Oracle® Application Server Containers for J2EE

Servlet Developer's Guide

10*g* Release 2 (10.1.2)

Part No. B14017-01

November 2004

Oracle Application Server Containers for J2EE Servlet Developer’s Guide, 10*g* Release 2 (10.1.2)

Part No. B14017-01

Primary Author: Brian Wright

Contributing Author: Tim Smith

Contributors: Bryan Atsatt, Ashok Banerjee, Bill Bishop, Olivier Caudron, Cania Chung, Olaf Heimburger, Gerald Ingalls, James Kirsch, Sunil Kunisetty, Philippe Le Mouel, David Leibs, Sastry Malladi, Jasen Minton, Debu Panda, Lenny Phan, Shiva Prasad, Paolo Ramasso, Charlie Shapiro, JJ Snyder, Joyce Yang, Serge Zloto, Sheryl Maring, Ellen Siegal

Contents

Send Us Your Comments

Oracle Application Server Containers for J2EE Servlet Developer's Guide, 10*g* Release 2 (10.1.2)

Part No. B14017-01

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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

- Electronic mail: appserverdocs_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 electronic mail address (optional).

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

Preface

This document introduces and explains the Oracle implementation of Java servlet technology, specified by an industry consortium led by Sun Microsystems. It summarizes standard features and covers Oracle implementation details and value-added features. The discussion includes basic servlets, data-access servlets, and servlet filters and event listeners.

Servlet technology is a component of the standard Java 2 Enterprise Edition (J2EE). The J2EE component of the Oracle Application Server is known as the Oracle Application Server Containers for J2EE (OC4J).

The OC4J servlet container in Oracle Application Server 10*g* Release 2 (10.1.2) is a complete implementation of the Sun Microsystems *Java Servlet Specification, Version 2.3*.

This preface contains the following sections:

- Intended Audience
- Documentation Accessibility
- Structure
- Related Documents
- Conventions

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 current version of the *Java Servlet Specification*, produced by Sun Microsystems. 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).

If you are developing applications that primarily use JavaServer Pages, refer to the *Oracle Application Server 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 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 does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Structure

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, a summary of servlet SSL features, and related examples. This chapter also introduces the OC4J standalone environment for the development stages.

Chapter 3, "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 were introduced in version 2.3 of the servlet specification.

Chapter 4, "JDBC and EJB Calls from Servlets"

Provides examples for using JDBC calls and EJB calls from servlets.

Chapter 5, "Deployment and Configuration Overview"

Discusses how to build and deploy a Web application in OC4J, and provides an overview of files for servlet and Web site configuration. This chapter is primarily useful for OC4J standalone users but also considers Oracle Application Server.

Chapter 6, "Configuration File Descriptions"

Documents all the elements and attributes of the `global-web-application.xml` and `orion-web.xml` files for servlet configuration, and the `default-web-site.xml` file (or other Web site XML files) for Web site configuration. This level of detail is primarily useful for OC4J standalone users.

Chapter 7, "Configuration with Enterprise Manager"

Shows and describes Oracle Enterprise Manager 10*g* pages for servlet and Web site configuration for deployment to an Oracle Application Server environment.

Appendix A, "Open Source Frameworks and Utilities"

Provides instructions for an OC4J standalone environment for installing and running open source framework utilities you can employ with OC4J. For the OC4J 10.1.2 implementation, this consists of Struts and log4j from the Apache Jakarta Project.

Appendix B, "Third-Party Licenses"

Contains the Third-Party License for third-party products included with Oracle Application Server and discussed in this document.

Related Documents

For more information, see the following Oracle resources.

Additional OC4J documents:

- *Oracle Application Server Containers for J2EE Servlet Developer's Guide*

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

- *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide*

  This book provides information about JavaServer Pages development and the JSP implementation and container in OC4J. This includes discussion of Oracle features such as the command-line translator and OC4J-specific configuration parameters.

- *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information as well as detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J. There is also a summary of tag libraries from other Oracle product groups.

- *Oracle Application Server Containers for J2EE Services Guide*

  This book provides information about standards-based Java services supplied with OC4J, such as JTA, JNDI, JMS, JAAS, and the Oracle Application Server Java Object Cache.

- *Oracle Application Server Containers for J2EE Security Guide*

  This document (not to be confused with the *Oracle Application Server 10g Security Guide*) describes security features and implementations particular to OC4J. This includes information about using JAAS, the Java Authentication and Authorization Service, as well as other Java security technologies.

- *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*

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

Oracle Application Server TopLink documents:

- *Oracle Application Server TopLink Getting Started Guide*
- *Oracle Application Server TopLink Mapping Workbench User's Guide*
- *Oracle Application Server TopLink Application Developer's Guide*

Java-related documents for Oracle Database:

- *Oracle Database Java Developer's Guide*
- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Database JPublisher User's Guide*

Additional Oracle Application Server documents:

- *Oracle Application Server Administrator's Guide*
- *Oracle Application Server Security Guide*
- *Oracle Application Server Certificate Authority Administrator's Guide*
- *Oracle Application Server Performance Guide*
- *Oracle Enterprise Manager Concepts*
- *Oracle HTTP Server Administrator's Guide*
- *Oracle Application Server Globalization Guide*
- *Oracle Application Server Web Cache Administrator's Guide*
- *Oracle Application Server Web Services Developer's Guide*
- *Oracle Application Server Upgrading to 10g Release 2 (10.1.2)*

Oracle JDeveloper documentation:

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

  `http://www.oracle.com/technology/products/jdev/content.html`

Additional Oracle Database documents:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML API Reference*
- *Oracle Database Application Developer's Guide - Fundamentals*
- *PL/SQL Packages and Types Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle Database SQL Reference*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle Database Reference*

Printed documentation is available for sale in the Oracle Store at:

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

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

`http://www.oracle.com/technology/membership/`

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://www.oracle.com/technology/documentation`

The following OTN Web site for Java servlets and JavaServer Pages is also available:

`http://www.oracle.com/technology/tech/java/servlets/`

For further servlet information, refer to the *Java Servlet Specification, Version 2.3* at the following location:

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

Resources 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`

- The servlet API Javadoc:

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

Conventions

The following conventions are used in this manual:

Convention	Meaning
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.
Boldface text	Boldface type in text indicates a GUI component such as a link or button to click.
Italics	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.
`Monospace (fixed-width) font`	Monospace typeface within text indicates items such as executables, file names, directory names, Java class names, Java method names, variable names, other programmatic elements (such as JSP tags or attributes, or XML elements or attributes), or database SQL commands or elements (such as schema names, table names, or column names).
`Italic monospace (fixed-width) font`	Italic monospace font represents placeholders or variables.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
|	A vertical bar represents a choice of two or more options. Enter one of the options. Do not enter the vertical bar.

1 Servlet Overview

Oracle Application Server Containers for J2EE (OC4J) enables you to develop and deploy standard J2EE-compliant applications. Applications are packaged in standard Enterprise archive (EAR) deployment files, which include standard Web archive (WAR) files to deploy the Web modules, and Java archive (JAR) files for any Enterprise JavaBeans (EJB) and application client modules in the application.

With Oracle Application Server 10*g* Release 2 (10.1.2), OC4J complies with *Java 2 Platform Enterprise Edition Specification, v1.3*, including full compliance with the Sun Microsystems *Java Servlet Specification, Version 2.3* in the OC4J servlet container. (Any mention of the servlet specification in this manual refers to this version unless otherwise noted.)

The most important concepts to understand about servlet development under OC4J are how a 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 5, "Deployment and Configuration Overview", 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 contains the following sections:

- Introduction to Servlets
- A First Servlet Example

Note: Sample servlet applications are included in the OC4J demos, available from the following location on the Oracle Technology Network (requiring an OTN membership, which is free of charge):

`http://www.oracle.com/technology/tech/java/oc4j/demos/`

Introduction to Servlets

The following sections offer a brief introduction to servlet technology:

- Review of Servlet Technology
- Advantages of Servlets
- The Servlet Interface and Request and Response Objects
- Servlets and the Servlet Container

- Introduction to Servlet Sessions
- Introduction to Servlet Contexts
- Introduction to Servlet Configuration Objects
- Introduction to Servlet Filters
- Introduction to Event Listeners
- JSP Pages and Other J2EE Component Types

Note: The terms *Web module* and *Web application* are interchangeable in most uses and are both used throughout this document. If there is a distinction, it is that "Web module" typically indicates a single component, whether or not it composes an independent application, while "Web application" typically indicates a working application that may consist of multiple modules or components.

Review of Servlet Technology

In recent years, servlet technology has emerged as a powerful way to extend Web server functionality through dynamic Web pages. A servlet is a Java program that runs in a Web server, as opposed to an applet that runs in a client browser. Typically, 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. Alternatively, the servlet can be accessed directly from another application component or send its output to another component. Most servlets generate HTML text, but a servlet may instead generate XML to encapsulate data.

More specifically, 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 JavaServer Pages (JSP) and EJB modules, which are also server-side J2EE component types. These are used in conjunction with client-side components such as applets (part of the Java 2 Platform, Standard Edition specification) and application client programs. An application may consist of any number of any of these components.

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.

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 of running in the server is that the server is usually a robust machine with many resources, making the program more scalable. Running in the server also results in more direct access to the data. The Web server in which a servlet is running is on the same side of the network firewall as the data being accessed.

