Oracle9*i*AS Containers for J2EE

Servlet Developer’s Guide

Release 2 (9.0.3)

August 2002

Part No. A97680-01

Oracle9*i*AS Containers for J2EE Servlet Developer's Guide, Release 2 (9.0.3)

Part No. A97680-01

Primary Authors: Brian Wright, Tim Smith

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

Contents

Send Us Your Comments

Oracle9*i*AS Containers for J2EE Servlet Developer's Guide, Release 2 (9.0.3)

Part No. A97680-01

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: jpgreader_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), including discussion of basic servlets, data-access servlets, and servlet filters and event listeners. It also provides an overview of OC4J deployment and configuration, with detailed descriptions of key configuration files.

Because this manual is intended for developers, its content is largely targeted toward users of the OC4J standalone development environment; however, there is ample consideration of OC4J within an Oracle9*i*AS production environment.

As for Oracle9*i*AS release 2 (9.0.3), the OC4J servlet container fully complies with the Sun Microsystems *Java Servlet Specification, Version 2.3*.

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 in general, nor does it document the Java Servlet API in detail.

You should be 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).

Because this is a developer's guide, and development and testing are more convenient in an OC4J standalone environment, key aspects of OC4J standalone are discussed, and the assumption is that most developers will be using a standalone environment.

If you are developing applications that primarily use JavaServer Pages, refer to the *Oracle9iAS Containers for J2EE Support for JavaServer Pages Developer's Guide*.

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"

Summarizes servlet technology and servlet development in general, introduces the OC4J servlet container, and provides a simple "Hello World" example.

Chapter 2, "Servlet Development"

Describes how the OC4J servlet container supports servlet development and invocation, including a discussion of key development considerations. 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 and Event Listeners"

Explains the use of filters to affect servlet input or output, and event listeners to track session and application events and manage resources accordingly. These features are new in the Servlet 2.3 specification.

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 Developer's Guide*

  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 standards-based Java services supplied with OC4J, such as JTA, JNDI, JMS, JAAS/JAZN, 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 to Release 2 (9.0.3)*

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 are also of possible 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*

You can obtain the Sun Microsystems *Java Servlet Specification, Version 2.3* at the following location:

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

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 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/`

If you already have a user name 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`

To access the documentation search engine directly, please visit

`http://tahiti.oracle.com`

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 document. 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.
`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]`

`. . .`	Horizontal ellipsis points indicate either:
	■ That we have omitted parts of the code that are not directly related to the example
	■ That you can repeat a portion of the code
		
		`CREATE TABLE ... AS `*`subquery`*`;`
		`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 and deploy 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.

With Oracle9*i*AS release 2 (9.0.3), OC4J complies with the J2EE 1.3 specification, including full servlet 2.3 compliance in the OC4J servlet container.

The most important concepts to understand about servlet development under OC4J are how the Web application is built and how it is deployed. If you are new to servlets, see Chapter 2, "Servlet Development". If OC4J is a new development environment for you, see 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 is organized as follows:

Introduction to Servlets

This section offers a brief introduction to servlet technology, covering the following topics:

- Review of Servlet Technology
- Advantages of Servlets
- The Servlet Interface and Request and Response Objects
- Servlets and the Servlet Container
- Introduction to Servlet Sessions
- Servlet Contexts
- Introduction to Servlet Filters
- Introduction to Event Listeners
- Other J2EE Component Types

Review of Servlet Technology

In recent years, servlet technology has emerged as a powerful way to extend Web server functionality through dynamic HTML pages. A servlet is a Java program that runs in a Web server (as opposed to an applet, which is a Java program that runs in a client browser). The servlet takes an HTTP request from a browser, generates dynamic content (such as by querying a database), and provides an HTTP response back to the browser.

Prior to servlets, Common Gateway Interface (CGI) technology was used for dynamic content, with CGI programs being written in languages such as Perl and being called by a Web application through the Web server. CGI ultimately proved less than ideal, however, due to its architecture and scalability limitations.

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.

A servlet runs in a J2EE application server, such as OC4J. Servlets are one of the main application component types of a J2EE application, along with EJBs, another server-side J2EE component type. These are used in conjunction with client-side components such as applets (part of the Java 2 Standard Edition specification) and application client programs. A Web application might consist of any number of any of these components.

Advantages of Servlets

In the Java realm, servlet technology offers advantages over applet technology for server-intensive applications such as those accessing a database. One advantage is that a servlet runs in the server, which is usually a robust machine with many resources, minimizing use of client resources. An applet, by contrast, is downloaded into the client browser and runs there. Another advantage is more direct access to the data. The Web server or data server in which a servlet is running is on the same side of the network firewall as the data being accessed. An applet running on a client machine, outside the firewall, requires special measures (such as signed applets) to allow the applet to access any server other than the one from which it was downloaded.

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

- 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.
- Servlet technology, in addition to improved scalability, offers the well-known Java advantages of object orientation, platform independence, security, and robustness.
- Servlets are fully integrated with the Java language and its standard APIs, such as JDBC (for Java database connectivity, of particular interest to database programmers).
- Servlets are fully integrated into the J2EE framework, which provides an extensive set of services that your Web application can use, such as JNDI, JMS, and RMI.
- A servlet handles concurrent requests (through either a single servlet instance or multiple servlet instances, depending on the thread model), and servlets have a well-defined lifecycle. For higher performance, servlets can be loaded when OC4J starts. (See "Servlet Preloading" on page 2-4.)
- The servlet request and response objects provide a convenient way to handle HTTP requests and send text and data back to the client.

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.

The Servlet Interface and Request and Response Objects

A Java servlet, by definition, implements the `javax.servlet.Servlet` interface. This interface specifies methods to initialize a servlet, process requests, get the configuration and other basic information of a servlet, and terminate a servlet instance.

For Web applications, you can implement the `Servlet` interface by extending the `javax.servlet.http.HttpServlet` abstract class. (Alternatively, for protocol-independent servlets, you can extend the `javax.servlet.GenericServlet` class.) The `HttpServlet` class includes the following methods:

- `init(...)`—This is to initialize the servlet.
- `destroy(...)`—This is to terminate the servlet.
- `doGet(...)`—This is for HTTP `GET` requests.
- `doPost(...)`—This is for HTTP `POST` requests.
- `doPut(...)`—This is for HTTP `PUT` requests.
- `doDelete(...)`—This is for HTTP `DELETE` requests.
- `service(...)`—This is to receive HTTP requests and, by default, dispatch them to the appropriate `do`*`XXX`*`()` methods.
- `getServletInfo(...)`—This is for use by the servlet to provide information about itself.

A servlet class that extends `HttpServlet` implements some or all of these methods, as appropriate, overriding the original implementations as necessary to process the request and return the response as desired. For example, most servlets override the `doGet()` method or `doPost()` method or both to process HTTP `GET` and `POST` requests.

Each method takes as input an `HttpServletRequest` instance (an instance of a class that implements the `javax.servlet.http.HttpServletRequest` interface) and an `HttpServletResponse` instance (an instance of a class that implements the `javax.servlet.http.HttpServletResponse` interface).

The `HttpServletRequest` instance provides information to the servlet regarding the HTTP request, such as request parameter names and values, the name of the remote host that made the request, and the name of the server that received the request. The `HttpServletResponse` instance provides HTTP-specific functionality in sending the response, such as specifying the content length and MIME type and providing the output stream.

Servlets and the Servlet Container

Unlike a Java client program (but like an applet), a servlet has no static `main()` method. Therefore, a servlet must execute under the control of an external container.

Servlet containers, sometimes referred to as *servlet engines*, execute and manage servlets. It is the servlet container that calls servlet methods and provides services that the servlet needs when executing. A servlet container is usually written in Java and is either part of a Web server (if the Web server is also written in Java) or otherwise associated with and used by a Web server.

The servlet container provides the servlet easy access to properties of the HTTP request, such as its headers and parameters. When a servlet is called (such as when a servlet is specified by URL), the Web server passes the HTTP request to the servlet container. The container, in turn, passes the request to the servlet. In the course of managing a servlet, a servlet container performs the following tasks:

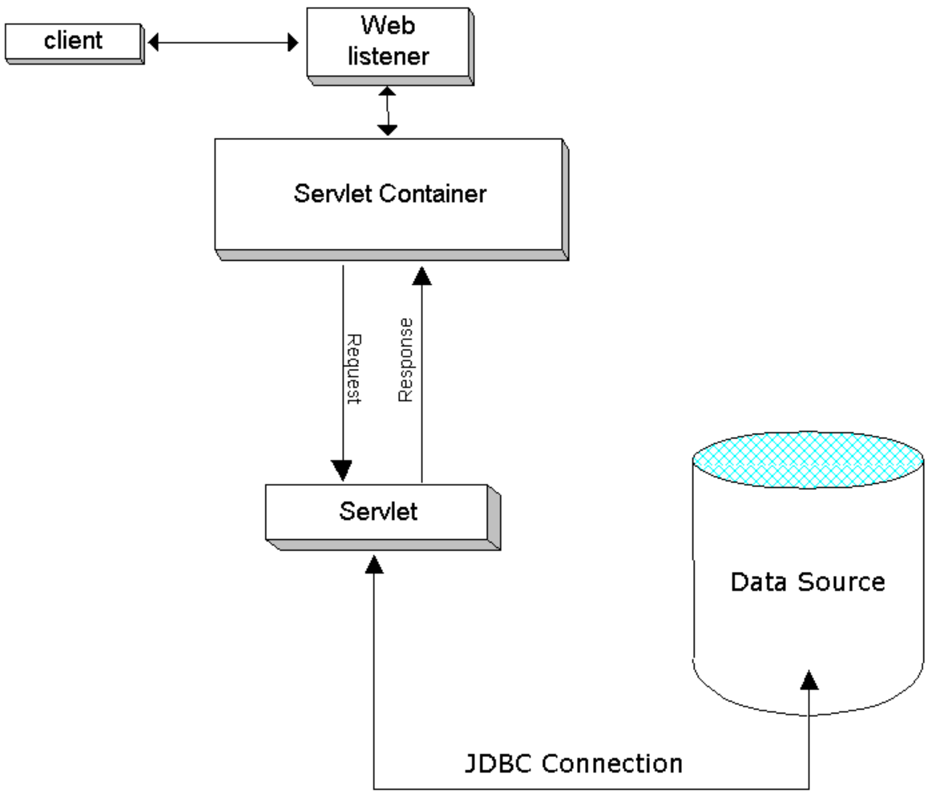
- It creates an instance of the servlet and calls its `init()` method to initialize it.
- It constructs a request object to pass to the servlet. The request includes, among other things:
  - any HTTP headers from the client
  - parameters and values passed from the client (for example, names and values of query strings in the URL)
  - the complete URI of the servlet request
- It constructs a response object for the servlet.
- It 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`).

- It calls the `destroy()` method of the servlet to discard it when appropriate, so that it can be garbage collected. (For performance reasons, it is typical for a servlet container to keep a servlet instance in memory for reuse, rather than destroying it each time it has finished its task. It would be destroyed only for infrequent events, such as Web server shutdown.)

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), 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



Introduction to Servlet Sessions

Servlets use HTTP sessions to keep track of which user each HTTP request comes from, so that a group of requests from a single user can be managed in a stateful way. Servlet session tracking is similar in nature to session tracking in previous technologies, such as CGI.

This section provides an introduction to servlet sessions. See "Servlet Sessions" on page 2-16 for more information and examples.

HttpSession Interface

In the standard servlet API, each user is represented by an instance of a class that implements the `javax.servlet.http.HttpSession` interface. Servlets can set and get information about the session in this `HttpSession` object, which must be of application-level scope.

A servlet uses the `getSession()` method of an `HttpServletRequest` object to retrieve or create an `HttpSession` object for the user. This method takes a boolean argument to specify whether a new session object should be created for the user if one does not already exist.

The `HttpSession` interface specifies the following public methods to get and set session information:

- `void setAttribute(String name, Object value)`

  This method binds the specified object to the session, under the specified name.

- `Object getAttribute(String name)`

  This method retrieves the object that is bound to the session under the specified name (or `null` if there is no match).

Depending on the implementation of the servlet container and the servlet itself, sessions may expire automatically after a set amount of time or may be invalidated explicitly by the servlet. Servlets can manage session lifecycle with the following methods, specified by the `HttpSession` interface:

- `void invalidate()`

  This method immediately invalidates the session, unbinding any objects from it.

- `void setMaxInactiveInterval(int interval)`

  This method sets a session timeout interval, in seconds, as an integer.

- `boolean isNew()`

  This method returns `true` within the request that created the session; it returns `false` otherwise.

- `long getCreationTime()`

  This method returns the time when the session object was created, measured in milliseconds since midnight, January 1, 1970.

- `long getLastAccessedTime()`

  This method returns the time of the last request associated with the client, measured in milliseconds since midnight, January 1, 1970.

For complete information about `HttpSession` methods, you can refer to the Sun Microsystems Javadoc at the following location:

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

Introduction to 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.

The following session-tracking mechanisms are supported. See "Session Tracking" on page 2-16 for more information.

- cookies

  The servlet container sends a cookie to the client, which returns the cookie to the server upon each HTTP request. This associates the request with the session ID indicated by the cookie. This is the most frequently used mechanism and is supported by any servlet container that adheres to the servlet 2.2 or higher specification.

- URL rewriting

  In case cookies might be disabled, the servlet can call the `encodeURL()` method of the response object, or the `encodeRedirectURL()` method for redirects, to append a session ID to the URL path for each request. This allows the request to be associated with the session. This is the most frequently used mechanism where clients do not accept cookies.

Servlet Contexts

A *servlet context* is used to maintain state information for all instances of a Web application within any single JVM (that is, for all servlet and JSP page instances that are part of the Web application). This is similar to the way a session maintains state information for a single client on the server; however, a servlet context is not specific to any single user and can handle multiple clients. There is usually one servlet context for each Web application running within a given JVM. You can think of a servlet context as an "application container".

Any servlet context is an instance of a class that implements the `javax.servlet.ServletContext` interface, with such a class being provided with any Web server that supports servlets.

A `ServletContext` object provides information about the servlet environment (such as name of the server) and allows sharing of resources between servlets in the group, within any single JVM. (For servlet containers supporting multiple simultaneous JVMs, implementation of resource-sharing varies.)

A servlet context maintains the session objects of the users who are running the application, and provides a scope for the running instances of the application. Through this mechanism, each application is loaded from a distinct class loader and its runtime objects are distinct from those of any other application. In particular, the `ServletContext` object is distinct for an application, much as each `HttpSession` object is distinct for each user of the application.

Beginning with the Sun Microsystems *Java Servlet Specification, Version 2.2,* most implementations can provide multiple servlet contexts within a single host, which is what allows each Web application to have its own servlet context. (Previous implementations usually provided only a single servlet context with any given host.)

The `ServletContext` interface specifies methods that allow a servlet to communicate with the servlet container that runs it, which is one of the ways that the servlet can retrieve application-level environment and state information. Methods specified in `ServletContext` include those listed here. For complete information, you can refer to the Sun Microsystems Javadoc at the following location:

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

- `void setAttribute(String name, Object value)`

  This method binds the specified object to the specified attribute name in the servlet context. Using attributes, a servlet container can give information to the servlet that is not otherwise provided through the `ServletContext` interface.

  > **Note:** For a servlet context, `setAttribute()` is a local operation only—it is not intended to be distributed to other JVMs within a cluster. (This is in accordance with the servlet 2.3 specification.)

- `Object getAttribute(String name)`

  This method returns the attribute with the given name, or `null` if there is no attribute by that name. The attribute is returned as a `java.lang.Object` instance.

- `java.util.Enumeration getAttributeNames()`

  This method returns a `java.util.Enumeration` instance containing the names of all available attributes of the servlet context.

- `void removeAttribute(String attrname)`

  This method removes the specified attribute from the servlet context.

- `String getInitParameter(String name)`

  This method returns a string that indicates the value of the specified context-wide initialization parameter, or `null` if there is no parameter by that name. This allows access to configuration information that is useful to the Web application associated with this servlet context (the name of a system that has critical information, for example).

- `Enumeration getInitParameterNames()`

  This method returns a `java.util.Enumeration` instance containing the names of the initialization parameters of the servlet context.

- `RequestDispatcher getNamedDispatcher(String name)`

  This returns a `javax.servlet.RequestDispatcher` instance that acts as a wrapper for the specified servlet.

- `RequestDispatcher getRequestDispatcher(String path)`

  This returns a `javax.servlet.RequestDispatcher` instance that acts as a wrapper for the resource located at the specified path.

- `String getRealPath(String path)`

  This returns the real path, as a string, for the specified virtual path.

- `URL getResource(String path)`

  This returns a `java.net.URL` instance with a URL to the resource that is mapped to the specified path.

- `String getServerInfo()`

  This method returns the name and version of the servlet container.

- `String getServletContextName()`

  This returns the name of the Web application with which the servlet context is associated, according to the `<display-name>` element of the `web.xml` file.

Introduction to Servlet Filters

Request objects (instances of a class that implements `HttpServletRequest`) and response objects (instances of a class that implements `HttpServletResponse`) are typically passed directly between the servlet container and a servlet.

The servlet 2.3 specification, however, allows *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.

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 "Servlet Filters" on page 4-2 for more information.

Introduction to Event Listeners

The servlet 2.3 specification adds the capability to track key events in your Web applications, through *event listeners*. This functionality allows more efficient resource management and automated processing based on event status.

When creating listener classes, there are standard interfaces you can implement for servlet context lifecycle events, servlet context attribute changes, HTTP session lifecycle events, and HTTP session attribute changes. A listener class can implement one, some, or all of the interfaces as appropriate.

An event listener class is declared in the `web.xml` deployment descriptor and invoked and registered upon application startup. When an event occurs, the servlet container calls the appropriate listener method.

