Component Palette to display the HTML palette shown in Figure 5-10.

Figure 5-10. HTML palette

In this palette, choose the Form option. Double-clicking the form displayed in the Design view or right-clicking and selecting the Edit Tag option will take you to the dialog box shown in Figure 5-11. Here state the action name as *name.welcome* and the form name as *NameForm*. The form name does not really matter in this particular case; the action matters. Based on the Servlet mapping you provided in the web.xml file earlier, all requests matching the pattern *.welcome will be directed to the WelcomeServlet. Thus, the action *name.welcome* will have the request directed to the WelcomeServlet.

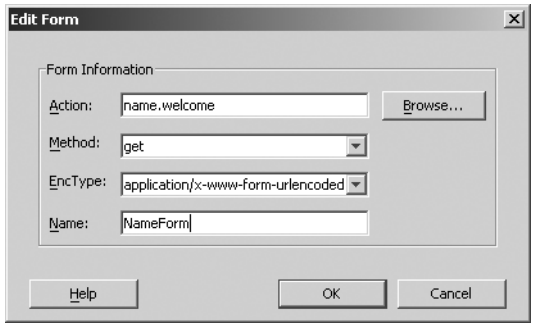

Figure 5-11. The form details

TIP *The HTML palette options provide for specifying event handling as well as styles to be applied. It is always advisable to use these features, especially in the case of JavaScript where it can be a real pain debugging the simplest of problems. The IDE will not make the silly mistakes that humans regularly commit.*

Now to get the user's first name and last name as input, you need a table that will hold the requisite input fields. The Table option presents a user interface (UI) that will take all parameters that an HTML table can take. After this, place two text fields into the table to get the first and last name from the user. Next, use the Submit Button option in the HTML palette to insert a new Submit button for the form.

NOTE *The text fields have to be named* fname *and* lname *because these are the request parameters expected by the WelcomeServlet. The input fields and the submit button have to be within the form tag structure—in other words, after <form> and before </form> for the data to be submitted as part of the form.*

Welcome.jsp should now look like this:

```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
    <title>
    Hello World
    </title>
  </head>
  <body>
    <h2>
      The current time is:
    </h2>
    <p>
      <%= new java.util.Date() %>
    </p>
    <form action="name.welcome" name="NameForm">
      <table  cellspacing="2" cellpadding="1" border="1" width="40%">
        <tr>
            <td>First Name</td>
            <td><input type="text" name="fname"></td>
        </tr>
        <tr>
            <td>Last Name</td>
            <td><input type="text" name="lname"></td>
        </tr>
      </table>
      <p>
        <input type="submit" name="Submit" value="Submit Name">
      </p>
    </form>
  </body>
</html>
```

The Structure window is a handy feature while working with JSP and HTML. This window shows the structure of the page as a tree of tags. It becomes even more useful if you commit blunders. For example, if you misspell the contentType attribute in the page directive, the Structure window immediately points out the error, as shown in Figure 5-12. The window constantly keeps getting updated while you work. You do not even need to save the file to have the errors pointed out.

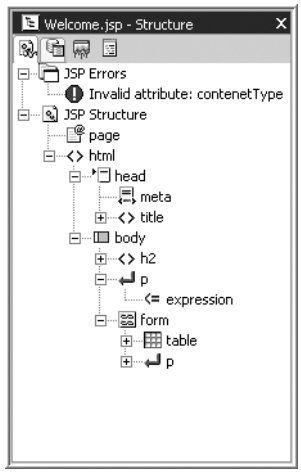

Figure 5-12. Structure window

To run the JSP, choose the Run Welcome.jsp command. The embedded OC4J server will start, and your default browser will open to the URL for the JSP. On the page displayed, input your first name and last name and then click the Submit button. The WelcomeServlet should be invoked, displaying a welcome message with your first and last name entered.

This example was more about HTML than JSP, but you will use and explore JSPs further while dealing with JDeveloper's integration with Struts.

JSP Standard Tag Libraries

Custom tag libraries hold a set of tags that are used on very similar lines to normal HTML tags but perform special functions that are beyond what HTML tags can achieve. These tag libraries provide tags that can format dates, modify strings and perform other similar tasks. Some Java code runs in the background to get these tags to perform the task.

With the growing adoption and usage of custom tag libraries has arisen the need to have tag libraries provide functionality in a standardized fashion. Although free third-party tag libraries are available, a need existed for the standardization of some commonly used tags.