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. After 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 security, robustness, object orientation, and platform independence.
- Servlets are fully integrated with the Java language and its standard APIs, such as JDBC for Java database connectivity.
- Servlets are fully integrated into the J2EE framework, which provides an extensive set of services that your Web application can use, such as Java Naming and Directory Interface (JNDI) for component naming and lookup, Java Transaction API (JTA) for managing transactions, Java Authentication and Authorization Service (JAAS) for security, Remote Method Invocation (RMI) for distributed applications, and Java Message Service (JMS). The following Web site contains information about the J2EE framework and services:

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

- 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. In addition, servlets can optionally be loaded when OC4J starts, so that any initialization is handled in advance instead of at the first user request. See "Servlet Preloading" on page 2-6.
- The servlet request and response objects offer 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 (JVM) 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(...)`: Initialize the servlet.
- `destroy(...)`: Terminate the servlet.
- `doGet(...)`: Execute an HTTP `GET` request.
- `doPost(...)`: Execute an HTTP `POST` request.
- `doPut(...)`: Execute an HTTP `PUT` request.
- `doDelete(...)`: Execute an HTTP `DELETE` request.
- `service(...)`: Receive HTTP requests and, by default, dispatch them to the appropriate `do`*`XXX`*`()` methods.

- `getServletInfo(...)`: Retrieve information about the servlet.

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, `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, a servlet has no static `main()` method. Therefore, a servlet must execute under the control of an external container.

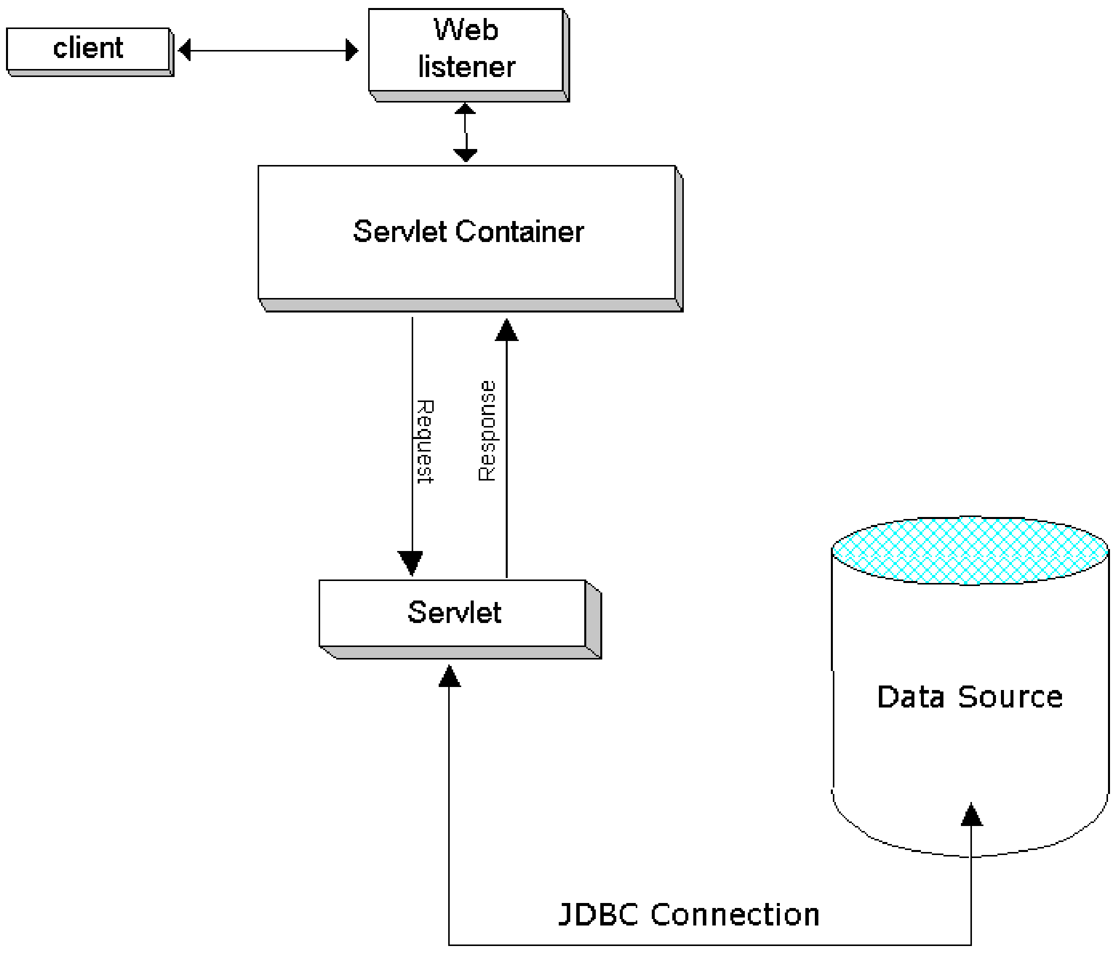
Servlet containers, sometimes referred to as *servlet engines*, execute and manage servlets. The servlet container calls servlet methods and provides services that the servlet needs while executing. A servlet container is usually written in Java and is either part of a Web server (if the Web server is also written in Java) or is otherwise associated with and used by a Web server. OC4J includes a fully standards-compliant servlet container.

The servlet container provides the servlet with easy access to properties of the HTTP request, such as its headers and parameters. When a servlet is called, such as when it 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 Administrator's 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 2-25 for more information and examples.

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.

OC4J supports the following session-tracking mechanisms. See "Session Tracking" on page 2-25 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 process 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 specification.

- URL rewriting

  Instead of using cookies, 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 process allows the request to be associated with the session. This is the most frequently used mechanism for situations in which clients do not accept cookies.

Introduction to the HttpSession Interface

In the standard servlet API, each client session 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 client if one does not already exist within the application.

See "Features of the HttpSession Interface" on page 2-28 for more information.

Introduction to Servlet Contexts

A *servlet context* is used to maintain 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). There is one servlet context for each Web application running within a given JVM; this is always a one-to-one correspondence. You can think of a servlet context as a container for a specific application.

Servlet Context Basics

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 provides the scope for the running instances of the application. Through this mechanism, each application is loaded from a distinct classloader 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 version 2.2 of the servlet specification, 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.)

How to Obtain a Servlet Context

Use the `getServletContext()` method of a servlet configuration object to retrieve a servlet context. See "Introduction to Servlet Configuration Objects" on page 1-8.

Servlet Context Methods

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, 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 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.

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

- `RequestDispatcher getRequestDispatcher(String path)`

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

- `String getRealPath(String path)`

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

- `URL getResource(String path)`

  This method 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 method 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 Configuration Objects

A *servlet configuration object* contains initialization and startup parameters for a servlet and is an instance of a class that implements the `javax.servlet.ServletConfig` interface. Such a class is provided with any J2EE-compliant Web server.

You can retrieve a servlet configuration object for a servlet by calling the `getServletConfig()` method of the servlet. This method is specified in the `javax.servlet.Servlet` interface, with a default implementation in the `javax.servlet.http.HttpServlet` class.

The `ServletConfig` interface specifies the following methods:

- `ServletContext getServletContext()`

  Retrieve a servlet context for the application. See "Introduction to Servlet Contexts" on page 1-6.

- `String getServletName()`

  Retrieve the name of the servlet.

- `Enumeration getInitParameterNames()`

  Retrieve the names of the initialization parameters of the servlet, if any. The names are returned in a `java.util.Enumeration` instance of `String` objects. (The `Enumeration` instance is empty if there are no initialization parameters.)

- `String getInitParameter(String name)`

  This returns a `String` object containing the value of the specified initialization parameter, or `null` if there is no parameter by that name.

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 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 3-1 for more information.

Introduction to Event Listeners

The servlet 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, you can implement standard interfaces 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 3-11 for more information.

JSP Pages and Other J2EE Component Types

In addition to servlets, an application may include other server-side components, such as JSP pages and EJBs. It is especially common for servlets to be used in combination with JSP pages in a Web application. 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 and are also essentially servlets.

Note: Wherever this manual mentions functionality that applies to servlets, you can assume it applies to JSP pages as well unless stated otherwise.

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

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

A First Servlet Example

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

Hello World Code

This servlet prints "Hi There!" back to the client. The comments note some of the basic aspects of writing a servlet.

```
// You must import at least the following packages for any servlet you write.
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet, the base servlet implementation.
public class HelloServlet extends HttpServlet {

  // Override the base implementation of doGet(), as desired.
  public void doGet (HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    // Set the MIME type for the response content.
    resp.setContentType("text/html");

    // Get an output stream to use in sending the output to the client.
    ServletOutputStream out = resp.getOutputStream();
    // Put together the HTML code for the output.
    out.println("<html>");
    out.println("<head><title>Hello World</title></head>");
    out.println("<body>");
    out.println("<h1>Hi There!</h1>");
    out.println("</body></html>");
  }
}
```

Compiling and Deploying the Servlet

To try out the sample servlet code in an OC4J standalone environment, save it as `HelloServlet.java` in the `/WEB-INF/classes` directory of the OC4J default Web application. (See "OC4J Default Application and Default Web Application" on page 5-25.)

Next, compile the servlet. First verify that `servlet.jar`, supplied with OC4J, is in your classpath. This contains the Sun Microsystems `javax.servlet` and `javax.servlet.http` packages.

Note: 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 the appropriate 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 6-1 for more information about `development` and `source-directory`.