See "Event Listeners" on page 4-16 for more information.

Other J2EE Component Types

In addition to servlets, a Web application might include other server-side components such as JavaServer Pages (JSP) and Enterprise JavaBeans (EJB).

While servlets are managed by the OC4J servlet container, EJBs are managed by the OC4J EJB container and JSP pages are managed by the OC4J JSP container. These containers form the core of OC4J.

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.

For more information about JSP pages and EJBs, see the following:

- JSP and EJB primer chapters in the *Oracle9iAS Containers for J2EE User's Guide*
- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Developer's Guide*
- *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*

A First Servlet Example

Looking at a basic example is the best way to demonstrate the general framework for writing a servlet. 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

This servlet 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 `/WEB-INF/classes` directory of the OC4J default Web application. (See "OC4J Default Web Application and Key Directories" on page 2-9.) 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. This is enabled through the setting `development="true"` in the `<orion-web-app>` element of the `global-web-application.xml` file in the OC4J configuration files directory. The `source-directory` attribute may also have to be set 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.

See "Element Descriptions for global-web-application.xml and orion-web.xml" on page 3-16 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>/servlet/HelloWorldServlet
```

If you are developing in an OC4J standalone environment, use port 8888 to access the OC4J Web listener directly.

This example assumes that "`/`" is the context path of the Web application, as is true by default in OC4J standalone for an application deployed to the default Web application.

See "Servlet Invocation" on page 2-11 for general information about invoking servlets in OC4J in various situations and environments.

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 and Key Considerations
- Servlet Invocation
- Servlet Sessions
- Use of JDBC in Servlets
- EJB Calls from Servlets

Note: The assumption in this chapter is that you are in an OC4J standalone development environment. For considerations in configuring and deploying a production application with Oracle Enterprise Manager in an Oracle9*i*AS environment, see "OC4J Deployment and Configuration with Oracle9iAS and Enterprise Manager" on page 3-3.

Servlet Development Basics and Key Considerations

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 for initialization work at the time the servlet is loaded by the container, or for finalization work 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 objects as parameters.

This chapter provides sample servlets that are more advanced than the `HelloWorldServlet` in "A First Servlet Example" on page 1-13. For simplicity, you can test each of these servlets using the OC4J standalone default Web application. To do this, save the Java source files in the `/WEB-INF/classes` directory of the default Web application.

To test some of the servlets, you might have to make changes to the `web.xml` file in the default Web application `/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 section covers features and issues to consider before developing your applications. In all, the following topics are covered:

- Code Template
- Servlet Loading and Lifecycle
- Servlet Preloading
- Servlet Class Loading
- Servlet Information Exchange
- Servlet Threading
- Servlet Security Considerations
- OC4J Default Web Application and Key Directories
- Oracle9i JDeveloper Support for Servlet Development

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()`.

The reason for overriding the `init()` method would be to perform special actions that are required only once in the servlet lifetime, such as the following:

- 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

Servlet Loading and Lifecycle

Servlets have a predictable and manageable lifecycle:

- 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 on 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 "Servlet Filters" on page 4-2 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.

Also see the next section, "Servlet Preloading".

Servlet Preloading

Typically, the servlet container instantiates and loads a servlet class when it is first requested. However, you can arrange the preloading of servlets through settings in the Web site XML file (such as `default-web-site.xml` or `http-web-site.xml`) and the `web.xml` file. Preloaded servlets are loaded and initialized when the OC4J server starts up, or when the Web module is deployed or redeployed.

Preloading requires the following steps:

1. Specify the attribute setting `load-on-startup="true"` in the `<web-app>` subelement of the `<web-site>` element of the Web site XML file. See "The default-web-site.xml, http-web-site.xml, and Other Web Site XML Files" on page 3-30 for information about Web site XML files.

2. For any servlet you want to preload, there must be a `<load-on-startup>` subelement under the `<servlet>` element in the `web.xml` file for the Web module.

Table 2–1 explains the behavior of the `<load-on-startup>` element in `web.xml`.

Table 2–1 File web.xml <load-on-startup> Behavior

Value Range	Behavior
Less than zero (<0) For example: `<load-on-startup>-1</load-on-startup)`	Servlet is *not* preloaded.
Greater than or equal to zero (>=0) For example: `<load-on-startup>1</load-on-startup>`	Servlet is preloaded. The order of its loading, with respect to other preloaded servlets in the same Web application, is according to the load-on-startup value, lowest number first. (For example, 0 is loaded before 1, which is loaded before 2.)
Empty element For example: `<load-on-startup/>`	The behavior is as if the load-on-startup value is `Integer.MAX_VALUE`, ensuring that the servlet is loaded after any servlets with load-on-startup values greater than or equal to zero.

Servlet Class Loading

Regarding class loading for servlets, there are two important considerations in OC4J:

- whether to load WAR file classes before system classes
- how to load classes so that cached Java objects can be shared across servlets

Loading WAR File Classes Before System Classes in OC4J

The servlet 2.3 specification recommends, but does not require, loading "local classes"—classes in the WAR file—before system classes. By default, the OC4J servlet container does *not* load local classes first, but this is configurable through the `<web-app-class-loader>` element in `global-web-application.xml` or `orion-web.xml`. This element has two attributes, listed here.

- `search-local-classes-first`: Set this to "`true`" to search and load WAR file classes before system classes. The default is "`false`".
- `include-war-manifest-class-path`: Set this to "`false`" to *not* include the class path specified in the WAR file manifest `Class-Path` attribute when searching and loading classes from the WAR file (regardless of the `search-local-classes-first` setting). The default is "`true`".

Also see "Element Descriptions for global-web-application.xml and orion-web.xml" on page 3-16.

Sharing Cached Java Objects Across OC4J Servlets

In order to take advantage of the distributed functionality of the Oracle9*i* Application Server Java Object Cache, or to share a cached object between servlets, some minor modification to an application deployment is necessary. Any user-defined objects that will be shared between servlets or distributed between JVMs must be loaded by the system class loader; however, by default, objects loaded by a servlet are loaded by the context class loader. Objects loaded by the context class loader are visible only to the servlets within the servlet context corresponding to that class loader. The object definition would not be available to other servlets or to the cache in another JVM. If an object is loaded by the system class loader, however, the object definition will be available to other servlets and to the cache on other JVMs.

With OC4J, the system classpath is derived from the manifest of the `oc4j.jar` file and any associated `.jar` files, including `cache.jar`. The classpath in the environment is ignored. To include a cached object in the classpath for OC4J, the `.class` file should either be copied to the following directory:

```
[Oracle_Home]/javacache/sharedobjects/classes
```

or it should be added to the following `.jar` file:

```
[Oracle_Home]/javacache/cachedobjects/share.jar
```

Both the `classes` directory and the `share.jar` file are included in the manifest for `cache.jar`, and are therefore included in the system classpath.

For information about the Oracle9*i* Application Server Java Object Cache, see the *Oracle9iAS Containers for J2EE Services Guide*.

Servlet Information Exchange

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

- parameters from the request object
- the HTTP session object
- the servlet context 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.

Servlet Threading

If there is an additional servlet request while a servlet is already running, servlet container behavior depends on whether the servlet uses a single-thread model or a multiple-thread model. In a single-thread case, the servlet container prevents multiple simultaneous `service()` calls from being dispatched to a single servlet instance—it may spawn multiple separate servlet instances instead. In a multiple-thread model, the container can make multiple simultaneous `service()` calls to a single servlet instance, using a separate thread for each call.

When a servlet can be invoked from more than one thread, you must ensure that the servlet code is thread-safe. One option, though not generally advisable, is to synchronize the servlet (specifically, the `service()` method). This can significantly reduce performance.

As an alternative, the servlet specification provides that a servlet can implement the `javax.servlet.SingleThreadModel` interface to guarantee synchronized access to the whole servlet. For stateful servlets, it is advisable to use this single-thread model.

Servlet Security Considerations

There are several considerations regarding the security of your Web application running in the OC4J servlet container:

- OC4J includes standard support for security constraints and security roles through the `<security-role>` element of the `web.xml` deployment descriptor. For general information, refer to the Sun Microsystems *Java Servlet Specification, Version 2.3*. OC4J also offers related support through the `global-web-application.xml` file `<security-role-mapping>` element. See "The global-web-application.xml and orion-web.xml Files" on page 3-14.

- Secure Socket Layer (SSL) and HTTPS support are through Oracle HTTP Server, not OC4J.

  In Oracle9*i*AS release 2, the OC4J servlet container does not directly support SSL and HTTPS. This affects some `default-web-site.xml` elements and attributes, specifically:

  – `<web-site>` element `secure` and `protocol` attributes—Some settings are unsupported or not recommended.

  – `<web-site>` element `<ssl-config>` subelement—This is unsupported.

  See "The default-web-site.xml, http-web-site.xml, and Other Web Site XML Files" on page 3-30 for information about these elements and attributes.

- There is a significant security risk when users are allowed to invoke servlets by class name. (Invoking by class is described in "Servlet Invocation by Class Name During OC4J Development" on page 2-11.)

  Invoking by class name bypasses security constraints specified in the `web.xml` file, and should be considered only in a development environment. It is also true that when a servlet is invoked by class, any exception it throws may reveal the physical path of the servlet location, which is highly undesirable.

  To resolve this problem, particularly in a production environment, configure the application so that the `servlet-webdir` attribute of the `<orion-web-app>` element of the `global-web-application.xml` file does *not* start with a slash ("`/`"). This effectively disables the dynamic servlet class name lookup. Note that the default setting, "`/servlet`", *does* start with a slash.

  The following configuration, for example, would remedy the situation:

  ```
  <orion-web-app ... servlet-webdir="null" ... >
     ...
  </orion-web-app>
  ```

- To guard against having session ID numbers guessed or "hacked" for possibly destructive purposes, OC4J uses `java.security.SecureRandom` functionality to generate random session ID numbers.

OC4J Default Web Application and Key Directories

OC4J is installed with a default configuration that includes a default Web site and a *default application*. The default application includes a *default Web application*. Therefore, you can start OC4J immediately without any additional configuration.

For simplicity, this chapter assumes you are using an OC4J standalone environment and deploying to the default Web application. An OC4J standalone environment includes the following key directories.

- J2EE home: `[Oracle_Home]/j2ee/home`

  This will be referred to as `j2ee/home` in the remainder of this document.
- configuration files directory: `j2ee/home/config`

In addition, there are the following key directories of the OC4J standalone default Web application:

- default Web application root: `j2ee/home/default-web-app`
- default Web application classes: `j2ee/home/default-web-app/WEB-INF/classes`

Note: In an Oracle9*i*AS environment, directories are configurable using Enterprise Manager. See the *Oracle9iAS Containers for J2EE User's Guide*.

Oracle9*i* JDeveloper Support for Servlet Development

Visual Java programming tools now typically support servlet coding. In particular, Oracle9*i* JDeveloper supports servlet development and includes the following features:

- wizards to help generate servlet code
- integration of the OC4J servlet container to support the full application development cycle—editing, debugging, and running servlets
- debugging of deployed servlets
- an extensive set of data-enabled and Web-enabled JavaBeans, known as JDeveloper Web beans
- the JSP Element Wizard, which offers a convenient way to add predefined Web beans to a page

- support for incorporating custom JavaBeans
- a deployment option for servlet applications that rely on the JDeveloper Business Components for Java (BC4J)

Servlet Invocation

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, up to 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 by class name in a development scenario:

- Servlet Invocation by Class Name During OC4J Development
- Servlet Invocation in an Oracle9iAS Production Environment
- Servlet Invocation in an OC4J Standalone Environment

Servlet Invocation by Class Name During OC4J Development

For a development or testing scenario in OC4J, there is a mechanism for invoking a servlet by class name. This might simplify the URL for invocation. The presumption here is that you would use this in an OC4J standalone environment while developing your application.

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. Also assume a context path of "`/`" (the default for the default Web application in OC4J standalone).

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

`http://`	The network protocol. Other protocols are `ormi`, `ftp`, `https`, and so on.
`<`*`host`*`>`	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 (as defined in `/etc/hosts` on a UNIX system, for example).
`<:`*`port`*`>`	The port that the Web server listens on. (If you do not specify a port, most browsers assume port 80.) For AJP protocol through Oracle HTTP Server, the server port is defined in the `port` attribute of the `<web-site>` element in `default-web-site.xml` (or other OC4J Web site XML file, as appropriate). For HTTP protocol in OC4J standalone, the port is according to the `port` attribute of the `<web-site>` element in `http-web-site.xml`.
`/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://<host><:port>/foo/servlet/foo.bar.SessionServlet
```

Important: Allowing the invocation of servlets by class name presents a significant security risk; OC4J should not be configured to operate in this mode in a production environment. See "Servlet Security Considerations" on page 2-7 for information.

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.

Servlet Invocation in an Oracle9*i*AS Production Environment

In a production environment, using the `servlet-webdir` attribute in `global-web-application.xml` or `orion-web.xml` is inadvisable for security reasons. Instead, standard servlet settings and mappings in the application `web.xml` file 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.

In an Oracle9*i*AS environment, the context path is "`/j2ee`" to use AJP protocol through the Oracle HTTP Server for an application that is deployed to the OC4J default Web application. Here is an example:

```
http://<host><:port>/j2ee/MyServlet
```

Use port 7777 for access through the Oracle HTTP Server with Oracle9*i*AS Web Cache enabled.

If you are not using the default Web application, specify the context path while deploying the application, in the Enterprise Manager page where you are prompted for "URL binding". See the *Oracle9iAS Containers for J2EE User's Guide* for more information. That document also has information about OC4J port settings and other default settings. For general information about Enterprise Manager, see the *Oracle Enterprise Manager Administrator's Guide*.

Notes:

- This discussion assumes the Web application is bound to a Web site that uses AJP protocol, according to settings in the `default-web-site.xml` file.
- For production applications, Oracle recommends that you always use the Oracle HTTP Server, as discussed in this section. See the *Oracle HTTP Server Administration Guide* for general information.

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 name 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 Invocation in an OC4J Standalone Environment

The information about servlet configuration through `web.xml` in the previous section, "Servlet Invocation in an Oracle9iAS Production Environment", also applies in an OC4J standalone environment if you are not invoking by class name.

When testing with OC4J standalone, you can use port 8888 to access OC4J through its own Web listener. The default context path is "`/`". Here is an example:

```
http://<host>:8888/MyServlet
```

Note: The OC4J standalone Web site uses HTTP protocol without going through the Oracle HTTP Server and AJP, and is configured according to settings in the `http-web-site.xml` file . These settings include the context path, for example (according to the `root` attribute of the `<web-app>` subelement under the `<web-site>` element).

Servlet Sessions

Servlet sessions were introduced in "Introduction to Servlet Sessions" on page 1-7. This section provides details and examples, covering the following topics:

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

Session Tracking

This section provides an overview of servlet session tracking and features, then describes the OC4J implementation.

Overview of 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. The client decides to buy some of the items offered for sale at the Web site, and clicks the **BUY ITEM** buttons. 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.

When a servlet creates an HTTP session object (through the request object `getSession()` method), the client interaction is considered to be stateful.

Session tracking involves identifying user sessions by ID numbers and tying requests to their session through use of the ID number. The typical mechanisms for this are cookies or URL rewriting.

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.

Cookies

A number of approaches have been used in attempting to add a measure of statefulness to the HTTP protocol. The most widely accepted is the use of cookies, used to transmit an identifier between server and client, in conjunction with stateful servlets that can maintain session objects. Session objects are simply dictionaries that store values (Java objects) together with their associated keys (Java strings).

Cookie usage is as follows:

1. With the first response from a stateful servlet after a session is created, the server (container) sends a cookie with a session identifier back to the client, often along with a small amount of other useful information (all less than 4 KB). The container sends the cookie, named `JSESSIONID`, in the HTTP response header.
2. Upon each subsequent request from the same Web client session, if the client supports cookies it sends the cookie back to the server as part of the request, and the cookie value is used by the server to look up session state information to pass to the servlet.
3. With subsequent responses, the container sends the updated cookie back to the client.

The servlet code is not required to do anything to send a cookie; this is handled by the container. Sending cookies back to the server is handled automatically by the Web browser, unless the end-user disables cookies.

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

An alternative to using cookies is URL rewriting, through the `encodeURL()` method of the response object. This is where the session ID is encoded into the URL path of a request. See "Session Servlet Example" on page 2-20 for an example of URL rewriting.

The name of the path parameter is `jsessionid`, as in the following example:

```
http://<host><:port>/myapp/index.html?jsessionid=6789
```

Similarly to the functionality of cookies, the value of the rewritten URL is used by the server to look up session state information to pass to the servlet.

Although cookies are typically enabled, the only way for you to ensure session tracking is to use `encodeURL()` in your servlets, or `encodeRedirectURL()` for redirects.

Note: In the OC4J implementation, calls to the `encodeURL()` and `encodeRedirectURL()` methods will result in no action if cookies are enabled.

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 these techniques in OC4J applications, because they have many drawbacks, including performance penalties and loss of confidentiality.

Session Tracking in OC4J

For session-tracking in OC4J, the servlet container will first attempt to accomplish tracking through cookies. If cookies are disabled, session tracking can be maintained only by using the `encodeURL()` method of the response object, or the `encodeRedirectURL()` method for redirects. You must include the `encodeURL()` or `encodeRedirectURL()` calls in your servlet if cookies might be disabled.

The use of session cookies is disabled by the following setting in the `global-web-application.xml` or `orion-web.xml` file:

```
<session-tracking cookies="disabled" ... >
   ...
</session-tracking>
```

Cookies are enabled by default.

