Oracle9*i*AS Containers for J2EE

Servlet Developer's Guide

Release 2 (9.0.2)

January 2002

Part No. A95878-01

ORACLE

Oracle9*i*AS Containers for J2EE Servlet Developer's Guide, Release 2 (9.0.2)

Part No. A95878-01

Primary Authors: Brian Wright, Tim Smith

Contributors: Jasen Minton, Joyce Yang, Sunil Kunisetty, Bryan Atsatt, Ashok Banerjee, Charlie Shapiro, Philippe Le Mouel, Paolo Ramasso, Olaf Heimburger, Sheryl Maring, Mike Sanko, Ellen Barnes

Contents

2 Servlet Development

3 Deployment and Configuration

4 Servlet Filters

A Third Party Licenses

Send Us Your Comments

Oracle9*i*AS Containers for J2EE Servlet Developer's Guide, Release 2 (9.0.2)

Part No. A95878-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: jpgcomment_us@oracle.com
- FAX: (650) 506-7225 Attn: Java Platform Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Java Platform Group, Information Development Manager
  500 Oracle Parkway, Mailstop 4op9
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This guide describes the servlet container of the Oracle9*i*AS Containers for J2EE (OC4J) application server, including discussion of basic servlets, data-access servlets, and servlet filters. It also provides an overview of OC4J deployment and configuration, with detailed descriptions of key configuration files.

This preface contains these topics:

Intended Audience

The guide is intended for J2EE developers who are writing Web applications that use servlets and possibly JavaServer Pages (JSP). It provides the basic information you will need regarding the OC4J servlet container.

It does not attempt to teach servlet programming, nor does it document the Java Servlet API. To learn about these topics, see the documentation available from Sun Microsystems, or look at one of the trade books on servlet programming.

If you are developing applications that primarily use JavaServer Pages, read the *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Organization

This document contains:

Chapter 1, "Servlet Overview"

Introduces the OC4J servlet container and briefly discusses servlet development in general, using a simple "Hello World" example.

Chapter 2, "Servlet Development"

Describes how the OC4J servlet container supports servlet development and invocation. Provides several complete examples.

Chapter 3, "Deployment and Configuration"

Discusses how to configure the OC4J servlet environment and deploy a Web application in OC4J.

Chapter 4, "Servlet Filters"

Explains the use of filters (new in the Servlet 2.3 specification) to affect servlet input or output.

Appendix A, "Third Party Licenses"

This appendix includes the Third Party License for third party products included with Oracle9*i* Application Server and discussed in this document.

Related Documentation

See the following additional OC4J documents available from the Oracle Java Platform group:

- *Oracle9iAS Containers for J2EE User's Guide*

  This book provides some overview and general information for OC4J; primer chapters for servlets, JSP pages, and EJBs; and general configuration and deployment instructions.

- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*

  This book provides information for JSP developers who want to run their pages in OC4J. It includes a general overview of JSP standards and programming considerations, as well as discussion of Oracle value-added features and steps for getting started in the OC4J environment.

- *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information and detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J.

- *Oracle9iAS Containers for J2EE Services Guide*

  This book provides information about basic Java services supplied with OC4J, such as JTA, JNDI, and the Oracle9*i* Application Server Java Object Cache.

- *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*

  This book provides information about the EJB implementation and EJB container in OC4J.

Also available from the Oracle Java Platform group:

- *Oracle9i JDBC Developer's Guide and Reference*
- *Oracle9i SQLJ Developer's Guide and Reference*
- *Oracle9i JPublisher User's Guide*
- *Oracle9i Java Stored Procedures Developer's Guide*

The following documents are available from the Oracle9*i* Application Server group:

- *Oracle9i Application Server Administrator's Guide*
- *Oracle Enterprise Manager Administrator's Guide*
- *Oracle HTTP Server Administration Guide*
- *Oracle9i Application Server Performance Guide*
- *Oracle9i Application Server Globalization Support Guide*
- *Oracle9iAS Web Cache Administration and Deployment Guide*
- *Oracle9i Application Server: Migrating from Oracle9i Application Server 1.x*

The following are available from the JDeveloper group:

- Oracle JDeveloper online help
- Oracle JDeveloper documentation on the Oracle Technology Network:

  `http://otn.oracle.com/products/jdev/content.html`

The following documents from the Oracle Server Technologies group may also contain information of interest:

- *Oracle9i XML Developer's Kits Guide - XDK*
- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle9i Supplied Java Packages Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle9i SQL Reference*
- *Oracle9i Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle9i Database Reference*
- *Oracle9i Database Error Messages*

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/admin/account/membership.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/docs/index.htm`

The following resources are available from Sun Microsystems:

- Web site for Java Servlet technology, including the latest specifications:

  `http://java.sun.com/products/servlet/index.html`

- Web site for JavaServer Pages, including the latest specifications:

  `http://java.sun.com/products/jsp/index.html`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the data files and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
lowercase italic monospace (fixed-width) font	Lowercase italic monospace font represents place holders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
`...`	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS subquery;`
	■ That you can repeat a portion of the code	`SELECT col1, col2, ... , coln FROM employees;`
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates place holders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

1

Servlet Overview

Oracle9*i*AS Containers for J2EE (OC4J) enables you to develop standard J2EE-compliant applications. Applications are packaged in standard EAR (Enterprise ARchive) deployment files, which include standard WAR (Web ARchive) files to deploy the Web modules, and JAR files for any EJB and application client modules in the application.

The most important thing to understand about servlet development under OC4J is how the Web application is built and deployed. If OC4J is a new development environment for you, see Chapter 2, "Servlet Development", and Chapter 3, "Deployment and Configuration", to learn how applications are deployed under OC4J.

This chapter introduces the Java servlet and provides an example of a basic servlet. It also briefly discusses how you can use servlets in a J2EE application to address some server-side programming issues.

This chapter covers the following topics:

Information Sources

This section points you to other information sources about servlets and OC4J.

Servlet Information

You should be generally familiar with the Sun Microsystems *Java(TM) Servlet Specification, Version 2.3*. This is especially true if you are developing a distributable Web application, in which sessions can be replicated to servers running under more than one Java Virtual Machine (JVM).

You can obtain the Servlet 2.3 specification at the following location:

```
http://jcp.org/aboutJava/communityprocess/first/jsr053/index.html
```

This guide is not a complete reference for servlet development. For example, it does not cover the standard servlet APIs. For servlet API documentation, refer to the Javadoc available from Sun Microsystems at the following location:

```
http://java.sun.com/products/servlet/2.3/javadoc/index.html
```

In addition, several trade press books are available to teach you how to develop servlets and deploy them in J2EE-compatible applications. In particular, the books from O'Reilly & Associates (`http://www.oreilly.com`) and Wrox (`http://www.wrox.com`) are useful.

Additional OC4J Documents

You should use this guide together with the following additional Oracle9*i*AS publications:

- *Oracle9iAS Containers for J2EE User's Guide*, for general information about OC4J features, simple primers to help you get started, and information about deployment to OC4J
- *Oracle9iAS Containers for J2EE Services Guide*, for information about basic J2EE services supplied with OC4J, such as JTA, JNDI, JMS, JCA, and security. It also covers the Oracle Java Object Cache (formerly OCS4J)
- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*, for information about developing JSP pages for OC4J
- *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*, for coverage of EJB development for OC4J

Introduction to Servlets

A servlet is a Java program that runs in a J2EE application server, such as OC4J. A servlet is the server-side counterpart of a Java applet. Servlets are one of the four application component types of a J2EE application, others being applets and application client programs on the client side, and EJBs on the server side. Servlets are managed by the OC4J servlet container; EJBs are managed by the OC4J EJB container. These containers, together with the JavaServer Pages container, form the core of OC4J.

JavaServer Pages (JSP) is another server-side component type. JSP pages also involve the servlet container, because the JSP container itself is a servlet and is therefore executed by the servlet container. The JSP container translates JSP pages into page implementation classes, which are executed by the JSP container but function similarly to servlets. See the JSP chapter in the *Oracle9iAS Containers for J2EE User's Guide* and the *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference* for more information about JavaServer Pages.

Most servlets generate HTML text, which is then sent back to the client for display by the Web browser, or is sent on to other components in the application. Servlets can also generate XML, to encapsulate data, and send this to the client or to other components.

The remainder of this section covers the following topics:

Advantages of Servlets

Servlet programming offers advantages over earlier models of server-side Web application development, including the following:

- Servlet programming is a mature and popular standard, effectively replacing early techniques, such as CGI scripts, for generating dynamic HTML.
- A servlet handles concurrent requests. There is only a single instance of each servlet, and servlets have a well-defined lifecycle. For higher performance, servlets can be loaded when OC4J starts.
- Servlets are fully integrated into the J2EE framework.

- The servlet request and response objects provide a convenient way to handle HTTP requests and send text and data back to the client.
- Servlets are fully integrated with the Java language and its standard APIs. The J2EE framework provides an extensive set of services that your Web application can use, such as JNDI, JMS, RMI, and security.

Because servlets are written in the Java programming language, they are supported on any platform that has a Java virtual machine and a Web server that supports servlets. Servlets can be used on different platforms without recompiling. You can package servlets together with associated files such as graphics, sounds, and other data to make a complete Web application. This simplifies application development and deployment.

In addition, you can port a servlet-based application from another Web server to OC4J with little effort. If your application was developed for a J2EE-compliant Web server, then the porting effort is minimal.

Servlets outperform earlier technologies for generating dynamic HTML, such as server-side "includes" or CGI scripts. Once a servlet is loaded into memory, it can run on a single lightweight thread; CGI scripts must be loaded in a different process for every request.

Servlets and the Servlet Container

As with an applet, but unlike a Java client program, a servlet has no static `main()` method. Therefore, a servlet must execute under the control of a servlet container, because it is the container that calls servlet methods and provides services that the servlet needs when executing.

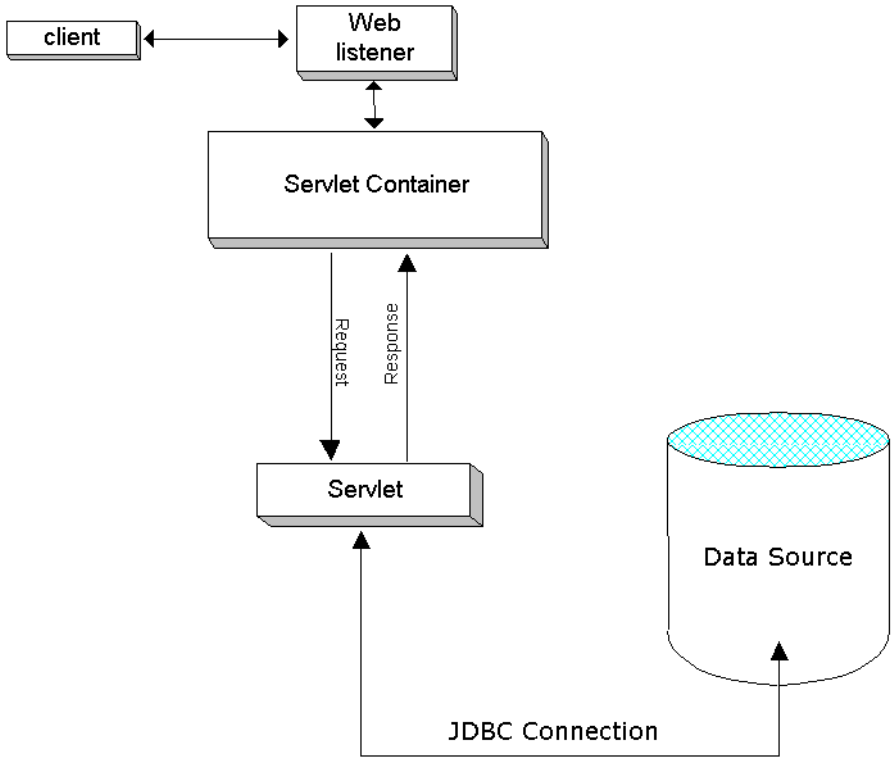
The servlet overrides the appropriate methods from the servlet base class—`javax.servlet.HttpServlet` class for HTTP servlets, or possibly `javax.servlet.GenericServlet` for protocol-independent servlets—in order to process the request and return the response. For example, most servlets override the `HttpServlet` `doGet()` method or `doPost()` method or both to process `HTTP GET` and `POST` requests.

The servlet container provides the servlet easy access to properties of the HTTP request, such as its headers and parameters. Also, a servlet can use other Java APIs, such as JDBC to access a database, RMI to call remote objects, JMS for asynchronous messaging, or many other Java and J2EE services.

Figure 1–1 shows how a servlet relates to the servlet container and to a client, such as a Web browser. When the Web listener is the Oracle HTTP Server (powered by

Apache), then the connection to the OC4J servlet container goes through the `mod_oc4j` module. See the *Oracle HTTP Server Administration Guide* for details.

Figure 1–1 Servlets and the Servlet Container



Request Objects, Response Objects, and Filters

The `HttpServlet` methods, such as `doGet()` and `doPost()`, take two parameters: a `javax.servlet.http.HttpServletRequest` object and a `javax.servlet.http.HttpServletResponse` object (instances of classes that implement the `HttpServletRequest` and `HttpServletResponse` interfaces). The servlet container passes these objects to the servlet, or to the next filter if there is a filter chain.

The Servlet 2.3 specification enables servlet filters, which are Java programs that execute on the server and can be interposed between the servlet (or group of servlets) and the servlet container for special request or response processing. See Chapter 4, "Servlet Filters", for more information.

The request and response objects support methods that let you write efficient servlet code. "A First Servlet Example" on page 1-7 shows that you can get a stream writer object from the response and use it to write statements to the response stream.

Session Tracking

Servlets provide convenient ways to keep the client and a server session in synchronization, enabling stateful servlets to maintain session state on the server over the whole duration of a client browsing session.

These techniques involve cookies or URL rewriting, and the `javax.servlet.http.HttpSession` object. See "Session Tracking" on page 2-10 for more information.

A First Servlet Example

Looking at a basic example is the best way to see how servlets are coded and what they can do. This section shows the code for a simple servlet, but with a twist for globalization. The code is commented to explain the basics of servlet development.

Hello World Code

A "Hello World" example is a good way to demonstrate the basic framework that you use to write a servlet. This servlet just prints the date and a greeting back to the client, but in Swedish.

Here is the code:

```
import java.io.*;
import java.text.*;
import java.util.*;
// The first three package imports support I/O, and the locale info and date
// formatting. The next two imports include the packages that support
// servlet development.
import javax.servlet.*;
import javax.servlet.http.*;
// HTTP servlets extend the javax.servlet.http.HttpServlet class.
public class HelloWorldServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    // doGet() overrides the HttpServlet method. Each method of this class
    // has request and/or response parameters.

    // Set the content type of the response.
    res.setContentType("text/plain");

    // Get a print writer stream to write output to the response. You
    // could also get s ServletOutputStream object to do this.
    PrintWriter out = res.getWriter();

    // This statement tells the client the language of the content--Swedish.
    // However, many Web browsers will ignore this info.
    res.setHeader("Content-Language", "sv");

    // Set the locale information, so the date will be formatted in a Swedish-
    // friendly way, and the right words are used for months and so on.
    Locale locale = new Locale("sv", "");
```

```
    // Get the date format.
    DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                           DateFormat.LONG,
                                                           locale);

    // Also set the local time zone.
    dateFormat.setTimeZone(TimeZone.getDefault());

    // Now use the printer object to send some HTML header info to the
    // output stream.
    out.println("<HTML><HEAD><TITLE>Hej V\u00e4rlden!</TITLE></HEAD>");
    out.println("<BODY>");

    // Send the date to the output.
    out.println(dateFormat.format(new Date()));

   // And then, greet the client--
    out.println("<p>In Swedish (p\u00E5 Svenska):");
    out.println("<H2>Hej V\u00E4rlden!</H2>");

    // Don't forget to close the HTML tags.
    out.println("</BODY></HTML>");
  }
}
```

Compiling and Deploying the Servlet

To try out this servlet in your OC4J server, enter the code using your favorite text editor, and save it as `HelloWorldServlet.java` in the `j2ee/home/default-web-apps/WEB-INF/classes` directory. This is the location where the container finds servlet classes for the OC4J default application. Next, compile the servlet, using a Java 1.3.x-compliant compiler.

