3

Getting Started with Web Applications

A web application is a dynamic extension of a web or application server. There are two types of web applications:

- *Presentation-oriented*: A presentation-oriented web application generates interactive web pages containing various types of markup language (HTML, XML, and so on) and dynamic content in response to requests. Chapters 11 through 22 cover how to develop presentation-oriented web applications.
- *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. Chapters 8 and 9 cover how to develop service-oriented web applications.

In the Java 2 platform, *web components* provide the dynamic extension capabilities for a web server. web components are either Java servlets, JSP pages, or web service endpoints. The interaction between a web client and a web application is illustrated in Figure 3–1. 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 it can pass the request to another web component. Eventu-

ally a web component generates a `HTTPServletResponse` object. The web server converts this object to an HTTP response and returns it to the client.

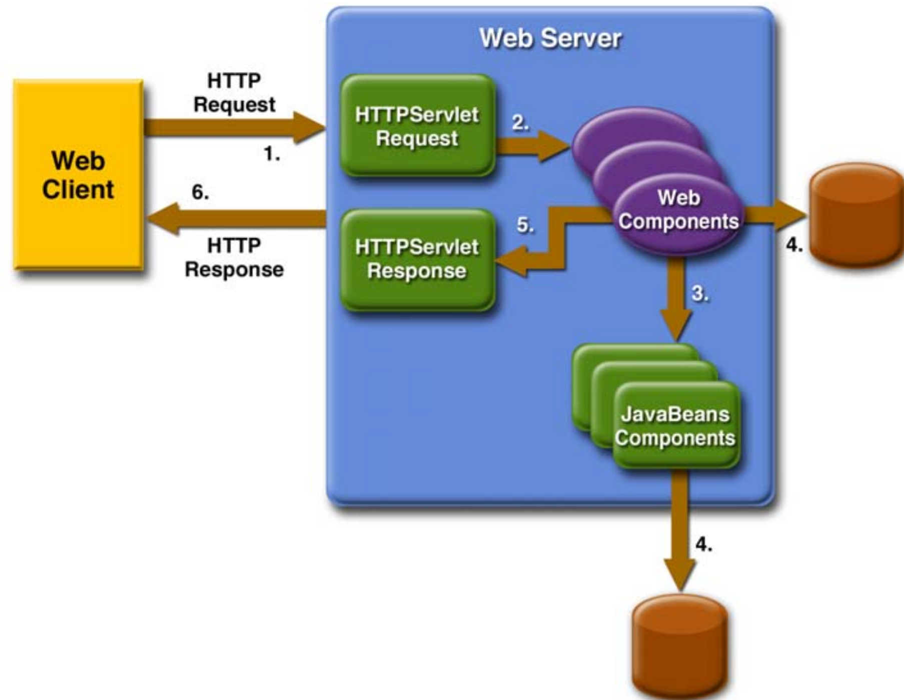

Figure 3–1 Java Web Application Request Handling

Servlets are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited for service-oriented applications (web service endpoints are implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, Scalable Vector Graphics (SVG), Wireless Markup Language (WML), and XML.

Since the introduction of Java Servlet and JSP technology, additional Java technologies and frameworks for building interactive web applications have been

developed. These technologies and their relationships are illustrated in Figure 3–2.

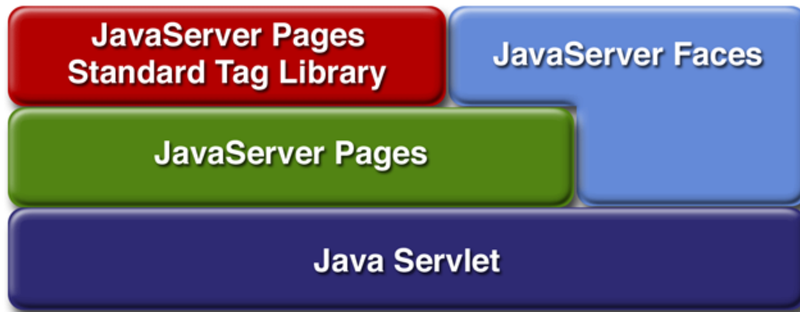

Figure 3–2 Java Web Application Technologies

Notice that Java Servlet technology is the foundation of all the web application technologies, so you should familiarize yourself with the material in Chapter 11 even if you do not intend to write servlets. Each technology adds a level of abstraction that makes web application prototyping and development faster and the web applications themselves more maintainable, scalable, and robust.