Notes:

- OC4J does not support *auto-encoding*, where session IDs are automatically encoded into the URL by the servlet container. This is a non-standard and expensive process.
- An `encodeURL()` or `encodeRedirectURL()` call will *not* encode the session ID into the URL if the cookie mechanism is found to be working properly.
- The `encodeURL()` method replaces the servlet 2.0 `encodeUrl()` method (note capitalization), which is deprecated.

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 under 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>
```

The `<session-timeout>` element takes integer values. According to the servlet 2.3 specification, a value of 0 (zero) or less specifies the default behavior that a session never times out. For example:

```
<session-config>
  <session-timeout>-1</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

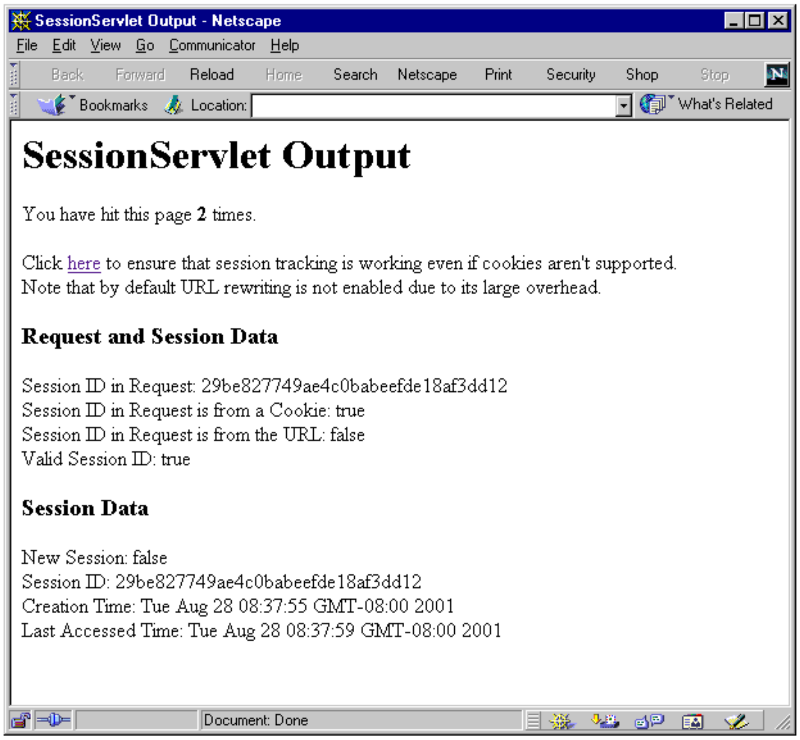
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;

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 `SessionServlet.java` in the OC4J default Web application `/WEB-INF/classes` directory. If you use the setting `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 3-16 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 select the HREF that causes URL rewriting.

Figure 2–1 Session Servlet Display


SessionServlet Output - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: What's Related
Document: Done

SessionServlet Output

You have hit this page **2** times.

Click here to ensure that session tracking is working even if cookies aren't supported.
Note that by default URL rewriting is not enabled due to its large overhead.

Request and Session Data

Session ID in Request: 29be827749ae4c0babeefde18af3dd12
Session ID in Request is from a Cookie: true
Session ID in Request is from the URL: false
Valid Session ID: true

Session Data

New Session: false
Session ID: 29be827749ae4c0babeefde18af3dd12
Creation Time: Tue Aug 28 08:37:55 GMT-08:00 2001
Last Accessed Time: Tue Aug 28 08:37:59 GMT-08:00 2001

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, copy 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 root directory of the OC4J default Web application.

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 consists of 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 `data-sources.xml` file in the OC4J configuration files directory:

```
<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 release 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 document root of the OC4J default Web application, and save the Java servlet in the `/WEB-INF/classes` directory of the default Web application. The `GetEmpInfo.java` file is automatically compiled when the servlet is invoked by the form.

To test the example directly through the OC4J listener, such as in OC4J standalone, invoke the `EmpInfo.html` page from a Web browser as follows:

`http://<host>:8888/EmpInfo.html`

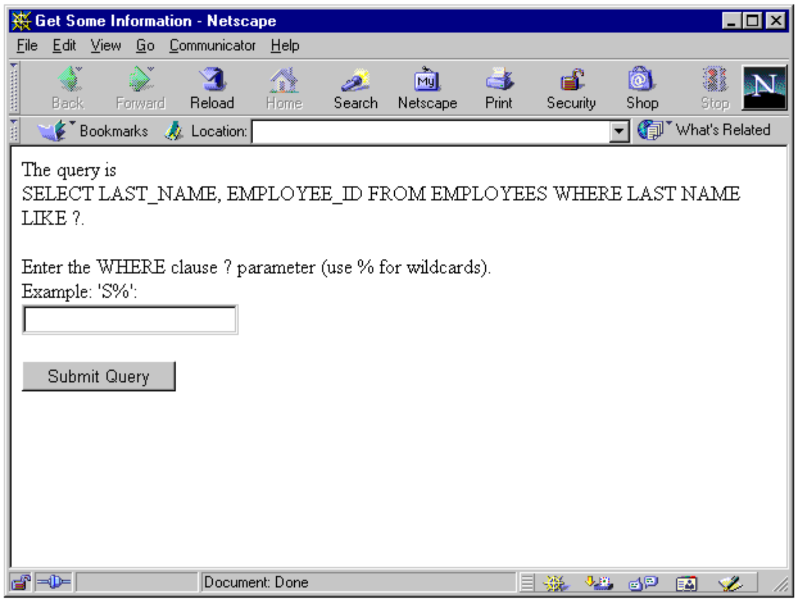
This assumes "`/`" is the context path of the OC4J standalone default Web application.

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

Note: For general information about invoking servlets in OC4J, see "Servlet Invocation" on page 2-11.

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

Figure 2–2 Employee Information Query

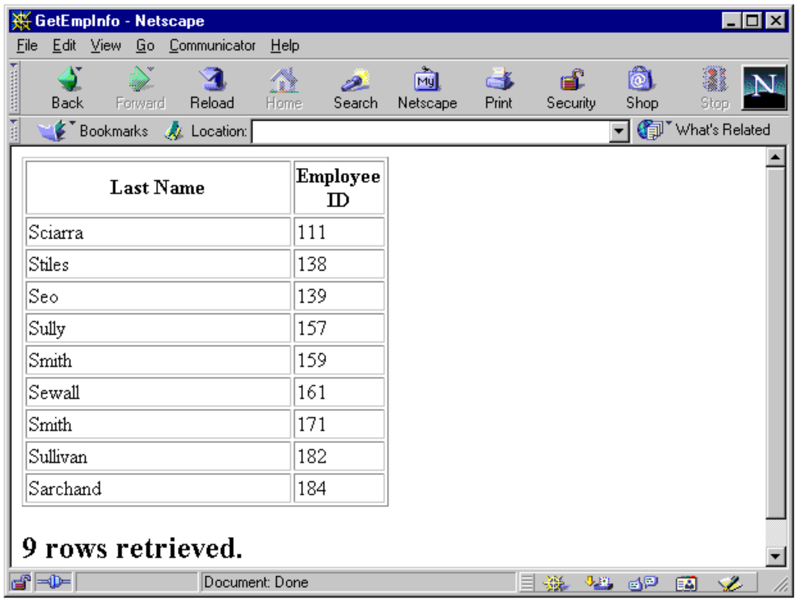

Get Some Information - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:
What's Related
Submit Query
Document: Done

The query is
SELECT LAST_NAME, EMPLOYEE_ID FROM EMPLOYEES WHERE LAST NAME LIKE ?.

Enter the WHERE clause ? parameter (use % for wildcards).
Example: 'S%':

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


GetEmpInfo - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: What's Related
Document: Done

Last Name	Employee ID
Sciarra	111
Stiles	138
Seo	139
Sully	157
Smith	159
Sewall	161
Smith	171
Sullivan	182
Sarchand	184

9 rows retrieved.

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 general 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 ensuring the definition of a 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 2-38.
- 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 2-39.

Notes:

- The EJB material that follows is from an EJB 1.1 perspective; however, there is a brief introduction to EJB 2.0 local interfaces in "EJB 2.0 Local Interfaces" on page 2-41. For more information about EJB 2.0 features, and EJBs in general, refer to the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*.
- For additional servlet-EJB examples, see the demo programs that come with the OC4J distribution.
- OC4J provides an EJB tag library. See the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for information.

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:

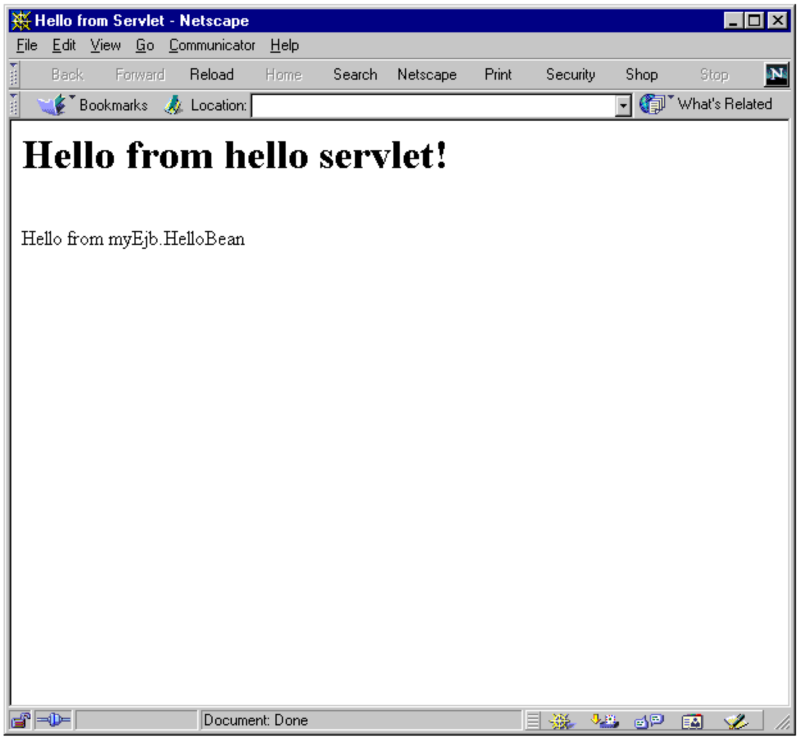
1. Imports the EJB package for access to the bean home and remote interfaces.
2. Prints a message from the servlet.
3. Creates an output string, with an error default.
4. Uses JNDI to look up the EJB home interface.
5. Creates the EJB remote object from the home.
6. Invokes the `helloWorld()` method on the remote object, which returns a `String` object.
7. Prints 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. 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


Hello from Servlet - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:
What's Related
Document: Done

Hello from hello servlet!

Hello from myEjb.HelloBean

EJB Code: HelloBean Stateful Session Bean

The EJB, as shown here, 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, and application-level descriptor.

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) must be 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>
```

However, presuming you are developing in an OC4J standalone environment (where OC4J runs apart from Oracle9*i*AS), the `<context-root>` element is ignored. You must specify the context root through appropriate entries in `http-web-site.xml` or the relevant Web site XML file.

(In an Oracle9*i*AS production environment, for a new context root—such as `/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. Assuming you use Enterprise Manager 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.)

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 Configuration To perform the application deployment for testing purposes, add the following entry to the `server.xml` file in the OC4J configuration files directory (specifying the appropriate path information).

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

In an OC4J standalone development environment, you can accomplish this through the `admin.jar` tool. That tool is documented in the *Oracle9iAS Containers for J2EE User's Guide*, OC4J standalone version.

Then bind the Web module to a Web site. You will need the following entry in the Web site XML file in the OC4J configuration files directory:

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

Notes:

- In an Oracle9*i*AS production environment, use Enterprise Manager for deployment and configuration.
- See "Introduction to Web Application Deployment and Configuration" on page 3-2 for additional information and important considerations.

Remote EJB Lookup within the Same Application

To perform a remote EJB lookup in OC4J, 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 2-31:

```
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.

Remote servers are configured in `rmi.xml` in `<server>` elements, through 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

This section discusses servlet coding steps to look up an EJB from a different application (deployed to a different OC4J instance). This functionality uses RMI protocol, and you must specify an appropriate initial context factory for the lookup. As of the OC4J 9.0.3 implementation, you can use RMI over either ORMI or IIOP to look up an EJB that is in a different application.

The `remote` flag discussed in the preceding section, "Remote EJB Lookup within the Same Application", is not relevant—the lookup is according to the ORMI or IIOP URL. If the host and port are the same as for the calling servlet, then the lookup is local; otherwise, the lookup is remote.

EJB Lookup Outside the Application over ORMI

Here are the servlet coding steps for EJB lookup outside the application (to a different OC4J instance) using RMI over ORMI. Assume that the EJB remote and home interfaces are packaged in the application WAR file.

1. Populate the environment entries for the lookup. These are static fields of the `javax.naming.Context` interface, which is implemented by the `javax.naming.InitialContext` class.

   ```
   ...
   Hashtable env = new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY,
                   "com.evermind.server.rmi.RMIInitialContextFactory");
   env.put(Context.SECURITY_PRINCIPAL, "admin");
   ```

```
env.put(Context.SECURITY_CREDENTIALS, "debu");
env.put(Context.PROVIDER_URL, "ormi://renmis/ejbsamples");
...
```

– The `INITIAL_CONTEXT_FACTORY` setting specifies the initial context factory to use. This example specifies the OC4J initial context factory for ORMI.

– The `SECURITY_PRINCIPAL` setting specifies the identity of the principal for authenticating the caller to the service.

– The `SECURITY_CREDENTIALS` setting specifies the credentials of the principal for authenticating the caller to the service.

– The `PROVIDER_URL` setting specifies the URL for the lookup.

Notes:

- If you omit the host and port in the ORMI (or IIOP) URL, the host is assumed to be `localhost` and a local lookup is performed.
- In situations such as in this example, where an ORMI or IIOP initial context factory is specified, EJB references declared in `web.xml` cannot be looked up.
- If no initial context factory is specified, then `ApplicationInitialContextFactory` is used by default. This is appropriate when you want to use the context of the application for lookup. EJB references *are* looked up in `web.xml` in this case.

2. Look up the remote EJB and create the bean instance.

```
...
Context ctx = new InitialContext(env);
Object obj = ctx.lookup("MyCart");
home = (CartHome)PortableRemoteObject.narrow(obj,CartHome.class);
cart = (Cart)PortableRemoteObject.narrow(home.create(), Cart.class);
...
```

The EJB must be defined with the name `MyCart` in the `ejb-jar.xml` file of the application where it is being looked up.

EJB Lookup Outside the Application over IIOP

To use RMI over IIOP instead of over ORMI, you must specify the OC4J IIOP initial context factory, `IIOPInitialContextFactory`, as in the following example:

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.oracle.iiop.server.IIOPInitialContextFactory");
```

In this case, any principal (user name) and credentials (password) specified through the environment entries are not used. Instead, assuming Common Secure Interoperability version 2 (CSIv2) is enabled, the available credentials are taken from the current "subject" (a standard JAAS concept) as follows:

- If private credentials are available from the subject, then they are used to authenticate.
- Otherwise, if public credentials are available from the subject, then the identity of the originating client is delegated. This also depends on the configured CSIv2 policies of the bean being invoked.

You would also have to specify a `PROVIDER_URL` setting that is appropriate for IIOP, using a corbaname URL scheme. Given that IIOP is the default when using a corbaname, the following `PROVIDER_URL` setting for IIOP would be equivalent to the `PROVIDER_URL` setting given earlier for ORMI (`ormi://renmis/ejbsamples`):

```
corbaname::localhost:port#/renmis/ejbsamples
```

(The "#" is part of the syntax.)

See the *Oracle9iAS Containers for J2EE Services Guide* for additional information about IIOP and related syntax.

Note: CSIv2 is required by J2EE 1.3 and used by IIOP.

EJB 2.0 Local Interfaces

In the EJB 1.1 specification, an EJB (except for message-driven beans) has a remote interface that extends `javax.ejb.EJBObject` and a home interface that extends `javax.ejb.EJBHome`. In this model, all EJBs are defined as remote objects, adding unnecessary overhead to EJB calls in situations where the EJBs are actually local (packaged in the same application, for example).

The EJB 2.0 specification adds support for local connections through *local interfaces*. An EJB is classified as local if it implements the local versions of the remote and home interfaces—`javax.ejb.EJBLocalObject` and `javax.ejb.EJBLocalHome`, respectively.

For a complete example of the use of local interfaces, see the OC4J How-To document at the following location:

```
http://otn.oracle.com/tech/java/oc4j/htdocs/how-to-ejb-local-interfaces.html
```

For additional information, refer to the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*.

3 Deployment and Configuration

This chapter provides an overview of OC4J deployment and describes how to assemble and configure a Web application in OC4J. It covers the following topics:

- Introduction to Web Application Deployment and Configuration
- Application Assembly
- Configuration File Descriptions
- Web Module Configuration in Oracle Enterprise Manager

Introduction to Web Application Deployment and Configuration

Because this is a developer's guide, much of it targets an OC4J standalone user. In a standalone environment, you can use the `admin.jar` tool for configuration, and optionally deploy to the OC4J default Web application for simplicity.

In a production environment, OC4J is installed within Oracle9*i*AS with the goal of managing J2EE enterprise systems. Oracle9*i*AS can manage multiple clustered OC4J processes and is managed and configured through the Oracle Enterprise Manager. Through Enterprise Manager, you can manage and configure your OC4J processes across multiple application server instances and hosts. Thus, you cannot locally manage your OC4J process by using the `admin.jar` tool or by hand-editing the configuration files. This undermines the management provided by Enterprise Manager.

This section discusses each of these scenarios a little further, and provides an overview OC4J configuration files, organized as follows:

- Deployment and Configuration with OC4J Standalone
- OC4J Deployment and Configuration with Oracle9iAS and Enterprise Manager
- Overview of Configuration Files

Note: Users of previous Oracle9*i*AS releases, can refer to *Oracle9i Application Server Migrating to Release 2 (9.0.3)* for information about issues in migrating to Oracle9*i*AS release 2.

