Part II The Web Tier

Part II explores the technologies in the web tier.

This part contains the following chapters:

- Chapter 6, Expression Language
- Chapter 7, Using JavaServer Faces Technology in Web Pages
- Chapter 8, Using Converters, Listeners, and Validators
- Chapter 9, Developing with JavaServer Faces Technology
- Chapter 10, JavaServer Faces Technology Advanced Concepts
- Chapter 11, Configuring JavaServer Faces Applications

- Chapter 12, Using Ajax with JavaServer Faces Technology
- Chapter 13, Advanced Composite Components
- Chapter 14, Creating Custom UI Components
- Chapter 15, Java Servlet Technology
- Chapter 16, Internationalizing and Localizing Web Applications

Getting Started with Web Applications

A **web application** is a dynamic extension of a web or application server.

Web applications are of the following types:

- **Presentation-oriented:** A presentation-oriented web application generates interactive web pages containing various types of markup language (HTML, XHTML, XML, and so on) and dynamic content in response to requests.

Development of presentation-oriented web applications is covered in Chapter 4, JavaServer Faces Technology through Chapter 9, Developing with JavaServer Faces Technology.

- **Service-oriented:** A service-oriented web application implements the endpoint of a web service.

  Presentation-oriented applications are often clients of service-oriented web applications.

  Development of service-oriented web applications is covered in Chapter 18, Building Web Services with JAX-WS and Chapter 19,

Building RESTful Web Services with JAX-RS in Part III, Web Services.

The following topics are addressed here:

- Web Applications
- Web Application Lifecycle
- Web Modules: The `hello1` Example
- Configuring Web Applications: The `hello2` Example
- Further Information about Web Applications

Web Applications

In the Java EE platform, web components provide the dynamic extension capabilities for a web server.

Web components can be Java servlets, web pages implemented with JavaServer Faces technology, web service endpoints, or JSP pages.

Figure 3-1 illustrates the interaction between a web client and a web application that uses a servlet.

The client sends an HTTP request to the web server.

A web server that implements Java Servlet and JavaServer Pages technology converts the request into an `HTTPServletRequest` object.

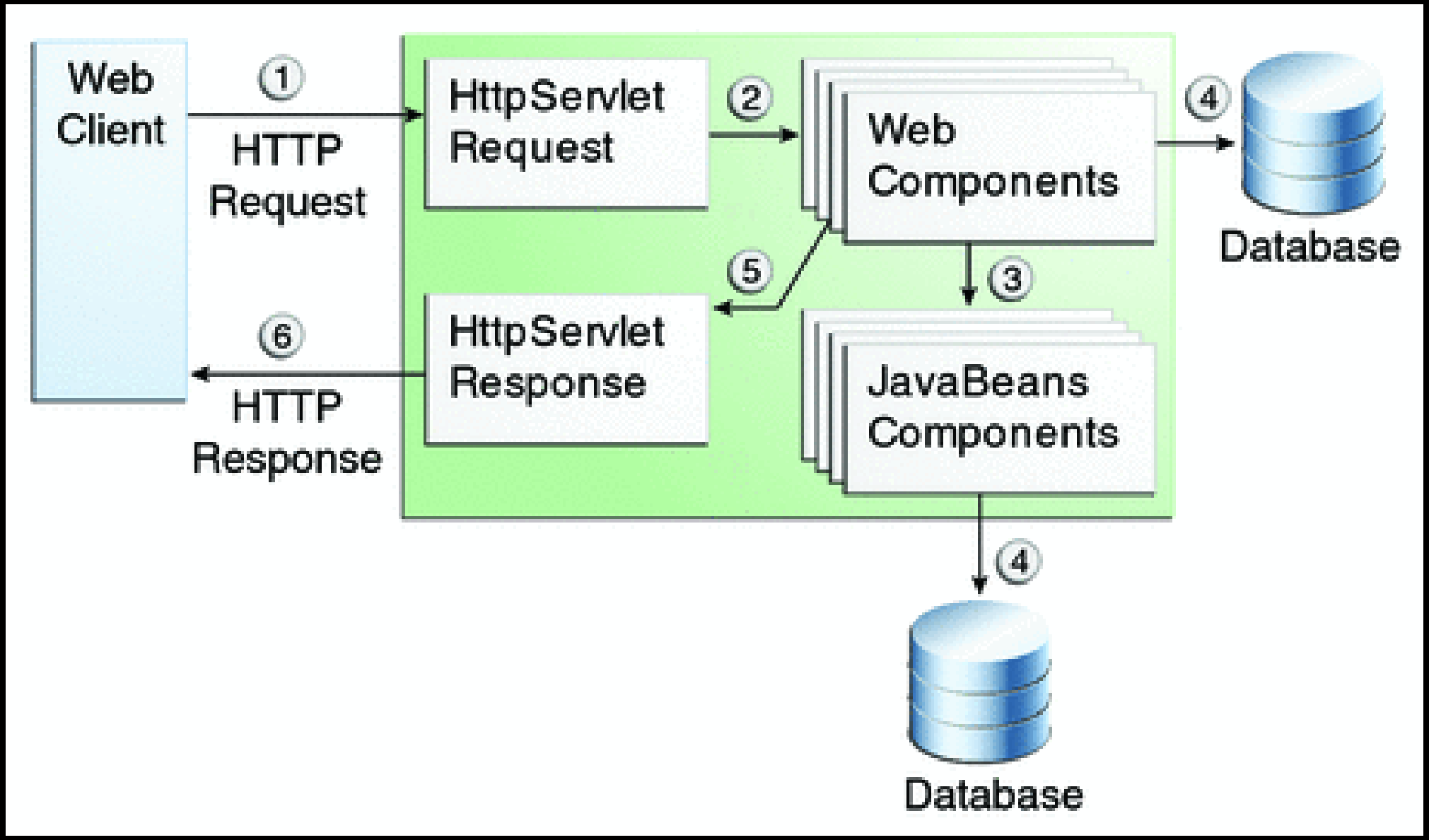
This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic content.