The JSP Standard Tag Libraries (JSTL) is a specification and not an implementation for tags; you can find it at http://java.sun.com/products/jstl. JDeveloper provides Component Palettes for JSTL Core, Formatting, SQL, and XML. Using the JSTL tags wherever possible is a far better alternative than developing your own tags or writing code in your JSP.

Jakarta Struts

With the ever-growing complexity of Web applications has arisen the need to have frameworks in place that can handle the mundane and can streamline the application as a whole. Most frameworks are based on the Model-View-Controller (MVC) pattern and are primarily meant to have a clear distinction between the presentation and the logic in an application. Many frameworks go way beyond these humble targets and provide tons of other features. Some of the more popular frameworks are Apache Struts, Apache Cocoon, Expresso, and WebWork.

Apache Jakarta Struts is by far the most popular of the frameworks in use, so most of the newer IDEs have some degree of built-in Struts support. JDeveloper also provides great integration with Struts and has wizards and tools to enable the creation and development of various Struts components.

Figure 5-13 depicts a common Struts architecture. The struts-config file is the file that the controller Servlet in Struts refers to in order to determine which Action class to use. All requests should go through the controller Servlet, and it is this Servlet that should determine the flow of the application. On finding the appropriate Action class, the Action class can either do the necessary processing on its own or can pass on to a core logic class. Then based on conditions, the Action class can tell the controller Servlet where the request should be forwarded. No JSP names are stated in the Java code; the actual JSP to use is stated in the configuration file.

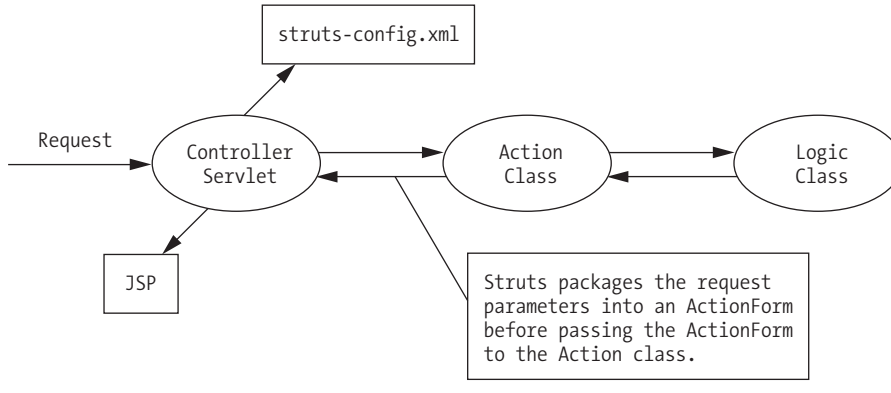

Figure 5-13. Struts architecture

The Struts framework provides the controller Servlet, and the user is expected to create the struts-config.xml file that adheres to a certain set of rules specified in a document type definition (DTD). Apart from this division of tasks that forms the core of the Struts framework, it also provides many other features to make Web development easier.

Elements of Struts

The key elements of the Struts framework that you need to understand to use the framework well are as follows:

Action: The controller Servlet based on the request received and on referring to the flow instructions stated in the struts-config file passes control to the action class stated in the struts-config file. Custom development using Struts begins with the Action class. The Action class can perform all the required logic and handling or in turn call a core logic class and then forward to the appropriate page.

ActionForm: Because all Web development revolves around the request parameters received from the client, Struts has the concept of an ActionForm to make things simpler. An ActionForm is an ordinary Java class adhering to JavaBean conventions. That is basically having the variables as private members and providing public get and set methods for the variables to get the value and set the value, respectively. The framework invokes the setter methods for all parameters that it finds in the request. Resultantly, the Action class does not need to work with the request as such but just a prepopulated ActionForm.

struts-config.xml: This file is the heart of Struts. For any development with Struts, this is the file to fiddle with and make it do things you want. All Actions used in the application, the ActionForms being used, the relations between these, the JSP pages to forward to are stated in the struts-config.xml file.

Struts Tags

As part of version 1.1-b2 of Struts that comes integrated with JDeveloper, Struts provides six tag libraries: template, html, logic, nested, bean, and tiles. The tag libraries make JSP development a lot easier and elegant. These tags, if used properly, can more or less eliminate any need to use Java code within your JSP. The libraries used most often are as follows:

- **bean**: This library provides many tags to make using and defining new beans possible.
- **logic**: This library provides tags that, based on conditions, can perform various actions. It is mostly used as a better option to using Java `if else` statement or `for` loops.
- **html**: This library provides tags to replace most core HTML tags and enable the creation of dynamic HTML pages.