For convenience during development and testing, use the OC4J auto-compile feature for servlet code. Set the attribute `development="true"` in the `<orion-web-app>` element of the `global-web-application.xml` configuration file. You may also have to set the `source-directory` attribute appropriately. With auto-compile enabled, after you change the servlet source and save it in a specified directory, the OC4J server automatically compiles and redeploys the servlet the next time it is invoked.

You can find `global-web-application.xml` in the `config` directory under the `j2ee/home` directory. See "Element Descriptions for global-web-application.xml

and orion-web.xml" on page 3-15 for more information about `development` and `source-directory`.

Running the Servlet

Assuming the OC4J server is up and running, by default you can invoke the servlet and see its output from a Web browser as follows, where *<host>* is the name of the host that the OC4J server is running on, and *<port>* is the Web listener port:

```
http://<host>:<port>/j2ee/servlet/HelloWorldServlet
```

This example assumes that `/j2ee` is the root context of the Web application. By default, this is the case for the OC4J default Web application. Typically, use port 7777 for access through the Oracle HTTP Server with Oracle9*i*AS Web Cache enabled.

For related information, see "Invoking a Servlet" on page 2-5 and "The global-web-application.xml and orion-web.xml Files" on page 3-12.

2

Servlet Development

This chapter provides basic information for developing servlets for OC4J and the Oracle9*i* Application Server, covering the following topics:

- Servlet Development Basics
- Invoking a Servlet
- Servlet Loading and Initialization
- Servlet Sessions
- Use of JDBC in Servlets
- EJB Calls from Servlets

Servlet Development Basics

Most HTTP servlets follow a standard form. They are written as public classes that extend the `HttpServlet` class. A servlet overrides the `init()` and `destroy()` methods when code is required to perform initialization work at the time the servlet is loaded by the container, or when finalization code is required when the container shuts the servlet down. Most servlets override either the `doGet()` method or the `doPost()` method of `HttpServlet`, to handle HTTP `GET` or `POST` requests. These two methods take request and response parameters.

This chapter provides sample servlets that are more advanced than the `HelloWorldServlet` in "A First Servlet Example" on page 1-7. You can test each of these servlets using the OC4J default Web application. To do this, save the Java source files in the following directory:

```
j2ee/home/default-web-app/WEB-INF/classes
```

To test some of the servlets, you might have to make changes to the `web.xml` file in the `j2ee/home/default-web-app/WEB-INF` directory, as directed. When you change and save the `web.xml` file, OC4J restarts and picks up the changes to the default Web application.

This chapter emphasizes the servlet code itself, so deployment is done to the default Web application for simplicity. Chapter 3, "Deployment and Configuration", describes Web application development, deployment, and testing under the J2EE paradigm that you would use for production applications.

Code Template

Here is a code template for servlet development:

```
public class myServlet extends HttpServlet {

  public void init(ServletConfig config) {
  }

  public void destroy() {
  }

  public void doGet(HttpServletRequest request, HttpServletResponse)
                   throws ServletException, IOException {
  }

  public void doPost(HttpServletRequest request, HttpServletResponse)
                   throws ServletException, IOException {
```

```
}

public String getServletInfo() {
  return "Some information about the servlet.";
}
```

Overriding the `init()`, `destroy()`, and `getServletInfo()` methods is optional. The simplest servlet just overrides either `doGet()` or `doPost()`.

Servlet Lifecycle

Servlets have a predictable and manageable lifecycle:

- The container creates a new instance of a servlet when either the servlet is first invoked by a client, or OC4J starts. This depends upon whether a `<load-on-startup>` element is declared for the servlet in the application `web.xml` file.
- When the servlet is loaded, its configuration details are read from `web.xml`. These can include initialization parameters.
- There is only one instance of a servlet. Client requests share servlet instances.
- Client requests invoke the `service()` method of the generic servlet, which then delegates the request to `doGet()` or `doPost()` (or another overridden request-handling method), depending upon the information in the request headers.
- Filters can be interposed between the container and the servlet to modify the servlet behavior, either during request or response. See Chapter 4, "Servlet Filters", for more information.
- A servlet can forward requests to other servlets.
- The servlet creates a response object, which the container passes back to the client in HTTP response headers. Servlets can write to the response using a `java.io.PrintWriter` or `javax.servlet.ServletOutputStream` object.
- The container calls the `destroy()` method before the servlet is unloaded.

Servlet Behavior

A servlet typically receives information from one or more sources, including the following:

- parameters from the request object
- the `HttpSession` object
- the `ServletContext` object
- information from data sources outside the servlet (for example: databases, file systems, or external sensors)

The servlet adds information to the response object, and the container sends the response back to the client.

Thread Safety

Because a servlet can be invoked from more than one thread, you must ensure that servlet code is thread-safe. Critical sections of code must be synchronized, although you must do this selectively and carefully, because it can affect performance. The servlet specification provides that a servlet can implement the `SingleThreadModel` to guarantee synchronized access to the whole servlet, but this practice is not recommended for OC4J applications.

Session Maintenance

The servlet specification provides a convenient way to enable stateful servlet sessions, using cookies and the `javax.servlet.http.HttpSession` object. See "Cookies" on page 2-10 for more information.

Servlet Context

There is a single servlet context for each Web application.

The `javax.servlet.ServletContext` object is contained within the `javax.servlet.ServletConfig` object, which the Web server provides to the servlet and which is used by the servlet container to pass information to the servlet during initialization.

Invoking a Servlet

A servlet or JSP page is invoked by the container when a request for the servlet arrives from a client. The client request might come from a Web browser or a Java client application, or from another servlet in the application using the request forwarding mechanism, or from a remote object on a server.

A servlet is requested through its URL mapping. The URL mapping for a servlet consists of two parts: the *context path* and the *servlet path*. The context path is that part of the URL from the first forward slash after the host name or port number, and before the servlet path. The servlet path continues from the slash at the end of the context path (if there is a context path) to the end of the URL string, or until a '?' or ';' that delimits the servlet path from the additional material, such as query strings or rewritten parts of the URI. In a typical deployment scenario, the context path and servlet path are determined through settings in a standard `web.xml` file.

The remainder of this section covers the following topics, including some special OC4J features for invoking a servlet in a development or testing environment:

- Action by the Servlet Container Upon Request
- Invoking a Servlet by Class Name in OC4J
- Configuration for Servlet Invocation in a Deployment Environment

Action by the Servlet Container Upon Request

When the servlet container receives a request for a servlet, it does the following:

- Loads and initializes the servlet, if that has not already been done. See "Servlet Loading and Initialization" on page 2-9.
- Constructs a request object to pass to the servlet. The request includes, among other things:
  - any HTTP headers from the client
  - parameters and values passed from the client (for example, names and values of query strings in the URL)
  - the complete URI of the servlet request

  You can determine all the available information passed in the request object by looking at the Javadoc for the `HttpServletRequest` interface, at the following location:

  `http://java.sun.com/products/servlet/2.3/javadoc/index.html`

- Constructs a response object for the servlet.
- Invokes the servlet `service()` method. Note that for HTTP servlets, the generic service method is usually overridden in the `HttpServlet` class. The service method dispatches requests to the servlet `doGet()` or `doPost()` methods, depending on the HTTP header in the request (`GET` or `POST`).

If there is a filter or a chain of filters to be invoked before the servlet, these are called by the container with the request and response objects as parameters. The filters pass these objects, perhaps modified, or alternatively create and pass new objects, to the next object in the chain using the `doChain()` method. See Chapter 4, "Servlet Filters", for more information about this topic.

Invoking a Servlet by Class Name in OC4J

In a development or testing environment in OC4J, there is a mechanism for invoking a servlet by class name. This may simplify the URL for invocation.

Setting the `servlet-webdir` attribute in the `<orion-web-app>` element of the `global-web-application.xml` file or `orion-web.xml` file defines a special URL component. Anything following this URL component is assumed to be a servlet class name, including applicable package information, within the appropriate servlet context. By default in OC4J, this setting is `"/servlet"`.

The following URL shows how to invoke a servlet called `SessionServlet`, with explanations following. In this example, assume `SessionServlet` is in package `foo.bar`, and executes in the OC4J default Web application.

`http://<hostname><:port>/j2ee/servlet/foo.bar.SessionServlet`

`http://`	The network protocol. Other protocols are `ormi`, `ftp`, `https`, and so on.
`<hostname>`	The network name of the server that the Web application is running on. If the Web client is on the same system as the application server, you can use `localhost`. Otherwise use the host name, for example as defined in `/etc/hosts` on a UNIX system.

<:*port*>	The port that the Web server listens on. (If you do not specify a port, most browsers assume port 80.) The server port is defined in the `port` attribute of the `<web-site>` element in the following file: `j2ee/home/config/default-web-site.xml`
`/j2ee`	`/j2ee` is the context path of the OC4J default Web application.
`/servlet`	This is according to the default `servlet-webdir` setting.
`/foo.bar.SessionServlet`	Because there is a `servlet-webdir` setting, this portion of the URL is simply the servlet package and class name.

This mechanism applies to any servlet context, however, and not just for the default Web application. If the context path is `foo`, for example, the URL to invoke by class name would be as follows:

`http://<hostname><:port>/foo/servlet/foo.bar.SessionServlet`

Note: See the *Oracle9iAS Containers for J2EE User's Guide* for information about defined ports and what listeners they are mapped to, and for information about how to alter these settings. Depending on the port you specify, you can access OC4J directly through its own listener (useful in development environments), or through the Oracle HTTP Server (highly recommended for deployment environments).

Configuration for Servlet Invocation in a Deployment Environment

In a deployment environment, using the `servlet-webdir` attribute in `global-web-application.xml` or `orion-web.xml` is inadvisable for security reasons. Instead, you should use standard servlet settings and mappings in the application `web.xml` file to specify the context path and servlet path.

In `web.xml`, the `<servlet-name>` subelement of the `<servlet>` element defines a name for the servlet and relates it to a servlet class. The `<servlet-mapping>` subelement relates servlet names to path mappings. The servlet name as well as the mapping names are arbitrary—it is not necessary for the class that is invoked to

have the same base name, or even a similar base name, to either the <servlet-name> or any of the <servlet-mapping> settings.

Note that because of the default OC4J mount point, each context path must start with "/j2ee/" for the servlet to be routed to OC4J through the Oracle HTTP Server. If you want to use something other than "/j2ee/", create a new Oc4jMount directive in the mod_oc4j.conf file. Copy the default mount directive and replace "j2ee" as desired. Here is the default directive:

```
Oc4jMount /j2ee/*
```

See the *Oracle9iAS Containers for J2EE User's Guide* for more information.

Notes:

- If you use OEM to deploy the application, the new mount point is added automatically.
- This discussion assumes the Web application is bound to a Web site that uses AJP protocol, according to settings in default-web-site.xml or the relevant Web site XML file.

There is also a relevant element in the default-web-site.xml file (or other Web site XML file). The <frontend> subelement of the <web-site> element specifies a perceived front-end host and port of the Web site as seen by HTTP clients. When the site is behind something like a load balancer or firewall, the <frontend> specification is necessary to provide appropriate information to the Web application for functionality such as URL rewriting. Attributes are host, for the hostname of the front-end server, such as "www.acme.com", and port, for the port number of the front-end server, such as "80". Using this front-end information, the back-end server that is actually running the application knows to refer to www.acme.com instead of to itself in any URL rewriting. This way, subsequent requests properly come in through the front-end again, instead of trying to access the back-end directly.

Servlet Loading and Initialization

The container instantiates and loads a servlet class when it is first requested, unless you specify that the class should be loaded and initialized when the OC4J server starts up. In the application `web.xml` file, you can specify a `<load-on-startup>` subelement in the `<servlet>` element to have the server load and initialize the servlet on start-up. For example, the following element names the servlet represented by the `PrimeSearcher.class` file as `PSearcher`, and specifies that it should be loaded when the server starts:

```
<servlet>
  <servlet-name>PSearcher</servlet-name>
  <servlet-class>PrimeSearcher</servlet-class>
  <load-on-startup/>
</servlet>
```

When the servlet is loaded, either at server start-up time or when requested, the container indirectly calls the servlet `init()` method. A servlet can override the `HttpServlet` `init()` method to perform actions that are required only once in the servlet lifetime, such as the following examples:

- establishing data source connections
- getting initialization parameters from the configuration and storing the values in local variables
- recovering persistent data that the servlet requires
- creating expensive session objects such as hashtables
- logging the servlet version to the `log()` method of the `ServletContext` object

See "Database Query Servlet" on page 2-17 for an example that shows a servlet that uses the `init()` method to get a data source object at start-up.

Servlet Sessions

This section discusses servlet sessions, covering the following topics:

- Session Tracking
- Session Cancellation
- Session Servlet Example
- Session Replication

Session Tracking

The HTTP protocol is stateless by design. This is fine for stateless servlets that simply take a request, do a few computations, output some results, and then in effect go away. But many, if not most, server-side applications must keep some state information and maintain a dialogue with the client. The most common example of this is a shopping cart application. A client accesses the server several times from the same browser, and visits several Web pages. They decide to buy some of the items offered for sale at the Web site, and clicks on the BUY ITEM boxes. If each transaction were being served by a stateless server-side object, and the client provided no identification on each request, it would be impossible to maintain a filled shopping cart over several HTTP requests from the client. In this case, there would be no way to relate a client to a server session, so even writing stateless transaction data to persistent storage would not be a solution.

Note: Do not use `HttpSession` objects to store persistent information that must last beyond the normal duration of a session. You can store persistent data in a database if you need the protection, transactional safety, and backup that a database offers. Alternatively, you can save persistent information on a file system or in a remote object. See the sample application called `stateless` in the demos supplied with OC4J to see how to store persistent information in a remote object.

Cookies

A number of approaches have attempted to add a measure of statefulness to the HTTP protocol. The most widely accepted at the current time is the use of cookies, to let the client transmit an identifier to the server, together with stateful servlets that can maintain session objects. Session objects are simply dictionaries that store a value (a Java object) together with a key (a Java string).

When a client first connects to a stateful servlet, the server (container) sends a cookie that contains a session identifier back to the client, often along with a small amount of other useful information (all less than 4 KB). Then on each subsequent request from the same Web client session, the client sends the cookie back to the server. Cookies are sent and updated by the container in the response header—the servlet code does not need to do anything to send a cookie. Similarly, cookies are sent back to the server by the Web browser. A browser user only has to enable cookies on the browser to get cookie functionality.

The container uses the cookie for session maintenance. A servlet can retrieve cookies using the `getCookies()` method of the `HttpServletRequest` object, and can examine cookie attributes using the accessor methods of the `javax.servlet.http.Cookie` objects.

URL Rewriting

Most Web users have learned to keep cookies enabled on their browsers, although some still do not. An alternative to using cookies is URL rewriting, through the `encodeURL()` method of the response object. See "Session Servlet Example" on page 2-12 for an example of URL rewriting.

Other Session Tracking Methods

Other techniques have been used in the past to relate client and server sessions. These include server hidden form fields and user authentication mechanisms to store additional information. Oracle does not recommend that you use these techniques in OC4J applications, because they have many drawbacks, including performance penalties and loss of confidentiality.

Session Cancellation

`HttpSession` objects persist for the duration of the server-side session. A session is either terminated explicitly by the servlet, or it "times out" after a certain period and is cancelled by the container.

Cancellation Through a Timeout

The default session timeout for the OC4J server is 20 minutes. You can change this for a specific application by setting the `<session-timeout>` subelement in the `<session-config>` element of `web.xml`. For example, to reduce the session timeout to five minutes, add the following lines to the application `web.xml`:

```
<session-config>
  <session-timeout>5</session-timeout>
```

```
</session-config>
```

Cancellation by the Servlet

A servlet explicitly cancels a session by invoking the `invalidate()` method on the session object. You must obtain a new session object by invoking the `getSession()` method of the `HttpServletRequest` object.

Session Servlet Example

The `SessionServlet` code below implements a servlet that establishes an `HttpSession` object and prints some interesting data held by the request and session objects.