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

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

Deployment and Configuration with OC4J Standalone

For application development, or perhaps for a relatively simple Web solution, you can use a single OC4J instance—known as *OC4J standalone*—outside of the Oracle9*i*AS environment. To accomplish this, download the `oc4j_extended.zip` file from the Oracle Technology Network (OTN). Any standalone OC4J process is not managed by Enterprise Manager and cannot be used within an Oracle9*i*AS enterprise environment.

You can start, manage, and control standalone OC4J instances through `oc4j.jar` (the OC4J standalone executable) and the `admin.jar` command-line tool. Update configuration files through the `admin.jar` tool or by modifying the XML configuration files by hand (though this is generally not recommended). The `admin.jar` tool modifies `server.xml` and other configuration files for you, based on settings you specify to the tool. For information about `admin.jar` and about how to start, stop, configure, and manage your standalone process, download the standalone version of the *Oracle9iAS Containers for J2EE User's Guide* along with `OC4J_extended.zip`.

In a simple testing mode, you can also deploy to the OC4J default Web application. The default Web application is introduced and its key directories are documented in "OC4J Default Web Application and Key Directories" on page 2-9.

Note: During development, also consider the Oracle9*i* JDeveloper visual development tool for development and deployment. This tool offers a number of conveniences, as described in "Oracle9i JDeveloper Support for Servlet Development" on page 2-9.

OC4J Deployment and Configuration with Oracle9*i*AS and Enterprise Manager

In Oracle9*i*AS production environments, use Enterprise Manager to manage OC4J and other components and to deploy and configure your applications. The Web module configuration pages are discussed under "Web Module Configuration in Oracle Enterprise Manager" on page 3-38. See the *Oracle9iAS Containers for J2EE User's Guide* for further information about using Enterprise Manager with OC4J. You can also refer to the *Oracle9i Application Server Administrator's Guide* and *Oracle Enterprise Manager Administrator's Guide* for additional general information about Enterprise Manager.

Configure each OC4J instance and its properties—within the context of an application server instance—using Enterprise Manager. After configuration, you start, manage, and control all OC4J instances through Enterprise Manager. You can group several OC4J processes in a cluster. You must use either the Enterprise Manager tool or its command-line tools for starting, stopping, restarting, configuring, and deploying applications. You cannot use the OC4J standalone tool—`admin.jar`—for managing OC4J instances created in an Oracle9*i*AS instance.

Note that in an Oracle9*i*AS release 2 environment, if you modify configuration files without going through Enterprise Manager, 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*.

For additional information about deploying an application that has EJB modules, see the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*.

Overview of Configuration Files

This section lists key XML configuration files supported by OC4J, primarily of interest in an OC4J standalone environment (given that Enterprise Manager in Oracle9*i*AS automates much of the configuration process). You can divide these configuration files into three categories:

- server configuration files

  These files, located in the OC4J configuration files directory, are used by OC4J to configure the server on startup. (The configuration files directory is `j2ee/home/config` in a standalone environment, and is configurable in an Oracle9*i*AS environment.)

- J2EE-standard application-level configuration files

  These are industry standard files, used to configure a particular application.

- OC4J-specific application-level configuration files

  These are OC4J-specific files, used to configure a particular application. For the most part, they correspond to the J2EE standard files, supporting OC4J-specific extended functionality.

Among the server configuration files are the following:

- `server.xml`
- `application.xml`
- `global-web-application.xml`
- `default-web-site.xml` (`http-web-site.xml` for OC4J standalone)

The `global-web-application.xml` and *`xxx`*`-web-site.xml` files and their elements and attributes are described later in this chapter—see "The global-web-application.xml and orion-web.xml Files" on page 3-14 and "The default-web-site.xml, http-web-site.xml, and Other Web Site XML Files" on page 3-30. For more information about OC4J server configuration files, see *Oracle9iAS Containers for J2EE User's Guide*.

Standard application-level configuration files include the following:

- `application.xml`
- `web.xml`
- `ejb-jar.xml`
- `application-client.xml`

These are described in J2EE standard documentation. The `web.xml` file, of particular interest to servlet developers, is described in the Sun Microsystems *Java Servlet Specification, Version 2.3*.

OC4J-specific application-level configuration files include the following:

- `orion-application.xml`
- `orion-web.xml`
- `orion-web-jar.xml`
- `orion-application-client.xml`
- `oc4j-ra.xml`

For information, see *Oracle9iAS Containers for J2EE User's Guide* and *Oracle Enterprise Manager Administrator's Guide.*

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

- `server.xml`
- `default-web-site.xml`, `http-web-site.xml`, or appropriate Web site XML file
- `global-web-application.xml`
- `web.xml`
- optionally `orion-web.xml`

Note: Notice 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.

Application Assembly

How you design, assemble, and build your application is largely up to you. Presumably you will be designing J2EE-compliant modules. Also, 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 covers the following topics:

- Web Application Modules
- Application Directory Structure
- Application Build Mechanisms
- Application Packaging

Web Application Modules

An OC4J application can consist of one or more J2EE-compliant modules, including:

- *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 typically 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.

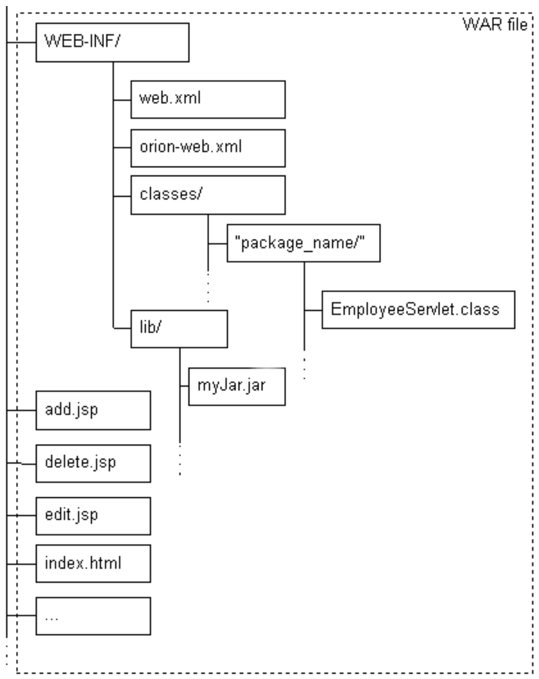
The examples in this chapter are derived from the sample application `stateless`, which is provided with OC4J. The actual application name is `employee`. This application includes both a Web and an EJB module, but building and deploying the Web module follows the same practice as a Web-only application.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

Application Directory Structure

Figure 3–1, 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, `http-web-site.xml` file, or other Web site XML file. (The `default-web-site.xml` file is for Oracle9*i*AS environments; `http-web-site.xml` is for OC4J standalone. See "The default-web-site.xml, http-web-site.xml, and Other Web Site XML Files" on page 3-30.)

Figure 3–1 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 the *`xxx`*`-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, if you are not using Oracle9*i* JDeveloper, you have several options:

- 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 is therefore 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.

or:

- Create a `make` file to automate the compilation and assembly process and use a standard UNIX `make` utility or the open-source `gmake` utility to execute it.

or:

- Compile each Java source file manually, using a Java 1.3.x-compatible compiler. This is a potentially error-prone process, perhaps appropriate for just the early stages of development.

A `build.xml` file or `make` file might include steps to create EAR, WAR, and JAR files as appropriate for deployment, or you can create them manually as described in the next section, "Application Packaging".

Application Packaging

This section describes the step for packaging an application for deployment. Once you have completed these steps, the application is ready to deploy as appropriate, depending on the target environment (such as OC4J standalone or Oracle9*i*AS). See "Introduction to Web Application Deployment and Configuration" on page 3-2 for an overview of OC4J deployment. The *Oracle9iAS Containers for J2EE User's Guide* covers deployment in detail for an Oracle9*i*AS environment. The standalone version of the user's guide, available with the OC4J standalone download from OTN, covers deployment for the standalone scenario.

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

The top-level archive file, for the entire application, is the EAR (Enterprise ARchive) file, which wraps any WAR and JAR files.

You can create these archive files using the standard Java JAR utility.

To deploy the application, follow these steps:

1. Ensure the creation of 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 3-14.
3. If your application has one or more EJB modules, create an `ejb-jar.xml` file for each. 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.

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, http-web-site.xml, and Other Web Site XML Files

Note: The detailed discussion in this section regarding configuration files and their elements and attributes assumes an OC4J standalone development environment. In an Oracle9*i*AS environment using Enterprise Manager, configuration is through Enterprise Manager Web module pages, and many of the files and their properties are invisible to the user.

For considerations in configuring and deploying a production application with Enterprise Manager in Oracle9*i*AS, see "OC4J Deployment and Configuration with Oracle9iAS and Enterprise Manager" on page 3-3.

Syntax Notes for Element Documentation

The elements described here do not use body values unless noted, and do not have subelements unless noted. Element attribute settings are in quotes. Here is the general syntax for an element with attributes but no subelements or body value:

```
<elementname attr1="value1" attr2="value2" ... />
```

If there are subelements (that have no subelements or body value themselves), the syntax is as follows:

```
<elementname ... >
   <subelement1 ... />
   <subelement2 ... />
   ...
</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 `global-web-application.xml`, in the OC4J configuration files directory, is the descriptor for the OC4J "global Web application". 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. 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, OC4J 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.

> **Note:** If you update `orion-web.xml` and later redeploy, you will lose your changes unless you include the updated `orion-web.xml` file in the application EAR file.

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 3-27, the `global-web-application.xml` file typically defines defaults for many settings of the `<web-app>` element and its subelements. The `<web-app>` element and subelements in the `web.xml` file are for desired settings specific to an application. When deploying an application, the `<web-app>` element and subelements in `orion-web.xml` are for overriding 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>`. This element and its subelements in `orion-web.xml` can 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
- URIs 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
- class loading

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. The `global-web-application.xml` file configures the global application and sets defaults, and the `orion-web.xml` can override these defaults for a particular application deployment as appropriate.

See "Syntax Notes for Element Documentation" on page 3-13 for general syntax information.

<orion-web-app ... >

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

Note: The `autoreload-jsp-pages` and `autoreload-jsp-beans` attributes of the `<orion-web-app>` element are not supported by the OC4J JSP container in Oracle9*i*AS release 2. You can use the JSP `main_mode` configuration parameter for equivalent functionality. See the *Oracle9iAS Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about this parameter.

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-class-loader>
<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).
- `jsp-print-null`: Set this flag to "`false`" to print an empty string instead of the default "null" string for null output from a JSP page. The default is "`true`".
- `jsp-timeout`: Specify an integer value, in seconds, after which any JSP page will be removed from memory if it has not been requested. This frees up resources in situations where some pages are called infrequently. The default value is 0 (zero), for no timeout.
- `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 invoking a servlet by name—anything appearing after this in a URL is assumed to be a class name. The default setting is `"/servlet"`. This is typically for use in an OC4J standalone environment during development and testing. For deployment, the

standard `web.xml` mechanisms for defining the context path and servlet path should be used.

Important: The default setting of `servlet-webdir`, or any setting that starts with a slash ("/"), presents a significant security risk and should not be used in a production environment. See "Servlet Security Considerations" on page 2-7 for more information.

- `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 for a scratch file can be retrieved by the following code in a servlet or JSP page, according to the servlet specification:

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

<classpath ... >

This specifies a codebase where classes used by this application can be found (servlets 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. The new value is specified 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 ... >

Use this element if, and only if, the application is to be clustered. Remove it or comment it out otherwise. 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 *Oracle9i Application Server 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.)

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. The servlet name is tied to a servlet class through its definition in

the `<web-app>` element of `global-web-application.xml`, `web.xml`, or `orion-web.xml`.

<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 information, 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 preprocess 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. The servlet name is tied to a servlet class through its definition in the `<web-app>` element of `global-web-application.xml`, `web.xml`, or `orion-web.xml`.
- `url-pattern`: This is an optional URL pattern to use as a qualifier for requests that are passed through the filter. For example:

```
url-pattern="/the/*.pattern"
```

For general information about servlet filters, see "Servlet Filters" on page 4-2.

Note: The functionality of the `<servlet-filter>` element is equivalent to that of the standard `<filter>` subelement of the `web.xml` `<web-app>` element. You can use either mechanism, but remember that `web.xml` settings override `global-web-application.xml` settings, and settings to `orion-web.xml` through Enterprise Manager (or directly to the file in the application deployment directory, for OC4J standalone) override `web.xml` settings.

<session-tracking ... >

This element specifies the session-tracking settings for this application. Session tracking is accomplished through cookies, assuming a cookie-enabled browser.

Note: If cookies are disabled, session tracking can be achieved only if your servlet explicitly calls the `encodeURL()` method of the response object, or the `encodeRedirectURL()` method for redirects.

For general information about servlet sessions, see "Servlet Sessions" on page 2-16.

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 log in to the application. Supported values are `"true"` and `"false"` (default).
- `cookies`: Specifies whether to send session cookies. Supported values are `"enabled"` (default) and `"disabled"`.
- `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 afterward.

<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 information, 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. For example:

  ```
  location="jdbc/TheDS"
  ```

- `name`: This is the resource reference name, which matches the name of a `resource-ref` element in the `web.xml` file. For example:

  ```
  name="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. The new value is specified 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. The `impliesAll` attribute or an appropriate combination of subelements—`<group>`, `<user>`, or both—should be used.

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`.

Important: OC4J has an automatic security mapping feature. By default, if a security role defined in `web.xml` has the same name as an OC4J group defined in `principals.xml`, then OC4J will map them. However, this feature is completely disabled if you do *any* explicit mapping through the `<security-role-mapping>` element. If you use `<security-role-mapping>` at all, OC4J assumes that you want explicit mapping only. This is to prevent unintended implicit mappings when a user might intend to declare explicit mappings only.

<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 number of seconds before expiration, or `"never"` for no expiration.
- `url-pattern`: This is the URL pattern that the expiration applies to, such as in the following example:

  ```
  url-pattern="*.gif"
  ```

<web-app-class-loader ... >

Use this element for class loading instructions.

Attributes:

- `search-local-classes-first`: Set this to "`true`" to search and load WAR file classes before system classes. The default is "`false`".
- `include-war-manifest-class-path`: Set this to "`false`" to *not* include the class path specified in the WAR file manifest `Class-Path` attribute when searching and loading classes from the WAR file (regardless of the `search-local-classes-first` setting). The default is "`true`".

<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`, defaults for `<web-app>` settings can be established. In `web.xml`, application-specific `<web-app>` settings can override the defaults. In `orion-web.xml`, deployment-specific `<web-app>` settings can 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, http-web-site.xml, 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 in Oracle9*i*AS, and `http-web-site.xml` for OC4J standalone. The documentation includes descriptions of the elements and attributes of these files. (All Web site XML files use the same DTD.)

Overview of Web Site XML Files

A Web site XML file contains the configurations for an OC4J Web site. The file `default-web-site.xml`, in the OC4J configuration files directory, configures the default OC4J Web site for Oracle9*i*AS and also defines default configurations for any additional Web site XML files. The file `http-web-site.xml` configures the HTTP Web site for an OC4J standalone environment.

The names of any additional Web site XML files are defined in the `server.xml` file, in the `path` attribute 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
- default Web application for this site
- additional Web applications for this site
- access-log format
- settings for user Web applications (for /~user/ sites)

Note: The `default-web-site.xml` file, for an Oracle9*i*AS production environment, sets up a default Web site that accesses OC4J through the Oracle HTTP Server and AJP (Apache JServ protocol).

The `http-web-site.xml` file, for an OC4J standalone development environment, sets up a default Web site that accesses the OC4J listener directly, using a protocol setting of `http` instead of `ajp13`.

Element Descriptions for Web Site XML Files

The element descriptions in this section apply to `default-web-site.xml`, `http-web-site.xml`, and the Web site XML files for any additional OC4J Web sites.

See "Syntax Notes for Element Documentation" on page 3-13 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>
```

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 in Oracle9*i*AS. 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 *Oracle9i Application Server 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) functionality. Possible values are `"true"` and `"false"` (default); however, because the OC4J servlet container in Oracle9*i*AS release 2 does not directly support SSL, leave this at the "`false`" setting.

  Note: SSL and HTTPS functionality is currently available through the Oracle HTTP Server.

- `protocol`: Specifies the protocol that the Web site is using. Possible values are `"http"`, `"https"`, and `"ajp13"` (Apache JServ Protocol, or AJP—default); however, in a production environment with Oracle9*i*AS release 2, you should use only the "`ajp13`" setting. The AJP protocol is for use with Oracle HTTP Server and `mod_oc4j`. Note that each port must have a corresponding protocol, and vice versa.

  The "`http`" setting is for development environments or OC4J standalone only. The "`https`" setting is not currently supported.

- 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 nonzero 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).

  In OC4J standalone, a port setting of 8888 is used for direct access to the OC4J listener.
- 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 other protocols, 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-delimited list of host names tied to this Web site.

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

You can use the body of this element for a brief description of the Web site.

<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 application startup. Otherwise, it is loaded upon the first request for it. Supported values are `"true"` and `"false"` (default).

  Preloading of individual servlets, through `<load-on-startup>` elements in the application `web.xml` file, is possible only if this `<web-app>` element `load-on-startup` attribute is enabled. See "Servlet Preloading" on page 2-4 for more information.
- `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`: This is the desired Web application name. For example, if the J2EE application name is `MyApp`, and this is the second of several Web sites, you might want a setting 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 setting `"/catalog"`, then it can be accessed as follows:

  ```
  http://www.site.com/catalog/CatalogApp
  ```
- `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. A possible use for this mode would be to share a Web application between an AJP site and an HTTP 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 could possibly 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. Oracle HTTP Server supports HTTPS.

<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`" by default in `default-web-site.xml` for an Oracle9*i*AS environment. In OC4J standalone, the default Web application is bound to "`/`" by default in `http-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/`*username*", where *username* 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:

  ```
  path="../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
  ```

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>
```

Web Module Configuration in Oracle Enterprise Manager

The direct use of `global-web-application.xml`, `orion-web.xml`, and `default-web-site.xml` elements and attributes described earlier in this chapter is generally for development in an OC4J standalone environment. In an Oracle9*i*AS environment, such as for production deployment, use Enterprise Manager for such configuration. This section covers Enterprise Manager pages related to Web module configuration and deployment. Some of the pages allow you to alter `orion-web.xml`, `global-web-application.xml`, and `default-web-site.xml` settings. Other pages display `web.xml` settings, which you can override through `orion-web.xml` settings.