NOTE *Before you use any scriptlets and end up having Java code in your JSP, look at all the tags available. In all probability, you will find the functionality you need. Also have a look at the JSTL at http://java.sun.com/products/jstl. In the long run, using standardized tags is a smarter option to using the existing Struts tags.*

You will now create a Struts application where you use the concepts discussed previously to create a simple login scenario.

Struts Application

Most Struts applications use JSPs extensively; there rarely arises the need to use Servlets. You will now create a Struts application where the flow is as follows:

1. Submit data from one JSP page.

2. Pass the request information encapsulated in an ActionForm to an Action.

3. Based on the userID and password entered, direct the user to a JSP that displays the appropriate message.

An addition to version 10*g* is the Struts page flow diagram. Because the primary task of Struts is proper and conditional management of flow, having such a flow diagram can be a valuable asset to quickly understand the workings of a Struts-based application.

Create an empty project named *StrutsApp.jpr* in a new directory named *StrutsApp*. Change the default package to com.jdevbook.chap5 using the Common ➤ Input Paths section in the Project Properties dialog box. For this project, from the New Gallery, select the item WebTier ➤ Struts ➤ Struts Controller Page Flow and click OK. The name is a little deceptive because what happens in reality is that JDeveloper completes all the groundwork required for a Struts application. The files created are struts-config.xml, web.xml, and ApplicationResources.properties. I have already talked of the first two files. ApplicationResources.properties is an ordinary text file that makes application internationalization a simple affair. You never need to hard-code any messages; just pick the messages from the properties file using simple Struts tags.

Struts Configuration in web.xml

To understand what JDeveloper has done for your Struts application, look at the web.xml file created for you. Because you want all requests to come through the controller Servlet, the web.xml file declares a new Servlet named *action* and has a mapping in which all requests that fulfill the URL pattern *.do will be directed to the action Servlet declared earlier. The tag libraries used in the application are also declared in the web.xml file.

NOTE *Using the URL pattern *.do is more of a convention than a rule. You can of course change it to anything you want. The pattern *.twinkle (read as* star dot twinkle *. . . humor intended) would be nice.*

However, one thing that JDeveloper does not do is that it does not add tag library declarations in the web.xml file for all the tag libraries that Struts provides. So for other tag libraries that you intend to use, the tag libraries have to be declared using the following bit of code and also the .tld file has to be placed in the corresponding location. However, if you use Struts Logic Component Palette to write the tags, JDeveloper does the rest automatically:

```
<taglib>
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
```

Creating the Application

The only action involved in this example is that of receiving a login request. For this action, you need to create an Action and an ActionForm as well as add a corresponding entry in the struts-config.xml file.

With the Struts page flow diagram, you can even get a base application running by just visually depicting how you want the flow of the application to be; just draw actions, forwards, and pages in the diagram and depict their relationships. JDeveloper will automatically generate the Java code as well as make the necessary modifications to the struts-config.xml file.

On the blank Struts page flow diagram, first use the page component to add a page to the diagram. In the diagram, name this page */login.jsp*. The / symbol before login.jsp is required. Next, add an Action to the diagram. Change the action name in the diagram to */Login*. Right-click, and select the Go to Code option or double-click the action. As shown in Figure 5-14, change the action name to *LoginAction*. As a convention, Action classes end with *Action*.

After you click OK, the class LoginAction will be created for you. As per usage, the class extends the Action class and overrides the execute method.

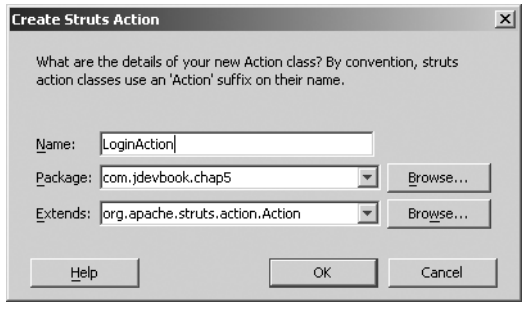

Figure 5-14. Viewing the Action details

Now that you have your Action class defined, you will define the form associated with the action. Right-click the action displayed in the diagram, and select the Go to Bean option. Change the form name to *LoginForm*, as shown in Figure 5-15.