The web component can then generate an `HTTPServletResponse` or can pass the request to another web component.

A web component eventually generates a `HTTPServletResponse` object.

The web server converts this object to an HTTP response and returns it to the client.

Figure 3-1 Java Web Application Request Handling



Servlets are Java programming language classes that dynamically process requests and construct responses.

Java technologies, such as JavaServer Faces and Facelets, are used for building interactive web applications.

(Frameworks can also be used for this purpose.)

Although servlets and Java Server Faces and Facelets pages can be used to accomplish similar things, each has its own strengths.

Servlets are best suited for service-oriented applications (web service endpoints can be implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data.

Java Server Faces and Facelets pages are more appropriate for generating text-based markup, such as XHTML, and are generally used for presentation–oriented applications.

Web components are supported by the services of a runtime platform called a web container.

A web container provides such services as request dispatching, security, concurrency, and lifecycle management.

A web container also gives web components access to such APIs as naming, transactions, and email.

Certain aspects of web application behavior can be configured when the application is installed, or deployed, to the web container.

The configuration information can be specified using Java EE annotations or can be maintained in a text file in XML format called a web application deployment descriptor (DD).

A web application DD must conform to the schema described in the Java Servlet specification.

This chapter gives a brief overview of the activities involved in developing web applications.

First, it summarizes the web application lifecycle and explains how to package and deploy very simple web applications on the GlassFish Server.

The chapter moves on to configuring web applications and discusses how to specify the most commonly used configuration parameters.

Web Application Lifecycle

A web application consists of web components; static resource files, such as images; and helper classes and libraries.

The web container provides many supporting services that enhance the capabilities of web components and make them easier to develop.

However, because a web application must take these services into account, the process for creating and running a web application is different from that of traditional stand-alone Java classes.

The process for creating, deploying, and executing a web application can be summarized as follows:

1. Develop the web component code.
2. Develop the web application deployment descriptor, if necessary.
3. Compile the web application components and helper classes referenced by the components.
4. Optionally, package the application into a deployable unit.
5. Deploy the application into a web container.
6. Access a URL that references the web application.

Developing web component code is covered in the later chapters.

Steps 2 through 4 are expanded on in the following sections and illustrated with a Hello, World-style presentation-oriented application.

This application allows a user to enter a name into an HTML form (Figure 3-2) and then displays a greeting after the name is submitted (Figure 3-3).