Running the Servlet

Assuming that the OC4J server is up and running and that invocation by class name is enabled with the `servlet-webdir` built-in default setting of "`/servlet/`", 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/HelloServlet
```

(See "Servlet Invocation by Class Name During OC4J Development" on page 2-22 for information about invocation by class name and about the OC4J `servlet-webdir` attribute.)

In an OC4J standalone environment, use port 8888 to access the OC4J Web listener directly. (See "OC4J Standalone for Development" on page 2-1 for an overview.)

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

Important: The way of invoking servlets that is shown here invokes directly by class name. This is suitable for a development environment but presents a significant security risk. Do not configure OC4J to operate in this mode in a production environment. See "Servlet Invocation by Class Name During OC4J Development" on page 2-22 and "Additional Security Considerations" on page 2-40 for more information.

---

2 Servlet Development

This chapter, consisting of the following sections, provides basic information for developing servlets for OC4J and the Oracle Application Server. The first section highlights the use of the standalone version of OC4J for convenience during your development and testing phases.

OC4J Standalone for Development

This manual assumes you are using an *OC4J standalone* environment for at least your initial development phases. This term refers to the use of a single OC4J instance outside the Oracle Application Server environment and Oracle Enterprise Manager 10*g*. Using OC4J standalone is typically more convenient for early development.

The following sections provide some overview and key considerations:

- Overview: Using OC4J Standalone
- Key OC4J Flags for Development
- Removal of tools.jar from OC4J Standalone

To obtain OC4J standalone, download the `oc4j_extended.zip` file from the Oracle Technology Network (OTN) at the following location:

`http://www.oracle.com/technology/tech/java/oc4j/content.html`

Notes:

- To use OC4J standalone, you must have a supported version of the Sun Microsystems JDK installed. A JDK is not provided with the OC4J standalone product.
- During development, also consider the Oracle JDeveloper visual development tool for development and deployment. This tool offers many conveniences, as described in "Oracle JDeveloper Support for Servlet Development" on page 2-19.

Overview: Using OC4J Standalone

You can start, manage, and control standalone OC4J instances through `oc4j.jar` (the OC4J standalone executable) and the `admin.jar` command-line utility, provided with the standalone product. Deploying an EAR file and binding its Web module through `admin.jar` result in automatic updates to key configuration files.

Note: Key aspects of the `admin.jar` utility are covered in Chapter 5, "Deployment and Configuration Overview", particularly in "Deploying an EAR File to OC4J Standalone" on page 5-27. For further information, see the *Oracle Application Server Containers for J2EE Stand Alone User's Guide.*

During testing, it is also possible to manually install an EAR file or individual files according to the J2EE directory structure, and to complete the process by manually updating key configuration files, which triggers OC4J to unpack and deploy the application.

If you have an independent Web application, you can deploy it as a WAR file (or as a directory structure) within the OC4J default J2EE application, rather than using an EAR file.

In addition, for a convenient testing mode, you can deploy individual servlets or JSP pages to the OC4J default Web application.

An OC4J standalone environment, by default, includes the following key directories:

- J2EE home: `j2ee/home`, relative to where you install OC4J
- Global configuration files directory: `j2ee/home/config`
- Default Web application root directory: `j2ee/home/default-web-app`
- Root target directory for deployed applications: `j2ee/home/applications`
- Root target directory for deployment descriptors (such as `orion-web.xml` and `orion-application.xml`): `j2ee/home/application-deployments`

In the simplest case, deploying a test servlet to the OC4J default Web application consists of placing the class file under the `/WEB-INF/classes` directory under the default Web application root directory.

Chapter 5 discussed more detailed deployment considerations, primarily targeting OC4J standalone users. See the following sections in particular:

- "General Overview of OC4J Deployment and Configuration" on page 5-1
- "OC4J Default Application and Default Web Application" on page 5-25

- "Deployment Scenarios to OC4J Standalone" on page 5-23

Additionally, for information about invoking a servlet in OC4J standalone, see "Servlet Invocation in an OC4J Standalone Environment" on page 2-25.

For detailed information about `admin.jar` and about how to start, stop, configure, and manage your standalone process, download the *Oracle Application Server Containers for J2EE Stand Alone User's Guide* when you download `OC4J_extended.zip`.

Key OC4J Flags for Development

There are several OC4J flags to be aware of during your development stages, presumably while using OC4J standalone. Note that these flags work independently of each other.

- OC4J `check-for-updates` flag

  In the OC4J `server.xml` file, the top-level `<application-server>` element includes a `check-for-updates` attribute that determines whether the OC4J task manager automatically checks for updates to XML configuration files (including `server.xml` itself), library JAR files, and JSP tag libraries. This is often referred to as OC4J *polling*. The default setting, for use during development, is "`true`". You can disable polling as follows:

  ```
  <application-server ... check-for-updates="false" ... >
     ...
  </application-server>
  ```

  For example, during manual operations (considered "expert modes") during development, you can install your application by hand, then manually update the `server.xml` file, global `application.xml` file, and `http-web-site.xml` file as appropriate to define and bind your Web application. With the default `check-for-updates="true"` setting, OC4J automatically detects the changes and deploys your application (unpacking the EAR or WAR file in the process, if applicable).

  See "OC4J Top-Level Server Configuration File: server.xml" on page 5-9 for more information about this file.

  Important: The `check-for-updates` flag is used only in OC4J standalone. It is disregarded in an Oracle Application Server environment, in which the Oracle Process Management and Notification system (OPMN) and Distributed Configuration Management subsystem (DCM) manage the OC4J file update facilities.

- `admin.jar -updateConfig` option

  If you manually update OC4J XML configuration files while `check-for-udpates="false"`, you can run the `admin.jar` utility with the `-updateConfig` option to trigger a one-time check for updates:

  ```
  % java -jar admin.jar -updateConfig
  ```

Important: If you want to re-enable checking after it had been disabled, you must use the `admin.jar -updateConfig` option after setting `check-for-udpates="true"`, so that OC4J notices this change. After that, automatic checking will be enabled again.

- Servlet `development` flag

  For convenience during development and testing, use the `development="true"` setting in the `<orion-web-app>` element of the `global-web-application.xml` file or `orion-web.xml` file. With this setting, whenever you update the servlet code under a particular directory—typically a `/WEB-INF/classes` directory, or according to the `source-directory` attribute of `<orion-web-app>`—the servlet is recompiled and redeployed automatically the next time it is invoked. See "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1 for more information about the `development` and `source-directory` attributes.

- JSP `main_mode` flag

  This flag directs the mode of operation of the JSP container, particularly for automatic retranslation of JSP pages and reloading of JSP-generated Java classes that have changed. During development, use the `recompile` (default) setting to check timestamps of JSP pages and to retranslate and reload them if they have been modified since they were last loaded. (Use the `justrun` setting to not check any timestamps, such as for production mode.) See the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for more information about this flag and how to set it.

Removal of tools.jar from OC4J Standalone

The OC4J 9.0.3 standalone implementation provided the `tools.jar` file from the Sun Microsystems JDK 1.3.1. This file includes the `java` front-end executable and `javac` compiler executable, for example, among many other components.

The OC4J 10.1.2 standalone implementation no longer provides the `tools.jar` file. Therefore, you must install a JDK that OC4J supports before installing OC4J itself. The JDK versions that OC4J supports for the OC4J 10.1.2 implementation are JDK 1.3.1 (for OC4J standalone only) and JDK 1.4. Oracle Application Server 10*g* Release 2 (10.1.2) includes JDK 1.4, so you should typically use this JDK version for OC4J standalone as well. However, there are migration issues to consider, particularly the JDK 1.4 requirement that all invoked classes must be in packages. See "JDK 1.4 Considerations: Cannot Invoke Classes Not in Packages" on page 2-13.

Notes: OC4J standalone uses `javac` from the same directory in which `java` is accessed through the command "`java -jar oc4j.jar`", ensuring use of the appropriate `javac` version.

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 down the servlet. Most servlets override either the `doGet()` method or the `doPost()` method of

HttpServlet, to handle HTTP GET or POST requests appropriately. 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 1-9.

The following sections cover features and issues to consider before developing your applications:

- Sample Code Template
- Servlet Lifecycle
- Servlet Preloading
- Servlet Classloading and Application Redeployment
- Servlet Information Exchange
- Servlet Includes and Forwards
- Servlet Thread Models and Related Considerations
- Servlet Performance and Monitoring
- JDK 1.4 Considerations: Cannot Invoke Classes Not in Packages

Sample Code Template

Here is a sample 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.";
  }
```

You can optionally override the init(), destroy(), and getServletInfo() methods, but 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 servlet configuration object and storing the values
- 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 Lifecycle

Servlets have a predictable and manageable lifecycle:

- When the servlet is loaded, its configuration details are read from `web.xml`. These details can include initialization parameters.
- There is only one instance of a servlet, unless the single-threaded model is used. See "Servlet Thread Models and Related Considerations" on page 2-11.
- 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 3-1 for more information.
- A servlet can forward requests to other servlets or include output from other servlets. See "Servlet Includes and Forwards" on page 2-10.
- Responses come back to the client through response objects, which the container passes back to the client in HTTP response headers. Servlets can write to a response object by using a `java.io.PrintWriter` or `javax.servlet.ServletOutputStream` object.
- The container calls the `destroy()` method before the servlet is unloaded.

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 `server.xml` file, 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 application is deployed or redeployed.

Preloading requires the following steps:

1. Verify that the relevant `<application>` element in the `server.xml` file has the attribute setting `auto-start="true"`. OC4J inserts this setting by default when you deploy an application.
2. Specify the attribute setting `load-on-startup="true"` in the relevant `<web-app>` subelement of the `<web-site>` element of the Web site XML file. See "Configuration for Web Site XML Files" on page 6-20 for information about the elements and attributes of Web site XML files.
3. For any servlet that 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.