The LoginForm class created extends ActionForm and overrides the reset and validate methods.

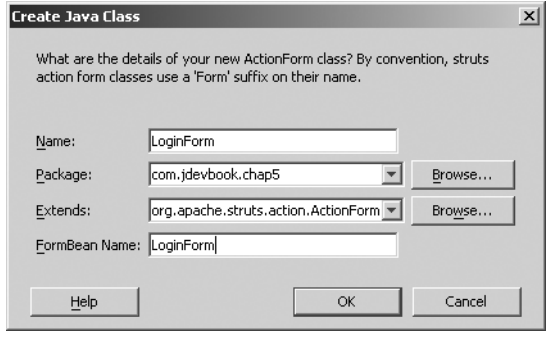

Figure 5-15. Viewing the ActionForm details

Now right click the struts-config.xml file, and choose the Edit Struts-Config option. The Struts Configuration Editor dialog box, as shown in Figure 5-16, should open. In the Form Bean tab for the action /Login, add login.jsp as the value of the input attribute. This value means that in case of an error, Struts takes the user back to the input page. Make it a point to explore most of the options provided in the Struts Configuration Editor dialog box.

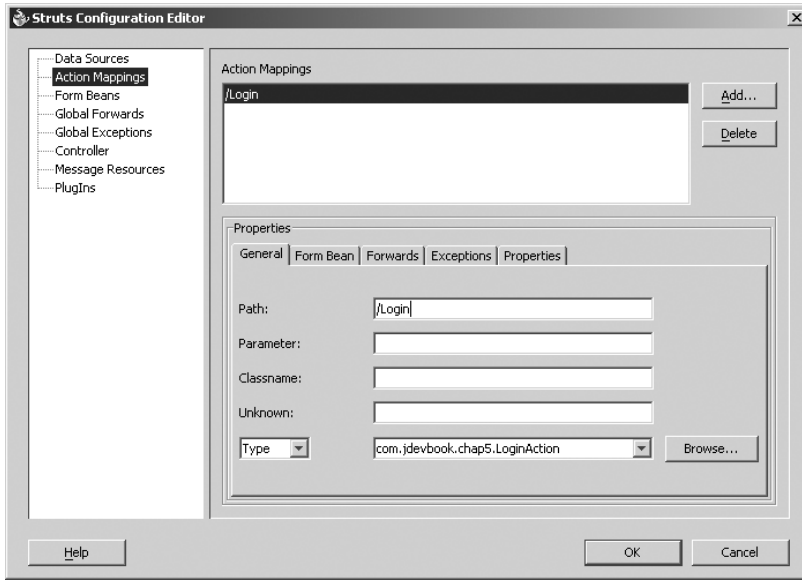

Figure 5-16. Using the Struts Configuration Editor dialog box

Now that you have the Action and ActionForm in place, you will move on to putting the JSPs in proper shape. Next you need to add a JSP to depict the successful flow. The Action will forward the request to this JSP if the request is successful. Add a new JSP to the diagram, and name it */result.jsp*.

NOTE *Double-click the JSPs depicted in the diagram for the files to be actually created. Only after the files are created can the JSPs be linked to actions or other JSPs in the diagram.*

Now use the Forward/Link component to link up all the elements you have drawn on the diagram, as shown in Figure 5-17.

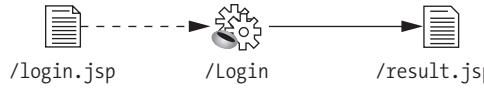

Figure 5-17. Struts flow diagram

You will now edit the login.jsp file to provide for submitting a form to the Login action. Open the login.jsp, file and choose the Struts HTML palette in the Component Palette list, to view a long list of all the HTML tags provided by Struts. The Struts HTML tags are replacements for many ordinary HTML tags. The Struts tags not only generate the relevant HTML tag, but also perform some additional Struts-specific functionality. You need a new HTML form to take a userID

and password as input and submit the same. As per normal HTML usage, the form has to be enclosed between the body tags of the HTML. Click the Form option in the palette and name the form *LoginForm* to match the form entry you created in struts-config.xml. The action would be named *Login*. You do not need to append the .do extension to the action because the tag will do that.