Figure 3-2 Greeting Form for `hello1` Web Application

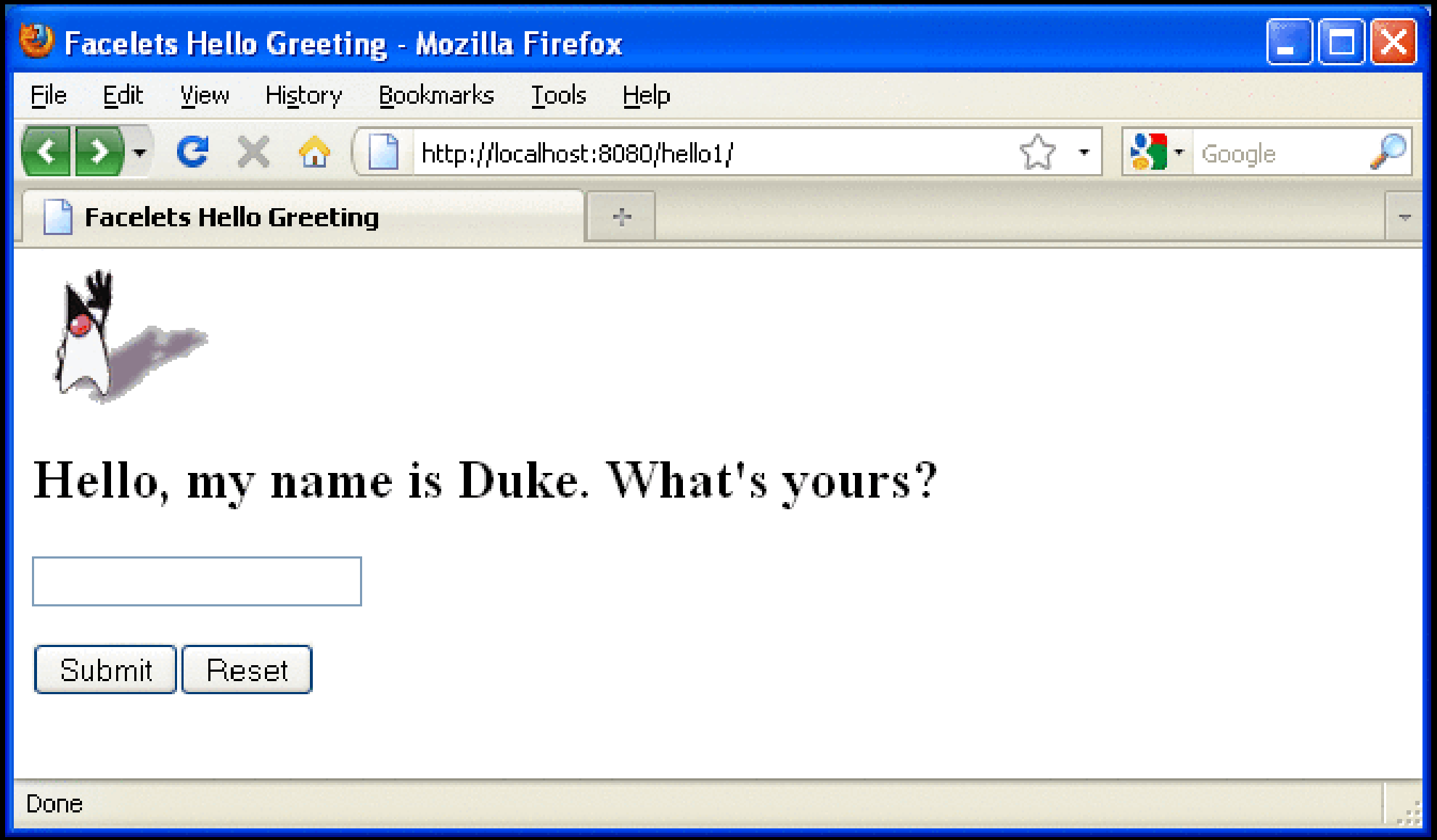


Figure 3-3 Response Page for `hello1` Web Application

The Hello application contains two web components that generate the greeting and the response.

This chapter discusses the following simple applications:

- `hello1`, a JavaServer Faces technology-based application that uses two XHTML pages and a backing bean

- `hello2`, a servlet-based web application in which the components are implemented by two servlet classes

The applications are used to illustrate tasks involved in packaging, deploying, configuring, and running an application that contains web components.

The source code for the examples is in the *tut-install*`/examples/web/hello1/` and *tut-install*`/examples/web/hello2/` directories.

Web Modules: The `hello1` Example

In the Java EE architecture, web components and static web content files, such as images, are called web resources.

A web module is the smallest deployable and usable unit of web resources.

A Java EE web module corresponds to a web application as defined in the Java Servlet specification.

In addition to web components and web resources, a web module can contain other files:

- Server-side utility classes, such as shopping carts
- Client-side classes, such as applets and utility classes

A web module has a specific structure.

The top-level directory of a web module is the document root of the application.

The document root is where XHTML pages, client-side classes and archives, and static web resources, such as images, are stored.

The document root contains a subdirectory named `WEB-INF`, which can contain the following files and directories:

- `classes`: A directory that contains server-side classes: servlets, enterprise bean class files, utility classes, and JavaBeans components

- `tags`: A directory that contains tag files, which are implementations of tag libraries

- `lib`: A directory that contains JAR files that contain enterprise beans, and JAR archives of libraries called by server-side classes

- Deployment descriptors, such as `web.xml` (the web application deployment descriptor) and `ejb-jar.xml` (an EJB deployment descriptor)

A web module needs a `web.xml` file if it uses JavaServer Faces technology, if it must specify certain kinds of security information, or if you want to override information specified by web component annotations.

You can also create application-specific subdirectories (that is, package directories) in either the document root or

the `WEB-INF/classes/` directory.

A web module can be deployed as an unpacked file structure or can be packaged in a JAR file known as a Web Archive (WAR) file.

Because the contents and use of WAR files differ from those of JAR files, WAR file names use a `.war` extension.

The web module just described is portable; you can deploy it into any web container that conforms to the Java Servlet specification.

To deploy a WAR on the GlassFish Server, the file must contain a runtime deployment descriptor.

The runtime DD is an XML file that contains such information as the context root of the web application and the mapping of the portable names of an application's resources to the GlassFish Server's resources.