Note: OC4J supports the specification of *startup classes* and *shutdown classes*. Startup classes are designated through the `<startup-classes>` element of the `server.xml` file and are called immediately after OC4J initializes. Shutdown classes are designated through the `<shutdown-classes>` element of `server.xml` and are called immediately before OC4J terminates.

Be aware that startup classes are called before any preloaded servlets.

See the *Oracle Application Server Containers for J2EE User’s Guide* for information about startup classes and shutdown classes.

Servlet Classloading and Application Redeployment

The following sections describe OC4J features and some important considerations regarding servlet classloading and application loading:

- OC4J Web Application Redeployment and Class Reloading Features
- Loading WAR File Classes Before System Classes in OC4J
- Sharing Cached Java Objects Across OC4J Servlets in Oracle Application Server

OC4J Web Application Redeployment and Class Reloading Features

In OC4J, any of the following circumstances, depending on OC4J polling, results in redeployment of a Web application and, upon request, reloading of servlet classes and any dependency classes.

Notes:

- "OC4J polling" refers to the automatic checking of library JAR files and XML configuration files by the OC4J task manager for updates. In an Oracle Application Server environment, this is controlled by OPMN and DCM. In OC4J standalone, it is controlled by the `server.xml` `check-for-updates` flag (set to "`true`" by default), described in "Key OC4J Flags for Development" on page 2-3.
- In this discussion, "redeployment" of a Web application refers to the process in which OC4J removes the Web application from its execution space, removes the classloader associated with execution of the Web application, reparses `web.xml` and `orion-web.xml`, and reinitializes servlet listeners, filters, and mappings.

- If a servlet `.class` file under `/WEB-INF/classes` changes, such as by recompilation, then when the servlet is next requested, the associated Web application is redeployed and the servlet class and any dependency classes are reloaded. This action does not depend on OC4J polling. Note that nothing happens until the servlet is next requested. Also note that if only *non*-servlet `.class` files under `/WEB-INF/classes` change, *nothing* is reloaded.

Note: Changing a servlet class file in a directory location specified in a `<classpath>` element in `global-web-application.xml` or `orion-web.xml` has the same effect as changing a servlet class file in `/WEB-INF/classes`. However, changing a JAR file or dependency class file (such as for a JavaBean) in a `<classpath>` location has no effect.

- If the `web.xml` file changes, or a library JAR file in `/WEB-INF/lib` changes, and OC4J polling is enabled, then the associated Web application is redeployed the next time the OC4J task manager runs, which by default is once each second. Any servlet class in the Web application and any dependency classes are reloaded upon the next request for the servlet. Alternatively, if polling is not enabled, you can trigger one-time polling and the resulting redeployment and reloading by using the `admin.jar -updateConfig` option.

Be aware of the following important considerations:

- In the preceding scenarios, a servlet and its dependency classes are reloaded immediately, instead of upon next request, if the servlet is set to be preloaded. This is according to `load-on-startup` settings. See "Servlet Preloading" on page 2-6.
- OC4J servlet reloading functionality does not extend to JSP page implementation classes. Changing a JSP page implementation `.class` file does *not* result in any reloading. JSP recompilation and reloading behavior is determined by the JSP `main_mode` flag, as described in the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide.*
- Classes in Web modules of a parent application are not visible to child applications, although other classes of the parent application (such as EJBs, for example) *are* visible.

- You can use `<library>` elements in the global `application.xml` file or `server.xml` file to specify directories or JAR or ZIP files for shared code. Upon startup, OC4J loads all classes in any JAR or ZIP file specified in any `<library>` element, and all classes in any JAR or ZIP file in any directory specified in any `<library>` element.

  To avoid unnecessary overhead, you should use `<library>` elements somewhat sparingly, specify particular JAR or ZIP files instead of entire directories wherever possible, and, where directories are specified, minimize the number of JAR or ZIP files in those directories.

  By default, `application.xml` includes a `<library>` element for the `j2ee/home/applib` directory.

Note: Changing a JAR or ZIP file specified in a `<library>` element, or a JAR or ZIP file in a directory specified in a `<library>` element, does *not* by itself result in redeployment of any Web applications or reloading of classes. The OC4J task manager does not poll these shared library locations.

Loading WAR File Classes Before System Classes in OC4J

The servlet specification recommends, but does not require, loading *local classes*, which are classes in the WAR file, before *system classes*, which are any other classes in the environment. Note that "classes in the WAR file" may include classes from the WAR file manifest classpath. 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:

- `search-local-classes-first`: Set this to "`true`" to search and load WAR file classes before system classes. The default setting is "`false`".
- `include-war-manifest-class-path`: Set this to "`false`" to *not* include the classpath 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 setting is "`true`".

Notes:

- If both attributes are set to "`true`", the overall classpath is constructed so that classes physically residing in the WAR file are loaded prior to any classes from the WAR file manifest classpath. So you can assume that in the event of any conflict, classes physically residing in the WAR file will take precedence.
- For compliance with the servlet specification, you cannot use `search-local-classes-first` functionality in loading classes in `java.*` or `javax.*` packages.

Also see "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1.

Sharing Cached Java Objects Across OC4J Servlets in Oracle Application Server

To take advantage of the distributed functionality of the Oracle 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 classloader; however, by default, objects loaded by a servlet are loaded by the context classloader. Objects loaded by the context classloader are visible only to the servlets within the servlet context corresponding to that classloader. The object definition will not be available to other servlets or to the cache in another JVM. If an object is loaded by the system classloader, 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, do one of the following with the `.class` file, assuming an Oracle Application Server environment:

- Copy it to the *`ORACLE_HOME`*`/javacache/cachedobjects/classes` directory.
- Add it to the *`ORACLE_HOME`*`/javacache/cachedobjects/share.jar` file.

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 Oracle Application Server Java Object Cache, see the *Oracle Application Server 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
- HTTP session object
- Servlet context object
- Sources of data outside the servlet (for example: databases, file systems, or external sensors)

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

Servlet Includes and Forwards

Many servlets use other servlets in the course of their processing, either by "including" another servlet or "forwarding" to another servlet.

In servlet terminology, a servlet *include* is the process by which a servlet includes response from another servlet within its own response. Processing and response are initially handled by the originating servlet, then are turned over to the included servlet, then revert back to the originating servlet once the included servlet is finished.

With a servlet *forward*, processing is handled by the originating servlet up to the point of the forward call, at which point the response is reset and the target servlet takes over processing of the request. When a response is reset, any HTTP header settings and any information in the output stream are cleared from the response. After a forward, the originating servlet must not attempt to set headers or write to the response. Also note that if the response has already been committed, then a servlet cannot forward to or include another servlet.

To forward to or include another servlet, you must obtain a request dispatcher for that servlet, using either of the following servlet context methods:

- `RequestDispatcher getRequestDispatcher(String path)`
- `RequestDispatcher getNamedDispatcher(String name)`

For `getRequestDispatcher()`, input the path of the target servlet. For `getNamedDispatcher()`, input the name of the target servlet, according to the `<servlet-name>` element for that servlet in the `web.xml` file.

In either case, the returned object is an instance of a class that implements the `javax.servlet.RequestDispatcher` interface. (Such a class is provided by the servlet container.) The request dispatcher is a wrapper for the target servlet. In general, the duty of a request dispatcher is to serve as an intermediary in routing requests to the resource that it wraps.

A request dispatcher has the following methods to effect any includes or forwards:

-
  ```
  void include(ServletRequest request,
               ServletResponse response)
  ```

-
  ```
  void forward(ServletRequest request,
               ServletResponse response)
  ```

As you can see, you pass in your request and response objects when you call these methods.

Notes:

- When a servlet forwards to or includes another servlet, default OC4J functionality enforces `web.xml` security constraints on the target servlet as well as the originating servlet. This does not comply with the servlet specification, but you can disable this behavior through the `<authenticate-on-dispatch>` element in `global-web-application.xml` or `orion-web.xml`. See "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1 for information about this element.
- When a servlet forwards to or includes another servlet, the default behavior is that any filters that apply to the originating servlet are *not* executed on the target servlet, but this behavior is configurable. See "Filtering of Forward or Include Targets" on page 3-2.

Servlet Thread Models and Related Considerations

For a servlet in a nondistributable environment, a servlet container uses only one servlet instance for each servlet declaration. In a distributable environment, a container uses one servlet instance for each servlet declaration in each JVM. Therefore, a servlet container, including the OC4J servlet container, generally processes concurrent requests to a servlet by using multiple threads for multiple concurrent executions of the servlet `service()` method.

Servlet developers must keep this in mind, making provisions for simultaneous processing through multiple threads and designing their servlets so that access to shared resources is somehow synchronized or coordinated. Shared resources fall into two main areas:

- In-memory data, such as instance or class variables
- External objects, such as files, database connections, and network connections

One option is to synchronize the `service()` method as a whole; however, this may adversely affect performance.

A better approach is to selectively protect instance or class fields, or access to external resources, through synchronization blocks.

As perhaps a last resort, the servlet specification supports a *single-thread model*. If a servlet implements the `javax.servlet.SingleThreadModel` interface, the servlet container must guarantee that there is never more than one request thread at a time in the `service()` method of any instance of the servlet. OC4J typically accomplishes this by creating a pool of servlet instances, with a separate instance handling each concurrent request. This process has significant performance impact on the servlet container, however, and should be avoided if at all possible. Furthermore, the `SingleThreadModel` interface will be deprecated in version 2.4 of the servlet specification.