NOTE *Based on the Servlet mapping in the web.xml file, Struts determines the extension to be used. In this example, you have mapped the extension *.do to the ActionServlet, so the .do extension to the action name is inserted automatically by the <html:form> tag. Changing all links to, say, *.twinkle involves only a simple Servlet mapping change to the web.xml file. View the source of the HTML generated to check the complete action name being used.*

You now need to create a text field to get the login userID and a password field to get the login password. Use the tags <html:text> and <html:password> to do this job for you. You can create the form's Submit button using the Submit option in the HTML palette.

Form Field Validations

Validating form input can at times be an especially annoying task. Struts provides many features that make validating user input a lot easier. With version 1.1, the Validator framework has also been integrated into Struts, making validations even easier. However, you will stick to validations using some of the concepts that the framework uses and not dwell on the validator framework as such.

The form is what carries the input to the action, so the ideal place to validate input is just after the form has been populated based on the values received in the request. For this, Struts provides a mechanism by which it calls a validate method in the form before actually calling the action class and passing the populated form instance to it. So if a validation fails at the form level, an appropriate error is returned and the user is taken back to the input page based on the value of the input attribute for the action.

The things you need to do to validate the userID and password are as follows:

- **Modify the validate method**: JDeveloper automatically created a validate method in the LoginForm class; however, you need to provide proper implementation to check the values received and act accordingly. If the validation of the fields fails, this method returns ActionErrors that holds the details of the exact errors that occurred.

- **Add an input attribute to the action stated in struts-config.xml**: The input attribute is what conveys to Struts the page to be displayed when an error occurs. In most forms, you go back to the same page, display the exact validation error, and allow the user to resubmit.

- **Use the <html:errors> tag to display the message on the JSP**: The ActionErrors returned by the validate method can be displayed in the JSP using this tag. The tag uses the key, which is part of the ActionError object, and fetches the corresponding message from the ApplicationResources.properties file.

- **Add appropriate error messages to the ApplicationResources.properties file**: This file holds the various messages to display. This particular filename is not mandatory and can be changed by editing the <message-resources> tag in the struts-config.xml file.

You will first provide an implementation to the validate method. You want to check that if the username or password is null or empty, you return ActionErrors and take the user back to the input page. You will take the names of the request parameters as the userID and password. So you have to define these variables and their corresponding getter and setter methods in the LoginForm class. Next you edit the validate method as follows:

```
 public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)
{
  //return super.validate(mapping, request);
    ActionErrors actErrors= new ActionErrors();
    if(userId==null || "".equals(userId))
    {
        actErrors.add("userId", new ActionError("error.userid"));
    }
    if(password == null || "".equals(password))
    {
        actErrors.add("password", new ActionError("error.password"));
    }
    return actErrors;
}
```

The values you are setting in theActionError objects are keys against which you maintain appropriate messages in the properties file. For the error.userid and error.password keys that are set into the ActionError object, you also need corresponding messages in the properties file. The properties you require are as follows:

```
error.userid=<LI>Invalid UserId</LI>
error.password=<LI>Invalid password</LI>
errors.header=<font color="red">Validation Errors</font><UL>
errors.footer=</UL><HR>
neo=Mr. Anderson! Welcome to the JTricks.
agent=JTricks likes you no more. Go away!
```

The login.jsp page has to undergo a small change with the <html:errors/> tag now appearing at the top of the page. If this tag finds any ActionError objects, it iterates through all the ActionError objects and displays all the error messages. The login.jsp will be as follows:

```
 <%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>untitled</title>
</head>
<body>
  <html:link page="/Login.do">
    <bean:message key="link.Login"/>
  </html:link>
  <br/>
      <html:errors />
      <html:form action="Login">
            <table  cellspacing="2" cellpadding="1" border="1" width="40%">
              <tr>
                   <td>UserId</td>
                   <td>
                     <html:text property="userId" maxlength="10" />
                   </td>
              </tr>
              <tr>
                   <td>Password</td>
                   <td>
                     <html:password property="password" maxlength="8" />
                   </td>
              </tr>
            </table>
            <p>
              <html:submit title="Submit" value="Submit" />
            </p>
      </html:form>
</body>
</html>
```

So whenever the validate method returns one or more ActionErrors, Struts takes the user back to the login.jsp page while displaying all the errors that

occurred. The login.jsp page will appear in the Design view as shown in Figure 5-18.