Web components are supported by the services of a runtime platform called a *web container*. A web container provides services such as request dispatching, security, concurrency, and life-cycle management. It also gives web components access to APIs such 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 is maintained in a text file in XML format called a *web application deployment descriptor* (DD). A DD must conform to the schema described in the Java Servlet Specification.

Most web applications use the HTTP protocol, and support for HTTP is a major aspect of web components. For a brief summary of HTTP protocol features see Appendix C.

This chapter gives a brief overview of the activities involved in developing web applications. First we summarize the web application life cycle. Then we describe how to package and deploy very simple web applications on the Sun Java System Application Server Platform Edition 8.1 2005Q1. We move on to configuring web applications and discuss how to specify the most commonly used configuration parameters. We then introduce an example—Duke's Book-

store—that we use to illustrate all the J2EE web-tier technologies and we describe how to set up the shared components of this example. Finally we discuss how to access databases from web applications and set up the database resources needed to run Duke's Bookstore.

Web Application Life Cycle

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.
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 3–3) and then displays a greeting after the name is submitted (Figure 3–4).

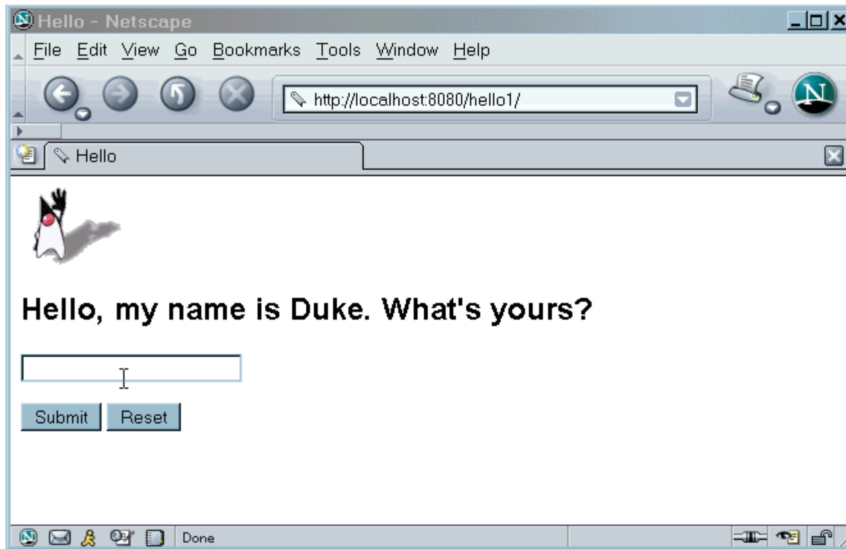

Figure 3–3 Greeting Form

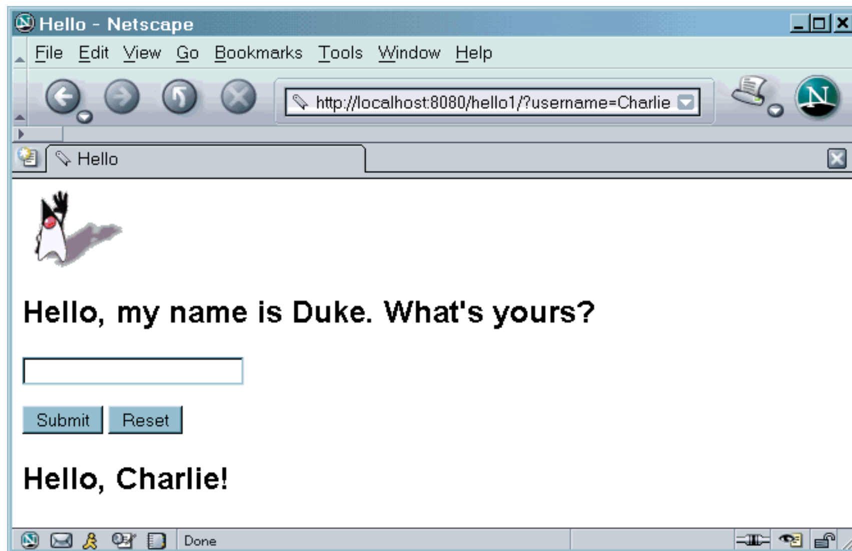

Figure 3–4 Response

The Hello application contains two web components that generate the greeting and the response. This chapter discusses two versions of the application: a JSP version called hello1, in which the components are implemented by two JSP pages (index.jsp and response.jsp) and a servlet version called hello2, in which the components are implemented by two servlet classes (GreetingServlet.java and ResponseServlet.java). The two versions are used to illustrate tasks involved in packaging, deploying, configuring, and running an application that contains web components. The section About the Examples (page xxxvi) explains how to get the code for these examples. After you install the tutorial bundle, the source code for the examples is in *<INSTALL>*/j2eetutorial14/examples/web/hello1/ and *<INSTALL>*/j2eetutorial14/examples/web/hello2/.