SessionServlet Code

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;

public class SessionServlet extends HttpServlet {

  public void doGet (HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException {

    // Get the session object. Create a new one if it doesn't exist.
    HttpSession session = req.getSession(true);

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<head><title> " + "SessionServlet Output " +
                "</title></head><body>");
    out.println("<h1> SessionServlet Output </h1>");

    // Set up a session hit counter. "sessionservlet.counter" is just the
    // conventional way to create a key for the value to be stored in the
    // session object "dictionary".
    Integer ival =
      (Integer) session.getAttribute("sessionservlet.counter");
    if (ival == null) {
      ival = new Integer(1);
    }
    else {
```

```
      ival = new Integer(ival.intValue() + 1);
    }

    // Save the counter value.
    session.setAttribute("sessionservlet.counter", ival);

    // Report the counter value.
    out.println(" You have hit this page <b>" +
                ival + "</b> times.<p>");

    // This statement provides a target that the user can click on
    // to activate URL rewriting. It is not done by default.
    out.println("Click <a href=" +
                res.encodeURL(HttpUtils.getRequestURL(req).toString()) +
                ">here</a>");
    out.println(" to ensure that session tracking is working even " +
                "if cookies aren't supported.<br>");
    out.println("Note that by default URL rewriting is not enabled" +
                " due to its large overhead.");

    // Report data from request.
    out.println("<h3>Request and Session Data</h3>");
    out.println("Session ID in Request: " +
                req.getRequestedSessionId());
    out.println("<br>Session ID in Request is from a Cookie: " +
                req.isRequestedSessionIdFromCookie());
    out.println("<br>Session ID in Request is from the URL: " +
                req.isRequestedSessionIdFromURL());
    out.println("<br>Valid Session ID: " +
                req.isRequestedSessionIdValid());

    // Report data from the session object.
    out.println("<h3>Session Data</h3>");
    out.println("New Session: " + session.isNew());
    out.println("<br> Session ID: " + session.getId());
    out.println("<br> Creation Time: " + new Date(session.getCreationTime()));
    out.println("<br>Last Accessed Time: " +
                new Date(session.getLastAccessedTime()));

    out.println("</body>");
    out.close();
  }

  public String getServletInfo() {
    return "A simple session servlet";
```

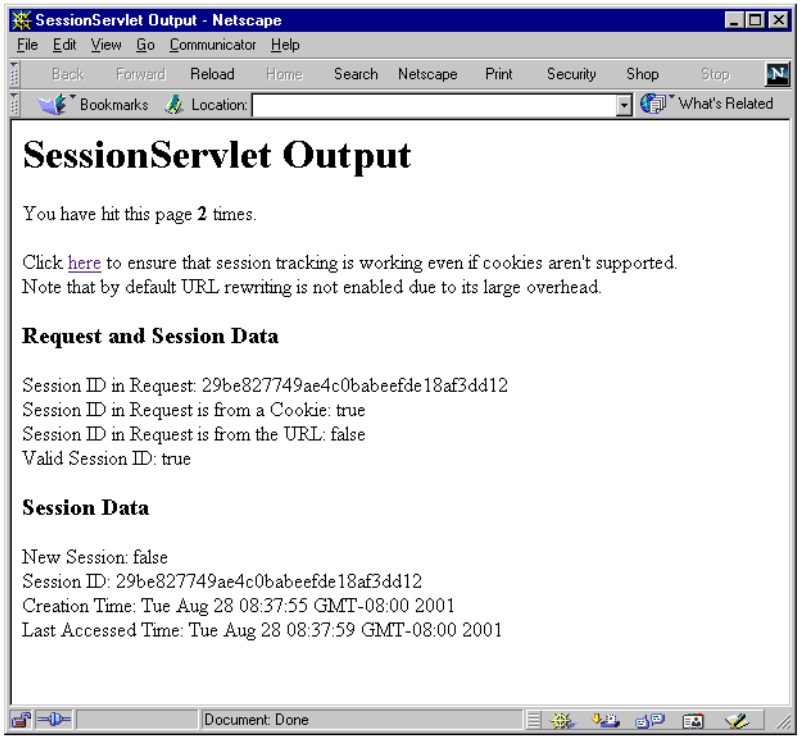
```
  }
}
```

Deploying and Testing

Enter the preceding code into a text editor, and save it in the file `j2ee/home/default-web-app/WEB-INF/classes/SessionServlet.java`. If you set the attribute `development="true"` in the `<orion-web-app>` element of the `global-web-application.xml` file, the servlet can be recompiled and redeployed automatically the next time it is invoked. You may also have to set the `source-directory` attribute appropriately. See "Element Descriptions for global-web-application.xml and orion-web.xml" on page 3-15 for more information about these attributes.

Figure 2–1 shows the output of this servlet when it is invoked the second time in a session by a Web browser that has cookies enabled. Experiment with different Web browser settings—for example, by disabling cookies—then click on the HREF that causes URL rewriting.

Figure 2–1 Session Servlet Display



Session Replication

The session object of a stateful servlet can be replicated to other OC4J servers in a load-balanced cluster island. If the server handling a request to a servlet should fail, the request can "failover" to another JVM on another server in the cluster island. The session state will still be available. The Web application must be marked as distributable in the `web.xml` file, by use of the standard `<distributable>` element.

Objects that are stored by a servlet in the `HttpSession` object are replicated, and must be serializable or remoteable for replication to work properly.

Note that a slight but noticeable delay occurs when an application is replicated to other servers in a load-balanced cluster island. It is, therefore, possible that the

servlet could have been replicated by the time a failure occurred in the original server, but that the session information had not yet been replicated.

Use of JDBC in Servlets

A servlet can access a database using a JDBC driver. The recommended way to use JDBC is by using an OC4J data source to get the database connection. See *Oracle9iAS Containers for J2EE Services Guide* for information about OC4J data sources.

For more information about JDBC, see the *Oracle9i JDBC Developer's Guide and Reference*.

Database Query Servlet

Part of the power of servlets comes from their ability to retrieve data from a database. A servlet can generate dynamic HTML by getting information from a database and sending it back to the client. A servlet can also update a database, based on information passed to it in the HTTP request.

The example in this section shows a servlet that gets some information from the user through an HTML form and passes the information to a servlet. The servlet completes and executes a SQL statement, querying the sample HR schema to get information based on the request data.

A servlet can get information from the client in many ways. This example reads a query string from the HTTP request.

HTML Form

The Web browser accesses a form in a page that is served through the Web listener. First, enter the following text into a file, naming the file `EmpInfo.html`.

```
<HTML>
<HEAD>
<TITLE>Get Employee Information</TITLE>
</HEAD>

<BODY>
<FORM METHOD=GET ACTION="/servlet/GetEmpInfo">
The query is<br>
SELECT LAST_NAME, EMPLOYEE_ID FROM EMPLOYEES WHERE LAST NAME LIKE ?.<p>

Enter the WHERE clause ? parameter (use % for wildcards).<br>
Example: 'S%':<br>
<INPUT TYPE=TEXT NAME="queryVal">
<P>
<INPUT TYPE=SUBMIT VALUE="Send Info">
```

```
</FORM>

</BODY>
</HTML>
```

Then save this file in the `j2ee/home/default-web-apps` directory.

Servlet Code: GetEmpInfo

The servlet that the preceding HTML page calls takes the input from a query string. The input is the completion of the WHERE clause in the SELECT statement. The servlet then appends this input to complete the database query. Most of the code in this servlet is taken up with the JDBC statements required to connect to the data server and retrieve the query rows.

This servlet makes use of the `init()` method to do a one-time lookup of a data source, using JNDI. The data source lookup assumes a data source such as the following has been defined in the `j2ee/home/config/data-sources.xml` file:

```
<data-source
   class="com.evermind.sql.DriverManagerDataSource"
   name="OracleDS"
   location="jdbc/OracleCoreDS"
   xa-location="jdbc/xa/OracleXADS"
   ejb-location="jdbc/OracleDS"
   connection-driver="oracle.jdbc.driver.OracleDriver"
   username="scott"
   password="tiger"
   url="jdbc:oracle:thin:@localhost:5521:oracle"
   inactivity-timeout="30"
/>
```

In Oracle9*i*AS 9.0.2, it is advisable to use only the `ejb-location` JNDI name in the JNDI lookup for a data source. See the *Oracle9iAS Containers for J2EE Services Guide* for more information about data sources.

Here is the servlet code:

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
// These packages are needed for the JNDI lookup.
import javax.naming.*;
// These packages support SQL operations and Oracle JDBC drivers.
import javax.sql.*;
```

```
import oracle.jdbc.*;


public class GetEmpInfo extends HttpServlet {

  DataSource ds = null;

  public void init() throws ServletException {
    try {
      InitialContext ic = new InitialContext();
      ds = (DataSource) ic.lookup("java:comp/env/jdbc/OracleDS");
    }
    catch (NamingException ne) {
      throw new ServletException(ne);
    }
  }


  public void doGet (HttpServletRequest req, HttpServletResponse resp)
                 throws ServletException, IOException {

    String queryVal = req.getParameter("queryVal");
    String query =
      "select last_name, employee_id from employees " +
      "where last_name like " + queryVal;

    resp.setContentType("text/html");

    PrintWriter out = resp.getWriter();
    out.println("<html>");
    out.println("<head><title>GetEmpInfo</title></head>");
    out.println("<body>");

    try {
      Connection conn = ds.getConnection();
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(query);

      out.println("<table border=1 width=50%>");
      out.println("<tr><th width=75%>Last Name</th>" +
                  "<th width=25%>Employee ID</th></tr>");

      for (int count = 0; ; count++ ) {
        if (rs.next()) {
          out.println("<tr><td>" + rs.getString(1) + "</td><td>" +
```

```
            rs.getInt(2) + "</td></tr>");
        }
        else {
          out.println("</table><h3>" + count + " rows retrieved</h3>");
          break;
        }
      }
      conn.close();
      rs.close();
      stmt.close();
    }
    catch (SQLException se) {
      se.printStackTrace(out);
    }

    out.println("</body></html>");
  }

  public void destroy() {
  }
}
```

Deployment and Testing of the Database Query Servlet

To deploy this example, save the HTML file in the `j2ee/home/default-web-app/` directory (the effective document root for the default application), and save the Java servlet in the `j2ee/home/default-web-app/WEB-INF/classes/` directory. The `GetEmpInfo.java` file is automatically compiled when the servlet is invoked by the form.

To test the example, invoke the `EmpInfo.html` page from a Web browser, as follows:

`http://<`*`hostname`*`><:`*`port`*`>/j2ee/EmpInfo.html`

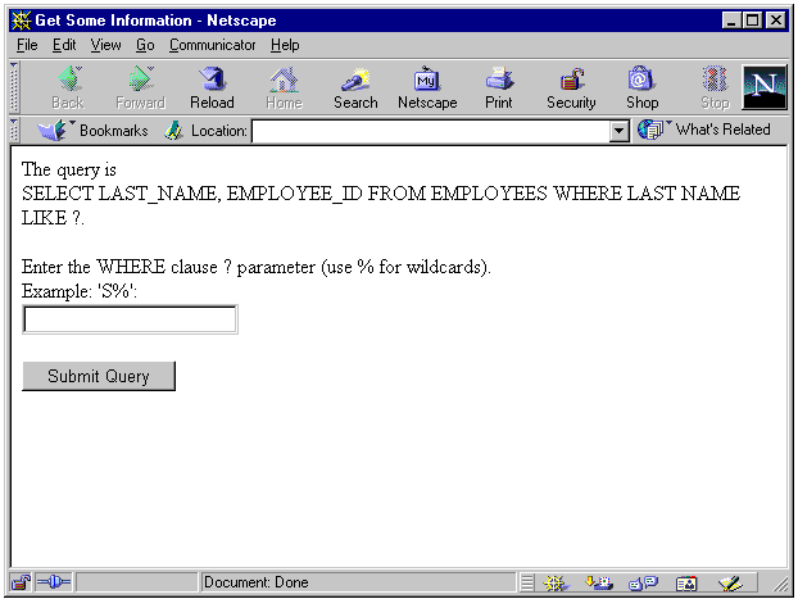
This assumes `/j2ee` is the root context of the OC4J default Web application.

Complete the form and click **Submit Query**.

Note: For the port setting, 7777 by default will access OC4J through the Oracle HTTP Server, powered by Apache, with the Oracle9*i*AS Web Cache enabled. Other port settings are possible to use the OC4J Web listener directly, which may be useful in development situations. See the *Oracle9iAS Containers for J2EE User's Guide* for information about OC4J port settings and default settings. For production applications, Oracle recommends that you use the Oracle HTTP Server, which is part of the Oracle9*i*AS distribution. See the *Oracle HTTP Server Administration Guide*.

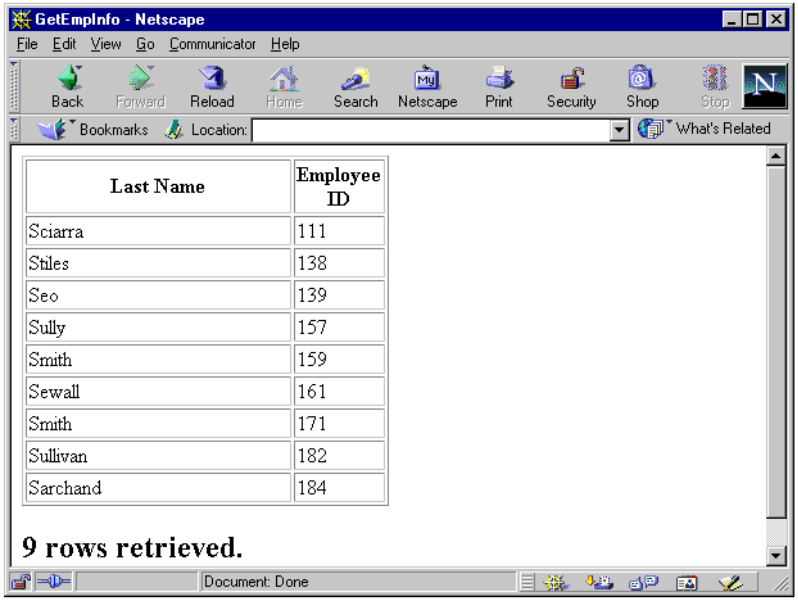
When you invoke `EmpInfo.html`, you will see a browser window that looks something like Figure 2–2.

Figure 2–2 Employee Information Query



Entering `'S%'` in the form and pressing **Submit Query** calls the `GetEmpInfo` servlet, and the results look something like Figure 2–3.

Figure 2–3 Employee Information Results



EJB Calls from Servlets

A servlet or a JSP page can call an EJB to perform additional processing. A typical application design often uses servlets as a front-end to do the initial processing of client requests, with EJBs being called to perform the business logic that accesses or updates a database. Container-managed-persistence (CMP) entity beans, in particular, are well-suited for such tasks.

There are three main scenarios for servlet-EJB interactions:

- The servlet calls an EJB within the same application, performing a local lookup. "Local" is the default lookup mode, so no special action is required. You just have to complete standard configuration for EJB usage, such as defining the reference name in an `<ejb-ref>` element in the `web.xml` file. See "Local EJB Lookup Within the Same Application" below, which includes a detailed example.
- The servlet calls an EJB within the same application, but performs the lookup remotely. This may be useful, for example, in a load-balancing situation where the Web tier and EJB tier are running in different OC4J instances. See "Remote EJB Lookup Within the Same Application" on page 2-31.
- The servlet looks up an EJB from another application. This is a remote lookup unless the specified host and port are the same as for the calling servlet. See "EJB Lookup Outside the Application" on page 2-31.

For additional servlet-EJB examples, see the demo programs that come with the OC4J distribution.

See the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference* for more information about EJB development in OC4J.

Local EJB Lookup Within the Same Application

This section presents an example of a single servlet, `HelloServlet`, that calls a single EJB, `HelloBean`, within the same application using a local lookup.

Here are the key steps of the servlet code:

1. Import the EJB package for access to the bean home and remote interfaces.
2. Print a message from the servlet.
3. Create an output string, with an error default.
4. Use JNDI to look up the EJB home interface.

5. Create the EJB remote object from the home.
6. Invoke the `helloWorld()` method on the remote object, which returns a `String` object.
7. Print the message from the EJB.