The following Enterprise Manager pages are discussed:

- Enterprise Manager OC4J Home Page
- Enterprise Manager Deploy Web Application Page
- Enterprise Manager Website Properties Page
- Enterprise Manager Web Module Page
- Enterprise Manager Web Module Properties Page
- Enterprise Manager Web Module Mappings Page
- Enterprise Manager Web Module Filtering and Chaining Page
- Enterprise Manager Web Module Environment Page
- Enterprise Manager Web Module Advanced Properties Page

Each page description notes the corresponding `web.xml`, `orion-web.xml` (or `global-web-application.xml`), or `default-web-site.xml` elements and attributes. The `orion-web.xml` elements and attributes are documented in "Element Descriptions for global-web-application.xml and orion-web.xml" on page 3-16. The `default-web-site.xml` elements and attributes are covered in "Element Descriptions for Web Site XML Files" on page 3-31. For information about `web.xml` elements, refer to the Sun Microsystems *Java Servlet Specification, Version 2.3*.

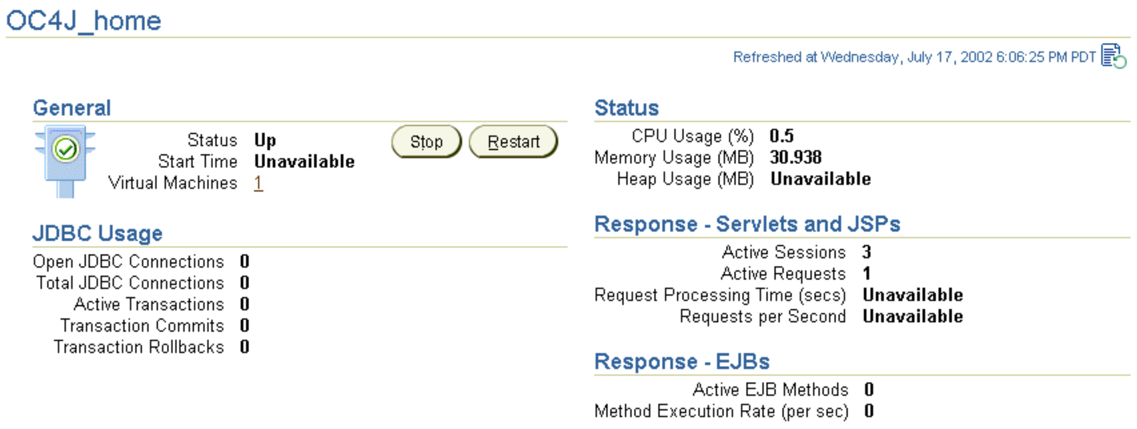
See the *Oracle9iAS Containers for J2EE User's Guide* for further information about using Enterprise Manager with OC4J. For general information about using Enterprise Manager to manage your Oracle9*i*AS environment, see the *Oracle9i Application Server Administrator's Guide*.

Enterprise Manager OC4J Home Page

From the Oracle9*i*AS Application Server Instance Home Page (the main page you reach when you first access Enterprise Manager), you can drill down to any of the running OC4J instances by selecting the name of the instance (`OC4J_home`, for example) in the System Components table. Enterprise Manager will display the OC4J Home Page for that instance.

Figure 3–2 and Figure 3–3 show portions of the OC4J Home Page for the `OC4J_home` instance.

Figure 3–2 Enterprise Manager OC4J Home Page (1 of 2)



OC4J_home

Refreshed at Wednesday, July 17, 2002 6:06:25 PM PDT

General

Status	Up
Start Time	Unavailable
Virtual Machines	1

JDBC Usage

Open JDBC Connections	0
Total JDBC Connections	0
Active Transactions	0
Transaction Commits	0
Transaction Rollbacks	0

Status

CPU Usage (%)	0.5
Memory Usage (MB)	30.938
Heap Usage (MB)	Unavailable

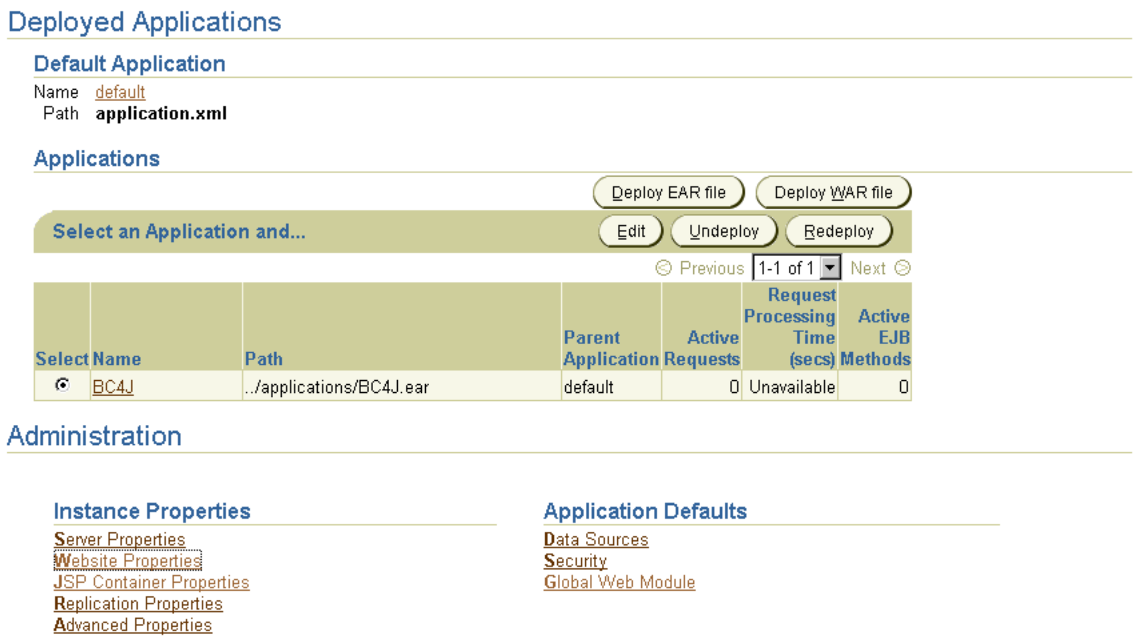
Response - Servlets and JSPs

Active Sessions	3
Active Requests	1
Request Processing Time (secs)	Unavailable
Requests per Second	Unavailable

Response - EJBs

Active EJB Methods	0
Method Execution Rate (per sec)	0

Figure 3–3 Enterprise Manager OC4J Home Page (2 of 2)


Deployed Applications

Default Application

Name default
Path **application.xml**

Applications

Select	Name	Path	Parent Application	Active Requests	Request Processing Time (secs)	Active EJB Methods
●	BC4J	../applications/BC4J.ear	default	0	Unavailable	0

Administration

Instance Properties
Server Properties
Website Properties
JSP Container Properties
Replication Properties
Advanced Properties

Application Defaults
Data Sources
Security
Global Web Module

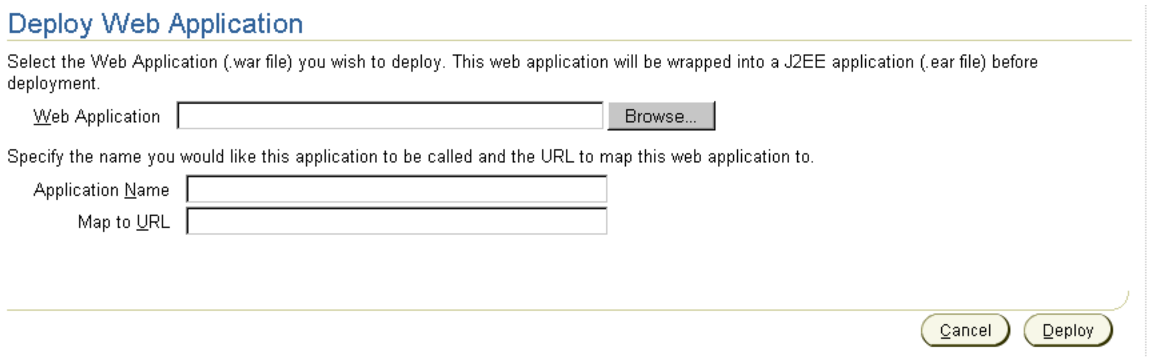
The OC4J Home Page enables you to access instance properties. In particular, relating to topics covered in this manual, note the following:

- Selecting **Website Properties** under Instance Properties in the Administration section of the page provides access to the Website Properties Page, through which you can access a variety of pages to update Web module properties.
- Clicking the **Deploy WAR file** button provides access to the Deploy Web Application Page.

Enterprise Manager Deploy Web Application Page

Figure 3–4 shows the key portion of the Enterprise Manager Deploy Web Application Page, which is the page for deploying a WAR file. Drill down to this page from any OC4J Home Page, such as the page for `OC4J_home`, by clicking the **Deploy WAR file** button.

Figure 3–4 Enterprise Manager Deploy Web Application Page


Deploy Web Application

Select the Web Application (.war file) you wish to deploy. This web application will be wrapped into a J2EE application (.ear file) before deployment.

Web Application

Specify the name you would like this application to be called and the URL to map this web application to.

Application Name

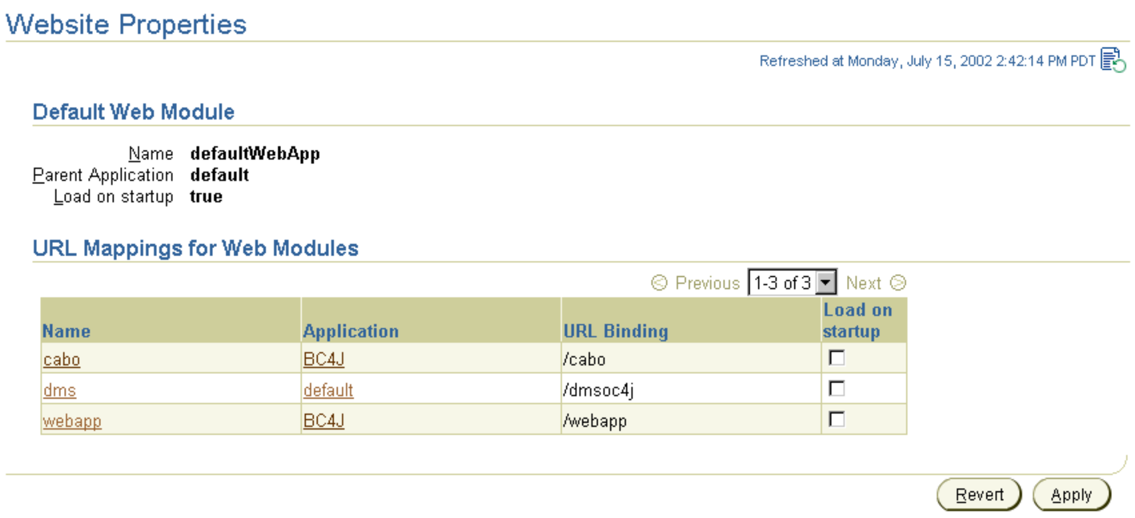
Map to URL

In the Deploy Web Application Page, click the **Browse** button to select a WAR file to deploy. Then specify the application name along with a URL to map to the application. Specifying the application name will result in an entry in the `server.xml` file, and there will also be a resulting entry in the `default-web-site.xml` file mapping the application name to the URL. This is accomplished through the `application` and `root` attributes of a `<web-app>` subelement of the `<web-site>` element. In addition, the `mod_oc4j.conf` configuration file for the Oracle HTTP Server `mod_oc4j` Apache mod is updated with appropriate mount points.

Enterprise Manager Website Properties Page

Figure 3–5 shows the key portion of the Enterprise Manager Website Properties Page for an OC4J instance. Drill down to this page from the OC4J Home Page, such as the page for `OC4J_home`, by selecting **Website Properties** under Instance Properties in the Administration section of the page.

Figure 3–5 Enterprise Manager Website Properties Page



Website Properties

Refreshed at Monday, July 15, 2002 2:42:14 PM PDT

Default Web Module

Name	**defaultWebApp**
Parent Application	**default**
Load on startup	**true**

URL Mappings for Web Modules

Name	Application	URL Binding	Load on startup
cabo	BC4J	/cabo	☐
dms	default	/dmsoc4j	☐
webapp	BC4J	/webapp	☐

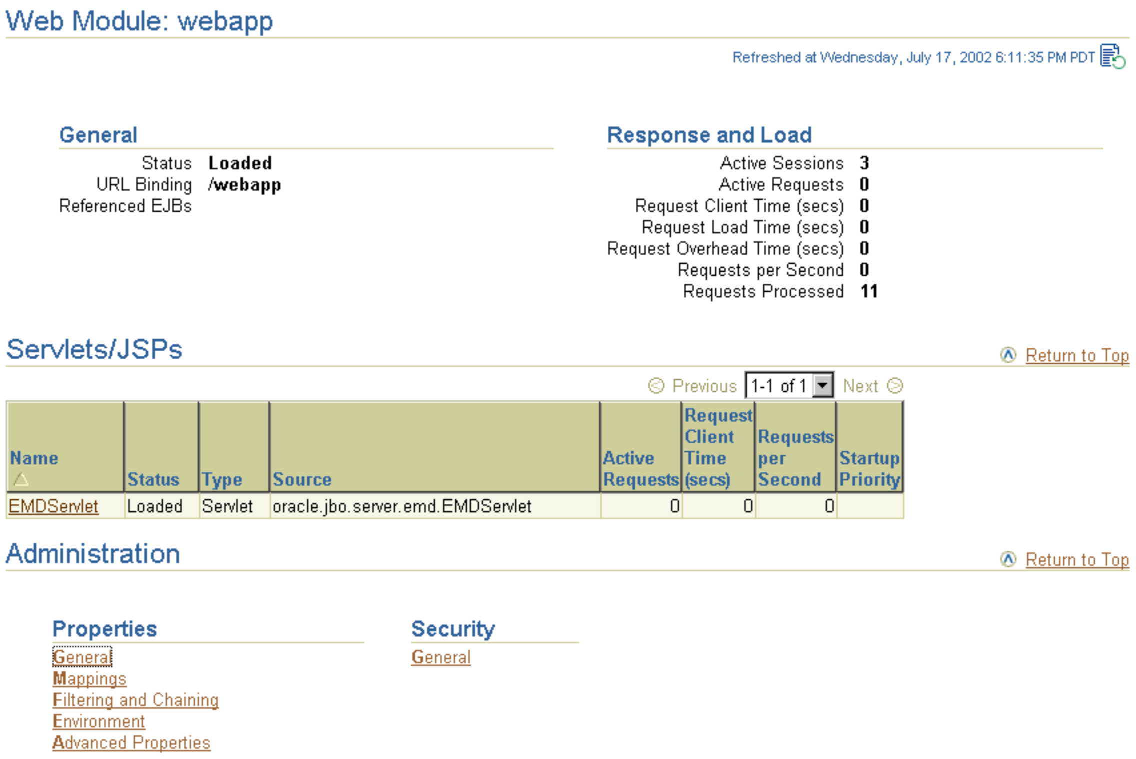
Among other things, this page enables you to specify whether each application should be loaded automatically when OC4J starts. (Otherwise, an application is not loaded until the first request for it.) This corresponds to the `load-on-startup` attribute of the appropriate `<web-app>` subelement of the `<web-site>` element in the `default-web-site.xml` file. (For general information about loading an application at OC4J startup, see "Servlet Preloading" on page 2-4.)

From the Website Properties Page, drill down to the Web Module Page for any particular Web module. In the sample page above, for example, you can select **`webapp`** to drill down to the Web Module Page for that module.

Enterprise Manager Web Module Page

Figure 3–6 shows the key portion of the Enterprise Manager Web Module Page for the module `webapp`. Drill down to the Web Module Page for a particular module by selecting the module name in the Website Properties Page.