Web Modules

In the J2EE 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 J2EE 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 (database beans, shopping carts, and so on). Often these classes conform to the JavaBeans component architecture.
- Client-side classes (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 JSP 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 contains the following files and directories:

- web.xml: The web application deployment descriptor
- Tag library descriptor files (see Tag Library Descriptors, page 602)
- classes: A directory that contains *server-side classes*: servlets, utility classes, and JavaBeans components
- tags: A directory that contains tag files, which are implementations of tag libraries (see Tag File Location, page 588)

- `lib`: A directory that contains JAR archives of libraries called by server-side classes

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 Application Server, the file must also contain a runtime deployment descriptor. The runtime deployment descriptor is an XML file that contains information such as the context root of the web application and the mapping of the portable names of an application's resources to the Application Server's resources. The Application Server web application runtime DD is named `sun-web.xml` and is located in `/WEB-INF/` along with the web application DD. The structure of a web module that can be deployed on the Application Server is shown in Figure 3–5.

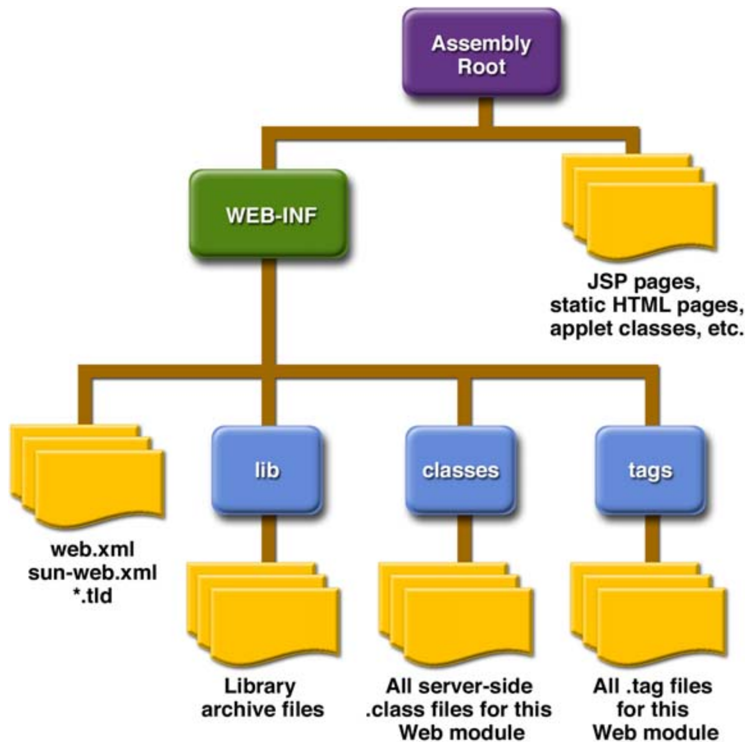

Figure 3–5 Web Module Structure

Packaging Web Modules

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 using the Application Server `deploytool` utility, by executing the `jar` command in a directory laid out in the format of a web module, or by using the `asant` utility. This tutorial allows you to use use either the first or the third approach. To build the `hello1` application, follow these steps:

1. In a terminal window, go to `<`*`INSTALL`*`>/j2eetutorial14/examples/web/hello1/`.
2. Run `asant build`. This target will spawn any necessary compilations and will copy files to the `<`*`INSTALL`*`>/j2eetutorial14/examples/web/hello1/build/` directory.

To package the application into a WAR named `hello1.war` using `asant`, use the following command:

```
asant create-war
```

This command uses `web.xml` and `sun-web.xml` files in the <*INSTALL*>/`j2eetutorial14/examples/web/hello1` directory.