Servlet Code: HelloServlet

```
package myServlet;

// Step 1: Import the EJB package.
import myEjb.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;                                  // for JNDI

public class HelloServlet extends HttpServlet {

  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<html><head><title>Hello from Servlet</title></head>");
    // Step 2: Print a message from the servlet.
    out.println("<body><h1>Hello from hello servlet!</h1></body>");

    // Step 3: Create an output string, with an error default.
    String s = "If you see this message, the ejb was not invoked properly!!";
    // Step 4: Use JNDI to look up the EJB home interface.
    try {
      HelloHome hh = (HelloHome)
                  (new InitialContext()).lookup("java:comp/env/ejb/HelloBean");

      // Step 5: Create the EJB remote IF.
      HelloRemote hr = hh.create();
      // Step 6: Invoke the helloWorld() method on the remote object.
      s = hr.helloWorld();
    } catch (Exception e) {
      e.printStackTrace(out);
    }
    // Step 7: Print the message from the EJB.
    out.println("<br>" + s);
```

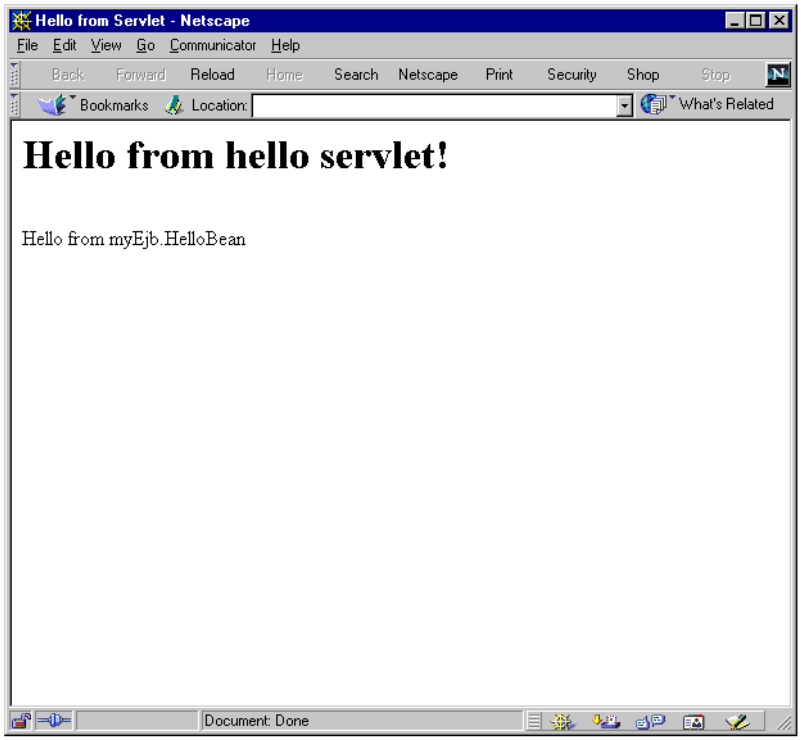
```
    out.println("</html>");
  }
}
```

Figure 2–4 shows the output to a Web browser when you invoke the servlet:

`http://<hostname><:port>/j2ee/myapp/doubleHello`

The output from the servlet is printed in H1 format at the top, then the output from the EJB is printed in text format below that.

Figure 2–4 Output from HelloServlet



EJB Code: HelloBean Stateful Session Bean

The EJB, as shown here, is simple. It implements a single method—`helloWorld()`—that returns a greeting to the caller. The home and remote EJB interface code is also shown below.

```
package myEjb;

import java.rmi.RemoteException;
import javax.ejb.*;

public class HelloBean implements SessionBean
{
  public String helloWorld () throws RemoteException {
    return "Hello from myEjb.HelloBean";
  }

  public void ejbCreate () throws RemoteException, CreateException {}
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

EJB Interface Code: Home and Remote Interfaces

Here is the code for the home interface:

```
package myEjb;

import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;

public interface HelloHome extends EJBHome
{
  public HelloRemote create () throws RemoteException, CreateException;
}
```

Here is the code for the remote interface:

```
package myEjb;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface HelloRemote extends EJBObject
{
  public String helloWorld () throws RemoteException;
}
```

Deployment of the Servlet-EJB Application

This section discusses the deployment steps for the Servlet-EJB sample application, including the Web archive, EJB archive, application-level descriptor, and deployment commands.

See Chapter 3, "Deployment and Configuration", for general information about deployment to OC4J.

Web Archive To deploy this application, an EJB deployment descriptor (`ejb-jar.xml`) and a Web deployment descriptor (`web.xml`) are required. The contents of `web.xml` for this example are as follows:

```
<?xml version="1.0"?>
<!DOCTYPE WEB-APP PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

  <display-name>HelloServlet</display-name>
  <description> HelloServlet </description>
  <servlet>
    <servlet-name> ServletCallingEjb </servlet-name>
    <servlet-class> myServlet.HelloServlet </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name> ServletCallingEjb </servlet-name>
    <url-pattern> /doubleHello </url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file> index.html </welcome-file>
  </welcome-file-list>
  <ejb-ref>
    <ejb-ref-name>ejb/HelloBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>myEjb.HelloHome</home>
    <remote>myEjb.HelloRemote</remote>
  </ejb-ref>
</web-app>
```

Next, create the directory structure that is required for Web application deployment, and move the Web deployment descriptor (`web.xml`) and the compiled servlet class file into the structure. The `web.xml` file must be in a `WEB-INF` directory, and the servlet class files (in their respective packages, as applicable) under the `WEB-INF/classes/` directory. Once you create the directory structure and

populate the directories, create a WAR file to contain the files. From the Web root directory, create the WAR file as follows:

```
% jar cvf myapp-web.war *
```

When created, the WAR file should look like this:

```
% jar -tf myapp-web.war
META-INF/
META-INF/MANIFEST.MF
WEB-INF/
WEB-INF/classes/
WEB-INF/classes/myServlet/
WEB-INF/classes/myServlet/HelloServlet.class
WEB-INF/web.xml
```

EJB Archive The contents of `ejb-jar.xml` are as follows. Note that the `<ejb-name>` value here corresponds to the `<ejb-ref-name>` value in the `web.xml` file above.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
1.12//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <description>Hello Bean</description>
      <ejb-name>ejb/HelloBean</ejb-name>
      <home>myEjb.HelloHome</home>
      <remote>myEjb.HelloRemote</remote>
      <ejb-class>myEjb.HelloBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
  </assembly-descriptor>
</ejb-jar>
```

Create a JAR file to hold the EJB components. The JAR file should look like this:

```
% jar tf myapp-ejb.jar
META-INF/
META-INF/MANIFEST.MF
myEjb/
```

```
META-INF/ejb-jar.xml
myEjb/HelloBean.class
myEjb/HelloHome.class
myEjb/HelloRemote.class
```

Application-Level Descriptor To deploy the application, create an application deployment descriptor—`application.xml`. This file describes the modules in the application:

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.2//EN" "http://java.sun.com/j2ee/dtds/application_1_2.dtd">

<application>
  <display-name>Servlet_calling_ejb_example</display-name>
  <module>
    <web>
      <web-uri>myapp-web.war</web-uri>
      <context-root>/foo</context-root>
    </web>
  </module>
  <module>
    <ejb>myapp-ejb.jar</ejb>
  </module>
</application>
```

Note the following regarding the `<context-root>` setting:

- For a new context root (`/foo` in this example) to route properly to OC4J through Oracle HTTP Server, there must be an appropriate `Oc4jMount` command in the `mod_oc4j.conf` file. (See the *Oracle9iAS Containers for J2EE User's Guide* for additional information.) If you use OEM to deploy the application, this is handled automatically. This discussion assumes the Web application is bound to a Web site that uses AJP protocol, according to settings in `default-web-site.xml` or the relevant Web site XML file.
- If you run in an OC4J standalone scenario (where OC4J runs apart from Oracle9*i*AS), then the `<context-root>` element is ignored. You must specify the context root by adding appropriate entries to `default-web-site.xml` or the relevant Web site XML file, and to `mod_oc4j.conf`.

Finally, create an EAR file to hold the application components. The EAR file should look like this:

```
% jar tf myapp.ear
META-INF/
META-INF/MANIFEST.MF
myapp-ejb.jar
myapp-web.war
META-INF/application.xml
```

Deployment Commands To perform the application deployment for testing purposes, you can use the OC4J `admin.jar` tool to issue two commands. The first command is as follows. (Specify the appropriate machine name.)

```
% java -jar $J2EE_HOME/admin.jar ormi://<machine_name> admin welcome  \
       -deploy -file ./lib/myapp.ear   \
       -deploymentName myapp
```

This command adds the following entry to `j2ee/home/config/server.xml`:

```
<application
   name="myapp"
   path="<your_path_to>/lib/myapp.ear"
   auto-start="true"
/>
```

Here is the second command:

```
% java -jar $J2EE_HOME/admin.jar ormi://<machine_name> admin welcome
       -bindWebApp myapp myapp-web default-web-site /myapp
```

This command binds the Web module to a Web site. It adds the following entry to `j2ee/home/config/default-web-site.xml`:

```
<web-app
   application="myapp"
   name="myapp-web"
   root="/myapp"
/>
```

Note: In a production environment, use Oracle Enterprise Manager (OEM) for deployment.

Remote EJB Lookup Within the Same Application

To perform a remote EJB lookup in OC4J, you must enable the EJB `remote` flag. This is an attribute in the `<ejb-module>` subelement of an `<orion-application>` element in the `orion-application.xml` file for the application to which the calling servlet belongs. (The default setting is `remote="false"`.) Here is an example of enabling this flag:

```
<orion-application ... >
   <ejb-module remote="true" ... />

   ...

</orion-application>
```

No changes are necessary to the servlet code. Recall the local EJB lookup from "Servlet Code: HelloServlet" on page 2-24:

```
HelloHome hh = (HelloHome)
               (new InitialContext()).lookup("java:comp/env/ejb/HelloBean");
```

Given a `remote="true"` setting, this code would result in a remote lookup of `ejb/HelloBean`. Where the lookup is performed is according to how EJB clustering is configured in the application `rmi.xml` file.

Configure remote servers in `rmi.xml` through `<server>` elements, using the `host`, `port`, `user`, and `password` attributes as appropriate. If multiple servers are configured, OC4J will search all of them, as necessary, for the intended EJB.

See the *Oracle9iAS Containers for J2EE Services Guide* for information about `rmi.xml`.

EJB Lookup Outside the Application

To look up an EJB from outside the application, use `ormi://...` syntax in the `lookup()` call. The `remote` flag discussed in "Remote EJB Lookup Within the Same Application" above is not relevant—the lookup is according to the `ormi` URL. If the host and port are the same as for the calling servlet, then the lookup is local; otherwise, the lookup is remote. Here is an example, where *appname* is the name of the application to which the EJB belongs:

```
HelloHome hh = (HelloHome)
  (new InitialContext()).lookup("ormi://host:port/appname/env/ejb/HELLOEJB");
```

This assumes that the name `ejb/HELLOEJB` is defined in an `<ejb-name>` element in the `ejb-jar.xml` file of the remote application. The `web.xml` file of the application to which the calling servlet belongs is not relevant.

See the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference* for information about `ejb-jar.xml`.

Note: If you omit the host and port in the `ormi` URL, the host is assumed to be `localhost` and a local lookup is performed.

3

Deployment and Configuration

This chapter describes how to deploy and configure a Web application in OC4J. It covers the following topics:

- Introduction to Web Application Deployment and Configuration
- Application Assembly
- Application Deployment
- Configuration File Descriptions

Introduction to Web Application Deployment and Configuration

This section provides an overview of OC4J Web applications, deployment, and configuration, covering the following topics:

- Web Application Modules
- Overview of OC4J Deployment
- Overview of Web Configuration Files

Note: For users of Oracle9*i*AS release 1.0.2.2, see *Migrating from Oracle9i Application Server 1.x* for information about issues in migrating to Oracle9*i*AS 9.0.2.

Web Application Modules

An OC4J application can consist of one or more J2EE-compliant *modules*. These include:

- *Web application modules* consist of JSP pages, servlet class files, HTML pages, and other resources that the application might require (such as data files, images, and sound files).
- *EJB modules* contain classes that implement Enterprise JavaBeans.
- A *client module* consists of Java class files that form a client application. The client application runs on a system that may or may not be the same as the server host, but normally is not the same.

A J2EE application might consist of only a single Web application module, the client being a Web browser. Or, it might consist of just a Java client and one or more EJB modules. Most business applications include both a Web application module (servlets, JSP pages, and HTML pages) and one or more EJB modules. Optionally, a Java client might be adopted as the front-end for the application, although there are many large applications that rely solely on a Web browser for client access.

The examples in this chapter are derived from the sample application `stateless`, which is provided with OC4J. The actual application name is `employee`. This application includes both a Web and an EJB module, but building and deploying the Web module follows the same practice as a Web-only application. The sample is also available at the following location:

```
http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html#Servlet
```

Overview of OC4J Deployment

For production, use Oracle Enterprise Manager (OEM) for deployment. OEM is recommended for managing OC4J and other components of Oracle9*i*AS in a production environment. Refer to the *Oracle9i Application Server Administrator's Guide* and *Oracle Enterprise Manager Administrator's Guide* for information.

OC4J also supports the `admin.jar` tool for deployment, typically in a development environment. This modifies `server.xml` and other configuration files for you, based on settings you specify to the tool. Or you can modify the configuration files manually (not generally recommended). Note that in Oracle9*i*AS 9.0.2, if you modify configuration files without going through OEM, you must use the `dcmctl` tool to inform Oracle9*i*AS Distributed Configuration Management (DCM) of the updates. (This does not apply in an OC4J standalone mode, where OC4J is being run apart from Oracle9*i*AS.) See "Use of admin.jar for Deployment" on page 3-11 regarding `dcmctl`.

For general OC4J deployment and configuration information and discussion of the `admin.jar` tool, see the *Oracle9iAS Containers for J2EE User's Guide*. For additional information about deploying an application that has EJB modules, see the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*.

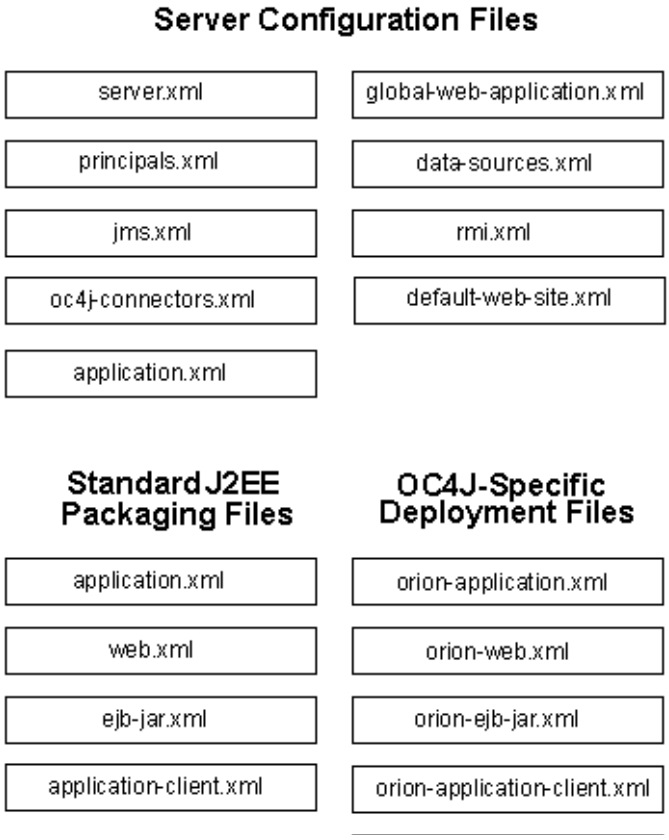
For more information about standard J2EE deployment, refer to the J2EE specification, which is available at the following location:

`http://java.sun.com/j2ee/docs.html`

Overview of Web Configuration Files

Figure 3–1 shows the XML configuration files that OC4J supports. OC4J uses the server configuration files to configure the server on start up. The server configuration files are located in the `j2ee/home/config` directory. The files shown at the bottom of the figure are application-specific configuration files. The files at the bottom-left are the J2EE-standard files: `web.xml`, `ejb-jar.xml`, `application.xml`, and `application-client.xml`. At the bottom-right are the corresponding OC4J-specific files to add application-specific and deployment-specific information.

Figure 3–1 OC4J Configuration Files



Note that one of the server configuration files is a global `application.xml` file, which is for overall defaults that apply to any application. In addition, each application has its own `application.xml` file, which applies to the particular application only.

Changes to the global `application.xml`, `global-web-application.xml`, `server.xml`, and `web.xml` are picked up automatically by OC4J.

Deploying a Web application on OC4J involves at least the following configuration files:

- `server.xml`

- `default-web-site.xml`, or appropriate Web site XML file for a separate Web site
- `global-web-application.xml`
- `web.xml`
- optionally `orion-web.xml`

The `server.xml` file (as well as other configuration files from Figure 3–1) is discussed in the *Oracle9iAS Containers for J2EE User's Guide*. The other files are discussed in this chapter. See "The global-web-application.xml and orion-web.xml Files" on page 3-12 and "The default-web-site.xml File and Other Web Site XML Files" on page 3-26. Also be aware that `web.xml` is defined in the *Java(TM) Servlet Specification, Version 2.2* (and higher); you can refer to that document from Sun Microsystems at the following location:

`http://java.sun.com/j2ee/docs.html`

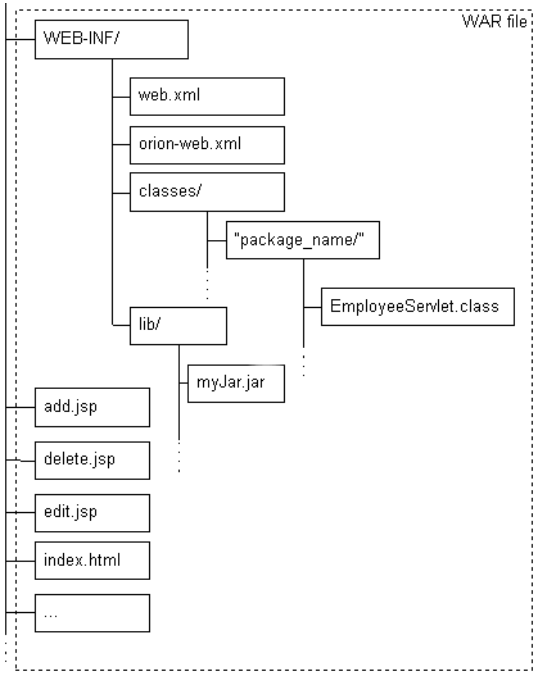
Application Assembly

How you assemble and build your application is up to you. Nevertheless, a standard directory structure is required for JAR and WAR deployment files, and it is simplest if you follow that when developing the application. This section discusses the standard directory structure, as well as application build mechanisms.

Application Directory Structure

Figure 3–2, shows the directory structure under the application root directory for a typical Web application. In OC4J, the root directory is *<app-name>/<web-app-name>*, according to the application name and corresponding Web application name. The application name is defined in the `server.xml` file and mapped to a Web application name in the `default-web-site.xml` file or other Web site XML file. (See "The default-web-site.xml File and Other Web Site XML Files" on page 3-26.)

Figure 3–2 Application Directory Structure



For easier application assembly and deployment, it is advisable to set up your Web application files in a pattern that is required for the deployment WAR file. The general rules are as follows:

- Put HTML files, JSP pages, and other resource files in the application root directory. The root directory is defined through the `root` attribute of the `<web-app>` element of `default-web-site.xml` or the Web site XML file for a particular Web site.
- Put servlet classes under the *<app_name>*`/`*<web-app-name>*`/WEB-INF/classes` directory, in subdirectories named after packages as appropriate. For example, if you have a servlet called `EmployeeServlet` in the `employee` package, then the class file should be as follows:

  <app_name>`/`*<web-app-name>*`/WEB-INF/classes/employee/EmpServlet.class`

- Put library files, such as JARs, that are required for the application in *<app_name>*`/`*<web-app-name>*`/WEB-INF/lib`.

Application Build Mechanisms

To build an application you have several options:

- **Preferred**—Create a `build.xml` file at the application root and use the `ant` utility to build the application. This utility is open-source and portable (between application servers, as well as operating systems) and so is ideal for Java-based applications. You can obtain `ant` and accompanying documentation at the following site:

  `http://jakarta.apache.org/ant/`

  Some of the sample applications that come with OC4J are set up to use `ant`. You can study the accompanying `build.xml` files for models.
- **Alternative**—Create a `makefile` to automate the compilation and assembly process and use a standard UNIX `make` utility or the open-source `gmake` utility to execute it.
- **Typically Not Advised**—Compile each Java source file manually, using a Java 1.3.x-compatible compiler. You will probably do this only at the very early stages of developing an application, if at all. It is a potentially error-prone mechanism.

A `build.xml` file or `makefile` might include steps to create EAR, WAR, and JAR file as appropriate for deployment, or you can create them manually. See "Application Deployment" on page 3-9 for information about these files.

Application Deployment

For J2EE-compatible deployment, each module requires a *deployment descriptor*. The descriptor is either a JAR file for EJB and client modules, or a WAR (Web ARchive) file for Web modules such as servlets and JSP pages.

The deployment descriptor for the entire application is the EAR (Enterprise ARchive) file, which wraps any WAR and JAR files.

You can create each of these deployment descriptors using the standard Java JAR utility. Specific examples appear below.

To deploy the application, follow these steps:

1. Create an `application.xml` file to specify the application modules. See the OC4J demos (such as the `stateless` application) and the *Oracle9iAS Containers for J2EE User's Guide* for more information about creating this file.
2. For each Web module in the application, create a `web.xml` descriptor file. This file is defined in the Servlet 2.3 specification. In addition, there is some introductory information about `web.xml` in "The global-web-application.xml and orion-web.xml Files" on page 3-12.
3. If your application has one or more EJB modules, then create an `ejb-jar.xml` file for each of these. See the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference* for more information about deploying EJB modules.
4. Create the WAR file for the Web module. When you are in the application root, issue the command:

   ```
   % jar -cvf <app_name>.war .
   ```

   This creates a JAR file with a `.war` extension. You can also examine the contents of the WAR file using the `jar` command. Here is an example, taken from the WAR file of the OC4J `stateless` sample application:

   ```
   % cd <app_root>/web
   % jar -tf employee-web.war
   META-INF/
   META-INF/MANIFEST.MF
   WEB-INF/
   WEB-INF/classes/
   WEB-INF/classes/employee/
   WEB-INF/classes/employee/EmployeeServlet.class
   WEB-INF/orion-web.xml
   WEB-INF/web.xml
   ```

```
delete.jsp
list.jsp
index.html
edit.jsp
add.jsp
```

The JAR utility creates the `META-INF/MANIFST.MF` file. You should not have to modify it.

5. Create the EAR file for the Web application. Use the `jar` command to create this file, as in the following example:

```
% jar -cvf employee.EAR .
```

Here is an example of an EAR file for the sample application `stateless`:

```
% jar -tf employee.ear
META-INF/
META-INF/MANIFEST.MF
META-INF/application.xml
META-INF/orion-application.xml
employee-ejb.jar
employee-web.war
employee-client.jar
```

For more information about EAR files, see the *Oracle9iAS Containers for J2EE User's Guide*.

6. If your application has one or more EJB modules, also include the EJB deployment descriptor in the EAR file. Here is a sample EJB JAR file:

```
% jar -tf employee-ejb.jar
META-INF/
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
META-INF/orion-ejb-jar.xml
employee/
employee/EmpRecord.class
employee/Employee.class
employee/EmployeeBean.class
employee/EmployeeHome.class
```

See the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference* for information about creating an EJB deployment descriptor and deploying an EJB application.

7. Deploy the application. In a production environment, use OEM.

Use of admin.jar for Deployment To deploy a Web application using the `admin.jar` command-line tool, typically in a development environment, go to the `j2ee/home` directory and issue the following commands in the application root directory.

```
% java -jar admin.jar ormi://localhost <admin_user> <admin_pw> \
       -deploy -file ./lib/<application_name>.ear  \
       -deploymentName <application_name>
```

This command adds the following entry to the `server.xml` file:

```
<application name="<app_name>"
             path="<your_path_to>/lib/.ear"
             auto-start="true"
/>
```

Then bind the Web access location into the `default-web-site.xml` file:

```
% java -jar admin.jar ormi://<hostname> <admin_user> <admin_pw> \
       -bindWebApp <app_name> <app_name>-web default-web-site \
            /<app_name>
```

This adds the following entry to the `default-web-site.xml` file:

```
<web-app
    application="<app_name>"
    name="<app_name>-web"
    root="/<app_name>"
/>
```

Note that in Oracle9*i*AS 9.0.2, if you modify configuration files without going through OEM, you must run the `dcmctl` tool, using its `updateConfig` command, to inform Oracle9*i*AS Distributed Configuration Management (DCM) of the updates. (This does not apply in an OC4J standalone mode, where OC4J is being run apart from Oracle9*i*AS.) Here is the `dcmctl` command:

```
dcmctl updateConfig -ct oc4j
```

The `dcmctl` tool is documented in the *Oracle9i Application Server Administrator's Guide*.

Configuration File Descriptions

This section discusses XML configuration files that are central to servlet development and invocation in an OC4J environment, including detailed element and attribute descriptions. The following topics are covered:

- Syntax Notes for Element Documentation
- The global-web-application.xml and orion-web.xml Files
- The default-web-site.xml File and Other Web Site XML Files

Syntax Notes for Element Documentation

The elements described here do not use body values unless specifically noted, and do not have subelements unless noted. If there is neither, the syntax is as follows (with "..." where attribute settings would appear):

```
<elementname ... />
```

If there are subelements, the syntax is as follows:

```
<elementname ... >
   ...subelements...
</elementname>
```

If a body value is used, the syntax is as follows:

```
<elementname ... >value</elementname>
```

The global-web-application.xml and orion-web.xml Files

This section describes the OC4J-specific `global-web-application.xml` and `orion-web.xml` files, and their relationships to the standard `web.xml` file. Overviews of these files and their features are followed by detailed descriptions of the elements supported by `global-web-application.xml` and `orion-web.xml`. This section is organized as follows:

- Overview of global-web-application.xml, orion-web.xml, and web.xml
- Standard Descriptor Configurations
- OC4J Descriptor Configurations
- Element Descriptions for global-web-application.xml and orion-web.xml
- Default global-web-application.xml File

Overview of global-web-application.xml, orion-web.xml, and web.xml

The file `j2ee/home/config/global-web-application.xml` is the descriptor for the OC4J global Web application, which is the parent of all Web applications on OC4J. The elements in this file define the default behavior of an OC4J Web application.

There is also, for each Web application, an application-specific `web.xml` file and an optional deployment-specific `orion-web.xml` file. Both of these files should be in the application `/WEB-INF` directory. Use of `web.xml` is standard, according to the Servlet 2.3 specification (and originally the Servlet 2.2 specification). Elements defined for the `orion-web.xml` file are a superset of those defined for `web.xml`, adding elements for OC4J-specific features. The `orion-web.xml` DTD is also used for `global-web-application.xml`—the two files support the same elements.

On deployment of a Web application, OEM or the `admin.jar` tool generates an `orion-web.xml` file, using the settings from the parent `global-web-application.xml` file. You can then update `orion-web.xml` as desired to override default values. You can also package `orion-web.xml` as part of your EAR file if you want to specify resource mappings or OC4J-specific configuration. In this case you will not have to override `orion-web.xml` after deployment.

The `global-web-application.xml`, `orion-web.xml`, and `web.xml` files all support a `<web-app>` element, which has many subelements. As you can see in "Default global-web-application.xml File" on page 3-23, the `global-web-application.xml` file typically defines defaults for many settings of the `<web-app>` element and its subelements. For desired settings specific to an application, use the `<web-app>` element and subelements in the `web.xml` file. When deploying an application, use the `<web-app>` element and subelements in `orion-web.xml` if you want to override any settings of the `web.xml` `<web-app>` element for this particular deployment.

OC4J-specific features are supported through the `<orion-web-app>` element and its many subelements in the `global-web-application.xml` and `orion-web.xml` files. The `<web-app>` element in these files is a subelement of `<orion-web-app>`. Use this element and its subelements in `orion-web.xml` to override `global-web-application.xml` settings of OC4J features for a particular application deployment.

Standard Descriptor Configurations

The `web.xml` descriptor file specifies the following servlet 2.3 standard configurations, among many others:

- names and classes of servlets in the Web module
- names of JSP pages
- servlet context initialization parameters
- location of any application-specific JSP tag libraries
- mappings of servlet names to URL patterns
- EJB references, including the JNDI names for looking up EJB home and remote interfaces

  (Only the Home interface JNDI name is provided, because only the Home interface is looked up through JNDI.)
- security constraints and security roles
- error code and error page mappings
- session timeout
- names of any filters in the Web application
- filter mappings—URL patterns that cause servlet filters to be triggered

  (Filter settings are outside the `<web-app>` element.)

OC4J Descriptor Configurations

The `global-web-application.xml` and `orion-web.xml` descriptor files, in addition to being able to specify almost all the same configurations as in the `web.xml` `<web-app>` element and subelements, can specify the following OC4J-specific configurations:

- additional servlet filtering and "servlet chaining"
- buffering
- character sets
- directory browsing
- document root
- locales
- classpath
- MIME mappings
- virtual directories

- access mask (to limit access to the servlet)
- clustering
- request and session tracking
- JNDI mappings
- security role mappings
- EJB mappings
- resource expiration settings

Element Descriptions for global-web-application.xml and orion-web.xml

The element descriptions in this section are applicable to either `global-web-application.xml` or to an application-specific `orion-web.xml` configuration file. Use `global-web-application.xml` to configure the global application and set defaults, and `orion-web.xml` to override these defaults for a particular application deployment as appropriate.

See "Syntax Notes for Element Documentation" on page 3-12 for general syntax information.

<orion-web-app ... >

This is the root element for specifying OC4J-specific configuration of a Web application.

Subelements:

```
<classpath>
<context-param-mapping>
<mime-mappings>
<virtual-directory>
<access-mask>
<cluster-config>
<servlet-chaining>
<request-tracker>
<servlet-filter>
<session-tracking>
<resource-ref-mapping>
<env-entry-mapping>
<security-role-mapping>
<ejb-ref-mapping>
<expiration-setting>
<web-app>
```

Attributes:

- `default-buffer-size`: Specifies the default size of the output buffer for servlet responses, in bytes. The default is `"2048"`.
- `default-charset`: This is the ISO character set to use by default. The default is `"iso-8859-1"`.
- `deployment-version`: This is the version of OC4J under which this Web application was deployed. If this value does not match the current version, then the application is redeployed. This is an internal server value and should not be changed.
- `development`: This is a convenience flag during development. If `development` is set to `"true"`, then each time you change the servlet source and save it in a particular directory, the OC4J server automatically compiles and redeploys the servlet the next time it is invoked. The directory is determined by the setting of the `source-directory` attribute. Supported values for `development` are `"true"` and `"false"` (default).
- `source-directory`: Specifies where to look for source files for classes to be auto-compiled if the `development` attribute is set to `"true"`. The default is `"WEB-INF/src"` if it exists, otherwise `"WEB-INF/classes"`.
- `directory-browsing`: Specifies whether to allow directory browsing. Supported values are `"allow"` and `"deny"` (default).
- `document-root`: Defines the path-relative or absolute directory to use as the root for served pages. The default setting is `"../"`.
- `file-modification-check-interval`: This is the amount of time, in milliseconds, for which a file-modification check is valid. Within that time period of the last check, further checks are not necessary. Zero or a negative number specifies that a check always occurs. The default is `"1000"`.
- `get-locale-from-user`: Specifies whether to determine the specific locale of the logged-in user before looking at the request headers for the information. Supported values are `"true"` and `"false"` (default, for performance reasons).
- `persistence-path`: Specifies where to store `HttpSession` objects for persistence across server restarts. Session objects must contain properly serializable or remoteable values, or EJB references, for this to work. There is no default.
- `servlet-webdir`: Specifies the servlet runner path for running a servlet by name—anything appearing after this in a URL is assumed to be a class name. This is typically for use during development and testing. For deployment, you

should instead use standard `web.xml` mechanisms for defining the context path and servlet path. The default is `"/servlet"`.

- `temporary-directory`: This is the absolute or relative path to a temporary directory that can be used by servlets and JSP pages for scratch files. The default is the `./temp` directory.

Note: The `File` object can be retrieved by the following code in a servlet or JSP page, according to the Servlet 2.2 specification:

```
File file = (File)application.getAttribute(
                        "javax.servlet.context.tempdir");
```

<classpath ... >

This specifies a codebase where classes used by this application can be found (servlet and JavaBeans, for example).

Attribute:

- `path`: This is the path or URL for the codebase, either absolute or relative to the location of the `orion-web.xml` file.

<context-param-mapping ... >*deploymentValue*</context-param-mapping>

In `orion-web.xml`, this overrides the value of a `context-param` setting in the `web.xml` file. It is used to keep the EAR assembly clean of deployment-specific values. Specify the new value in the tag body.

Attribute:

- `name`: This is the name of the `context-param` setting to override.

<mime-mappings ... >

This defines the path to a file containing MIME mappings to use.

Attribute:

- `path`: This is the path or URL for the file, either absolute or relative to the location of the `orion-web.xml` file.

<virtual-directory ... >

This adds a virtual directory mapping, used to include files that do not physically reside under the document root among the Web-exposed files.

Attributes:

- `real-path`: This is a real path, such as `/usr/local/realpath` on UNIX or `C:\testdir` in Windows.
- `virtual-path`: This is a virtual path to map to the specified real path.

<access-mask ... >

Use subelements of `<access-mask>` to specify optional access masks for this application. You can use host names or domains to filter clients, through `<host-access>` subelements, or you can use IP addresses and subnets to filter clients, through `<ip-access>` subelements, or you can do both.

Subelements:

```
<host-access>
<ip-access>
```

Attribute:

- `default`: Specifies whether to allow requests from clients that are not identified through a `<host-access>` or `<ip-access>` subelement. Supported values are `"allow"` (default) and `"deny"`. There are separate `mode` attributes for the `<host-access>` and `<ip-access>` subelements, which are used to specify whether to allow requests from clients that *are* identified through those subelements.

<host-access ... >

This subelement of `<access-mask>` specifies a host name or domain to allow or deny access.

Attributes:

- `domain`: This is the host or domain.
- `mode`: Specifies whether to allow or deny access to the specified host or domain. Supported values are "`allow`" (default) or "`deny`".

<ip-access ... >

This subelement of `<access-mask>` specifies an IP address and subnet mask to allow or deny access.

Attributes:

- `ip`: This is the IP address, as a 32-bit value (example: `"123.124.125.126`").
- `netmask`: This is the relevant subnet mask (example: `"255.255.255.0"`)
- `mode`: Specifies whether to allow or deny access to the specified IP address and subnet mask. Supported values are "`allow`" (default) or "`deny`".

<cluster-config ... >

Define this tag if the application is to be clustered. Clustered applications have their `ServletContext` and `HttpSession` data shared between the applications in the cluster. Shared objects must either be serializable or be remote RMI objects implementing the `java.rmi.Remote` interface.

See the *Oracle9iAS Performance Guide* for general information about clustering.

Attributes:

- `host`: This is the multicast host/IP for transmitting and receiving cluster data. The default is `"230.0.0.1"`.
- `id`: This is the ID (number) of this cluster node to identify itself within the cluster. The default is based on the local machine IP.
- `port`: This is the port through which to transmit and receive cluster data. The default is `"9127"`.

<servlet-chaining ... >

This element specifies a servlet to call when the response of the current servlet is set to a specified MIME type. The specified servlet will be called after the current servlet. This is known as *servlet chaining* and is useful for filtering or transforming certain kinds of output. Servlet chaining is an older servlet mechanism that is similar to servlet filtering (see `<servlet-filter>` below), which is specified in the Servlet 2.3 specification and covered in Chapter 4, "Servlet Filters".

Attributes:

- `mime-type`: This is the MIME type to trigger the chaining, such as`"text/html"`.
- `servlet-name`: This is the servlet to call when the specified MIME type is encountered.

<request-tracker ... >

This element specifies a servlet to use as the request tracker. A request tracker is called for each request, for use as desired. A request tracker might be useful for logging purposes, for example.

Attribute:

- `servlet-name`: This is the servlet to call as the request tracker.

<servlet-filter ... >

This element specifies a servlet to use as a filter. Filters are invoked for every request, and can be used to either pre-process the request or post-process the

response. Optionally, the filter would apply only to requests from servlets that match a specified URL pattern. Using `<servlet-filter>` to post-process a response is similar in nature to using `<servlet-chaining>` (see above), but is not based on MIME type.

Attributes:

- `servlet-name`: This is the servlet to call as the filter.
- `url-pattern`: This is an optional URL pattern to use as a qualifier for requests that are passed through the filter. For example: `"/the/*.pattern"`.

<session-tracking ... >

This element specifies the session-tracking settings for this application. Session tracking is accomplished through cookies, assuming a cookie-enabled browser. Session tracking through URL rewriting, also known as *auto-encoding*, is not currently supported.

The servlet to use as the session tracker is specified through a subelement.

Subelement:

```
<session-tracker>
```

Attributes:

- `autojoin-session`: Specifies whether users should be assigned a session as soon as they login to the application. Supported values are `"true"` and `"false"` (default).
- `cookie-domain`: This is the relevant domain for cookies. This is useful for sharing session state between nodes of a Web application running on different hosts.
- `cookie-max-age`: This number is sent with the session cookie and specifies a maximum interval (in seconds) for the browser to save the cookie. By default, the cookie is kept in memory during the browser session and discarded afterwards.
- `cookies`: Specifies whether to send session cookies. Supported values are `"enabled"` (default) and `"disabled"`.

<session-tracker ... >

This subelement of `<session-tracking>` specifies a servlet to use as the session tracker. Session trackers are invoked as soon as a session is created and are useful for logging purposes, for example.

Attribute:

`servlet-name`: This is the servlet to call.

<resource-ref-mapping ... >

Use this element to declare a reference to an external resource such as a data source, JMS queue, or mail session. This ties a resource reference name to a JNDI location when deploying.

Subelement:

`<lookup-context>`

Attributes:

- `location`: This is the JNDI location from which to look up the resource. Example: `"jdbc/TheDS"`.
- `name`: This is the resource reference name, which matches the name of a `resource-ref` element in the `web.xml` file. Example: `"jdbc/TheDSVar"`.

<lookup-context ... >

This subelement of `<resource-ref-mapping>` specifies an optional `javax.naming.Context` that will be used to retrieve the resource. This is useful when connecting to third-party modules, such as a third-party JMS server, for example. Either use the context implementation supplied by the resource vendor, or, if none exists, write an implementation that in turn negotiates with the vendor software.

Subelement:

`<context-attribute>`

Attribute:

- `location`: This is the name to look for in the foreign (such as third-party) context when retrieving the resource.

<context-attribute ... >

This subelement of `<lookup-context>` (which is a subelement of `<resource-ref-mapping>`) specifies an attribute to send to the foreign context.

The only mandatory attribute in JNDI is `java.naming.factory.initial`, which is the class name of the context factory implementation.

Attributes:

- `name`: Specifies the name of the attribute.
- `value`: Specifies the value of the attribute.

<env-entry-mapping ... >*deploymentValue*</env-entry-mapping>

In `orion-web.xml`, this element overrides the value of an `env-entry` setting in the `web.xml` file. It is used to keep the EAR assembly clean of deployment-specific values. Specify the new value in the tag body.

Attribute:

- `name`: This is the name of the `env-entry` setting to override.

<security-role-mapping ... >

This element maps a security role to specified users and groups, or to all users. It maps to a security role of the same name in the `web.xml` file. Use either the `impliesAll` attribute or an appropriate combination of subelements—`<group>`, or `<user>`, or both.

Subelements:

```
<group>
<user>
```

Attributes:

- `impliesAll`: Specifies whether this mapping implies all users. Supported values are `"true"` or `"false"` (default).
- `name`: This is the name of the security role. It must match a name specified in a `<role-name>` subelement of a `<security-role>` element in `web.xml`.

<group ... >

Use this subelement of `<security-role-mapping>` to specify a group to map to the security role of the parent `<security-role-mapping>` element. All the members of the specified group are included in this role.

Attribute:

- `name`: This is the name of the group.

<user ... >

Use this subelement of `<security-role-mapping>` to specify a user to map to the security role of the parent `<security-role-mapping>` element.

Attribute:

- `name`: This is the name of the user.

<ejb-ref-mapping ... >

This element creates a mapping between an EJB reference, defined in an `<ejb-ref>` element, and a JNDI location when deploying.

The <ejb-ref> element can appear within the <web-app> element of orion-web.xml or web.xml, and is used to declare a reference to an EJB.

Attributes:

- location: This is the JNDI location from which to look up the EJB home.
- name: This is the EJB reference name, which matches the <ejb-ref-name> setting of the <ejb-ref> element.

<expiration-setting ... >

This element sets the expiration for a given set of resources. This is useful for caching policies, such as for not reloading images as frequently as documents.

Attributes:

- expires: This is the time, in seconds, before expiration, or "never" for no expiration.
- url-pattern: This is the URL pattern that the expiration applies to, such as "*.gif", for example.

<web-app ... >

This element is used as in the standard web.xml file; see the Servlet 2.3 specification for details. In global-web-application.xml, you can establish defaults for <web-app> settings. In web.xml, you can specify application-specific <web-app> settings to override the defaults. In orion-web.xml, you can specify deployment-specific <web-app> settings to override the settings in web.xml.

Note: In a global-web-application.xml or orion-web.xml file, filter settings within the <web-app> element are not supported, because that would conflict with the <servlet-filter> subelement under the <orion-web-app> element.

Default global-web-application.xml File

This is an example of a default global-web-application.xml file (may be subject to change in the shipped product):

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE orion-web-app PUBLIC '//Evermind//Orion web-application'
'http://xmlns.oracle.com/ias/dtds/orion-web.dtd'>

<orion-web-app
```

```
        jsp-cache-directory="./persistence"
        servlet-webdir="/servlet"
        development="false"
>

        <!-- The mime-mappings for this server -->
        <mime-mappings path="./mime.types" />

        <web-app>
                <!--
                <servlet>
                        <servlet-name>xsl</servlet-name>
                        <servlet-class>com.evermind.servlet.XSLServlet
                        </servlet-class>
                        <init-param>
                                <param-name>defaultContentType</param-name>
                                <param-value>text/html</param-value>
                        </init-param>
                </servlet>
                -->

                <servlet>
                        <servlet-name>jsp</servlet-name>
                        <servlet-class>oracle.jsp.runtimev2.JspServlet
                        </servlet-class>
                </servlet>

                <servlet>
                        <servlet-name>rmi</servlet-name>
                    <servlet-class>com.evermind.server.rmi.RMIHttpTunnelServlet
                    </servlet-class>
                </servlet>

                <servlet>
                        <servlet-name>rmip</servlet-name>
                <servlet-class>com.evermind.server.rmi.RMIHttpTunnelProxyServlet
                </servlet-class>
                </servlet>

                <servlet>
                        <servlet-name>ssi</servlet-name>
                        <servlet-class>com.evermind.server.http.SSIServlet
                        </servlet-class>
                </servlet>
```

```
<servlet>
        <servlet-name>cgi</servlet-name>
        <servlet-class>com.evermind.server.http.CGIServlet
        </servlet-class>
</servlet>

<servlet>
        <servlet-name>perl</servlet-name>
        <servlet-class>com.evermind.server.http.CGIServlet
        </servlet-class>
        <init-param>
                <param-name>interpreter</param-name>
                <param-value>perl</param-value>
        </init-param>
</servlet>

<servlet>
        <servlet-name>php</servlet-name>
        <servlet-class>com.evermind.server.http.CGIServlet
        </servlet-class>
        <init-param>
        <param-name>interpreter</param-name>
                <param-value>php</param-value>
        </init-param>
</servlet>

<servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>/*.jsp</url-pattern>
</servlet-mapping>
<servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>/*.JSP</url-pattern>
</servlet-mapping>
<servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>/*.sqljsp</url-pattern>
</servlet-mapping>
<servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>/*.SQLJSP</url-pattern>
</servlet-mapping>

<servlet-mapping>
        <servlet-name>cgi</servlet-name>
```

```
                <url-pattern>/*.cgi</url-pattern>
            </servlet-mapping>

            <servlet-mapping>
                <servlet-name>perl</servlet-name>
                <url-pattern>/*.pl</url-pattern>
            </servlet-mapping>

            <servlet-mapping>
                <servlet-name>php</servlet-name>
                <url-pattern>/*.php</url-pattern>
            </servlet-mapping>

            <servlet-mapping>
                <servlet-name>php</servlet-name>
                <url-pattern>/*.php3</url-pattern>
            </servlet-mapping>

            <servlet-mapping>
                <servlet-name>php</servlet-name>
                <url-pattern>/*.phtml</url-pattern>
            </servlet-mapping>

            <servlet-mapping>
                <servlet-name>ssi</servlet-name>
                <url-pattern>/*.shtml</url-pattern>
            </servlet-mapping>

            <welcome-file-list>
                <welcome-file>index.html</welcome-file>
                <welcome-file>default.jsp</welcome-file>
            </welcome-file-list>
        </web-app>
</orion-web-app>
```

The default-web-site.xml File and Other Web Site XML Files

This section describes OC4J Web site XML files, including `default-web-site.xml` for the default OC4J Web site. The documentation includes descriptions of the elements and attributes of these files.

Overview of default-web-site.xml and Web Site XML Files

A Web site XML file contains the configurations for an OC4J Web site. The file `j2ee/home/config/default-web-site.xml` configures the default OC4J Web site, also defining default configurations for any additional Web site XML files.

The names of any additional Web site XML files are defined in the `server.xml` file, in the `path` attributes of any `<web-site>` elements. See the *Oracle9iAS Containers for J2EE User's Guide* for more information about the `server.xml` file.

Configuration settings in Web site XML files include the following:

- host name/IP as well as virtual host settings for this site
- port to listen on and whether the site is secure (using SSL)
- default web application for this site
- additional Web applications for this site
- access-log format
- settings for user Web applications (for /~user/ sites)
- SSL configuration

Note: To use HTTP protocol to access OC4J directly, instead of going through the Oracle HTTP Server and AJP protocol, use a Web site XML file that specifies a `port` setting and `protocol="http"` (instead of `ajp13`). These are attributes of the `<web-site>` element. This is not recommended for a production environment, however.

Element Descriptions for default-web-site.xml and Web Site XML Files

The element descriptions in this section apply to `default-web-site.xml` and the Web site XML files for any additional OC4J Web sites. Use `default-web-site.xml` to configure the default Web site and set overall defaults, and additional Web site XML files to override these defaults for particular OC4J Web sites, as appropriate.

See "Syntax Notes for Element Documentation" on page 3-12 for general syntax information.

<web-site ... >

This is the root element for configuring an OC4J Web site.

Subelements:

```
<description>
<frontend>
<web-app>
<default-web-app>
<user-web-apps>
<access-log>
<ssl-config>
```

Attributes:

- `cluster-island`: A *cluster island* is two or more Web servers that share session failover state for replication. Use the `cluster-island` attribute when clustering the Web tier between multiple OC4J instances. If this attribute is set to a cluster island ID (number spawning from 1 and up), then this Web site will participate as a back-end server in the island specified by the ID. The ID is a chosen number that depends on your clustering configuration. If only one island is used, the ID is always 1.

  See the *Oracle9iAS Performance Guide* for general information about clustering.

- `display-name`: This is for a user-friendly or informal Web site name to display in GUI configuration tools when the site is being administered.
- `host`: This is the host IP address for this site. If `"[ALL]"` is specified, then all IP addresses of the server are used.
- `log-request-info`: Specifies whether to log information about the incoming request (such as headers) if an error occurs. Supported values are `"true"` and `"false"` (default).
- `max-request-size`: Sets a maximum size, in bytes, for incoming requests. If a client sends a request that exceeds this maximum, it will receive a "request entity too large" error. The default maximum is 15000.
- `secure`: Specifies whether to support SSL (Secure Socket Layer). Supported values are `"true"` and `"false"` (default). If you enable this, use the `<ssl-config>` element for SSL configuration settings.
- `protocol`: Specifies the protocol that the Web site is using. Supported values are `"http"`, `"https"`, and `"ajp13"` (Apache JServ Protocol, or AJP—default). The `ajp13` protocol is for use with Oracle HTTP Server and `mod_oc4j` only, and is highly recommended for production environments. Note that each port must have a corresponding protocol, and vice versa.