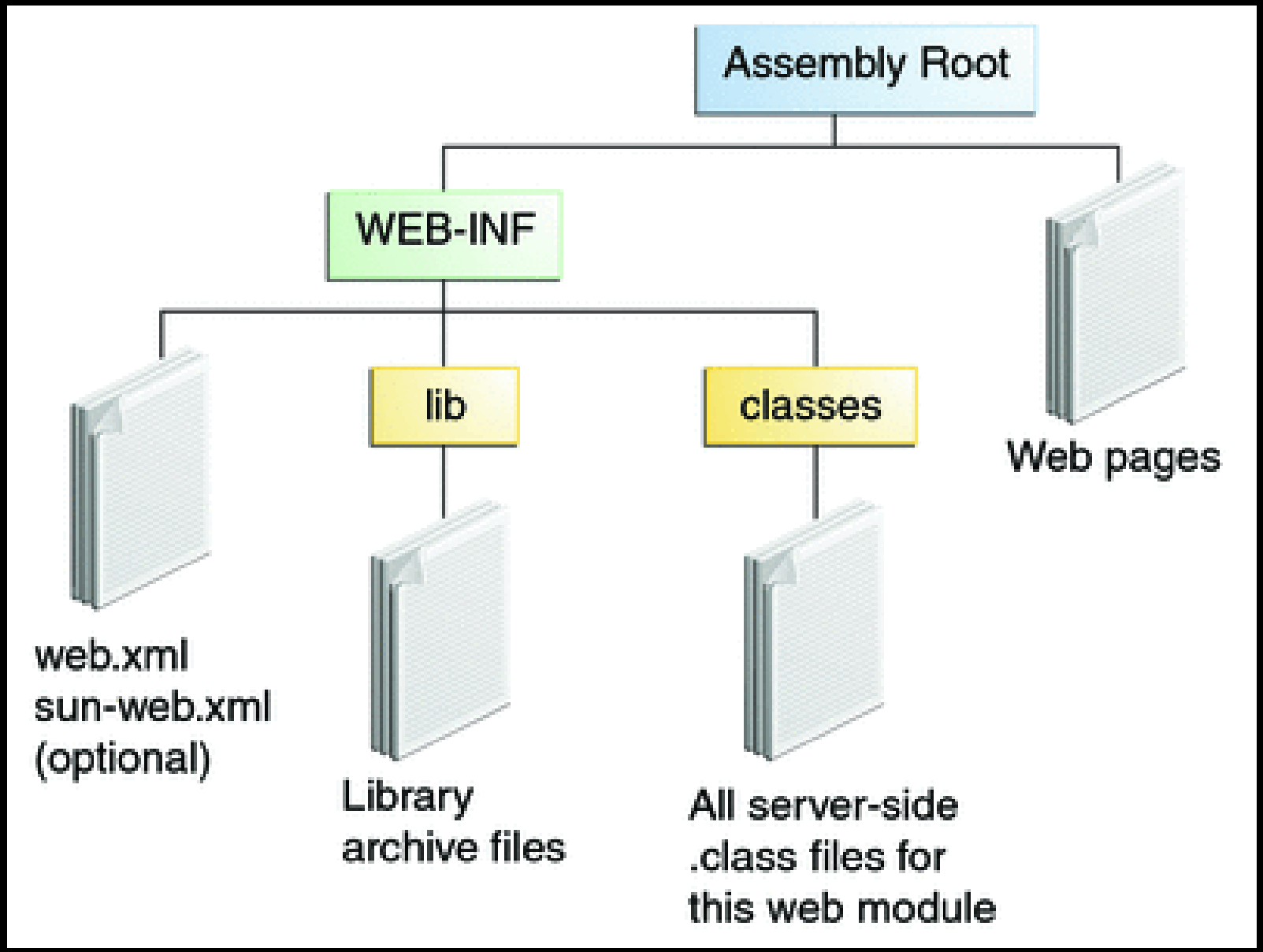
The GlassFish Server web application runtime DD is named `glassfish-web.xml` and is located in the `WEB-INF` directory.

The structure of a web module that can be deployed on the GlassFish Server is shown in Figure 3-4.

For example, the `glassfish-web.xml` file for the `hello1` application specifies the following context root:

```
<context-root>
/hello1
</context-root>
```

Figure 3-4 Web Module Structure



Examining the `hello1` Web Module

The `hello1` application is a web module that uses JavaServer Faces technology to display a greeting and response.

You can use a text editor to view the application files, or you can use NetBeans IDE.

To View the `hello1` Web Module Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/web/`

3. Select the `hello1` folder.

4. Select the Open as Main Project check box.

5. Expand the Web Pages node and double-click the `index.xhtml` file to view it in the right-hand pane.

The `index.html` file is the default landing page for a Facelets application.

For this application, the page uses simple tag markup to display a form with a graphic image, a header, a text field, and two command buttons:

```
<?xml version='1.0'
encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
"http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h=
"http://java.sun.com/jsf/html"
>
<h:head>
<title>
Facelets Hello Greeting
</title>
```

```
</h:head>
<h:body>
<h:form>
<h:graphicImage
url="duke.waving.gif"
/>
<h2>Hello, my name is Duke.
What's yours?
</h2>
<h:inputText id="username"
value="#{hello.name}"
```

```
required="true"
requiredMessage=
"A name is required."
maxlength="25">
</h:inputText>
<p></p>
<h:commandButton id="submit"
value="Submit" action="response">
</h:commandButton>
<h:commandButton id="reset"
value="Reset" type="reset">
```

```
</h:commandButton>
</h:form>
</h:body>
</html>
```

The most complex element on the page is the `inputText` text field.

The `maxlength` attribute specifies the maximum length of the field.

The `required` attribute specifies that the field must be filled out; the `requiredMessage` attribute provides the error message to be displayed if the field is left empty.

Finally, the `value` attribute contains an expression that will be provided by the `Hello` backing bean.

The Submit `commandButton` element specifies the action as `response`, meaning that when the button is clicked, the `response.xhtml` page is displayed.

6. Double-click the `response.xhtml` file to view it.

The response page appears.