To learn how to configure this web application, package the application using `deploytool` by following these steps:

1. Start `deploytool`.
2. Create a web application called `hello1` by running the New Web Component wizard. Select File→New→Web Component.
3. In the New Web Component wizard:
    a. Select the Create New Stand-Alone WAR Module radio button.
    b. In the WAR File field, enter <*INSTALL*>`/j2eetutorial14/examples/web/hello1/hello1.war`. The WAR Display Name field will show `hello1`.
    c. In the Context Root field, enter `/hello1`.
    d. Click Edit Contents to add the content files.
    e. In the Edit Contents dialog box, navigate to <*INSTALL*>/`j2eetutorial14/examples/web/hello1/build/`. Select `duke.waving.gif`, `index.jsp`, and `response.jsp` and click Add. Click OK.
    f. Click Next.
    g. Select the No Component radio button and click Next.
    h. Click Finish.
4. Select File→Save.

A sample `hello1.war` is provided in <*INSTALL*>`/j2eetutorial14/examples/web/provided-wars/`. To open this WAR with `deploytool`, follow these steps:

1. Select File→Open.
2. Navigate to the `provided-wars` directory.
3. Select the WAR.
4. Click Open Module.

Deploying Web Modules

You can deploy a web module to the Application Server in several ways:

- By pointing the Application Server at an unpackaged web module directory structure using `asadmin` or the Admin Console.
- By packaging the web module and
    - Copying the WAR into the `<J2EE_HOME>/domains/domain1/autodeploy/` directory.
    - Using the Admin Console, `asadmin`, `asant`, or `deploytool` to deploy the WAR.

All these methods are described briefly in this chapter; however, throughout the tutorial, we use `deploytool` or `asant` for packaging and deploying.

Setting the Context Root

A *context root* identifies a web application in a J2EE server. You specify the context root when you deploy a web module. A context root must start with a forward slash (`/`) and end with a string.

In a packaged web module for deployment on the Application Server, the context root is stored in `sun-web.xml`. If you package the web application with `deploytool`, then `sun-web.xml` is created automatically.

Deploying an Unpackaged Web Module

It is possible to deploy a web module without packaging it into a WAR. The advantage of this approach is that you do not need to rebuild the package every time you update a file contained in the web module. In addition, the Application Server automatically detects updates to JSP pages, so you don't even have to redeploy the web module when they change.

However, to deploy an unpackaged web module, you must create the web module directory structure and provide the web application deployment descriptor `web.xml`. Because this tutorial uses `deploytool` for generating deployment

descriptors, it does not document how to develop descriptors from scratch. You can view the structure of deployment descriptors in three ways:

- In `deploytool`, select Tools→Descriptor Viewer→Descriptor Viewer to view `web.xml` and Tools→Descriptor Viewer→Application Server Descriptor to view `sun-web.xml`.
- Use a text editor to view the `web.xml` and `sun-web.xml` files in the example directories.
- Unpackage one of the WARs in `<INSTALL>/j2eetutorial14/examples/web/provided-wars/` and extract the descriptors.

Since you explicitly specify the context root when you deploy an unpackaged web module, usually it is not necessary to provide `sun-web.xml`.

Deploying with the Admin Console

1. Expand the Applications node.
2. Select the Web Applications node.
3. Click the Deploy button.
4. Select the No radio button next to Upload File.
5. Type the full path to the web module directory in the File or Directory field. Although the GUI gives you the choice to browse to the directory, this option applies only to deploying a packaged WAR.
6. Click Next.
7. Type the application name.
8. Type the context root.
9. Select the Enabled box.
10. Click the OK button.

Deploying with asadmin

To deploy an unpackaged web module with `asadmin`, open a terminal window or command prompt and execute

```
asadmin deploydir full-path-to-web-module-directory
```

The build task for the hello1 application creates a build directory (including web.xml) in the structure of a web module. To deploy hello1 using asadmin deploydir, execute:

```
asadmin deploydir --contextroot /hello1
  <INSTALL>/j2eetutorial14/examples/web/hello1/build
```

After you deploy the hello1 application, you can run the web application by pointing a browser at

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

You should see the greeting form depicted earlier in Figure 3–3.

A web module is executed when a web browser references a URL that contains the web module's context root. Because no web component appears in http://localhost:8080/hello1/, the web container executes the default component, index.jsp. The section Mapping URLs to Web Components (page 99) describes how to specify web components in a URL.

Deploying a Packaged Web Module

If you have deployed the hello1 application, before proceeding with this section, undeploy the application by following one of the procedures described in Undeploying Web Modules (page 98).