- `port`: This is the port number for this Web site. Each port must have a corresponding protocol, and vice versa. Also note that for AJP, port 0 has a special meaning. Any non-zero port number is static, but with a `port` setting of `"0"`, the servlet container dynamically accesses any available port. This functionality is invisible to the user, who is only aware of the Oracle HTTP Server port specified through the browser (such as 7777, typical for access through the Oracle HTTP Server with Oracle9*i*AS Web Cache enabled).
- `use-keep-alives`: Typical behavior for a servlet container is to close a connection once a request has been completed. With a `use-keep-alives` setting of `"true"`, however, a connection is maintained across requests. For AJP protocol, connections are always maintained and this attribute is ignored. For HTTP and HTTPS, the default is `"true"`; disabling it may cause major performance loss.
- `virtual-hosts`: This optional setting is useful for virtual sites sharing the same IP address. The value is a comma-separated list of host names tied to this Web site.

<description>*This is the description.*</description>

You can optionally use the body of this element for a brief description of the Web site. The `<description>` element has no attributes or subelements.

<frontend ... >

This specifies a perceived front-end host and port of this Web site as seen by HTTP clients. When the site is behind something like a load balancer or firewall, the `<frontend>` specification is necessary to provide appropriate information to Web application code for functionality such as URL rewriting. Using the host and port specified in the `<frontend>` element, the back-end server that is actually running the application knows to refer to the front-end instead of to itself in any URL rewriting. This way, subsequent requests properly come in through the front-end again instead of trying to access the back-end directly.

Attributes:

- `host`: This is the host name of the front-end server, such as `"www.acme.com"`.
- `port`: This is the port number of the front-end server, such as `"80"`.

<web-app ... >

This element creates a reference to a Web application—a J2EE application, defined in the `server.xml` file, that is bound to this particular Web site. Each instance of a J2EE application bound to a particular Web site is a separate Web entity.

The Web application is bound at the location specified by the `root` attribute.

Attributes:

- `application`: This is the name of the J2EE application, as specified by the `application` attribute of an `<application>` element in the `server.xml` file.
- `load-on-startup`: Optional attribute to specify whether this Web application should be preloaded on startup. Otherwise, it is loaded upon the first request for it. Supported values are `"true"` and `"false"` (default).
- `max-inactivity-time`: Optional attribute to specify a period of minutes of inactivity after which the Web application will automatically be shut down. The default is no automatic shutdown.
- `name`: Specify the desired Web application name. For example, if the J2EE application name is `MyApp`, and this is Web site #2 of 4, you might specify a Web application name of `MyWebApp2`. This name must be the same as the corresponding name specified in a `<web-module>` element in the `application.xml` file, to be bound to this Web site under the specified root context.
- `root`: The path on this Web site to which the Web application should be bound. For example, if the Web application `CatalogApp` at Web site `www.site.com` is bound to the root `"/catalog"`, then it can be accessed as follows:

  ```
  http://www.site.com/catalog/CatalogApp
  ```

  It is advisable to use the `root` value defined in `application.xml`. This occurs automatically when you use Oracle Enterprise Manager.
- `shared`: This indicates whether multiple bindings (different Web sites and context roots) can be shared. Supported values are `"true"` and `"false"` (default). Sharing implies the sharing of everything that makes up a Web application, including sessions, servlet instances, and context values. The most common use for this mode is to share a Web application between an HTTP site and an HTTPS site at the same context path. If an HTTPS Web application is marked as shared, its session tracking strategy reverts from SSL session tracking to session tracking through cookies or URL rewriting. This may make the Web application less secure, but might be necessary to work around issues such as SSL session timeouts not being properly supported in some browsers.

<default-web-app ... >

This element creates a reference to the default Web application of this Web site.

The default Web application is bound to `"/j2ee"` in `default-web-site.xml`.

Attributes are the same as for the <web-app> element described immediately above, with the following exceptions:

- There is no need for a root attribute.
- The default setting of load-on-startup is "true".

<user-web-apps ... >

Use this element to support user directories and applications. Each user has his or her own Web application and associated web-application.xml file. User applications are reached at */username/* from the server root.

Attributes:

- max-inactivity-time: Optional attribute to specify a period of minutes of inactivity after which the user application will automatically be shut down. The default is no automatic shutdown.
- path: This is a path to specify the local directory of the user application, including a wildcard for the user name. The default path setting on UNIX, for example, is "/home/*", where "*" is replaced by the particular user name.

<access-log ... >

This element specifies information about the access log for this Web site, including the path and what information is included. This is where incoming requests are logged.

Attributes:

- format: Specify one or more of several supported variables that result in information being prepended to log entries. Supported variables are $time $request, $ip, $host, $path, $size, $method, $protocol, $user, $status, $referer, $time, $agent, $cookie, $header, and $mime. Between variables, you can type in any separator characters that you want to appear between values in the log message. The default setting is as follows:

  ```
  "$ip - $user - [$time] '$request' $status $size"
  ```

  As an example, this would result in log messages such as the following (with the second message wrapping around to a second line):

  ```
  148.87.1.180 - - [06/Nov/2001:10:23:18 -0800] 'GET / HTTP/1.1' 200 2929
  148.87.1.180 - - [06/Nov/2001:10:23:53 -0800] 'GET
  /webservices/statefulTest HTTP/1.1' 200 301
  ```

The user is null, the time is in brackets (as specified in the `format` setting), the request is in quotes (as specified), and the status and size in the first message are 200 and 2929, respectively.

- `path`: Specifies the path and name of the access log, such as `"./access.log"`. The default setting in `default-web-site.xml` is the following:

  ```
  "../log/default-web-access.log"
  ```

- `split`: Specifies how often to begin a new access log. Supported values are `"none"` (never), `"hour"`, `"day"`, `"week"`, or `"month"`. For a value other than `"none"`, logs are named according to the `suffix` attribute.

- `suffix`: Specifies timestamp information to append to the base file name of the logs (as specified in the `path` attribute) if splitting is used, to make a unique name for each file. The format used is that of `java.text.SimpleDateFormat`, and symbols used in `suffix` settings are according to the symbology of that class. For information about `SimpleDateFormat` and the format symbols that is uses, refer to the Sun Microsystems Javadoc at the following location:

  ```
  http://java.sun.com/products/jdk/1.2/docs/api/index.html
  ```

  The default `suffix` setting is `"-yyyy-MM-dd"`. These characters are case-sensitive, as described in the `SimpleDateFormat` documentation.

  As an example, assume the following `<access-log>` element (using the default `suffix` value):

  ```
  <access-log path="c:\foo\web-site.log" split="day" />
  ```

  Log files would be named such as in the following example:

  ```
  c:\foo\web-site-2001-11-17.log
  ```

<ssl-config ... >

This element specifies SSL configuration settings, if applicable. Use it when you set the `secure` attribute of the `<web-site>` element to `"true"`.

If the application uses a third-party `SSLServerSocketFactory` implementation, you can use `<property>` subelements of `<ssl-config>` to send parameters to the factory.

Subelements:

```
<property>
```

Attributes:

- `factory`: If you are not using JSSE (Java Secure Socket Extension), use the `factory` attribute to specify an implementation of `SSLServerSocketFactory`. The default setting is:

  ```
  "JSSE: com.evermind.ssl.JSSESSLServerSocketFactory"
  ```

- `keystore`: A relative or absolute path to the keystore database (a binary file) used by this Web site to store certificates and keys for the user base in this installation. A keystore is a `java.security.KeyStore` instance. The database can be created using standard JavaSoft tools.
- `keystore-password`: The required password to open the keystore.
- `needs-client-auth`: Indicates whether the client must submit a certificate for authorization to log in. Supported values are `"true"` for client-side authorization (client must submit certificate), and `"false"` (default), for server-side authentication (no certificate required).
- `provider`: Specify the provider if JSSE is used. By default, OC4J generally uses the Sun Microsystems implementation of SSL (using `com.sun.net.ssl.internal.ssl.Provider` for provider). However, the Oracle SSL implementation is also used in some cases, such as for SOAP and `http_client`.

<property ... >

If you are using a third-party `SSLServerSocketFactory` implementation for SSL, you can use `<property>` subelements of `<ssl-config>` to pass parameters to the factory.

Attributes:

- `name`: The name of a parameter to pass to the factory.
- `value`: The value of the specified parameter.

Default default-web-site.xml File

This is an example of a default `default-web-site.xml` file (may be subject to change in the shipped product):

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE web-site PUBLIC "Oracle9iAS XML Web-site"
"http://xmlns.oracle.com/ias/dtds/web-site.dtd">

<!-- change the host name below to your own host name. Localhost will -->
```

```
<!-- not work with clustering -->
<!-- also add cluster-island attribute as below
<web-site host="localhost" port="0"  protocol="ajp13"
          display-name="Default Oracle 9iAS Java WebSite" cluster-island="1" >
-->

<web-site port="0"  protocol="ajp13"
          display-name="Default Oracle9iAS Containers for J2EE Web Site">
        <!-- Uncomment the following line when using clustering -->
        <!-- <frontend host="your_host_name" port="80" /> -->
        <!-- The default web-app for this site, bound to the root -->
        <default-web-app application="default" name="defaultWebApp"
                         root="/j2ee" />
        <web-app application="default" name="dms" root="/dmsoc4j" />

        <web-app application="ojspdemos" name="ojspdemos-web"
                 root="/ojspdemos" />

        <!-- Uncomment the following to access these apps.
        <web-app application="callerInfo" name="callerInfo-web" root="/jazn" />
        <web-app application="news" name="news-web" root="/news" />
        <web-app application="logger" name="messagelogger-web"
                 root="/messagelogger" />
        <web-app application="ws_example" name="ws_example"
                 root="/webservices" />
        -->
        <!-- Access Log, where requests are logged to -->
        <access-log path="../log/default-web-access.log" />
</web-site>
```

4

Servlet Filters

This chapter describes servlet filters, which can be a useful part of Web-tier applications. Filters are new in the Servlet 2.3 specification, although many earlier Web servers have supported similar constructs.

This chapter covers the following topics:

Overview of Servlet Filters

When the servlet container calls a method in a servlet on behalf of the client, the HTTP request that the client sent is, by default, passed directly to the servlet. The response that the servlet generates is, by default, passed directly back to the client, with its content unmodified by the container. So, normally, the servlet must process the request and generate as much of the response as the application requires.

But there are many cases where some preprocessing of the request for servlets would be useful. In addition, it is sometimes useful to modify the response from a class of servlets. One example is encryption. A servlet, or a group of servlets in an application, might generate response data that is sensitive and should not go out over the network in clear-text form, especially when the connection has been made using a non-secure protocol such as HTTP. A filter can encrypt the responses. Of course, in this case the client must be able to decrypt the responses.

A common case for a filter is where you want to apply pre-processing or post-processing to requests and responses for a group of servlets, not just a single servlet. If you need to modify the request or response for just one servlet, there is no need to create a filter—just do what is required directly in the servlet itself.

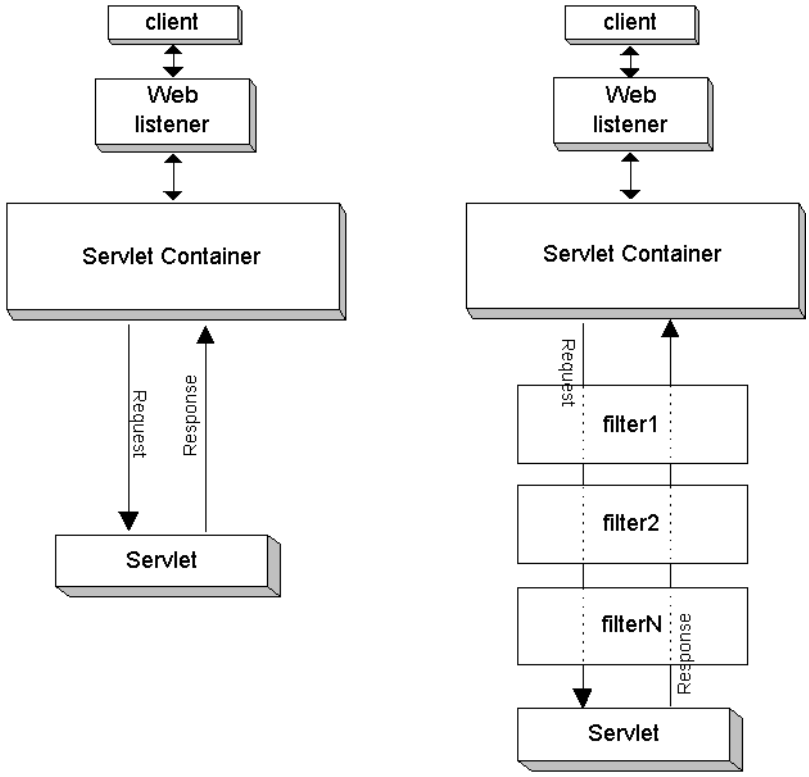
Note that filters are not servlets. They do not implement and override `HttpServlet` methods such as `doGet()` or `doPost()`. Rather, a filter implements the methods of the `javax.servlet.Filter` interface. The methods are:

- `init()`
- `destroy()`
- `doFilter()`

How the Servlet Container Invokes Filters

Figure 4–1 shows how the servlet container invokes filters. On the left is a scenario where no filters are configured for the servlet being called. On the right, several filters (1, 2, ..., N) have been configured in a chain to be invoked by the container before the servlet is called. Specify in the `web.xml` file which servlets or JSP pages cause the container to invoke the filters.

Figure 4–1 Servlet Invocation with and without Filters



The order in which filters are invoked depends on the order in which they are configured in the `web.xml` file. The first filter `web.xml` is the first one invoked during the request, and the last filter in `web.xml` is the first one invoked during the response (note the reverse order during the response).

Filter Examples

This section lists and describes three servlet filter examples.

Filter Example #1

This section provides a simple filter example. Any filter must implement the three methods in the `javax.servlet.Filter` interface or must extend a class that implements them. So the first step is to write a class that implements these methods. This class, which we will call `GenericFilter`, can be extended by other filters.

Generic Filter

Here is the generic filter code. Assume this generic filter is part of the `com.acme.filter` package, so you should set up a corresponding directory structure somewhere.

The numbers in comments at the right of the code match numbers in the "Code Notes" below.