Even simpler than the greeting page, the response page contains a graphic image, a header that displays the expression provided by the backing bean, and a single button whose `action` element transfers you back to the `index.xhtml` page:

```
<?xml version='1.0'
encoding='UTF-8' ?>
```

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xh
tml1-transitional.dtd">
<html
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h=
"http://java.sun.com/jsf/html">
<h:head>
```

```
<title>
Facelets Hello Response
</title>
</h:head>
<h:body>
<h:form>
<h:graphicImage
url="duke.waving.gif"/>
<h2>Hello, #{hello.name}!</h2>
<p></p>
```

```
<h:commandButton id="back"
value="Back" action="index" />
</h:form>
</h:body>
</html>
```

7. Expand the Source Packages node, then the `hello1` node.

8. Double-click the `Hello.java` file to view it.

The `Hello` class, called a backing bean class, provides getter and setter methods for the `name` property used in the Facelets page expressions.

By default, the expression language refers to the class name, with the first letter in lowercase (`hello.name`).

```
package hello1;
import
javax.faces.bean.ManagedBean;
import
javax.faces.bean.RequestScoped;
@ManagedBean
@RequestScoped
public class Hello {
private String name;
public Hello() { }
```

```
public String getName()
{ return name; }
public void setName
(String user_name)
{ this.name = user_name; }
}
```

9. Under the Web Pages node, expand the `WEB-INF` node and double-click the `web.xml` file to view it.

The `web.xml` file contains several elements that are required for a Facelets application.

All these are created automatically when you use NetBeans IDE to create an application:

- A context parameter specifying the project stage:

```
<context-param>
<param-name>
javax.faces.PROJECT_STAGE
</param-name>
<param-value>
Development
</param-value>
</context-param>
```

A context parameter provides configuration information needed by a web application.

An application can define its own context parameters.

In addition, JavaServer Faces technology and Java Servlet technology define context parameters that an application can use.

- A servlet element and its servlet-mapping element specifying the FacesServlet:

```
<servlet>
<servlet-name>
Faces Servlet
</servlet-name>
<servlet-class>
javax.faces.webapp.FacesServlet
</servlet-class>
```

```
<load-on-startup>
1
</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Faces
Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

A `welcome-file-list` element specifying the location of the landing page; note that the location is `faces/index.xhtml`, not just `index.xhtml`:

```
<welcome-file-list>
<welcome-file>
faces/index.xhtml
</welcome-file>
</welcome-file-list>
```

Packaging a Web Module

A web module must be packaged into a WAR in certain deployment scenarios and whenever you want to distribute the web module.

You package a web module into a WAR by executing the `jar` command in a directory laid out in the format of a web module, by using the Ant utility, or by using the IDE tool of your choice.

This tutorial shows you how to use NetBeans IDE or Ant to build, package, and deploy the `hello1` sample application.

To Set the Context Root

A context root identifies a web application in a Java EE server.

A context root must start with a forward slash (/) and end with a string.

In a packaged web module for deployment on the GlassFish Server, the context root is stored in `glassfish-web.xml`.

To view or edit the context root, follow these steps.

1. Expand the Web Pages and WEB-INF nodes of the `hello1` project.

2. Double-click `glassfish-web.xml`.

3. In the General tab, observe that the Context Root field is set to `/hello1`.

If you needed to edit this value, you could do so here.

When you create a new application, you type the context root here.

4. (Optional) Click the XML tab.

Observe that the context root value `/hello1` is enclosed by the `context-root` element.

You could also edit the value here.

To Build and Package the `hello1` Web Module Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/web/`

3. Select the `hello1` folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the `hello1` project and select Build.

To Build and Package the `hello1` Web Module Using Ant

1. In a terminal window, go to:

   `tut-install``/examples/web/hello1/`

2. Type the following command:

   `ant`

This command spawns any necessary compilations, copies files to the directory

tut-install`/examples/web/hello1/build/`, creates the WAR file, and copies it to the directory

tut-install`/examples/web/hello1/dist/`.

Deploying a Web Module

You can deploy a WAR file to the GlassFish Server by

- Using NetBeans IDE
- Using the Ant utility
- Using the `asadmin` command
- Using the Administration Console

- Copying the WAR file into the *domain-dir*`/autodeploy/` directory

Throughout the tutorial, you will use NetBeans IDE or Ant for packaging and deploying.

To Deploy the `hello1` Web Module Using NetBeans IDE

. **Right-click the `hello1` project and select Deploy.**

To Deploy the `hello1` *Web Module Using Ant*

1. In a terminal window, go to:

   `tut-install``/examples/web/hello1/`

2. Type the following command:

   ```
   ant deploy
   ```

Running a Deployed Web Module

Now that the web module is deployed, you can view it by opening the application in a web browser.

By default, the application is deployed to host `localhost` on port 8080.

The context root of the web application is `hello1`.

To Run a Deployed Web Module

1. Open a web browser.

2. Type the following URL:

   `http://localhost:8080/hello1/`

3. Type your name and click Submit.

The response page displays the name you submitted.

Click the Back button to try again.

Listing Deployed Web Modules

The GlassFish Server provides two ways to view the deployed web modules: the Administration Console and the `asadmin` command.

To List Deployed Web Modules Using the Administration Console

1. Open the URL `http://localhost:4848/` in a browser.

2. Select the Applications node.

   The deployed web modules appear in the Deployed Applications table.

To List Deployed Web Modules Using the `asadmin` *Command*

- **Type the following command:**

```
asadmin list-applications
```

Updating a Web Module

A typical iterative development cycle involves deploying a web module and then making changes to the application components.