Deploying with deploytool

To deploy the hello1 web module with deploytool:

1. Select the hello1 WAR you created in Packaging Web Modules (page 90).
2. Select Tools→Deploy.
3. Click OK.

You can use one of the following methods to deploy the WAR you packaged with deploytool, or one of the WARs contained in *<INSTALL>*/j2eetutorial14/examples/web/provided-wars/.

Deploying with the Admin Console

1. Expand the Applications node.
2. Select the Web Applications node.

3. Click the Deploy button.
4. Select the No radio button next to Upload File.
5. Type the full path to the WAR file (or click on Browse to find it), and then click the OK button.
6. Click Next.
7. Type the application name.
8. Type the context root.
9. Select the Enabled box.
10. Click the OK button.

Deploying with asadmin

To deploy a WAR with `asadmin`, open a terminal window or command prompt and execute

```
asadmin deploy full-path-to-war-file
```

Deploying with asant

To deploy a WAR with `asant`, open a terminal window or command prompt in the directory where you built and packaged the WAR, and execute

```
asant deploy-war
```

Listing Deployed Web Modules

The Application Server provides three ways to view the deployed web modules:

- `deploytool`
  a. Select localhost:4848 from the Servers list.
  b. View the Deployed Objects list in the General tab.
- Admin Console
  a. Open the URL `http://localhost:4848/asadmin` in a browser.
  b. Expand the nodes Applications→Web Applications.
- `asadmin`
  a. Execute
  `asadmin list-components`

Updating Web Modules

A typical iterative development cycle involves deploying a web module and then making changes to the application components. To update a deployed web module, you must do the following:

1. Recompile any modified classes.
2. If you have deployed a packaged web module, update any modified components in the WAR.
3. Redeploy the module.
4. Reload the URL in the client.

Updating an Unpackaged Web Module

To update an unpackaged web module using either of the methods discussed in Deploying an Unpackaged Web Module (page 92), reexecute the `deploydir` operation. If you have changed only JSP pages in the web module directory, you do not have to redeploy; simply reload the URL in the client.

Updating a Packaged Web Module

This section describes how to update the `hello1` web module that you packaged with `deploytool`.

First, change the greeting in the file `<INSTALL>/j2eetutorial14/examples/web/hello1/web/index.jsp` to

```
<h2>Hi, my name is Duke. What's yours?</h2>
```

Run `asant build` to copy the modified JSP page into the `build` directory. To update the web module using `deploytool` follow these steps:

1. Select the `hello1` WAR.
2. Select Tools→Update Module Files. A popup dialog box will display the modified file. Click OK.
3. Select Tools→Deploy. A popup dialog box will query whether you want to redeploy. Click Yes.
4. Click OK.

To view the modified module, reload the URL in the browser.

You should see the screen in Figure 3–6 in the browser.

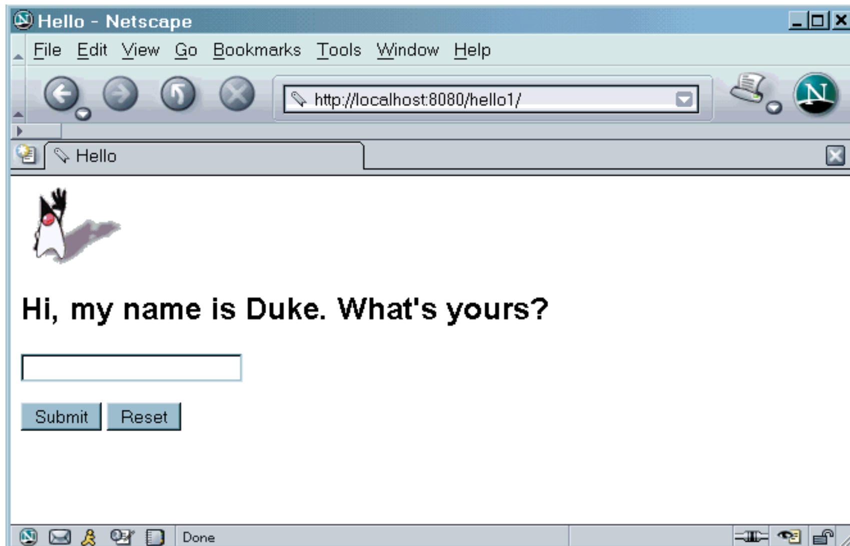

Figure 3–6 New Greeting

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 JSP or class files into the deployment directory for the application or module. The deployment directory for a web module named *context_root* is <*J2EE_HOME*>/domains/domain1/applications/j2ee-modules/*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.