For general information about multithreading, see the Sun Microsystems *Java Tutorial on Multithreaded Programming* at the following Web site:

`http://java.sun.com/Series/Tutorial/java/threads/multithreaded.html`

Servlet Performance and Monitoring

The following sections list servlet performance considerations and introduce the Oracle Application Server Dynamic Monitoring Service (DMS):

- General Performance Considerations
- Oracle Application Server Dynamic Monitoring Service

For general OC4J performance information, including coverage of DMS and the `dmstool` for performance metrics, refer to the *Oracle Application Server Performance Guide*.

General Performance Considerations

This section summarizes issues, mostly documented elsewhere in this manual, that may impact performance:

- Consider the optimal expiration setting for Web pages in your application. You can set the expiration for pages that match a given URL pattern, using the `<expiration-setting>` subelement of `<orion-web-app>` in `global-web-application.xml` or `orion-web.xml`. (See "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1.) A more appropriate setting decreases load on the application and improves performance.
- Be aware of performance implications relating to how multiple concurrent requests are synchronized or coordinated, and also be aware of related considerations regarding thread models. See "Servlet Thread Models and Related Considerations" on page 2-11.
- There are performance implications related to how session state is replicated in a distributable environment. Replication is triggered each time there is a `setAttribute()` call on the session object, so large numbers of such calls in a servlet may impact performance. In addition, be aware that for performance reasons, OC4J does not wait to confirm successful replication of session state. See "Session Replication in a Distributable Application" on page 2-29.
- Servlet configuration parameters can significantly affect performance. For information about the `file-modification-check-interval` attribute of the

`<orion-web-app>` element, see "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1. For information about the `use-keep-alives` attribute of the `<web-site>` element, see "Element Descriptions for Web Site XML Files" on page 6-20.

- Additional JSP-related configuration parameters can significantly affect performance. See "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1 for information about the `simple-jsp-mapping` and `enable-jsp-dispatcher-shortcut` attributes of the `<orion-web-app>` element.
- OC4J standalone supports a mode of "shared" operation for a single application through multiple Web sites, where a site is defined as a particular host and port. This feature is particularly intended for secure applications in which some but not all communications require HTTPS. Running the noncritical communications through an HTTP port improves performance. See "Element Descriptions for Web Site XML Files" on page 6-20 for information about the `shared` attribute of the `<web-app>` element.
- If you ever use OC4J standalone as a production environment (although this is not typical), remember to disable the `server.xml` `check-for-updates` flag. See "Key OC4J Flags for Development" on page 2-3.

Oracle Application Server Dynamic Monitoring Service

In an Oracle Application Server environment, DMS adds performance-monitoring features to several components, including OC4J. The goal of DMS is to provide information about runtime behavior through built-in performance measurements so that users can diagnose, analyze, and debug any performance problems. DMS provides this information in a package that can be used at any time, including during live deployment. Data are published through HTTP and can be viewed with a browser.

Standard configuration for the DMS servlets, such as the spy servlet and monitoring agent, is in the global `application.xml` file and the `default-web-site.xml` file.

In the OC4J `application.xml` file, the Web modules `dms` and `dms0` and the paths to their WAR files are specified. The `default-web-site.xml` file specifies that these Web modules are deployed to the OC4J default application and binds them to their context paths. Do not directly alter any of these DMS configurations.

Use the Oracle Enterprise Manager 10*g* to access DMS, display DMS information, and, if appropriate, alter DMS configuration.

JDK 1.4 Considerations: Cannot Invoke Classes Not in Packages

Among the migration considerations in moving to a Sun Microsystems JDK 1.4 environment, which is the environment shipped with Oracle Application Server 10*g* Release 2 (10.1.2), one is of particular importance to servlet and JSP developers.

To address security concerns and ambiguities with previous JDK versions, Sun Microsystems modified the Java compiler to reject import statements that import a type from the "unnamed namespace". So essentially, you cannot invoke a class (a method of a class) that is not within a package. Any attempt to do so will result in a fatal error at compilation time.

This issue especially affects JSP developers who invoke JavaBeans from their JSP pages, because such beans are often outside of any package (although version 2.0 of the JSP specification now requires beans to be within packages, to satisfy the new compiler requirements). Where JavaBeans outside of packages are invoked, JSP applications that were built and executed in an OC4J 9.0.3 / JDK 1.3.1 environment will no longer work in an OC4J 10.1.2 / JDK 1.4 environment.

Until you update your application so that all JavaBeans and other invoked classes are within packages, you can avoid this issue by reverting back to a JDK 1.3.1 environment for OC4J standalone. Note that JDK 1.3.x is not supported in a full Oracle Application Server 10.1.2 environment.

Notes:

- The `javac -source` compiler option allows JDK 1.3.1 code to be processed seamlessly by the JDK 1.4 compiler, but this option does not account for the "classes not in packages" issue.
- Only the JDK 1.3.1 and JDK 1.4 compilers are supported and certified by OC4J. It is possible to specify an alternative compiler by adding a `<java-compiler>` element to the `server.xml` file, and this may provide a workaround for the "classes not in packages" issue, but no other compilers are certified or supported by Oracle for use with OC4J. (Furthermore, do *not* update the `server.xml` file directly in an Oracle Application Server environment. Instead, use the Oracle Enterprise Manager 10*g*.)

For more information about the "classes not in packages" issue and other JDK 1.4 compatibility issues, refer to the following Web site:

`http://java.sun.com/j2se/1.4/compatibility.html`

In particular, click the link "Incompatibilities Between Java 2 Platform, Standard Edition, v1.4.0 and v1.3".

Additional Oracle Features

The followings sections describe additional features, mostly Oracle-specific, to consider in developing and running servlets in OC4J:

- OC4J Logging
- Servlet Debugging
- Oracle JDeveloper Support for Servlet Development
- Introduction to OC4J Support for Open Source Frameworks

OC4J Logging

The following sections provide an overview of OC4J logging features:

- OC4J Logs
- Oracle Diagnostic Logging Versus Text-Based Logging
- Additional Oracle Application Server Log Files

Note: Logging features discussed here are for log messages from the OC4J server. It is also possible to use open source frameworks and utilities with OC4J, such as those from the Apache Jakarta Project. This includes log4j, a complementary technology that you can use to insert log statements in your own code. See "Configuration and Use of Jakarta log4j in OC4J" on page A-7.

OC4J Logs

Several logs are available in OC4J. Because they are not specific to servlets, they are documented elsewhere, but this section provides a summary list and appropriate cross-references. For each log, you have the option of using text-based logging or ODL logging. (See the next section, "Oracle Diagnostic Logging Versus Text-Based Logging".) Note that for ODL, log file names always take the form `log`*`N`*`.xml`, where *`N`* is an integer. For text-based logging, you must specify the log file names.

For each log there is a configuration element in the appropriate OC4J configuration file to enable text-based logging, and a separate element to enable ODL logging. The presence of a logging configuration element enables the associated type of logging.

OC4J supports the following logs:

- Application log

  There is a log for each application deployed, as configured in `orion-application.xml`. For text-based logging, a typical name is `application.log`.

- Global application log

  There is a log for global logging for all applications, including the default application, as configured in the OC4J global `application.xml` file. For text-based logging, a typical name is `global-application.log`.

- JMS log

  There is a log for Java Message Service (JMS) functionality, as configured in `jms.xml`. For text-based logging, a typical name is `jms.log`.

- RMI log

  There is a log for remote method invocation functionality, as configured in `rmi.xml`. For text-based logging, a typical name is `rmi.log`.

- Server log

  There is a server-wide log, as configured in `server.xml`. For text-based logging, a typical name is `server.log`.

- Web site access log

  There is a Web site access log (one log file for each Web site to log all accesses of the site), as configured in the Web site XML file. For text-based logging, a typical name is `http-access.log`.

Note: For Web site access logging, you can use only one type of logging, not both.

Configuration of the Web access log is covered in this manual. Under "Element Descriptions for Web Site XML Files" on page 6-20, see the information about the `<access-log>` and `<odl-access-log>` subelements of the `<web-site>` element.

The *Oracle Application Server Containers for J2EE User's Guide* has information about how to enable logging to the other OC4J files.

Oracle Diagnostic Logging Versus Text-Based Logging

For each of the logs listed in the preceding section, "OC4J Logs", you have the option of using Oracle Diagnostic Logging (ODL), which offers some advantages over text-based logging.

ODL provides standardized logging across all components of OC4J, creating the log files in an XML format that can be loaded to a repository for reporting and viewing. You can view ODL logs from Oracle Enterprise Manager 10*g*, for example.

With ODL, it is also easier to manage the size and number of your log files. In many situations, text-based logging results in the need to periodically shut down the OC4J server and manually clean up the files.

To configure ODL logging for the Web site access log file, use the `<odl-access-log>` subelement of the `<web-site>` element in the Web site XML file. To use text-based logging, use the `<access-log>` subelement of the `<web-site>` element instead.

For each of the other OC4J logs, use the `<odl>` subelement of the `<log>` element in the appropriate XML configuration file if you want to use ODL logging. To use text-based logging, use the `<file>` subelement of the `<log>` element instead.

Note: Web site access logs commonly use standard XLF or CLF format (extended log file format or common log file format). Users can split the files according to a specified time period, such as time of day or day of month. ODL Web site logs, however, do not support XLF or CLF format and you cannot split files by time period. When you reach the maximum size of an ODL log file, a new file is automatically created. (Log file names are `log1.xml`, `log2.xml`, and so on.)