To update a deployed web module, follow these steps.

To Update a Deployed Web Module

1. Recompile any modified classes.

2. Redeploy the module.

3. Reload the URL in the client.

Dynamic Reloading

If dynamic reloading is enabled, you do not have to redeploy an application or module when you change its code or deployment descriptors.

All you have to do is copy the changed pages or class files into the deployment directory for the application or module.

The deployment directory for a web module named *context-root* is

domain-dir`/applications/`*context-root*.

The server checks for changes periodically and redeploys the application, automatically and dynamically, with the changes.

This capability is useful in a development environment because it allows code changes to be tested quickly.

Dynamic reloading is not recommended for a production environment, however, because it may degrade performance.

In addition, whenever a reload is done, the sessions at that time become invalid, and the client must restart the session.

In the GlassFish Server, dynamic reloading is enabled by default.

To Disable or Modify Dynamic Reloading

If for some reason you do not want the default dynamic reloading behavior, follow these steps in the Administration Console.

1. Open the URL `http://localhost:4848/` in a browser.

2. Select the GlassFish Server node.

3. Select the Advanced tab.

4. To disable dynamic reloading, deselect the Reload Enabled check box.

5. To change the interval at which applications and modules are checked for code changes and dynamically reloaded, type a number of seconds in the Reload Poll Interval field.

The default value is 2 seconds.

6. Click the Save button.

Undeploying Web Modules

You can undeploy web modules and other types of enterprise applications by using either NetBeans IDE or the Ant tool.

To Undeploy the `hello1` Web Module Using NetBeans IDE

1. Ensure that the GlassFish Server is running.

2. In the Services window, expand the Servers node, GlassFish Server instance, and the Applications node.

3. Right-click the `hello1` module and choose Undeploy.

4. To delete the class files and other build artifacts, right-click the project and choose Clean.

To Undeploy the `hello1` *Web Module Using Ant*

1. In a terminal window, go to:

   tut-install`/examples/web/hello1/`

2. Type the following command:

   ```
   ant undeploy
   ```

3. To delete the class files and other build artifacts, type the following command:

```
ant clean
```

Configuring Web Applications: The `hello2` Example

Web applications are configured by means of annotations or by elements contained in the web application deployment descriptor.

The following sections give a brief introduction to the web application features you will usually want to configure.

Examples demonstrate procedures for configuring the Hello, World application.

Mapping URLs to Web Components

When it receives a request, the web container must determine which web component should handle the request.

The web container does so by mapping the URL path contained in the request to a web application and a web component.

A URL path contains the context root and, optionally, a URL pattern:

```
http://host:port/
context-root[/url-pattern]
```

You set the URL pattern for a servlet by using the `@WebServlet` annotation in the servlet source file.

For example, the `GreetingServlet.java` file in the `hello2` application contains the following annotation, specifying the URL pattern as `/greeting`:

```
@WebServlet("/greeting")
public class GreetingServlet
extends HttpServlet {
  ...
```

This annotation indicates that the URL pattern `/greeting` follows the context root.

Therefore, when the servlet is deployed locally, it is accessed with the following URL:

```
http://localhost:8080/hello2/
greeting
```

To access the servlet by using only the context root, specify `"/"` as the URL pattern.

Examining the `hello2` Web Module

The `hello2` application behaves almost identically to the `hello1` application, but it is implemented using Java Servlet technology instead of JavaServer Faces technology.

You can use a text editor to view the application files, or you can use NetBeans IDE.

To View the `hello2` Web Module Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/web/`

3. Select the `hello2` folder.

4. Select the Open as Main Project check box.

5. Expand the Source Packages node, then the `servlets` node.

6. Double-click the `GreetingServlet.java` file to view it.

This servlet overrides the `doGet` method, implementing the `GET` method of HTTP.

The servlet displays a simple HTML greeting form whose Submit button, like that of `hello1`, specifies a response page for its action.

The following excerpt begins with the `@WebServlet` annotation that specifies the URL pattern, relative to the context root:

```
@WebServlet("/greeting")
public class GreetingServlet
extends HttpServlet {
@Override
public void
doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException,
IOException {
response.
setContentType("text/html");
```

```
response.setBufferSize(8192);
PrintWriter out =
response.getWriter();
// then write the data of
// the response
out.println("<html>" +
"<head><title>
Servlet Hello</title></head>");
// then write the data of
// the response
```

```
out.println(
"<body bgcolor=\"#ffffff\">"
+ "<img src=\"duke.waving.gif\"
alt=\"Duke waving\">"
+ "<h2>Hello, my name is Duke.
What's yours?</h2>"
+ "<form method=\"get\">"
+ "<input type=\"text\"
name=\"username\" size=\"25\">"
+ "<p></p>"
```

```
+ "<input type=\"submit\"
value=\"Submit\">"
+ "<input type=\"reset\"
value=\"Reset\">"
+ "</form>");
String username =
request.getParameter("username");
if (username != null &&
username.length() > 0) {
```

```
RequestDispatcher dispatcher =
getServletContext().
getRequestDispatcher("/response");
if (dispatcher != null) {
dispatcher.
include(request, response);
}
}
out.println("</body></html>");
out.close();
} ...
```

7. Double-click the `ResponseServlet.java` file to view it.

This servlet also overrides the `doGet` method, displaying only the response.