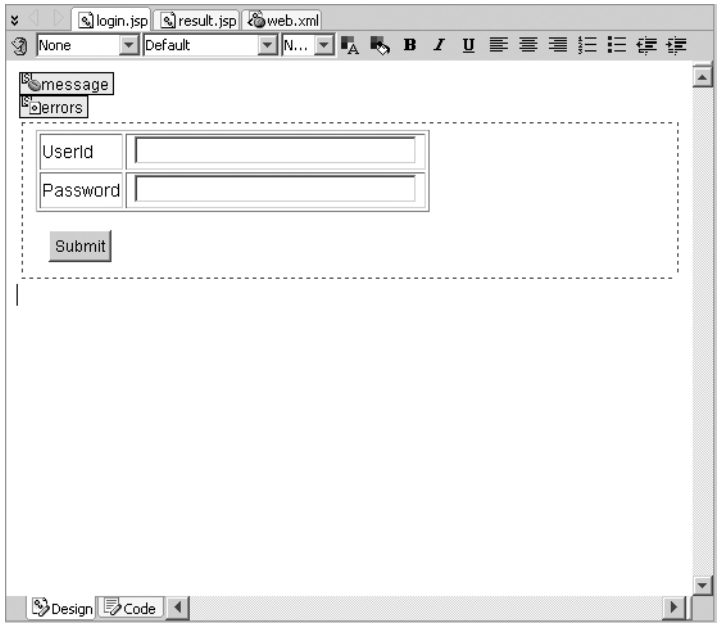

Figure 5-18. Accessing the Login.jsp Design view

Next you create another JSP named *result.jsp* to display a welcome message only if the user manages to log in successfully. You place the logic to check whether the access is legitimate in the LoginAction class. The execute method shell that JDeveloper has already created takes the validated form as input. The action fetches the userID and password, checks if the user is valid, and sets a flag into the request that states whether the access is valid. This flag is then used by result.jsp to display the appropriate message. The execute method in LoginAction is so modified:

```
 public ActionForward execute(ActionMapping mapping
  , ActionForm form
  , HttpServletRequest request
  , HttpServletResponse response) throws IOException, ServletException
{
    String userId=((LoginForm)form).getUserId();
    String password=((LoginForm)form).getPassword();

    String accessFlag="NO";
```

```
    if("neo".equals(userId) && "trinity".equals(password) )
    {
        accessFlag="YES";
    }

    request.setAttribute("ACCESS", accessFlag);
    return mapping.findForward("success");
}
```

Here you fetch the details from the form, and if the user is valid, you set the request attribute ACCESS value to YES. Otherwise, it is set to NO. The method then calls `mapping.findforward`. This call takes a string as input and forwards the request to the corresponding forward as stated in the struts-config.xml file. You now add a new forward to the action you created in struts-config. Because the forward tag is `<forward name="success" path="/result.jsp"/>` , returning success takes the user to the result.jsp page.

You next create a result.jsp page:

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>
Hello <bean:write name="LoginForm" property="userId"/>
</title>
</head>
<body>
  <logic:equal name="ACCESS" value="YES">
      <bean:message key="neo"/>
  </logic:equal>
    <logic:notEqual name="ACCESS" value="YES">
      <bean:message key="agent"/>
  </logic:notEqual>
</body>
</html>
```

Because you are using three tag libraries, these are specified in the taglib directive. The <bean:write> tag is capable of displaying values fetched from the request and the session. Because the same request that was fired from the login.jsp page is being forwarded to the result.jsp page, the request object can be used even in this JSP. Struts stores the form instance in the request or the session based on the scope attribute value of the action tag defined in the struts-config.xml file. Because

in this particular case the form is in the request, the userID is fetched from the form.

Next you use the logic tags to check if the ACCESS value is YES or NO. Based on this value the corresponding message is fetched from the ApplicationResources.properties file using the <bean:message> tag. The login functionality is now in place. Run the login.jsp file. In the input form displayed, only the userID *neo* and password *trinity* will get you a welcome message shown in Figure 5-19. If both values are invalid, you will get the screen shown in Figure 5-20.

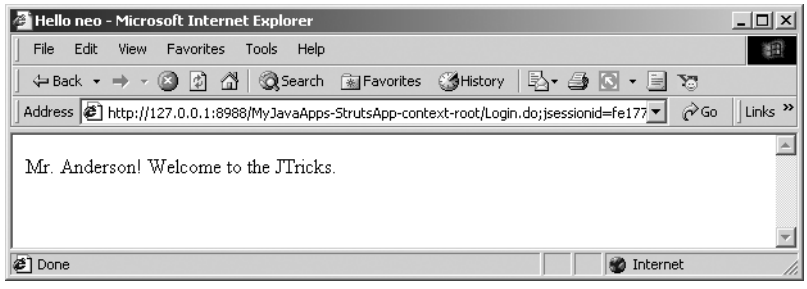

Figure 5-19. A valid login

If either or both values are incorrect, you will get the login page again with the appropriate messages displayed at the top of the page.

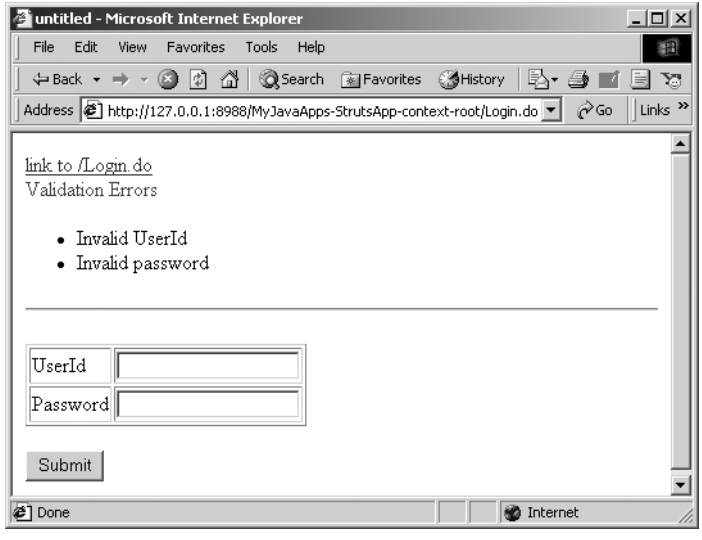

Figure 5-20. Invalid input values

An important feature of J2EE applications is the ease with which they can be ported across operating systems and application servers. Web applications are

packaged as Web archive (WAR) files. You will now look at how you can get this packaging done.

Deploy to a WAR File

A WAR file is a single file that holds all the files that are part of a Web application. The WAR file maintains the exact directory structure as required for Web applications. You can use the *jar* command to create a WAR file, either compressed or not. However, JDeveloper eliminates the need to manually create WAR files.

To simplify application deployment, JDeveloper has introduced the idea of creating deployment profiles. These profiles hold all the information required to easily deploy the application on various severs and to package the application into archives. Although enterprise applications consisting of EJBs are packaged as enterprise archive (EAR) files, Web applications such as StrutsApp that you created are packaged as WAR files.