To enable dynamic reloading, use the Admin Console:

1. Select the Applications node.
2. Check the Reload Enabled box to enable dynamic reloading.

3. Enter a number of seconds in the Reload Poll Interval field to set the interval at which applications and modules are checked for code changes and dynamically reloaded.
4. Click the Save button.

In addition, to load new servlet files or reload deployment descriptor changes, you must do the following:

1. Create an empty file named `.reload` at the root of the module:

   ```
   <J2EE_HOME>/domains/domain1/applications/j2ee-modules/
   context_root/.reload
   ```

2. Explicitly update the `.reload` file's time stamp each time you make these changes. On UNIX, execute

   ```
   touch .reload
   ```

For JSP pages, changes are reloaded automatically at a frequency set in the Reload Pool Interval. To disable dynamic reloading of JSP pages, set the reload-interval property to -1.

Undeploying Web Modules

You can undeploy web modules in four ways:

- `deploytool`
  a. Select localhost:4848 from the Servers list.
  b. Select the web module in the Deployed Objects list of the General tab.
  c. Click the Undeploy button.
- Admin Console
  a. Open the URL `http://localhost:4848/asadmin` in a browser.
  b. Expand the Applications node.
  c. Select Web Applications.
  d. Click the checkbox next to the module you wish to undeploy.
  e. Click the Undeploy button.
- `asadmin`
  a. Execute

     ```
     asadmin undeploy context_root
     ```

- `asant`
  a. In the directory where you built and packaged the WAR, execute
  ```
  asant undeploy-war
  ```

Configuring Web Applications

Web applications are configured via elements contained in the web application deployment descriptor. The `deploytool` utility generates the descriptor when you create a WAR and adds elements when you create web components and associated classes. You can modify the elements via the inspectors associated with the WAR.

The following sections give a brief introduction to the web application features you will usually want to configure. A number of security parameters can be specified; these are covered in Web-Tier Security (page 1125).

In the following sections, examples demonstrate procedures for configuring the Hello, World application. If Hello, World does not use a specific configuration feature, the section gives references to other examples that illustrate how to specify the deployment descriptor element and describes generic procedures for specifying the feature using `deploytool`. Extended examples that demonstrate how to use `deploytool` appear in later tutorial chapters.

Mapping URLs to Web Components

When a request is received by the web container it must determine which web component should handle the request. It 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 an alias:

```
http://host:port/context_root/alias
```

Setting the Component Alias

The *alias* identifies the web component that should handle a request. The alias path must start with a forward slash (`/`) and end with a string or a wildcard expression with an extension (for example, `*.jsp`). Since web containers automatically map an alias that ends with `*.jsp`, you do not have to specify an alias for a JSP page unless you wish to refer to the page by a name other than its file

name. To set up the mappings for the servlet version of the hello application with deploytool, first package it, as described in the following steps.

1. In a terminal window, go to <INSTALL>/j2eetutorial14/examples/web/hello2/.
2. Run asant build. This target will compile the servlets to the <INSTALL>/j2eetutorial14/examples/web/hello2/build/ directory.
3. Start deploytool.
4. Create a web application called hello2 by running the New Web Component wizard. Select File→New→Web Component.
5. In the New Web Component wizard:
    a. Select the Create New Stand-Alone WAR Module radio button.
    b. In the WAR File field, enter <INSTALL>/j2eetutorial14/examples/web/hello2/hello2.war. The WAR Display Name field will show hello2.
    c. In the Context Root field, enter /hello2.
    d. Click Edit Contents to add the content files.
    e. In the Edit Contents dialog box, navigate to <INSTALL>/j2eetutorial14/examples/web/hello2/build/. Select duke.waving.gif and the servlets package and click Add. Click OK.
    f. Click Next.
    g. Select the Servlet radio button and click Next.
    h. Select GreetingServlet from the Servlet Class combo box.
    i. Click Finish.
6. Select File→New→Web Component.
    a. Click the Add to Existing WAR Module radio button and select hello2 from the combo box. Because the WAR contains all the servlet classes, you do not have to add any more content.
    b. Click Next.
    c. Select the Servlet radio button and click Next.
    d. Select ResponseServlet from the Servlet Class combo box and click Finish.