Figure 3–6 Enterprise Manager Web Module Page



Web Module: webapp

Refreshed at Wednesday, July 17, 2002 6:11:35 PM PDT

General

Status	**Loaded**
URL Binding	**/webapp**
Referenced EJBs	

Response and Load

Active Sessions	**3**
Active Requests	**0**
Request Client Time (secs)	**0**
Request Load Time (secs)	**0**
Request Overhead Time (secs)	**0**
Requests per Second	**0**
Requests Processed	**11**

Servlets/JSPs

Name	Status	Type	Source	Active Requests	Request Client Time (secs)	Requests per Second	Startup Priority
EMDServlet	Loaded	Servlet	oracle.jbo.server.emd.EMDServlet	0	0	0	

Administration

Properties
General
Mappings
Filtering and Chaining
Environment
Advanced Properties

Security
General

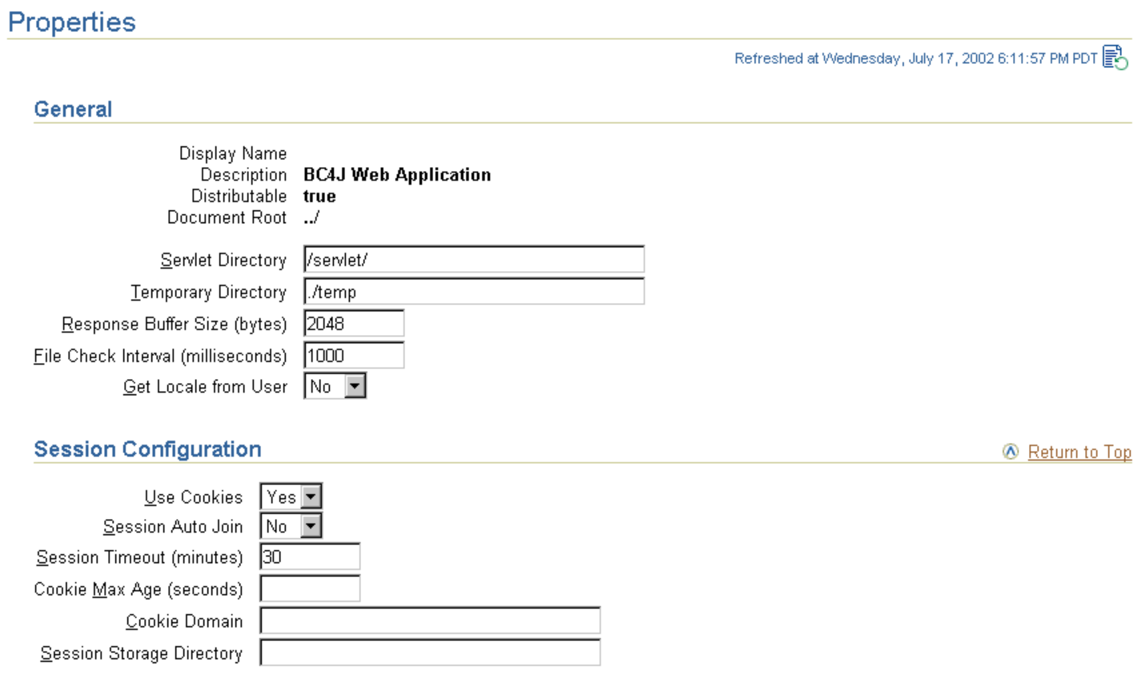
From the Web Module Page, you can access several categories of Web module properties through the following links under Properties in the Administration section of the page:

- **General** (to drill down to the Web Module Properties Page)
- **Mappings** (to drill down to the Web Module Mappings Page)
- **Filtering and Chaining** (to drill down to the Web Module Filtering and Chaining Page)
- **Environment** (to drill down to the Web Module Environment Page)
- **Advanced Properties** (to drill down to the Web Module Advanced Properties Page)

Enterprise Manager Web Module Properties Page

Figure 3–7 and Figure 3–8 show portions of the Enterprise Manager Web Module Properties Page for a particular module. Drill down to this page by selecting **General** under Properties in the Administration section of the Web Module Page.

Figure 3–7 Enterprise Manager Web Module Properties Page (1 of 2)


Properties

Refreshed at Wednesday, July 17, 2002 6:11:57 PM PDT

General

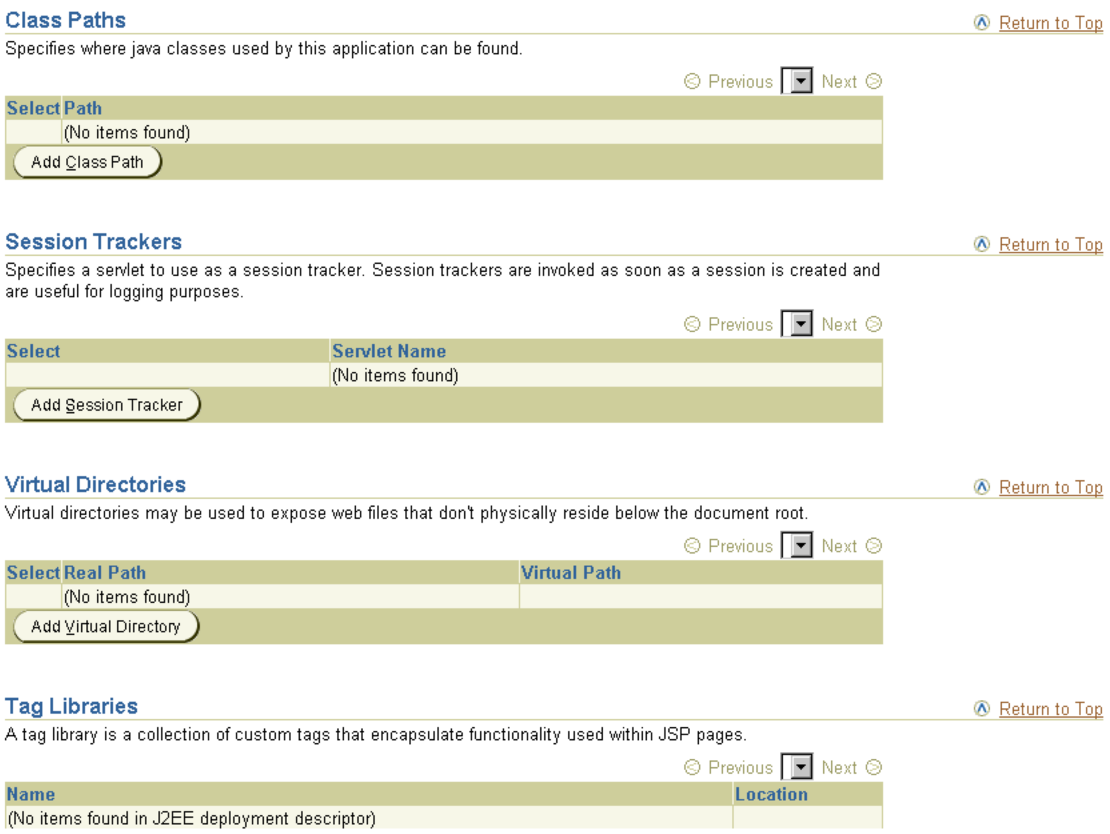
Display Name	
Description	**BC4J Web Application**
Distributable	**true**
Document Root	../
Servlet Directory	/servlet/
Temporary Directory	./temp
Response Buffer Size (bytes)	2048
File Check Interval (milliseconds)	1000
Get Locale from User	No

Session Configuration

Return to Top

Use Cookies	Yes
Session Auto Join	No
Session Timeout (minutes)	30
Cookie Max Age (seconds)	
Cookie Domain	
Session Storage Directory	

Figure 3–8 Enterprise Manager Web Module Properties Page (2 of 2)



Class Paths

Return to Top

Specifies where java classes used by this application can be found.

Previous Next

Select	Path
	(No items found)

Add Class Path

Session Trackers

Return to Top

Specifies a servlet to use as a session tracker. Session trackers are invoked as soon as a session is created and are useful for logging purposes.

Previous Next

Select	Servlet Name
	(No items found)

Add Session Tracker

Virtual Directories

Return to Top

Virtual directories may be used to expose web files that don't physically reside below the document root.

Previous Next

Select	Real Path	Virtual Path
	(No items found)	

Add Virtual Directory

Tag Libraries

Return to Top

A tag library is a collection of custom tags that encapsulate functionality used within JSP pages.

Previous Next

Name	Location
(No items found in J2EE deployment descriptor)	

Correspondence of these settings to `orion-web.xml` elements is as follows.

In the General section:

- Servlet Directory corresponds to the `servlet-webdir` attribute of the `<orion-web-app>` element.
- Temporary Directory corresponds to the `temporary-directory` attribute of the `<orion-web-app>` element.
- Response Buffer Size corresponds to the `default-buffer-size` attribute of the `<orion-web-app>` element.

- File Check Interval corresponds to the `file-modification-check-interval` attribute of the `<orion-web-app>` element.
- Get Locale from User corresponds to the `get-locale-from-user` attribute of the `<orion-web-app>` element.

In the Session Configuration section:

- Use Cookies corresponds to the `cookies` attribute of the `<session-tracking>` element, which is a subelement of the `<orion-web-app>` element.
- Session Auto Join corresponds to the `autojoin-session` attribute of the `<session-tracking>` element.
- Session Timeout corresponds to the `<session-timeout>` subelement of the `<session-config>` subelement of the standard `<web-app>` element. You can use a `<web-app>` subelement under `<orion-web-app>` in `orion-web.xml` for deployment-specific overrides of `<web-app>` settings in the application `web.xml` file.
- Cookie Max Age corresponds to the `cookie-max-age` attribute of the `<session-tracking>` element.
- Cookie Domain corresponds to the `cookie-domain` attribute of the `<session-tracking>` element.
- Session Storage Directory corresponds to the `persistence-path` attribute of the `<orion-web-app>` element.

In the Class Paths section:

- Adding a classpath here corresponds to setting the `path` attribute of a `<classpath>` subelement of the `<orion-web-app>` element.

In the Session Trackers section:

- Adding a session tracker here corresponds to setting the `servlet-name` attribute of a `<session-tracker>` element, which is a subelement of the `<session-tracking>` element.

In the Virtual Directories section:

- Adding a virtual directory here corresponds to setting the `real-path` and `virtual-path` attributes of a `<virtual-directory>` subelement of the `<orion-web-app>` element.

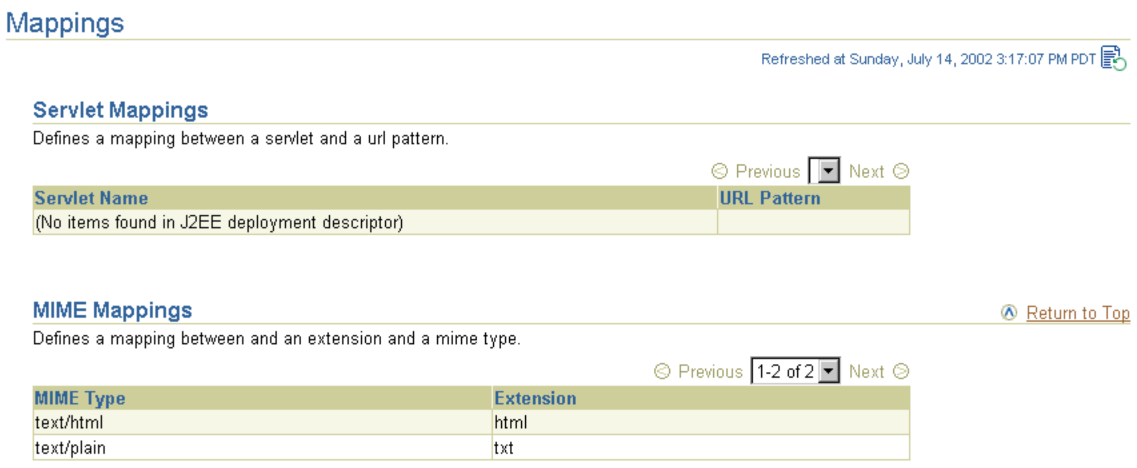
In the Tag Libraries section:

- This lists JSP tag libraries that are used in the application, according to contents of the WAR file.

Enterprise Manager Web Module Mappings Page

Figure 3–9 and Figure 3–10 show portions of the Enterprise Manager Web Module Mappings Page for a particular module. Drill down to this page by selecting **Mappings** under Properties in the Administration section of the Web Module Page.

Figure 3–9 Enterprise Manager Web Module Mappings Page (1 of 2)



Mappings

Refreshed at Sunday, July 14, 2002 3:17:07 PM PDT

Servlet Mappings

Defines a mapping between a servlet and a url pattern.

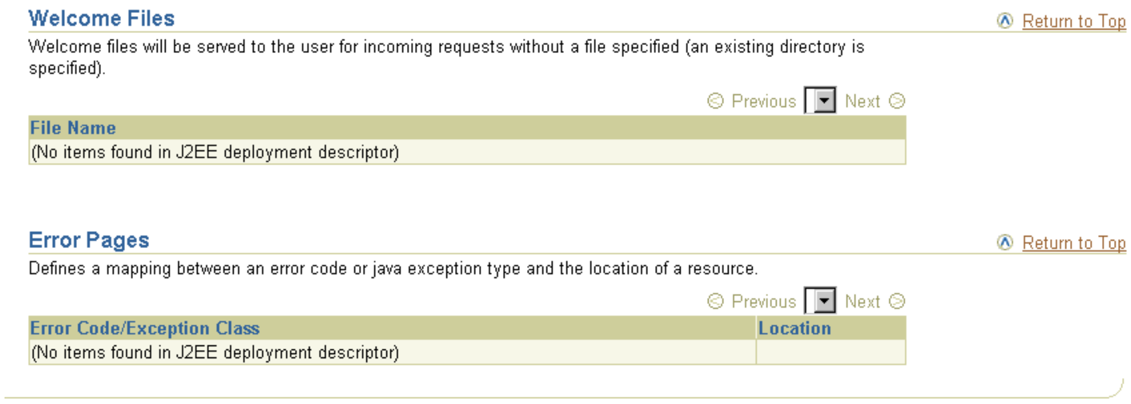
Servlet Name	URL Pattern
(No items found in J2EE deployment descriptor)	

MIME Mappings

Defines a mapping between and an extension and a mime type.

MIME Type	Extension
text/html	html
text/plain	txt

Figure 3–10 Enterprise Manager Web Module Mappings Page (2 of 2)



Welcome Files

Return to Top

Welcome files will be served to the user for incoming requests without a file specified (an existing directory is specified).

Previous Next

File Name
(No items found in J2EE deployment descriptor)

Error Pages

Return to Top

Defines a mapping between an error code or java exception type and the location of a resource.

Previous Next

Error Code/Exception Class	Location
(No items found in J2EE deployment descriptor)	

These settings all correspond to subelements of the `<web-app>` element in the `web.xml` file. You can use a `<web-app>` subelement under `<orion-web-app>` in `orion-web.xml` for deployment-specific overrides of these settings. You can use the Advanced Properties Page for this purpose—see "Enterprise Manager Web Module Advanced Properties Page" on page 3-51.

In the Servlet Mappings section:

- A servlet name and URL pattern specified here correspond to settings in the `<servlet-name>` and `<url-pattern>` subelements of a `<servlet-mapping>` subelement of the `<web-app>` element.

In the MIME Mappings section:

- A MIME type and extension specified here correspond to settings in the `<mime-type>` and `<extension>` subelements of a `<mime-mapping>` subelement of the `<web-app>` element.

In the Welcome Files section:

- A file name specified here corresponds to the setting in a `<welcome-file>` subelement of the `<welcome-file-list>` subelement of the `<web-app>` element.

In the Error Pages section:

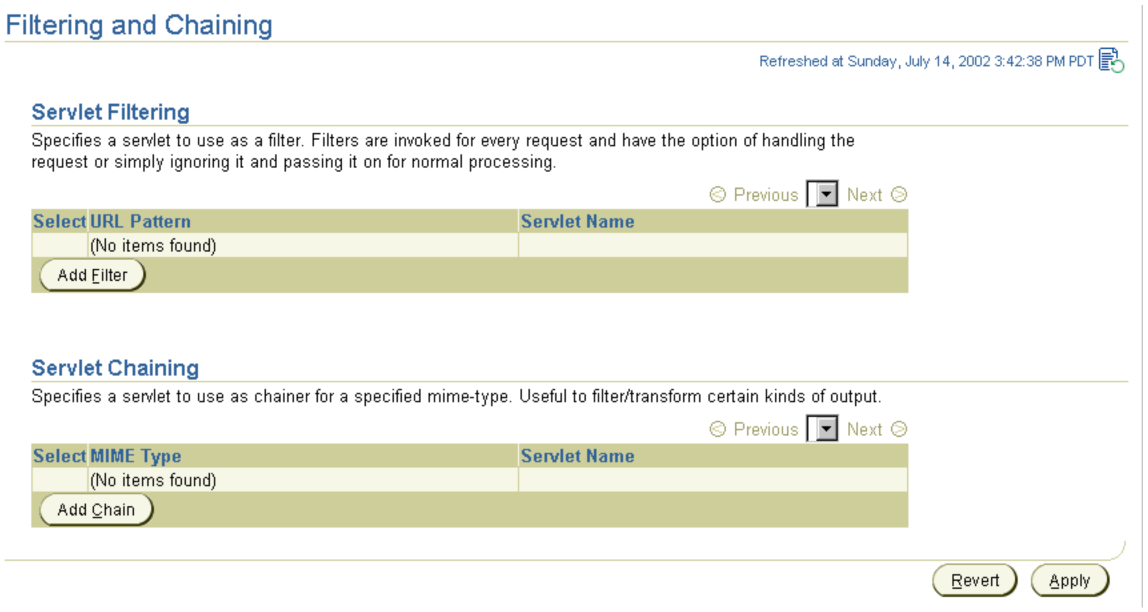
- An error code and location specified here correspond to settings in the `<error-code>` and `<location>` subelements of an `<error-page>` subelement of the `<web-app>` element.

- An exception class and location specified here correspond to settings in the `<exception-type>` and `<location>` subelements of an `<error-page>` subelement of the `<web-app>` element.

Enterprise Manager Web Module Filtering and Chaining Page

Figure 3–11 shows the key portion of the Enterprise Manager Web Module Filtering and Chaining Page for a particular module. Drill down to this page by selecting **Filtering and Chaining** under Properties in the Administration section of the Web Module Page.

Figure 3–11 Enterprise Manager Web Module Filtering and Chaining Page



Filtering and Chaining

Refreshed at Sunday, July 14, 2002 3:42:38 PM PDT

Servlet Filtering

Specifies a servlet to use as a filter. Filters are invoked for every request and have the option of handling the request or simply ignoring it and passing it on for normal processing.