The web.xml file in a Web application plays an important role in the deployment of the application. JDeveloper provides a neat visual environment to configure your web.xml file without having to manually edit the XML. To get this display, right-click the web.xml file listing and select the Settings option. The screen shown in Figure 5-21 should be displayed, making it easier to edit the web.xml file without having to worry about committing XML mistakes or using tags that are not valid as per the DTD concerned.

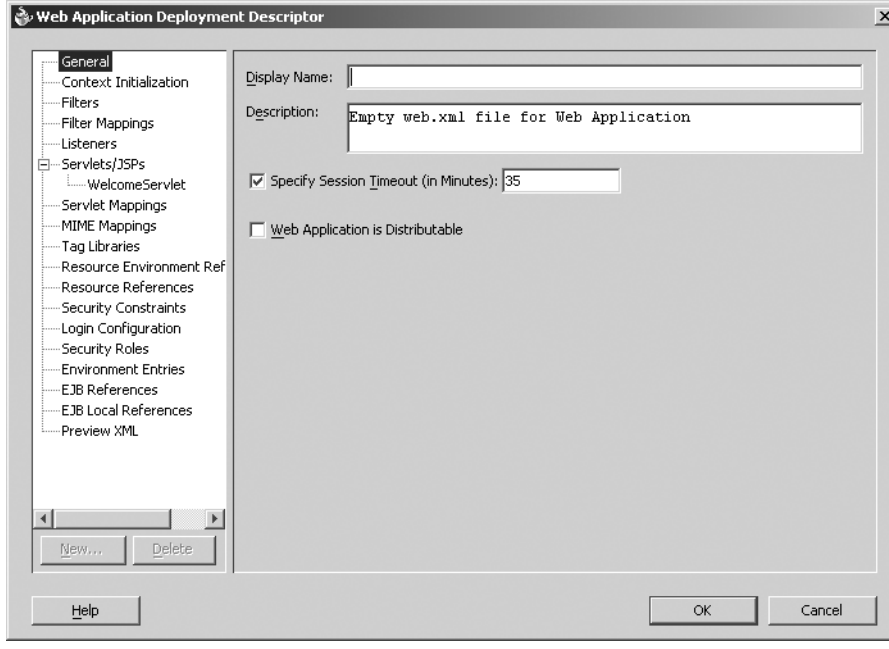

Figure 5-21. The Web Application Deployment Descriptor dialog box

For a Web application, the deployment depends greatly on the web.xml file. To create a deployment profile for StrutsApp, simply right-click the web.xml file as listed in the System Navigator and choose the Create WAR Deployment Profile option. Save the file in the StrutsApp directory as *StrutsApp.deploy*. You should now see the WAR Deployment Profile Settings dialog box, as shown in Figure 5-22. If you create a new Web application using the wizard provided, a WAR deployment profile is created by default and stored as webapp.deploy.

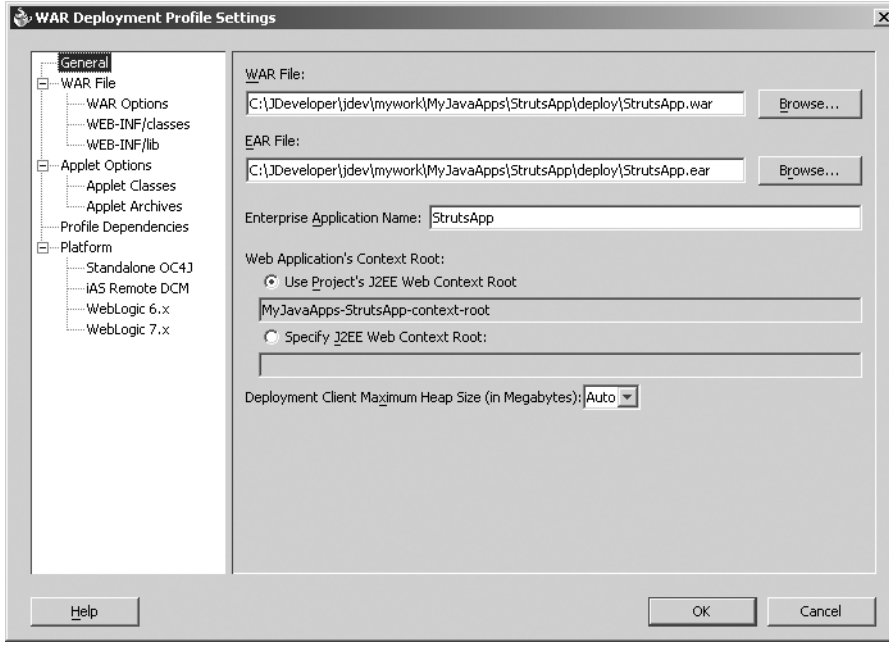

Figure 5-22. Using the WAR Deployment Profile Settings dialog box

In most cases, you do not need to change any of the settings because the default settings ensure that all the relevant files are packaged into the WAR. However, you can exclude certain files by choosing the WAR File option and deselecting all the directories and files you do not want to be part of the WAR file. If you do not want to ship your source files, you can remove the src directory from the list.

NOTE *JDeveloper creates class files for JSP in a directory under the classes directory named* .jsps. *Remember to remove this directory from the list of directories to be included in the WAR file.*

The Deployment Profile Settings dialog box also lets you tweak and set many other properties with respect to WAR creation and deployment on various supported servers. To actually deploy the StrutsApp application to a WAR file, simply right-click the StrutsApp.deploy file and choose the Deploy to WAR option. As specified in the deployment settings, by default the WAR file is created in a directory named *deploy* under the StrutsApp directory. The WAR file can now be easily ported to any other server, and the application can be up and running in no time whatsoever.

Summary

In this chapter you looked at the fundamentals of Servlets and JSPs and explored JDeveloper's features that enable quick and easy creation and deployment of Web applications.

JSPs are the workhorse of J2EE, and JDeveloper's integration and support for Struts and tag libraries makes JSP development a breeze. Deployment of J2EE applications has always been one of the relatively complex parts of J2EE. JDeveloper's ability to create deployment profiles that can be extensively configured and enable quick deployment to multiple supported servers is certainly a feature to write home about.