The following excerpt begins with the `@WebServlet` annotation, which specifies the URL pattern, relative to the context root:

```
@WebServlet("/response")
public class ResponseServlet
extends HttpServlet {
@Override
public void
doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException,
IOException {
PrintWriter out =
response.getWriter();
```

```
// then write the data of
// the response
String username =
request.getParameter("username");
if (username != null &&
username.length() > 0) {
out.println("<h2>Hello, " +
username + "!</h2>");
}
}
...
```

8. Under the Web Pages node, expand the `WEB-INF` node and double-click the `glassfish-web.xml` file to view it.

In the General tab, observe that the Context Root field is set to `/hello2`.

For this simple servlet application, a `web.xml` file is not required.

Building, Packaging, Deploying, and Running the `hello2` Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `hello2` example.

To Build, Package, Deploy, and Run the `hello2` *Example Using NetBeans IDE*

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   tut-install`/examples/web/`

3. Select the `hello2` folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the `hello2` project and select Build.

7. Right-click the project and select Deploy.

8. In a web browser, open the URL `http://localhost:8080/hello2/greeting`.

The URL specifies the context root, followed by the URL pattern.

The application looks much like the `hello1` application shown in Figure 3-2.

The major difference is that after you click the Submit button, the response appears below the greeting, not on a separate page.

To Build, Package, Deploy, and Run the `hello2` Example Using Ant

1. In a terminal window, go to:
   `tut-install/examples/web/hello2/`

2. Type the following command:
   `ant`

This target builds the WAR file and copies it to the *tut-install*`/examples/web/hello2/dist/` directory.

3. Type `ant deploy`.

Ignore the URL shown in the deploy target output.

4. In a web browser, open the URL `http://localhost:8080/hello2/greeting`.

The URL specifies the context root, followed by the URL pattern.

The application looks much like the `hello1` application shown in Figure 3-2.

The major difference is that after you click the Submit button, the response appears below the greeting, not on a separate page.

Declaring Welcome Files

The welcome files mechanism allows you to specify a list of files that the web container will use for appending to a request for a URL (called a valid partial request) that is not mapped to a web component.

For example, suppose that you define a welcome file `welcome.html`.

When a client requests a URL such as *`host:port/webapp/directory`*, where *directory* is not mapped to a servlet or XHTML page, the file *host*:*port*/*webapp*/*directory*/`welcome.html` is returned to the client.

If a web container receives a valid partial request, the web container examines the welcome file list and appends to the partial request each welcome file in the order specified and checks whether a static resource or servlet in the WAR is mapped to that request URL.

The web container then sends the request to the first resource that matches in the WAR.

If no welcome file is specified, the GlassFish Server will use a file named `index.html` as the default welcome file.

If there is no welcome file and no file named `index.html`, the GlassFish Server returns a directory listing.

By convention, you specify the welcome file for a JavaServer Faces application as `faces/`*`file-name`*`.xhtml`.

Setting Context and Initialization Parameters

The web components in a web module share an object that represents their application context.

You can pass initialization parameters to the context or to a web component.

To Add a Context Parameter Using NetBeans IDE

These steps apply generally to web applications, but do not apply specifically to the examples in this chapter.

1. Open the project.

2. Expand the project's node in the Projects pane.

3. Expand the Web Pages node and then the WEB-INF node.

4. Double-click `web.xml`.

If the project does not have a `web.xml` file, follow the steps in To Create a `web.xml` File Using NetBeans IDE.

5. Click General at the top of the editor pane.

6. Expand the Context Parameters node.

7. Click Add.

An Add Context Parameter dialog opens.

8. In the Parameter Name field, type the name that specifies the context object.

9. In the Parameter Value field, type the parameter to pass to the context object.

10. Click OK.

To Create a `web.xml` File Using NetBeans IDE

1. From the File menu, choose New File.

2. In the New File wizard, select the Web category, then select Standard Deployment Descriptor under File Types.

3. Click Next.

4. Click Finish.

An empty `web.xml` file appears in `web/WEB-INF/`.

To Add an Initialization Parameter Using NetBeans IDE

You can use the `@WebServlet` annotation to specify web component initialization parameters by using the `initParams` attribute and the `@WebInitParam` annotation.

For example:

```
@WebServlet(urlPatterns="/MyPattern",
initParams={@WebInitParam(name="ccc",
value="333")})
```

Alternatively, you can add an initialization parameter to the `web.xml` file.

To do this using NetBeans IDE, follow these steps.

These steps apply generally to web applications, but do not apply specifically to the examples in this chapter.

1. Open the project.

2. Expand the project's node in the Projects pane.

3. Expand the Web Pages node and then the WEB-INF node.

4. Double-click `web.xml`.

If the project does not have a `web.xml` file, follow the steps in To Create a `web.xml` File Using NetBeans IDE.

5. Click Servlets at the top of the editor pane.

6. Click the Add button under the Initialization Parameters table.

An Add Initialization Parameter dialog opens.

7. In the Parameter Name field, type the name of the parameter.

8. In the Parameter Value Field, type the parameter's value.

9. Click OK.

Mapping Errors to Error Screens

When an error occurs during execution of a web application, you can have the application display a specific error screen according to the type of error.