See the *Oracle Application Server Containers for J2EE User's Guide* for additional information about ODL.

Additional Oracle Application Server Log Files

In addition to the OC4J log files discussed previously, Oracle Application Server supports the following log files:

- OPMN log file (one log file for each OC4J instance, for Oracle Process Management and Notification functionality)
- `ons.log` (OPMN notification system log, configured in `opmn.xml`)
- `ipm.log` (OPMN process management log, configured in `opmn.xml`)

OPMN manages Oracle HTTP Server and OC4J processes within an application server instance.

For information about Oracle Application Server log file management, refer to *Oracle Application Server Administrator's Guide*.

Servlet Debugging

This discussion summarizes debugging features and considerations for servlet developers, with appropriate cross-references for additional information. It consists of the following sections:

- OC4J Debugging Flags
- Setting OC4J Debugging Flags
- Timing Considerations for Debugging in Oracle Application Server
- Debugging Through JDeveloper and Other IDEs

OC4J Debugging Flags

OC4J supports several flags to enable debugging output for its subsystems.

Here are the HTTP debugging flags:

- `http.error.debug` for all HTTP errors; otherwise some are consumed without being reported
- `http.cluster.debug` for debugging statements regarding HTTP clustering and session persistence
- `http.session.debug` for HTTP session errors and lifecycle statements
- `http.request.debug` for information from HTTP request stream
- `http.redirect.debug` for information about HTTP redirects
- `debug.http.contentLength` to print explicit content-length calls as well as extra `sendError` information
- `http.virtualdirectory.debug` to print the enforced virtual directory mappings upon startup
- `http.method.trace.allow` to enable the `traceHTTP()` method

Here are the AJP debugging flags (for Oracle Application Server with Oracle HTTP Server only):

- `ajp.debug` to print the incoming AJP stream
- `ajp.io.debug` to print the AJP response from the server

The AJP flags do not produce user-friendly output, but are necessary for debugging some AJP issues.

Here are the JDBC debugging flags:

- `datasource.verbose` for information about the creation of data sources and database connections
- `jdbc.debug` for detailed information about JDBC calls

Here is the EJB debugging flag:

- `ejb.cluster.debug` for information about EJB clustering

Here are the RMI debugging flags:

- `rmi.debug` for information about remote method invocations
- `rmi.verbose` for detailed information about RMI calls

Here is the Web services debugging flag:

- `ws.debug` for information about Web services

Setting OC4J Debugging Flags

The debugging flags are enabled through Java option settings such as the following:

```
-Dhttp.session.debug=true
```

If you are using OC4J standalone, specify option settings in the Java command line when you start OC4J. In an Oracle Application Server environment, use Oracle Enterprise Manager 10*g*. Specify option settings in the Java Options field under Command Line Options in the Application Server Control Console Server Properties Page for the OC4J instance. To get to this page, select **Server Properties** under Instance Properties in the Administration Page for the OC4J instance. See "Application Server Control Console OC4J Administration Page" on page 7-6. See the *Oracle Application Server Containers for J2EE User's Guide* for further information.

Timing Considerations for Debugging in Oracle Application Server

Because of the way OPMN functions, there are timing issues to consider when debugging in an Oracle Application Server environment. Specifically, whenever debugging results in the halting of a process, OPMN terminates that process after the halt goes beyond the timeout period.

To remedy this situation, you must set an appropriate timeout value, using the `<timeout>` element in the `opmn.xml` file.

For information about `opmn.xml`, particulaly for starting and stopping Oracle Application Server, refer to the *Oracle Application Server Administrator's Guide*.

Debugging Through JDeveloper and Other IDEs

If you use Oracle JDeveloper as your development environment, you can take advantage of its debugging features.

For debugging, JDeveloper offers local and remote debugging of JSP pages, servlets, and other Java source files. You can start by setting breakpoints in the source files within JDeveloper and running a debugging session with the source selected. While debugging an application such as a servlet in JDeveloper, you have complete control over the execution flow and can view and modify variable values, as well as perform advanced application performance monitoring, such as viewing class instance counts and memory usage. JDeveloper will follow calls from your application into other source files or offer to generate stub classes for class sources that are not available. Remote debugging, after the code to be debugged is launched and the JDeveloper debugger is attached to it, is similar to local debugging.

See "Oracle JDeveloper Support for Servlet Development" on page 2-19 for a general summary of JDeveloper features for servlet development.

Note: Other key IDE vendors offer plug-in modules that allow seamless integration with OC4J. This provides developers with the capability to build, deploy, and debug J2EE applications running on OC4J directly from within the IDE. You can refer to the following Web site for more information:

`http://www.oracle.com/technology/products/ias/9ias_partners.html`

(To access the preceding Web site, you must have an Oracle Technology Network membership, but it is free of charge.)

Oracle JDeveloper Support for Servlet Development

Visual Java programming tools now typically support servlet coding. In particular, Oracle 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
- Support for incorporating custom JavaBeans
- A deployment option for servlet applications that rely on Oracle Application Development Framework (Oracle ADF) Business Components

Also see "Debugging Through JDeveloper and Other IDEs" on page 2-18.

For general information about JDeveloper, refer to the JDeveloper online help or to the following site on the Oracle Technology Network:

`http://www.oracle.com/technology/products/jdev/content.html`

Introduction to OC4J Support for Open Source Frameworks

OC4J supports some common open source utilities and frameworks. For Oracle Application Server 10*g* Release 2 (10.1.2), this document discusses support for two in particular:

- Jakarta Struts
- Jakarta log4j

The focus is on configuring and using these open source utilities in the OC4J standalone environment. See Appendix A, "Open Source Frameworks and Utilities".

Servlet Invocation

A servlet is invoked by the container when a request for the servlet arrives from a client. The client request may 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 following sections cover servlet invocation, including some special OC4J features for invoking a servlet by class name in a development or testing scenario:

- Summary of URL Components
- Servlet Invocation by Class Name During OC4J Development
- Servlet Invocation in an Oracle Application Server Production Environment
- Servlet Invocation in an OC4J Standalone Environment

Summary of URL Components

Before discussing servlet invocation, it is useful to summarize the components of a URL. Here is the generic construct (though note that *`pathinfo`* is usually empty):

```
protocol://host:port/contextpath/servletpath/pathinfo
```

You could also have additional information following any delimiters, such as request parameter settings following a question mark ("`?`") delimiter:

```
protocol://host:port/contextpath/servletpath/pathinfo?param=value
```

Table 2–2 describes the components of the generic construct.

Table 2–2 URL Components

Component	Description
Protocol	The network protocol to be used when invoking the Web application. Examples are `http`, `https`, `ftp`, and `ormi` (for EJBs).
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), such as: `www.example.com`
Port	The port that the Web server listens on. If a URL does not specify a port, most browsers assume port 80 for HTTP protocol or port 443 for HTTPS. For OC4J, the port number is specified in the `port` attribute of the `<web-site>` element in the Web site XML file, such as `default-web-site.xml` for an Oracle Application Server environment or `http-web-site.xml` for OC4J standalone. (For each port, there must be one associated protocol, according to the `<web-site>` element `protocol` attribute.)

Context path	The designated root path for the servlet context. You specify the context path when you deploy an application. For OC4J, the specified context path is reflected in the setting of the `root` attribute of the `<web-app>` element (a subelement of `<web-site>`) in the Web site XML file.
	
	Each servlet context is associated with a directory path in the server file system.
	
	The `<web-app>` element also indicates the J2EE application name (and EAR file name) through its `application` attribute, and the Web module name (and WAR file name) through its `name` attribute. The J2EE application name, Web module name, and context path are all mapped together in this way. Here is an example:
	
	`<web-app application="ojspdemos" name="ojspdemos-web"`
	`        root="/ojspdemos" />`

Table 2–2 (Cont.) URL Components

Component	Description

Servlet path	The designated path, beyond the context path, for the particular servlet you want to invoke. You specify the servlet path through standard mappings in the application `web.xml` file. A servlet class is mapped to an arbitrary servlet name through `<servlet-class>` and `<servlet-name>` subelements of a `<servlet>` element. The servlet name is mapped to a servlet path through `<servlet-name>` and `<url-pattern>` subelements of a `<servlet-mapping>` element. (You can map a single servlet class to multiple servlet names and multiple servlet paths.) Here is an example:
	`<web-app>`
	`  ...`
	`  <servlet>`
	`    <servlet-name>logout</servlet-name>`
	`    <servlet-class>`
	`       oracle.security.jazn.samples.http.Logout`
	`    </servlet-class>`
	`  </servlet>`
	`  ...`
	`  <servlet-mapping>`
	`    <servlet-name>logout</servlet-name>`
	`    <url-pattern>/logout/*</url-pattern>`
	`  </servlet-mapping>`
	`  ...`
	`</web-app>`

Path information	(This is typically empty.) Beyond the context path and servlet path, a URL can contain additional information that is supplied to the servlet through the HTTP request object. Such information is presumably understood by the servlet. This information is separate from any request parameter settings or other URL components that follow delimiters such as question marks. Such delimiters would follow any path information.

Note: The name specified in a `<servlet-name>` element is the name you input to the servlet context `getNamedDispatcher()` method if you want a request dispatcher for that servlet.