```
package com.acme.filter;                                                    //1.
import javax.servlet.*;

public
class GenericFilter implements javax.servlet.Filter {
  public FilterConfig filterConfig;                                         //2.

  public void doFilter(final ServletRequest request,                        //3.
                       final ServletResponse response,
                       FilterChain chain)
      throws java.io.IOException, javax.servlet.ServletException {
    chain.doFilter(request,response);                                       //4.
  }

  public void init(final FilterConfig filterConfig) {              //5.
    this.filterConfig = filterConfig;
  }

  public void destroy() {                                                   //6.
  }
}
```

Save this code in a file called `GenericFilter.java` in the package directory.

Code Notes

1. The filter examples in this chapter are kept in this package.
2. This declares a variable to save the filter configuration object.
3. The `doFilter()` method contains the code that implements the filter.
4. In the generic case, just call the filter chain.
5. The `init()` method saves the filter configuration in a variable.
6. The `destroy()` method can be overridden to accomplish any required finalization.

Filter Code: HelloWorldFilter.java

This filter overrides the `doFilter()` method of the `GenericFilter` class above. It prints a message on the console when it is called on entrance, next adds a new attribute to the servlet request, then calls the filter chain. In this example there is no other filter in the chain, so the container passes the request directly to the servlet. Enter the following code in a file called `HelloWorldFilter.java`:

```
package com.acme.filter;

import javax.servlet.*;

public class HelloWorldFilter extends GenericFilter {
  private FilterConfig filterConfig;

  public void doFilter(final ServletRequest request,
                       final ServletResponse response,
                       FilterChain chain)
       throws java.io.IOException, javax.servlet.ServletException  {
    System.out.println("Entering Filter");
    request.setAttribute("hello","Hello World!");
    chain.doFilter(request,response);
    System.out.println("Exiting HelloWorldFilter");
  }
}
```

JSP Code: filter.jsp

To keep the example simple, the "servlet" to process the filter output is written as a JSP page. Here it is:

```
<HTML>
<HEAD>
<TITLE>Filter Example 1</TITLE>
```

```
</HEAD>
<BODY>
<HR>
<P><%=request.getAttribute("hello")%></P>
<P>Check your console output!</P>
<HR>
</BODY>
</HTML>
```

The JSP page gets the new request attribute, `hello`, that the filter added, and prints its value on the console. Put the `filter.jsp` page in the document root of the application—in this case, in `j2ee/home/default-web-app`—and make sure your console window is visible when you invoke `filter.jsp` from your browser.

Setting Up Example #1

To test the filter examples in this chapter, we will use the OC4J default application. Configure the filter in the `web.xml` file of the default application by editing `j2ee/home/default-web-app/WEB-INF/web.xml`. Add the following lines to this file, in the `<web-app>` element:

```
<!-- Filter Example #1 -->
<filter>
  <filter-name>helloWorld</filter-name>
  <filter-class>com.acme.filter.HelloWorldFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>helloWorld</filter-name>
  <url-pattern>/filter.jsp</url-pattern>
</filter-mapping>
<!-- end Filter Example #1 -->
```

The `<filter>` element defines the name of the filter and the Java class that implements the filter. The `<filter-mapping>` element defines the URL pattern that specifies to which targets the `<filter-name>` should apply. In this simple example, the filter applies to only one target: the JSP code in `filter.jsp`.

Running Example #1

Invoke `filter.jsp` from your Web browser as follows, and watch the output on your console:

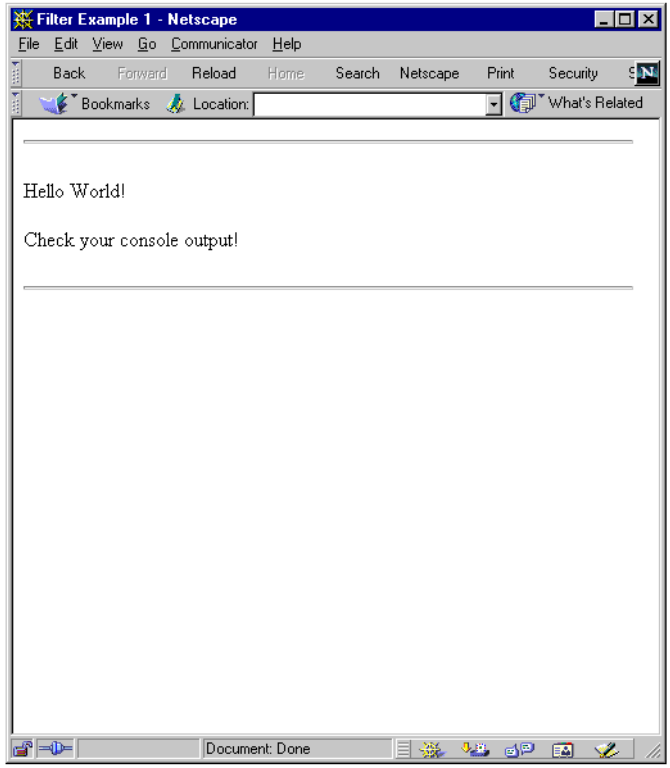
`http://<`*`hostname`*`>:<`*`TTCport`*`>/j2ee/filter.jsp`

The console output should look something like this:

```
<hostname>% Entering Filter
Exiting HelloWorldFilter
```

The output to the Web browser is something like what is shown in Figure 4–2.

Figure 4–2 Example #1 Output



Filter Example #2

You can configure a filter with initialization parameters in the `web.xml` file. This section provides a filter example that uses the following `web.xml` entry, which demonstrates a parameterized filter:

```
<!-- Filter Example #2 -->
<filter>
  <filter-name>message</filter-name>
```

```
    <filter-class>com.acme.filter.MessageFilter</filter-class>
    <init-param>
      <param-name>message</param-name>
      <param-value>A message for you!</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>message</filter-name>
    <url-pattern>/filter2.jsp</url-pattern>
  </filter-mapping>
  <!-- end Filter Example #2 -->
```

Here, the filter named `message` has been configured with an initialization parameter, also called `message`. The value of the `message` parameter is "A message for you!"

Filter Code: MessageFilter.java

The code to implement the `message` filter example is shown below. Note that it uses the `GenericFilter` class from "Filter Example #1" on page 4-4.

```
package com.acme.filter;
import javax.servlet.*;

public class MessageFilter extends GenericFilter {
  public void doFilter(final ServletRequest request,
                       final ServletResponse response,
                       FilterChain chain)
      throws java.io.IOException, javax.servlet.ServletException {
    System.out.println("Entering MessageFilter");
    String message = filterConfig.getInitParameter("message");
    request.setAttribute("message",message);
    chain.doFilter(request,response);
    System.out.println("Exiting MessageFilter");
  }
}
```

This filter uses the `filterConfig` object that was saved in the generic filter. The `filterConfig.getInitParameter()` method returns the value of the initialization parameter.

JSP Code: filter2.jsp

As in the first example, this example uses a JSP page to implement the "servlet" that tests the filter. The filter named in the `<url-pattern>` tag above is `filter2.jsp`.

Here is the code, which you can enter into a file `j2ee/home/default-web-app/filter2.jsp`:

```
<HTML>
<HEAD>
<TITLE>Lesson 2</TITLE>
</HEAD>
<BODY>
<HR>
<P><%=request.getAttribute("message")%></P>
<P>Check your console output!</P>
<HR>
</BODY>
</HTML>
```

Running Example #2

Make sure that you have entered the filter configuration in the `web.xml` file, as shown above. Then access the JSP page with your browser:

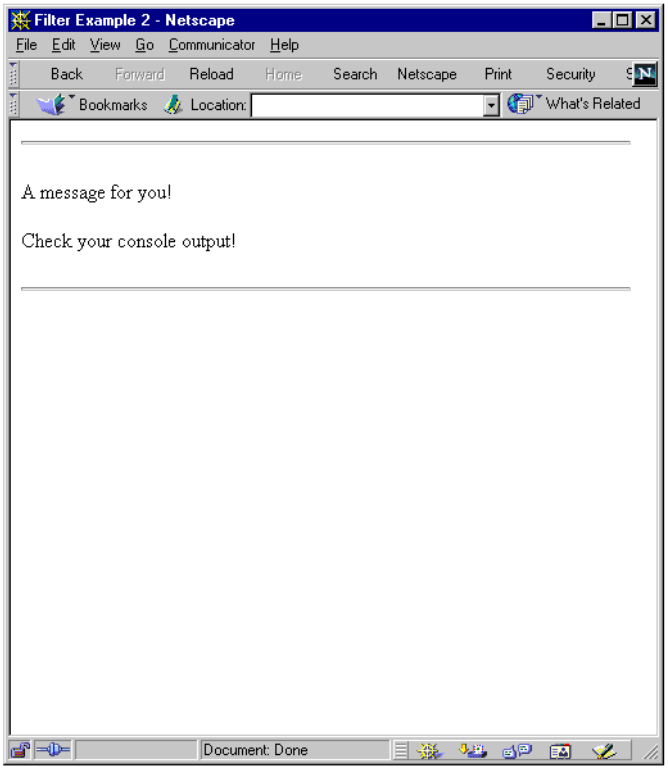
`http://<hostname>:<port>/j2ee/filter2.jsp`

The console output should show something like the following:

```
Auto-deploying file:/private/tssmith/appserver/default-web-app/ (Assembly had
been updated)...
Entering MessageFilter
Exiting MessageFilter
```

Note the message from the server showing that it redeployed the default application after the `web.xml` file was edited, and note the messages from the filter as it was entered and exited. The Web browser screen should show something like what is shown in Figure 4–3.

Figure 4–3 Example #2 Output



Filter Example #3

A particularly useful function for a filter is to manipulate the response to a request. To accomplish this, use the standard `javax.servlet.http.HttpServletResponseWrapper` class, a custom `javax.servlet.ServletOutputStream` object, and a filter. To test the filter, you also need a target to be processed by the filter. In this example, the target that is filtered is a JSP page.

There are three new classes to write to implement this example:

- `FilterServletOutputStream`—a new implementation of `ServletOutputStream` for response wrappers

- `GenericResponseWrapper`—a basic implementation of the response wrapper interface
- `PrePostFilter`—the code that implements the filter

This example uses the `HttpServletResponseWrapper` class to wrap the response before it is sent to the target. This class is an object that acts as a wrapper for the `ServletResponse` object (using a Decorator design pattern, as described in *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides; Addison-Wesley Press). It is used to wrap the real response so that it can be modified after the target of the request has delivered its response.

The HTTP servlet response wrapper developed in this example uses a custom servlet output stream that lets the wrapper manipulate the response data after the servlet (or JSP page, in this example) is finished writing it out. Normally, this cannot be done after the servlet output stream has been closed (essentially, after the servlet has committed it). That is the reason for implementing a filter-specific extension to the `ServletOutputStream` class in this example.

Output Stream: FilterServletOutputStream.java

The `FilterServletOutputStream` class is used to manipulate the response of another resource. This class overrides the three `write()` methods of the standard `java.io.OutputStream` class.

Here is the code for the new output stream:

```
package com.acme.filter;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public
class FilterServletOutputStream extends ServletOutputStream {

  private DataOutputStream stream;

  public FilterServletOutputStream(OutputStream output) {
    stream = new DataOutputStream(output);
  }

  public void write(int b) throws IOException  {
    stream.write(b);
  }
```

```
  public void write(byte[] b) throws IOException  {
    stream.write(b);
  }

  public void write(byte[] b, int off, int len) throws IOException  {
    stream.write(b,off,len);
  }

}
```

Save this code in the following directory and compile it:

```
j2ee/home/default-web-app/WEB-INF/classes/com/acme/filter
```

Servlet Response Wrapper: GenericResponseWrapper.java

To use the custom `ServletOutputStream` class, implement a class that can act as a response object. This wrapper object is sent back to the client in place of the original response generated by the servlet (or JSP page).

The wrapper must implement some utility methods, such as to retrieve the content type and content length of its content. The `GenericResponseWrapper` class accomplishes this:

```
package com.acme.filter;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class GenericResponseWrapper extends HttpServletResponseWrapper {
  private ByteArrayOutputStream output;
  private int contentLength;
  private String contentType;

  public GenericResponseWrapper(HttpServletResponse response) {
    super(response);
    output=new ByteArrayOutputStream();
  }

  public byte[] getData() {
    return output.toByteArray();
  }

  public ServletOutputStream getOutputStream() {
```

```
    return new FilterServletOutputStream(output);
  }

  public PrintWriter getWriter() {
    return new PrintWriter(getOutputStream(),true);
  }

  public void setContentLength(int length) {
    this.contentLength = length;
    super.setContentLength(length);
  }

  public int getContentLength() {
    return contentLength;
  }

  public void setContentType(String type) {
    this.contentType = type;
    super.setContentType(type);
  }


  public String getContentType() {
    return contentType;
  }
}
```

Save this code in the following directory and compile it:

```
j2ee/home/default-web-app/WEB-INF/classes/com/acme/filter
```

Writing the Filter

This filter adds content to the response of the servlet (or JSP page) after that target is invoked. This filter extends the filter from "Generic Filter" on page 4-4.

Filter Code: PrePostFilter.java

```
package com.acme.filter;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class PrePostFilter extends GenericFilter {

  public void doFilter(final ServletRequest request,
                       final ServletResponse response,
                       FilterChain chain)
       throws IOException, ServletException {
  OutputStream out = response.getOutputStream();
  out.write("<HR>PRE<HR>".getBytes());
  GenericResponseWrapper wrapper = new
         GenericResponseWrapper((HttpServletResponse) response);
  chain.doFilter(request,wrapper);
  out.write(wrapper.getData());
  out.write("<HR>POST<HR>".getBytes());
  out.close();
  }
}
```

Save this code in the following directory and compile it:

```
j2ee/home/default-web-app/WEB-INF/classes/com/acme/filter
```

JSP code: filter3.jsp

As in the previous examples, create a simple JSP page and place it in the root of `default-web-app`:

```
<HTML>
<HEAD>
<TITLE>Filter Example 3</TITLE>
</HEAD>
<BODY>
This is a testpage. You should see<br>
this text when you invoke filter3.jsp, <br>
as well as the additional material added<br>
by the PrePostFilter.
<br>
</BODY>
</HTML
```

Save this JSP code as `j2ee/home/default-web-app/filter3.jsp`.

Configuring the Filter

Add the following `<filter>` element to `web.xml`, after the configuration of the `message` filter:

```
<!-- Filter Example #3 -->
<filter>
  <filter-name>prePost</filter-name>
  <display-name>prePost</display-name>
  <filter-class>com.acme.filter.PrePostFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>prePost</filter-name>
  <url-pattern>/filter3.jsp</url-pattern>
</filter-mapping>
<!-- end Filter Example #3 -->
```

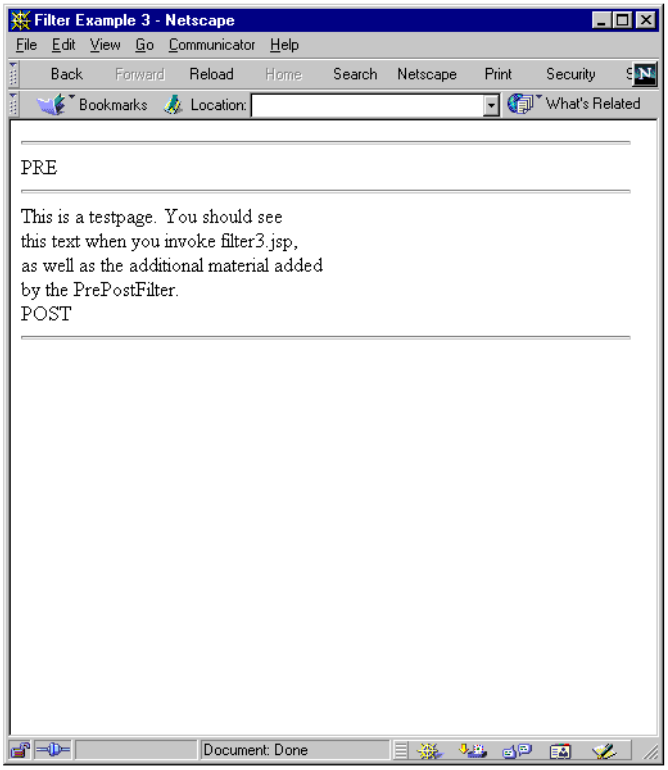
Running Example #3

In your Web browser, enter a URL such as the following:

`http://<hostname>:<port>/j2ee/filter3.jsp`

You should see a page that looks something like what is shown in Figure 4–4.

Figure 4–4 Example3 Output



A

Third Party Licenses

This appendix includes the Third Party License for third party products included with Oracle9*i* Application Server and discussed in this document. Topics include:

- Apache HTTP Server
- Apache JServ

Apache HTTP Server

Under the terms of the Apache license, Oracle is required to provide the following notices. However, the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

The Apache Software License

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
```

```
*    permission of the Apache Software Foundation.
*
* THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND ANY EXPRESSED OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
* USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
* OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
* ====================================================================
*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Software Foundation.  For more
* information on the Apache Software Foundation, please see
* <http://www.apache.org/>.
*
* Portions of this software are based upon public domain software
* originally written at the National Center for Supercomputing Applications,
* University of Illinois, Urbana-Champaign.
*/
```

Apache JServ

Under the terms of the Apache license, Oracle is required to provide the following notices. However, the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

Apache JServ Public License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and / or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).

- The names "Apache JServ", "Apache JServ Servlet Engine" and "Java Apache Project" must not be used to endorse or promote products derived from this software without prior written permission.
- Products derived from this software may not be called "Apache JServ" nor may "Apache" nor "Apache JServ" appear in their names without prior written permission of the Java Apache Project.
- Redistribution of any form whatsoever must retain the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).

THIS SOFTWARE IS PROVIDED BY THE JAVA APACHE PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JAVA

APACHE PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

G

H

I

J

L

M

O

P

R

S

T

U

W