In particular, you can specify a mapping between the status code returned in an HTTP response or a Java programming language exception returned by any web component and any type of error screen.

You can have multiple `error-page` elements in your deployment descriptor.

Each element identifies a different error that causes an error page to open.

This error page can be the same for any number of `error-page` elements.

To Set Up Error Mapping Using NetBeans IDE

These steps apply generally to web applications, but do not apply specifically to the examples in this chapter.

1. Open the project.

2. Expand the project's node in the Projects pane.

3. Expand the Web Pages node and then the WEB-INF node.

4. Double-click `web.xml`.

If the project does not have a `web.xml` file, follow the steps in To Create a `web.xml` File Using NetBeans IDE.

5. Click Pages at the top of the editor pane.

6. Expand the Error Pages node.

7. Click Add.

The Add Error Page dialog opens.

8. Click Browse to locate the page that you want to act as the error page.

9. In the Error Code field, type the HTTP status code that will cause the error page to be opened, or leave the field blank to include all error codes.

10. In the Exception Type field, type the exception that will cause the error page to load.

To specify all exceptions, type `java.lang.Throwable`.

11. Click OK.

Declaring Resource References

If your web component uses such objects as enterprise beans, data sources, or web services, you use Java EE annotations to inject these resources into your application.

Annotations eliminate a lot of the boilerplate lookup code and configuration elements that previous versions of Java EE required.

Although resource injection using annotations can be more convenient for the developer, there are some restrictions on using it in web applications.

First, you can inject resources only into container-managed objects, since a container must have control over the creation of a component so that it can perform the injection into a component.

As a result, you cannot inject resources into such objects as simple JavaBeans components.

However, JavaServer Faces managed beans are managed by the container; therefore, they can accept resource injections.

Components that can accept resource injections are listed in Table 3-1.

This section explains how to use a couple of the annotations supported by a servlet container to inject resources.

Chapter 33, Running the Persistence Examples, explains how web applications use annotations supported by the Java Persistence API.

Chapter 40, Getting Started Securing Web Applications, explains how to use annotations to specify information about securing web applications.

Table 3-1 Web Components That Accept Resource Injections

Component	Interface/Class
Servlets	`javax.servlet.Servlet`
Servlet filters	`javax.servlet.ServletFilter`

Event listeners	`javax.servlet.ServletContextListener`
	`javax.servlet.ServletContextAttributeListener`
	`javax.servlet.ServletRequestListener`
	`javax.servlet.ServletRequestAttributeListener`
	`javax.servlet.http.HttpSessionListener`
	`javax.servlet.http.HttpSessionAttributeListener`
	`javax.servlet.http.HttpSessionBindingListener`

Taglib listeners	Same as above
Taglib tag handlers	`javax.servlet.jsp.tagext.JspTag`
Managed beans	Plain Old Java Objects

Declaring a Reference to a Resource

The `@Resource` annotation is used to declare a reference to a resource, such as a data source, an enterprise bean, or an environment entry.

The `@Resource` annotation is specified on a class, a method, or a field.

The container is responsible for injecting references to resources declared by the `@Resource` annotation and mapping it to the proper JNDI resources.

In the following example, the `@Resource` annotation is used to inject a data source into a component that needs to make a connection to the data source, as is done when using JDBC technology to access a relational database:

```
@Resource
javax.sql.DataSource catalogDS;
public getProductsByCategory() {
// get a connection and execute
// the query
Connection conn =
catalogDS.getConnection(); ..
}
```

The container injects this data source prior to the component's being made available to the application.

The data source JNDI mapping is inferred from the field name `catalogDS` and the type, `javax.sql.DataSource`.

If you have multiple resources that you need to inject into one component, you need to use the `@Resources` annotation to contain them, as shown by the following example:

```
@Resources ({
@Resource (name="myDB"
type=java.sql.DataSource),
@Resource(name="myMQ"
type=javax.jms.ConnectionFactory)
})
```

The web application examples in this tutorial use the Java Persistence API to access relational databases.

This API does not require you to explicitly create a connection to a data source.

Therefore, the examples do not use the `@Resource` annotation to inject a data source.

However, this API supports the `@PersistenceUnit` and `@PersistenceContext` annotations for injecting `EntityManagerFactory` and `EntityManager` instances, respectively.

Chapter 33, Running the Persistence Examples describes these annotations and the use of the Java Persistence API in web applications.

Declaring a Reference to a Web Service

The `@WebServiceRef` annotation provides a reference to a web service.

The following example shows uses the `@WebServiceRef` annotation to declare a reference to a web service.

`WebServiceRef` uses the `wsdlLocation` element to specify the URI of the deployed service's WSDL file:

```
...
import javax.xml.ws.WebServiceRef;
...
public class ResponseServlet
extends HTTPServlet {
```

```
@WebServiceRef(wsdlLocation=
"http://localhost:8080/helloservice
/hello?wsdl")
static HelloService service;
```

Further Information about Web Applications

For more information on web applications, see

- JavaServer Faces 2.0 specification:

  http://jcp.org/en/jsr/detail?id=314

- JavaServer Faces technology web site:

http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

- Java Servlet 3.0 specification:

http://jcp.org/en/jsr/detail?id=315

- Java Servlet web site:

http://www.oracle.com/technetwork/java/index-jsp-135475.html