For more information about the OC4J configuration elements and attributes discussed in Table 2–2, see "Element Descriptions for Web Site XML Files" on page 6-20. For information about elements and attributes of the `web.xml` file, refer to the servlet specification

Consider the following sample URL:

```
http://www.example.com:8888/foo/bar/mypath/MyServlet/info1/info2?user=Amy
```

In the process of invoking a servlet according to a URL supplied by a client browser, the servlet container takes the following steps:

1. It examines everything in the URL after the port number, then examines its own configuration settings (such as in a Web site XML file) for recognized context paths, then determines what part of the URL is the context path.

   Assume for this example that `/foo/bar` is the context path.

2. It examines everything in the URL after the context path, then examines the servlet mappings in the `web.xml` file for recognized servlet paths, then determines what part of the URL is the servlet path.

   At this point, the servlet can be invoked. The servlet container does not use any information beyond the servlet path.

   Assume for this example that `/mypath/MyServlet` is the servlet path.

3. If anything remains in the URL after the servlet path and preceding any URL delimiters (such as "?" in this example, which delimits request parameter settings), that portion of the URL is taken as extra information and is passed to the servlet through the HTTP request object.

   Assume for this example that `/info1/info2` is the extra path information.

Note that the context path, servlet path, and any path information can all be "compound" components, with one or more forward-slashes in between parts. The preceding example shows this. In many cases, the context path may be simple, such as just `foo`, and the servlet path may also be simple, such as just `MyServlet`, and any path information may be simple as well. But it is impossible to know by just looking at a URL what part of it is the context path, what part is the servlet path, and what part is extra path information (if any). You must examine the configuration in the Web site XML file and `web.xml` file to determine this.

Notes:

- See the *Oracle Application Server Containers for J2EE User's Guide* or *Oracle Application Server Containers for J2EE Stand Alone User's Guide* for information about defined ports and what listeners they are mapped to, and for information about how to alter these settings.
- Cookie names are based on the host name, port number, and path (just the context path by default, but possibly including the servlet path as well).
- The concepts of servlet contexts and context paths were introduced in version 2.2 of the servlet specification.
- You can retrieve the context path, servlet path, and path information through the `getContextPath()`, `getServletPath()`, and `getPathInfo()` methods of the HTTP request object.

Servlet Invocation by Class Name During OC4J Development

For a development or testing scenario in OC4J, there is a convenience mechanism for invoking a servlet by class name. For security reasons, use this mechanism *only* 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 used to invoke servlets by class name. This URL component follows the context path in the URL, and anything following this URL component is assumed to be a servlet class name, including applicable package information, within the appropriate servlet context. The servlet class name appears instead of a servlet path in the URL. (Technically, the `servlet-webdir` value is the servlet path and acts as a servlet itself, and the class name of the servlet you wish to execute is taken as path information.)

Generally speaking, for any given application, OC4J behavior for invocation by class name is determined by the `servlet-webdir` setting in the `orion-web.xml` file for that application, if there is a setting. But note the following:

- Any setting of `servlet-webdir` in the `global-web-application.xml` file acts as a default value (as is true with configuration settings in `global-web-application.xml` in general). If there is no `servlet-webdir` setting in `global-web-application.xml`, however, then the default value is `""` (empty quotes). This setting disables invocation by class name. The default value is used in the event that `orion-web.xml` is not provided with the application deployment, or does not have a `servlet-webdir` setting.
- You can disable servlet invocation by class name in either of two ways:
  - Set the system property `http.webdir.enable` to a value of `false`. This results in any `servlet-webdir` setting being ignored.
  - Set a `servlet-webdir` value of `""` (empty quotes), either through `global-web-application.xml` or `orion-web.xml`.

For information about OC4J system properties, see the *Oracle Application Server Containers for J2EE Stand Alone User's Guide*, or the *Oracle Application Server Containers for J2EE User's Guide* for an Oracle Application Server environment. See the *Oracle Application Server Release Notes* for your platform for information about the default value of the `http.webdir.enable` system property and any default setting of `servlet-webdir` in the `global-web-application.xml` file that is shipped with OC4J.

The following URL invokes a servlet called `SessionServlet` by its class name, assuming a setting of `servlet-webdir="/servlet/"`. 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://www.example.com:8888/servlet/foo.bar.SessionServlet
```

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 is as follows:

```
http://www.example.com:8888/foo/servlet/foo.bar.SessionServlet
```

Important: Allowing the invocation of servlets by class name presents a significant security risk. Do not configure OC4J to operate in this mode in a production environment. See "Additional Security Considerations" on page 2-40 for information.

Servlet Invocation in an Oracle Application Server Production Environment

The following sections describe Oracle HTTP Server and OC4J features for servlet invocation in an Oracle Application Server Environment:

- Key Features for Invocation in Oracle Application Server
- Use of Perceived Front-End Hosts by OC4J

Key Features for Invocation in Oracle Application Server

In an Oracle Application Server production environment, OC4J should always be accessed through the Oracle HTTP Server. Oracle HTTP Server uses AJP (Apache JServ protocol) to communicate to OC4J, but this is invisible to the end user.

When a servlet is requested, the OC4J servlet container interprets the URL, as "Summary of URL Components" on page 2-19 describes.

Whatever port number you use is mapped to AJP protocol through a `<web-site>` element in the `default-web-site.xml` file. (This is the typical name, but Web site XML file names are defined according to settings in the `server.xml` file and can be changed as desired.) The port mapping is defined through the `port` and `protocol` attributes of the `<web-site>` element, with `port` set as desired and `protocol` set to "`ajp13`". By default, port 7777 is for access through the Oracle HTTP Server with Oracle Application Server Web Cache enabled.

Whenever you use Enterprise Manager to deploy an application, you are prompted for a URL mapping. The mapping you specify results in a new OC4J mount point in `mod_oc4j.conf`. If you specify a URL mapping of "`/mypath`", for example, this is the context path of your Web application and is defined as a new OC4J mount point. Then you invoke a servlet with a URL such as the following:

```
http://www.example.com:7777/mypath/MyServlet
```

See "Application Server Control Console Deploy Application (EAR) Page" on page 7-3 and "Application Server Control Console Deploy Web Application (WAR) Page" on page 7-5 for information about the Enterprise Manager EAR and WAR deployment pages.

For an overview of deployment to Oracle Application Server, see "OC4J Deployment in Oracle Application Server" on page 5-39. For further information, see the *Oracle Application Server Containers for J2EE User’s Guide*. For general information about Enterprise Manager, see *Oracle Enterprise Manager Concepts*.

See the *Oracle HTTP Server Administrator’s Guide* for information about Oracle HTTP Server configuration, mount points, and the `mod_oc4j.conf` file.

Use of Perceived Front-End Hosts by OC4J

An additional element in the `default-web-site.xml` file (or other Web site XML file) is relevant in servlet invocation. The `<frontend>` subelement of the `<web-site>` element can specify a perceived front-end host and port of the Web site as seen by HTTP clients. When the site is behind 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.example.com`"), and `port`, for the port number of the front-end server (such as "`8080`"). Using this front-end information, the back-end server that is actually running the application knows to refer to `www.example.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.

The specified front-end `host` and `port` settings are also reflected back to the servlet and are the values you receive if you call the `getServerName()` or `getServerPort()` method of the HTTP request object.

Servlet Invocation in an OC4J Standalone Environment

In OC4J standalone, a 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. (This is the typical name, but Web site XML file names are according to settings in the `server.xml` file and can be changed as desired.)

When a servlet is requested, the OC4J servlet container interprets the URL, as "Summary of URL Components" on page 2-19 describes.

Whatever port number you use is mapped to HTTP protocol through a `<web-site>` element in the `http-web-site.xml` file (or other Web site XML file, as applicable). The port mapping is defined through the `port` and `protocol` attributes of the `<web-site>` element, with `port` set as desired and `protocol` set to "`http`". By default, port 8888 is for direct access to OC4J through its own Web listener.

In OC4J standalone, the default context path is "`/`" to use HTTP protocol for an application deployed to the OC4J default Web application. Here is an example:

```
http://www.example.com:8888/MyServlet
```

If you are not using the default Web application, specify the context path while deploying the application. You can either do this through the `admin.jar` utility, or by manual deployment and manual edits of the `http-web-site.xml` file (not recommended). Deployment for OC4J standalone is discussed in "Deployment Scenarios to OC4J Standalone" on page 5-23, but for complete information see the *Oracle Application Server Containers for J2EE Stand Alone User's Guide*. That document also has information about OC4J port settings and other default settings.

If you specify "`/mypath`" as the context path, for example, you will invoke the servlet with a URL such as the following:

```
http://www.example.com:7777/mypath/MyServlet
```

Servlet Sessions

Servlet sessions were introduced in "Introduction to Servlet Sessions" on page 1-5. The following sections provide details and examples:

- Session Tracking
- Features of the HttpSession Interface
- Session Cancellation
- Session Replication in a Distributable Application
- Session Servlet Example

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, perform a few computations, output some results, and then in effect go away. But 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.

Session tracking involves identifying user sessions by ID numbers and tying requests to their session through use of the ID number. This process is typically performed using cookies or URL rewriting.

The OC4J servlet container, to comply with the servlet specification, implements session tracking through HTTP session objects, which are instances of a class that implements the `javax.servlet.http.HttpSession` interface.

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

An HTTP session object has scope over the Web application only. You cannot use session objects to share data between applications. Nor can you use session objects to share data between different clients of the same application. There is one HTTP session object for each client in each application.

Note: To share information between clients or applications, you can store such 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

Several approaches have been used 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 (assuming the client supports cookies), the client sends the cookie back to the server as part of the request, and the server uses the cookie value 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; the container handles this. Sending cookies back to the server is handled automatically by the Web browser, unless the 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. In this mechanism, the session ID is encoded into the URL path of a request. See "Session Servlet Example" on page 2-31 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
```

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

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: To comply with the servlet specification, calls to the `encodeURL()` and `encodeRedirectURL()` methods 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, including server hidden form fields and user authentication mechanisms to store additional information. Oracle does not recommend these techniques in OC4J applications. They have many drawbacks, including performance penalties and loss of confidentiality.