Select	URL Pattern	Servlet Name
	(No items found)	

Servlet Chaining

Specifies a servlet to use as chainer for a specified mime-type. Useful to filter/transform certain kinds of output.

Select	MIME Type	Servlet Name
	(No items found)	

Correspondence of these settings to `orion-web.xml` elements is as follows.

In the Servlet Filtering section:

- Adding a filter here is equivalent to setting the `servlet-name` and `url-pattern` attributes of a `<servlet-filter>` subelement of the `<orion-web-app>` element. The servlet name you specify is tied to a servlet class through its standard configuration in the `web.xml` file.

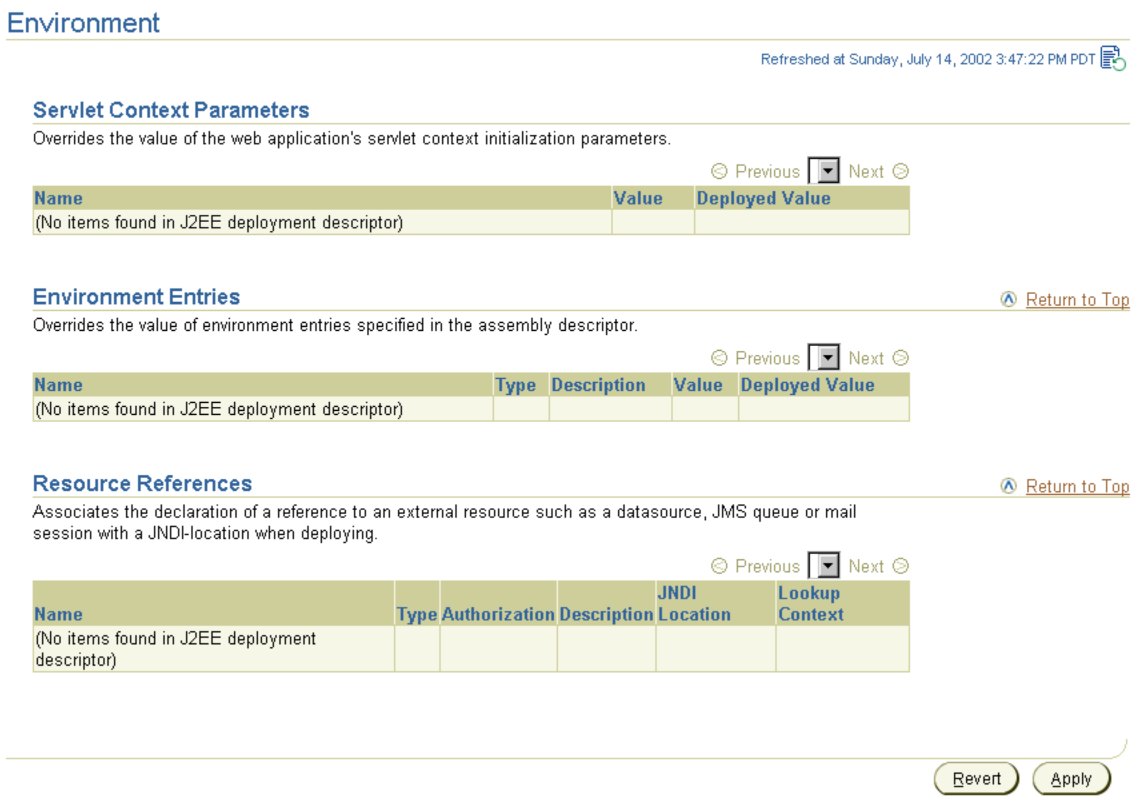
In the Servlet Chaining section:

- Adding a chain here is equivalent to setting the `servlet-name` and `mime-type` attributes of a `<servlet-chaining>` subelement of the `<orion-web-app>` element. The servlet name you specify is tied to a servlet class through its standard configuration in the `web.xml` file.

Enterprise Manager Web Module Environment Page

Figure 3–12 shows most of the Enterprise Manager Web Module Environment Page for a particular module. Drill down to this page by selecting **Environment** under Properties in the Administration section of the Web Module Page.

Figure 3–12 Enterprise Manager Web Module Environment Page



Environment

Refreshed at Sunday, July 14, 2002 3:47:22 PM PDT

Servlet Context Parameters

Overrides the value of the web application's servlet context initialization parameters.

Name	Value	Deployed Value
(No items found in J2EE deployment descriptor)		

Environment Entries

Overrides the value of environment entries specified in the assembly descriptor.

Name	Type	Description	Value	Deployed Value
(No items found in J2EE deployment descriptor)				

Resource References

Associates the declaration of a reference to an external resource such as a datasource, JMS queue or mail session with a JNDI-location when deploying.

Name	Type	Authorization	Description	JNDI Location	Lookup Context
(No items found in J2EE deployment descriptor)					

This page shows settings for servlet context parameter overrides, environment entry overrides, and resource references. The overrides indicate settings in the `orion-web.xml` file that override corresponding `web.xml` settings.

Correspondence of these settings to `web.xml` and `orion-web.xml` elements is as follows.

In the Servlet Context Parameters section:

- This section displays settings of `web.xml` `<context-param>` elements that can be overridden for this deployment, along with any Deployed Value overrides that have already been specified. Enter a new value in the Deployed Value column to specify a new override. Doing so creates a `<context-param-mapping>` entry in `orion-web.xml`.

In the Environment Entries section:

- This section displays settings of `web.xml` `<env-entry>` elements that can be overridden for this deployment, along with any Deployed Value overrides that have already been specified. Enter a new value in the Deployed Value column to specify a new override. Doing so creates an `<env-entry-mapping>` entry in `orion-web.xml`.

In the Resource References section:

- This section displays a combination of `web.xml` and `orion-web.xml` settings. The name and type of a resource reference correspond to `<res-ref-name>` and `<res-type>` subelements under a `<resource-ref>` subelement of the `<web-app>` element in the `web.xml` file. The JNDI location and lookup context correspond to settings under a `<resource-ref-mapping>` element and its `<lookup-context>` subelement, under the `<orion-web-app>` element in the `orion-web.xml` file.

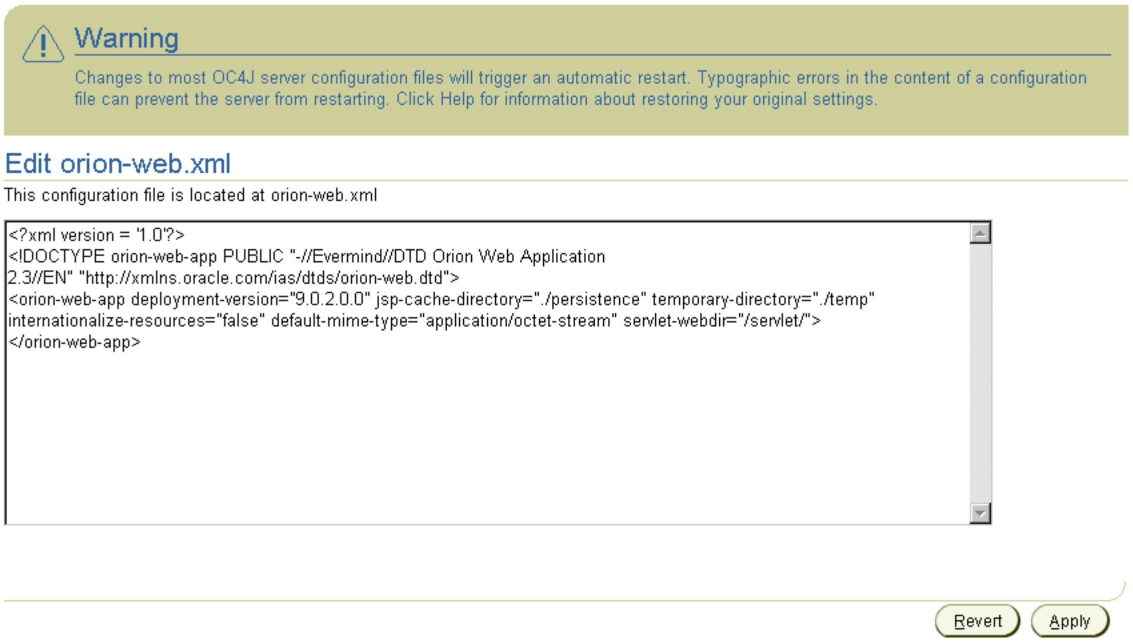
Enterprise Manager Web Module Advanced Properties Page

Figure 3–13 shows the key portion of the Enterprise Manager Web Module Advanced Properties Page for a particular module. Drill down to this page by selecting **Advanced Properties** under Properties in the Administration section of the Web Module Page.

You can use the Web Module Advanced Properties Page to edit `orion-web.xml` or `global-web-application.xml` for any settings not covered by the previously discussed Enterprise Manager Web module pages. (In fact, you can make any `orion-web.xml` or `global-web-application.xml` entries through the

Advanced Properties Page; however, it is advisable to use the previously described pages whenever possible because of their error handling and reporting features.)

Figure 3–13 Enterprise Manager Web Module Advanced Properties Page



Warning

Changes to most OC4J server configuration files will trigger an automatic restart. Typographic errors in the content of a configuration file can prevent the server from restarting. Click Help for information about restoring your original settings.

Edit orion-web.xml

This configuration file is located at orion-web.xml

```
<?xml version = '1.0'?>
<!DOCTYPE orion-web-app PUBLIC "-//Evermind//DTD Orion Web Application
2.3//EN" "http://xmlns.oracle.com/ias/dtds/orion-web.dtd">
<orion-web-app deployment-version="9.0.2.0.0" jsp-cache-directory="./persistence" temporary-directory="./temp"
internationalize-resources="false" default-mime-type="application/octet-stream" servlet-webdir="/servlet/">
</orion-web-app>
```

4
Servlet Filters and Event Listeners

This chapter describes the following servlet features, which are new in the servlet 2.3 specification:

- Servlet Filters
- Event Listeners

Servlet Filters

Servlet filters are used for preprocessing Web application requests and post-processing responses. While this is new functionality in the servlet 2.3 specification, earlier Web servers have supported similar constructs.

This section covers the following topics:

- Overview of Servlet Filters
- How the Servlet Container Invokes Filters
- Filter Examples

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 scenario for a filter is where you want to apply preprocessing 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.

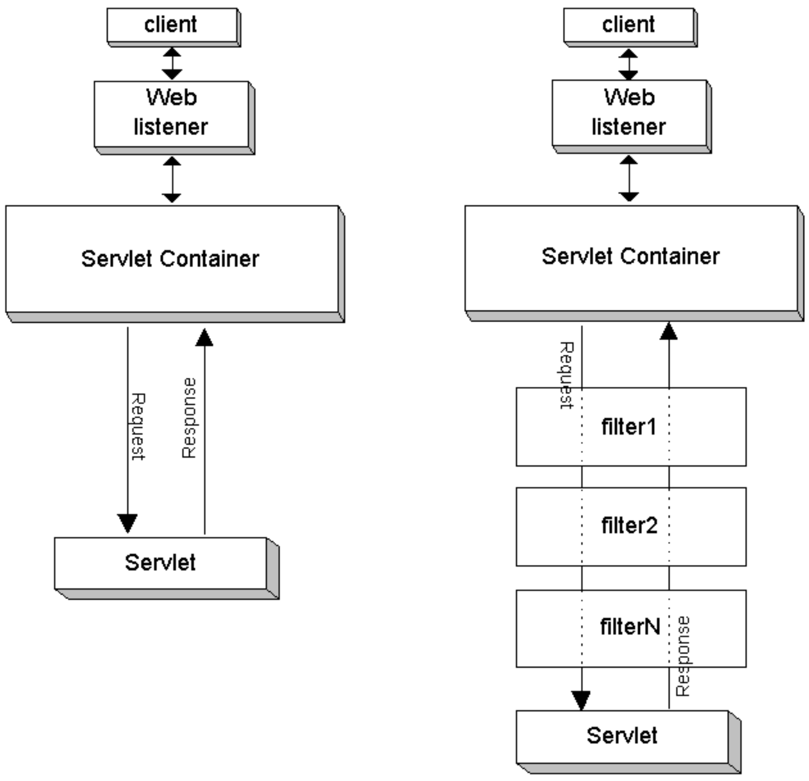
Note that filters are not servlets. They do not implement and override `HttpServlet` methods such as `doGet()` or `doPost()`. Rather, a filter implements the methods of the `javax.servlet.Filter` interface. The methods are:

- `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. The `web.xml` file specifies 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 in `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.

```
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. The numbered code notes refer to the following:

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 OC4J standalone default Web application 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, the OC4J standalone default Web application will be used. The filter should be configured in the `web.xml` file in the default Web application `/WEB-INF` directory. You will need the following entries 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`.

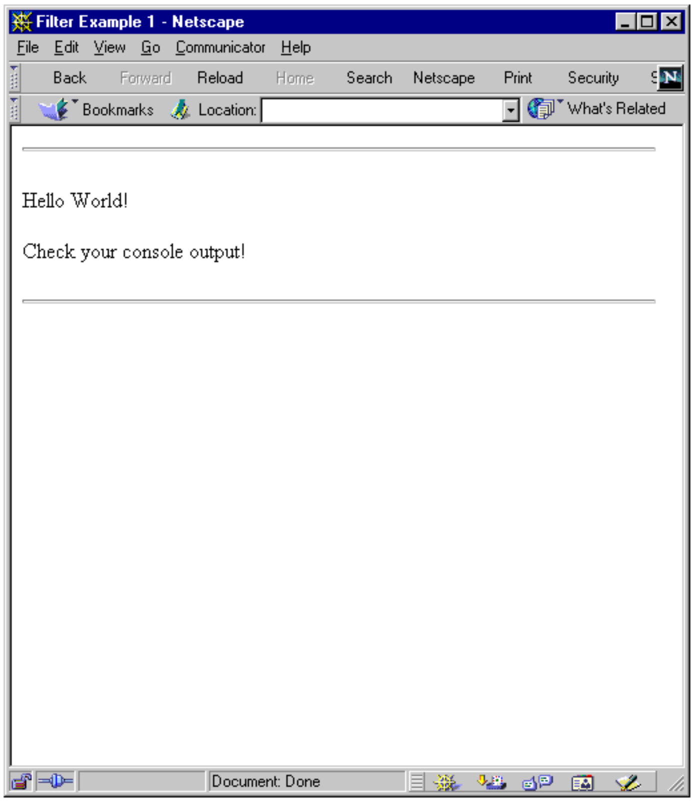
Note: There is a `<servlet-filter>` element for the `global-web-application.xml` file or `orion-web.xml` file that has equivalent functionality. You can use either mechanism, but remember that `web.xml` settings override `global-web-application.xml` settings, and settings to `orion-web.xml` through Oracle Enterprise Manager (or directly to the file in the application deployment directory, for OC4J standalone) override `web.xml` settings.

Running Example 1 Invoke `filter.jsp` from your Web browser. 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 1 - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security
Bookmarks Location: What's Related
Document: Done

Hello World!

Check your console output!

Filter Example 2

A filter can be configured 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 `filter2.jsp` in the OC4J standalone default Web application root directory:

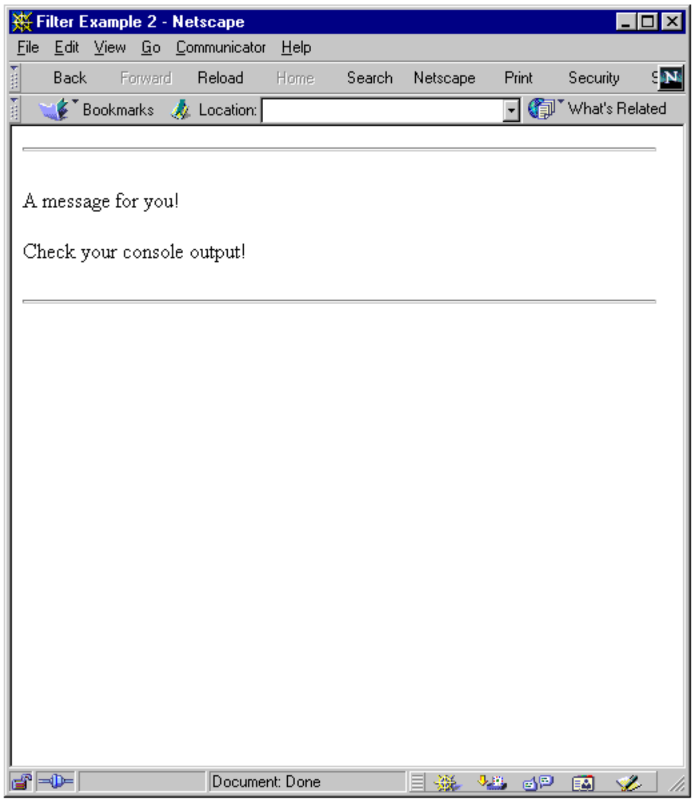
```
<HTML>
<HEAD>
<TITLE>Lesson 2</TITLE>
</HEAD>
<BODY>
<HR>
<P><%=request.getAttribute("message")%></P>
<P>Check your console output!</P>
<HR>
</BODY>
</HTML>
```

Running Example 2 Make sure that the filter configuration has been entered in the `web.xml` file, as shown above. Then access the JSP page with your browser. 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 Web 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 2 - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security
Bookmarks Location:
What's Related
Document: Done

A message for you!

Check your console 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`—This is a new implementation of `ServletOutputStream` for response wrappers.

- `GenericResponseWrapper`—This is a basic implementation of the response wrapper interface.
- `PrePostFilter`—This is 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 software design textbooks). 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 under the default Web application root directory, and compile it:

```
/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 under the default Web application root directory, and compile it:

```
/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.

Here is the 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 under the default Web application root directory, and compile it:

```
/WEB-INF/classes/com/acme/filter
```

As in the previous examples, create a simple JSP page and place it in the root directory of the default Web application. Here is `filter3.jsp`:

```
<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
```

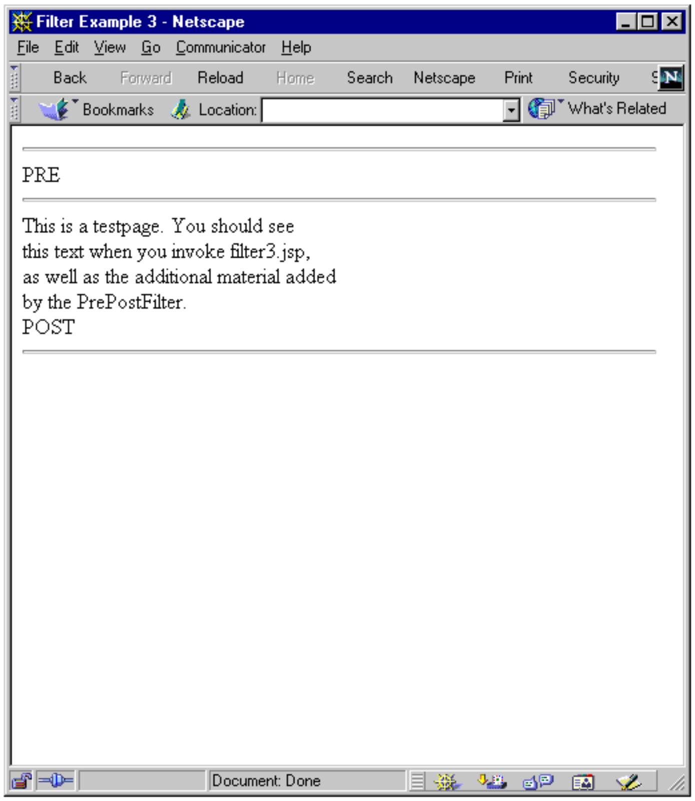
Save this JSP code in `filter3.jsp` in the root directory of the default Web application.

Configuring the Filter The following `<filter>` element must be added 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 Invoke the servlet in your Web browser. You should see a page that looks something like what is shown in Figure 4–4.

Figure 4–4 Example 3 Output


Filter Example 3 - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security
Bookmarks Location:
What's Related
Document: Done

```
PRE

This is a testpage. You should see
this text when you invoke filter3.jsp,
as well as the additional material added
by the PrePostFilter.
POST
```

Event Listeners

The servlet 2.3 specification adds the capability to track key events in your Web applications through *event listeners*. This functionality allows more efficient resource management and automated processing based on event status. This section describes servlet event listeners, covering the following topics:

- Event Categories and Listener Interfaces
- Typical Event Listener Scenario
- Event Listener Declaration and Invocation
- Event Listener Coding and Deployment Guidelines
- Event Listener Methods and Related Classes
- Event Listener Sample

Event Categories and Listener Interfaces

There are two levels of servlet events:

- servlet context (application) level event

  This involves resources or state held at the level of the JVM in which the application is running; that is, associated with the application servlet context object.

- session level event

  This involves resources or state associated with the series of requests from a single user session; that is, associated with the HTTP session object.

At each of these two levels, there are two event categories:

- lifecycle changes
- attribute changes

You can create one or more event listener classes for each of the four event categories. A single listener class can monitor multiple event categories.

Create an event listener class by implementing the appropriate interface or interfaces of the `javax.servlet` package or `javax.servlet.http` package. Table 4–1 summarizes the four categories and the associated interfaces.

Table 4–1 Event Listener Categories and Interfaces

Event Category	Event Descriptions	Interface
Servlet context lifecycle changes	Servlet context creation (at which point it is ready to service its first request) Imminent shutdown of the servlet context	javax.servlet. ServletContextListener
Servlet context attribute changes	Addition of servlet context attributes Removal of servlet context attributes Replacement of servlet context attributes	javax.servlet. ServletContextAttributeListener
Session lifecycle changes	Session creation Session invalidation Session timeout	javax.servlet.http. HttpSessionListener
Session attribute changes	Addition of session attributes Removal of session attributes Replacement of session attributes	javax.servlet.http. HttpSessionAttributeListener

Typical Event Listener Scenario

Consider a Web application consisting of servlets that access a database. A typical use of the event listener mechanism would be to create a servlet context lifecycle event listener to manage the database connection. This listener might function as follows:

1. The listener is notified of application startup.
2. The application logs in to the database and stores the connection object in the servlet context.
3. Servlets use the database connection to perform SQL operations.
4. The listener is notified of imminent application shutdown (shutdown of the Web server, or removal of the application from the Web server).
5. Prior to application shutdown, the listener closes the database connection.

Event Listener Declaration and Invocation

Event listeners are declared in the application `web.xml` deployment descriptor, through `<listener>` elements under the top-level `<web-app>` element. Each listener has its own `<listener>` element, with a `<listener-class>` subelement specifying the class name. Within each event category, event listeners should be specified in the order in which you would like them to be invoked when the application runs.

After the application starts up, and before the application services the first request, the servlet container creates and registers an instance of each listener class that you have declared. For each event category, listeners are registered in the order in which they are declared. Then, as the application runs, event listeners for each category are invoked in the order of their registration. All listeners will remain active until after the last request is serviced for the application.

Upon application shutdown, session event listeners are notified first, in reverse order of their declarations, then application event listeners are notified in reverse order of their declarations.

Here is an example of event listener declarations, from the Sun Microsystems *Java Servlet Specification, Version 2.3*:

```
<web-app>
   <display-name>MyListeningApplication</display-name>
   <listener>
      <listener-class>com.acme.MyConnectionManager</listenerclass>
   </listener>
   <listener>
      <listener-class>com.acme.MyLoggingModule</listener-class>
   </listener>
   <servlet>
      <display-name>RegistrationServlet</display-name>
      ...
   </servlet>
</web-app>
```

Assume that `MyConnectionManager` and `MyLoggingModule` both implement the `ServletContextListener` interface, and that `MyLoggingModule` also implements the `HttpSessionListener` interface.

When the application runs, both listeners will be notified of servlet context lifecycle events, and the `MyLoggingModule` listener will also be notified of session lifecycle events. For servlet context lifecycle events, the `MyConnectionManager` listener will be notified first, because of the declaration order.

Event Listener Coding and Deployment Guidelines

Be aware of the following rules and guidelines for event listener classes:

- In a multithreaded application, attribute changes may occur simultaneously. There is no requirement for the servlet container to synchronize the resulting notifications—it is the listener classes themselves that are responsible for maintaining data integrity in such a situation.
- Each listener class must have a public zero-argument constructor.
- Each listener class file must be packaged in the application WAR file, either under `/WEB-INF/classes` or in a JAR file in `/WEB-INF/lib`.

Note: In a distributed environment, the scope of event listeners is one for each deployment descriptor declaration for each JVM. There is no requirement for distributed Web containers to propagate servlet context events or session events to additional JVMs. This is discussed in the Sun Microsystems *Java Servlet Specification, Version 2.3*.

Event Listener Methods and Related Classes

This section lists event listener methods, which are called by the servlet container when a servlet context event or session event occurs. These methods take different types of event objects as input, so these event classes and their methods are also discussed.

ServletContextListener Methods, ServletContextEvent Class

The `ServletContextListener` interface specifies the following methods:

- `void contextInitialized(ServletContextEvent sce)`

  The servlet container calls this method to notify the listener that the servlet context has been created and the application is ready to process requests.

- `void contextDestroyed(ServletContextEvent sce)`

  The servlet container calls this method to notify the listener that the application is about to be shut down.

The servlet container creates a `javax.servlet.ServletContextEvent` object that is input for calls to `ServletContextListener` methods. The

`ServletContextEvent` class includes the following method, which your listener can call:

- `ServletContext getServletContext()`

  Use this to retrieve the servlet context object that was created or is about to be destroyed, from which you can obtain information as desired. See "Servlet Contexts" on page 1-9 for information about the `javax.servlet.ServletContext` interface.

ServletContextAttributeListener Methods, ServletContextAttributeEvent Class

The `ServletContextAttributeListener` interface specifies the following methods:

- `void attributeAdded(ServletContextAttributeEvent scae)`

  The servlet container calls this method to notify the listener that an attribute was added to the servlet context.

- `void attributeRemoved(ServletContextAttributeEvent scae)`

  The servlet container calls this method to notify the listener that an attribute was removed from the servlet context.

- `void attributeReplaced(ServletContextAttributeEvent scae)`

  The servlet container calls this method to notify the listener that an attribute was replaced in the servlet context.

The container creates a `javax.servlet.ServletContextAttributeEvent` object that is input for calls to `ServletContextAttributeListener` methods. The `ServletContextAttributeEvent` class includes the following methods, which your listener can call:

- `String getName()`

  Use this to get the name of the attribute that was added, removed, or replaced.

- `Object getValue()`

  Use this to get the value of the attribute that was added, removed, or replaced. In the case of an attribute that was replaced, this method returns the old value, not the new value.

HttpSessionListener Methods, HttpSessionEvent Class

The `HttpSessionListener` interface specifies the following methods:

- `void sessionCreated(HttpSessionEvent hse)`

  The servlet container calls this method to notify the listener that a session was created.

- `void sessionDestroyed(HttpSessionEvent hse)`

  The servlet container calls this method to notify the listener that a session was destroyed.

The container creates a `javax.servlet.http.HttpSessionEvent` object that is input for calls to `HttpSessionListener` methods. The `HttpSessionEvent` class includes the following method, which your listener can call:

- `HttpSession getSession()`

  Use this to retrieve the session object that was created or destroyed, from which you can obtain information as desired. See "Introduction to Servlet Sessions" on page 1-7 for information about the `javax.servlet.http.HttpSession` interface.

HttpSessionAttributeListener Methods, HttpSessionBindingEvent Class

The `HttpSessionAttributeListener` interface specifies the following methods:

- `void attributeAdded(HttpSessionBindingEvent hsbe)`

  The servlet container calls this method to notify the listener that an attribute was added to the session.

- `void attributeRemoved(HttpSessionBindingEvent hsbe)`

  The servlet container calls this method to notify the listener that an attribute was removed from the session.

- `void attributeReplaced(HttpSessionBindingEvent hsbe)`

  The servlet container calls this method to notify the listener that an attribute was replaced in the session.

The container creates a `javax.servlet.http.HttpSessionBindingEvent` object that is input for calls to `HttpSessionAttributeListener` methods.

The `HttpSessionBindingEvent` class includes the following methods, which your listener can call:

- `String getName()`

  Use this to get the name of the attribute that was added, removed, or replaced.

- `Object getValue()`

  Use this to get the value of the attribute that was added, removed, or replaced. In the case of an attribute that was replaced, this method returns the old value, not the new value.

- `HttpSession getSession()`

  Use this to retrieve the session object that had the attribute change.

Event Listener Sample

This section provides code for a sample that uses a servlet context lifecycle and session lifecycle event listener. This includes the following components:

- `SessionLifeCycleEventExample`—This is the event listener class, implementing the `ServletContextListener` and `HttpSessionListener` interfaces.
- `SessionCreateServlet`—This servlet creates an HTTP session.
- `SessionDestroyServlet`—This servlet destroys an HTTP session.
- `index.jsp`—This is the application welcome page (the user interface), from which you can invoke `SessionCreateServlet` or `SessionDestroyServlet`.
- `web.xml`—This is the deployment descriptor, where the servlets and listener class are declared.

To download and run this application, go to the following link:

```
http://otn.oracle.com/tech/java/oc4j/htdocs/oc4j-how-to.html
```

If you do not already have a complimentary Oracle Technology Network membership, select the membership link at the following address:

```
http://otn.oracle.com/
```

Welcome Page—index.jsp

Here is the welcome page, the user interface that enables you to invoke the session-creation servlet by selecting the **Create New Session** link, or the session-destruction servlet by selecting the **Destroy Current Session** link.

```
<%@page session="false" %>
<H2>OC4J - HttpSession Event Listeners </H2>
<P>
This example demonstrates the use of the HttpSession Event and Listener that is
new with the Java Servlet 2.3 API.
</P>
<P>
[<a href="servlet/SessionCreateServlet">Create New Session</A>] &nbsp;
[<a href="servlet/SessionDestroyServlet">Destroy Current Session</A>]
</P>
<P>
Click the Create link above to start a new HttpSession. An HttpSession
listener has been configured for this application. The servler container
will send an event to this listener when a new session is created or
destroyed. The output from the event listener will be visible in the
console window from where OC4J was started.
</P>
```

Deployment Descriptor—web.xml

The servlets and the event listener are declared in the `web.xml` file. This results in `SessionLifeCycleEventExample` being instantiated and registered upon application startup. Because of this, its methods are automatically called by the servlet container, as appropriate, upon the occurrence of servlet context or session lifecycle events. Here are the `web.xml` entries:

```
<web-app>
   <listener>
      <listener-class>SessionLifeCycleEventExample</listener-class>
   </listener>
   <servlet>
      <servlet-name>sessioncreate</servlet-name>
      <servlet-class>SessionCreateServlet</servlet-class>
   </servlet>
   <servlet>
      <servlet-name>sessiondestroy</servlet-name>
      <servlet-class>SessionDestroyServlet</servlet-class>
   </servlet>
   <welcome-file-list>
```

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

Listener Class—SessionLifeCycleEventExample

This is the listener class. Its `sessionCreated()` method is called by the servlet container when an HTTP session is created, which occurs when you select the **Create New Session** link in `index.jsp`. When `sessionCreated()` is called, it calls the `log()` method to print a "CREATE" message indicating the ID of the new session.

The `sessionDestroyed()` method is called when the HTTP session is destroyed, which occurs when you select **Destroy Current Session**. When `sessionDestroyed()` is called, it calls the `log()` method to print a "DESTROY" message indicating the ID and duration of the terminated session.

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

public class SessionLifeCycleEventExample
   implements ServletContextListener, HttpSessionListener
{
   ServletContext servletContext;

   /* Methods from the ServletContextListener */
   public void contextInitialized(ServletContextEvent sce)
   {
      servletContext = sce.getServletContext();
   }

   public void contextDestroyed(ServletContextEvent sce)
   {
   }

   /* Methods for the HttpSessionListener */
   public void sessionCreated(HttpSessionEvent hse)
   {
      log("CREATE",hse);
   }
   public void sessionDestroyed(HttpSessionEvent hse)
   {

         HttpSession _session = hse.getSession();
         long _start = _session.getCreationTime();
         long _end = _session.getLastAccessedTime();
         String _counter = (String)_session.getAttribute("counter");
         log("DESTROY, Session Duration:"
                   + (_end - _start) + "(ms) Counter:" + _counter, hse);
   }

   protected void log(String msg, HttpSessionEvent hse)
   {
      String _ID = hse.getSession().getId();
      log("SessionID:" + _ID + "    " + msg);
   }

   protected void log(String msg)
   {
      System.out.println("[" + getClass().getName() + "] " + msg);
   }
}
```

Session Creation Servlet—SessionCreateServlet.java

This servlet is invoked when you select the **Create New Session** link in `index.jsp`. Its invocation results in a request object and associated session object being created by the servlet container. Creation of the session object results in the servlet container calling the `sessionCreated()` method of the event listener class.

```
import java.io.*;
import java.util.Enumeration;
import java.util.Date;

import javax.servlet.*;
import javax.servlet.http.*;


public class SessionCreateServlet extends HttpServlet {

    public void doGet (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
      {
         //Get the session object
         HttpSession session = req.getSession(true);

        // set content type and other response header fields first
```

```
         res.setContentType("text/html");

         // then write the data of the response
         PrintWriter out = res.getWriter();

         String _sval = (String)session.getAttribute("counter");
         int _counter=1;

         if(_sval!=null)
         {
            _counter=Integer.parseInt(_sval);
            _counter++;
       }


       session.setAttribute("counter",String.valueOf(_counter));

       out.println("<HEAD><TITLE> " + "Session Created Successfully ..
          Look at OC4J Console to see whether the HttpSessionEvent invoked "
          + "</TITLE></HEAD><BODY>");
       out.println("<P>[<A HREF=\"SessionCreateServlet\">Reload</A>]&nbsp;");
       out.println("[<A HREF=\"SessionDestroyServlet\">Destroy Session</A>]");
       out.println("<h2>Session created Successfully</h2>");
       out.println("Look at the OC4J Console to see whether the HttpSessionEvent
                   was invoked");
       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("<BR>Number of Accesses: " + session.getAttribute("counter"));

   }
}
```

Session Destruction Servlet—SessionDestroyServlet.java

This servlet is invoked when you select the **Destroy Current Session** link in `index.jsp`. Its invocation results in a call to the `invalidate()` method of the session object. This in turn results in the servlet container calling the `sessionDestroyed()` method of the event listener class.

```
import java.io.*;
import java.util.Enumeration;

import javax.servlet.*;
import javax.servlet.http.*;

public class SessionDestroyServlet extends HttpServlet {

    public void doGet (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
      {
         //Get the session object
         HttpSession session = req.getSession(true);
         // Invalidate Session
         session.invalidate();

         // set content type and other response header fields first
         res.setContentType("text/html");

         // then write the data of the response
         PrintWriter out = res.getWriter();

         out.println("<HEAD><TITLE> " + "Session Destroyed Successfully ..
           Look at OC4J Console to see whether the HttpSessionEvent invoked "
           + "</TITLE></HEAD><BODY>");
         out.println("<P>[<A HREF=\"../index.jsp\">Restart</A>]");
         out.println("<h2> Session Destroyed Successfully</h2>");
         out.println("Look at the OC4J Console to see whether the
                      HttpSessionEvent was invoked");
         out.close();
   }
}
```

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

The Apache Software License

Apache JServ

Apache JServ Public License

Index