Then, to set the aliases, follow these steps:

1. Select the GreetingServlet web component.
2. Select the Aliases tab.

3. Click Add to add a new mapping.
4. Type `/greeting` in the aliases list.
5. Select the `ResponseServlet` web component.
6. Click Add.
7. Type `/response` in the aliases list.
8. Select File→Save.

To run the application, first deploy the web module, and then open the URL `http://localhost:8080/hello2/greeting` in a browser.

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 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 JSP 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 in the WAR that matches.

If no welcome file is specified, the Application Server will use a file named `index.`*`XXX`*, where *`XXX`* can be `html` or `jsp`, as the default welcome file. If there is no welcome file and no file named `index.`*`XXX`*, the Application Server returns a directory listing.

To specify welcome files with `deploytool`, follow these steps:

1. Select the WAR.
2. Select the File Ref's tab in the WAR inspector.
3. Click Add File in the Welcome Files pane.
4. Select the welcome file from the drop-down list.

The example discussed in Encapsulating Reusable Content Using Tag Files (page 586) has a welcome file.

Setting Initialization Parameters

The web components in a web module share an object that represents their application context (see Accessing the Web Context, page 471). You can pass initialization parameters to the context or to a web component.

To add a context parameter with `deploytool`, follow these steps:

1. Select the WAR.
2. Select the Context tab in the WAR inspector.
3. Click Add.

For a sample context parameter, see the example discussed in The Example JSP Pages (page 484).

To add a web component initialization parameter with `deploytool`, follow these steps:

1. Select the web component.
2. Select the Init. Parameters tab in the web component inspector.
3. Click Add.

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 (see Handling Errors, page 450) and any type of error screen. To set up error mappings with `deploytool`:

1. Select the WAR.
2. Select the File Ref's tab in the WAR inspector.
3. Click Add Error in the Error Mapping pane.
4. Enter the HTTP status code (see HTTP Responses, page 1400) or the fully qualified class name of an exception in the Error/Exception field.
5. Enter the name of a web resource to be invoked when the status code or exception is returned. The name should have a leading forward slash (`/`).

Note: You can also define error screens for a JSP page contained in a WAR. If error screens are defined for both the WAR and a JSP page, the JSP page's error page takes precedence. See Handling Errors (page 493).

For a sample error page mapping, see the example discussed in The Example Servlets (page 442).

Declaring Resource References

If your web component uses objects such as databases and enterprise beans, you must declare the references in the web application deployment descriptor. For a sample resource reference, see Specifying a Web Application's Resource Reference (page 106). For a sample enterprise bean reference, see Specifying the Web Client's Enterprise Bean Reference (page 892).

Duke's Bookstore Examples

In Chapters 11 through 22 a common example—Duke's Bookstore—is used to illustrate the elements of Java Servlet technology, JavaServer Pages technology, the JSP Standard Tag Library, and JavaServer Faces technology. The example emulates a simple online shopping application. It provides a book catalog from which users can select books and add them to a shopping cart. Users can view and modify the shopping cart. When users are finished shopping, they can purchase the books in the cart.

The Duke's Bookstore examples share common classes and a database schema. These files are located in the directory `<INSTALL>/j2eetutorial14/examples/web/bookstore/`. The common classes are packaged into a JAR. To create the `bookstore` library JAR, follow these steps:

1. In a terminal window, go to `<INSTALL>/j2eetutorial14/examples/web/bookstore/`.
2. Run `asant build` to compile the bookstore files.
3. Run `asant package-bookstore` to create a library named `bookstore.jar` in `<INSTALL>/j2eetutorial14/examples/bookstore/dist/`.

The next section describes how to create the bookstore database tables and resources required to run the examples.

Accessing Databases from Web Applications

Data that is shared between web components and is persistent between invocations of a web application is usually maintained in a database. web applications use the JDBC API to access relational databases. For information on this API, see

```
http://java.sun.com/docs/books/tutorial/jdbc
```

In the JDBC API, databases are accessed via `DataSource` objects. A `DataSource` has a set of properties that identify and describe the real world data source that it represents. These properties include information such as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on.

Web applications access a data source using a connection, and a `DataSource` object can be thought of as a factory for connections to the particular data source that the `DataSource` instance represents. In a basic `DataSource` implementation, a call to the `getConnection` method returns a connection object that is a physical connection to the data source. In the Application Server, a data source is referred to as a JDBC resource. See DataSource Objects and Connection Pools (page 1109) for further information about data sources in the Application Server.

If a `DataSource` object is registered with a JNDI naming service, an application can use the JNDI API to access that `DataSource` object, which can then be used to connect to the data source it represents.

To maintain the catalog of books, the Duke's Bookstore examples described in Chapters 11 through 22 use the PointBase evaluation database included with the Application Server.

This section describes how to

- Populate the database with bookstore data
- Create a data source in the Application Server
- Specify a web application's resource reference
- Map the resource reference to the data source defined in the Application Server

Populating the Example Database

To populate the database for the Duke's Bookstore examples, follow these steps:

1. In a terminal window, go to `<INSTALL>/j2eetutorial14/examples/web/bookstore/`.
2. Start the PointBase database server. For instructions, see Starting and Stopping the PointBase Database Server (page 29).
3. Run `asant create-db_common`. This task runs a PointBase commander tool command to read the file `books.sql` and execute the SQL commands contained in the file.
4. At the end of the processing, you should see the following output:

```
...
[java] SQL> INSERT INTO books VALUES('207', 'Thrilled', 'Ben',
[java]  'The Green Project: Programming for Consumer Devices',
[java]  30.00, false, 1998, 'What a cool book', 20);
[java] 1 row(s) affected

[java] SQL> INSERT INTO books VALUES('208', 'Tru', 'Itzal',
[java]  'Duke: A Biography of the Java Evangelist',
[java]  45.00, true, 2001, 'What a cool book.', 20);
[java] 1 row(s) affected
```

Creating a Data Source in the Application Server

Data sources in the Application Server implement connection pooling. To define the Duke's Bookstore data source, you use the installed PointBase connection pool named PointBasePool.

You create the data source using the Application Server Admin Console, following this procedure:

1. Expand the JDBC node.
2. Select the JDBC Resources node.
3. Click the New... button.
4. Type `jdbc/BookDB` in the JNDI Name field.
5. Choose `PointBasePool` for the Pool Name.
6. Click OK.

Specifying a Web Application's Resource Reference

To access a database from a web application, you must declare a resource reference in the application's web application deployment descriptor (see Declaring Resource References, page 103). The resource reference specifies a JNDI name, the type of the data resource, and the kind of authentication used when the resource is accessed. To specify a resource reference for a Duke's Bookstore example using `deploytool`, follow these steps:

1. Select the WAR (created in Chapters 11 through 22).
2. Select the Resource Ref's tab.
3. Click Add.
4. Type `jdbc/BookDB` in the Coded Name field.
5. Accept the default type `javax.sql.DataSource`.
6. Accept the default authorization `Container`.
7. Accept the default `Sharable` selected.

To create the connection to the database, the data access object `database.BookDBAO` looks up the JNDI name of the bookstore data source object:

```
public BookDBAO () throws Exception {
  try  {
    Context initCtx = new InitialContext();
    Context envCtx = (Context)
      initCtx.lookup("java:comp/env");
    DataSource ds = (DataSource) envCtx.lookup("jdbc/BookDB");
    con =  ds.getConnection();
    System.out.println("Created connection to database.");
  } catch (Exception ex) {
    System.out.println("Couldn't create connection." +
      ex.getMessage());
    throw new
      Exception("Couldn't open connection to database: "
      + ex.getMessage());
  }
```

Mapping the Resource Reference to a Data Source

Both the web application resource reference and the data source defined in the Application Server have JNDI names. See JNDI Naming (page 1107) for a discussion of the benefits of using JNDI naming for resources.

To connect the resource reference to the data source, you must map the JNDI name of the former to the latter. This mapping is stored in the web application runtime deployment descriptor. To create this mapping using `deploytool`, follow these steps:

1. Select localhost:4848 in the Servers list to retrieve the data sources defined in the Application Server.
2. Select the WAR in the Web WARs list.
3. Select the Resource Ref's tab.
4. Select the Resource Reference Name, `jdbc/BookDB`, defined in the previous section.
5. In the Sun-specific Settings frame, select `jdbc/BookDB` from the JNDI Name drop-down list.

Further Information

For more information about web applications, refer to the following:

- Java Servlet specification:
  `http://java.sun.com/products/servlet/download.html#specs`
- The Java Servlet web site:
  `http://java.sun.com/products/servlet`