Session Tracking in OC4J

For session-tracking in OC4J, the servlet container first attempts to accomplish tracking through cookies. If cookies are disabled, the server can maintain session tracking 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 may be disabled.

You can use the following setting in the `global-web-application.xml` or `orion-web.xml` file to disable the use of session cookies:

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

Cookies are enabled by default.

Notes:

- OC4J does not support *auto-encoding*, in which session IDs are automatically encoded into the URL by the servlet container. This is a nonstandard 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.

Features of the HttpSession Interface

The servlet container uses HTTP session objects—instances of a class that implements the `javax.servlet.http.HttpSession` interface—in tracking and managing user sessions. 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 configuration 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. A negative value indicates no timeout. A value of 0 results in an immediate timeout.
- `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 most recent request associated with the client session, measured in milliseconds since midnight, January 1, 1970. If the client session has not yet been accessed, this method returns the session creation time.

For an example of how a servlet can use an HTTP session object, see "Session Servlet Example" on page 2-31.

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

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

Session Cancellation

HTTP session 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`. Specify the timeout in minutes, as an integer. For example, to reduce the session timeout to five minutes, add the following lines to the application `web.xml` file:

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

According to the servlet specification, a negative value specifies the default behavior that a session never times out. For example:

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

A value of 0 results in an immediate timeout.

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 Replication in a Distributable Application

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, and the session state will still be available.

The following sections provide more information:

- Overview of Session Replication and Requirements
- Possible Clustering Error Conditions and Related Environment Flags
- Session Replication Details and Logistics

Overview of Session Replication and Requirements

To enable replication of the session state of an application between OC4J servers, you must mark the Web application as *distributable*, by use of the standard `<distributable>` element in the `web.xml` file. The presence of this subelement of the `<web-app>` element, as follows, specifies that the application is distributable:

```
<web-app ... >
   ...
   <distributable/>
   ...
</web-app>
```

Note: In an Oracle Application Server environment, accomplish this through Oracle Enterprise Manager 10*g*. See the discussion of clustering in the *Oracle Application Server Containers for J2EE User's Guide* for details.

Objects that are stored by a servlet in the `HttpSession` object are replicated. For replication to work properly, objects must be *serializable* (directly or indirectly implementing the `java.io.Serializable` interface) or *remoteable* (directly or indirectly implementing the `java.rmi.Remote` interface). Furthermore, any objects that are referenced by objects in the session object must themselves be serializable or remoteable.

Possible Clustering Error Conditions and Related Environment Flags

Replicated data is sent asynchronously to the other OC4J servers in the cluster island. For performance reasons, OC4J does not wait to confirm successful replication. Therefore, either of the following scenarios is possible, though highly unlikely:

- Broadcast latency, where session replication messages are not received and processed by the other OC4J servers before a client is rerouted:
    1. A client sends a request and receives a response from the OC4J server.
    2. The server broadcasts a replication message to the other OC4J servers in the cluster island with the updated state for the client.
    3. The client sends another request before the broadcast of the updated state has been received and processed by all OC4J servers.
    4. The original server fails, and the client is rerouted to one of the OC4J servers that does not yet have the new state information, resulting in the client receiving old data.
- Failure before response to client, where a server fails after it has broadcast its replication message to other servers, but before it has completed its response to the client:
    1. A client sends a request to the OC4J server.
    2. The server broadcasts a replication message to the other OC4J servers in the cluster island with the updated state for the client.
    3. The server fails, however, before completing its response to the client. This results in the client being unaware of the processing completed by the server, even though other OC4J servers are aware.

Because of the possible error scenarios, OC4J and Oracle HTTP Server maintain *session affinity*, meaning they make every effort to always route requests and responses through the same OC4J JVM. The session cookie, `JSESSIONID`, maintains the required, detailed routing information across HTTP requests to ensure that subsequent requests through Oracle HTTP Server are dispatched to the originating JVM wherever possible.

In addition, the OC4J 10.1.2 implementation supports two environment flags that you can use to reduce the risk of either error scenario occurring:

- `cluster.thread.priority`: By default, OC4J clustering threads run with the same priority as the other main OC4J threads. You can, however, set this flag to any integer value from 6 through 10 to give clustering threads higher priority, with 10 being the highest priority.
- `cluster.failover.delay`: In the event that an OC4J server fails, this flag results in a delay of the specified number of milliseconds before a client is rerouted to an alternate server. The default is no delay. A setting between 7000 and 9000 is probably sufficient to avoid the first of the error scenarios described above.

Session Replication Details and Logistics

For a distributable application, session replication is triggered each time there is a `setAttribute()` call on the session object. The name and value specified in the call are serialized and replicated, with the serialized value being stored using the specified name as the key. The value is deserialized only upon first access by a failed-over servlet.

Note that you must explicitly call `setAttribute()` whenever you update a data item belonging to the session object. For example, if you call `getAttribute()` on the session object to retrieve a bean and then call a method on the bean to change its state, you must then call `setAttribute()` on the session object to update the bean in the session. This is in contrast to the situation in a nondistributable environment, in which the bean is passed to you by reference and updated directly within the session object as soon as you call the method on the bean.

Also note the performance implications of this functionality. A servlet with a large number of `setAttribute()` calls may have lower performance because of the small overhead introduced when performing state replication.

Note: You can observe the runtime status of replication and session state updates by enabling the OC4J debugging flags `http.session.debug` and `http.cluster.debug`. See "OC4J Debugging Flags" on page 2-17 and "Setting OC4J Debugging Flags" on page 2-18.

Session Servlet Example

The following `SessionServlet` code implements a servlet that establishes an `HttpSession` object and prints 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
    // 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";
  }
}
```

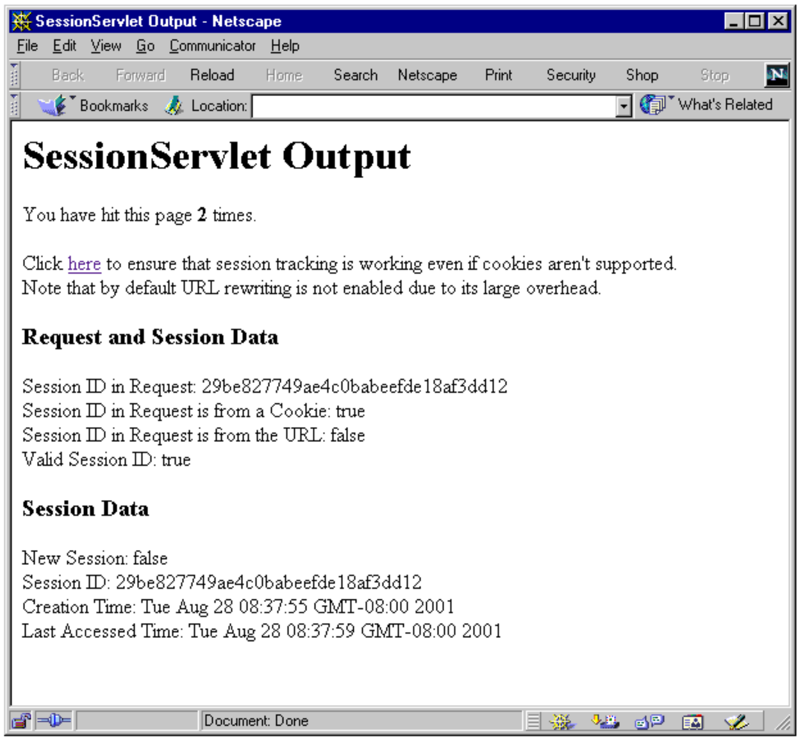
Deploying and Testing

In OC4J standalone, save the preceding code into a file `SessionServlet.java` in the OC4J default Web application `/WEB-INF/classes` directory. By default, the default Web application root directory is `j2ee/home/default-web-app`. (See "OC4J Default Application and Default Web Application" on page 5-25 for more information.)

For convenience, use the `development="true"` setting in the `<orion-web-app>` element of the `global-web-application.xml` file. See "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1 for more information about the `development` flag.

Figure 2–1 shows the output of this servlet when a Web browser that has cookies enabled invokes it the second time in a session. 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

Servlet Security

OC4J supports Secure Socket Layer (SSL) communication between Oracle HTTP Server and OC4J in an Oracle Application Server environment, using secure AJP. This is the secure version of Apache JServ Protocol, the protocol Oracle HTTP Server uses to communicate with OC4J. The following sections provide details:

- Use of Security Features
- Configuration of Oracle HTTP Server and OC4J for SSL
- SSL Common Problems and Solutions

This discussion is followed by a section of general security considerations:

- Additional Security Considerations

Notes:

- Secure communication between a client and Oracle HTTP Server is independent of secure communication between Oracle HTTP Server and OC4J. (Also note that the secure AJP protocol used between Oracle HTTP Server and OC4J is not visible to the end user.) This section covers only secure communication between Oracle HTTP Server and OC4J.
- In addition, OC4J standalone supports SSL communication directly between a client and OC4J, using HTTPS. See the *Oracle Application Server Containers for J2EE Stand Alone User's Guide*, available when you download the standalone version from OTN.

See the following documents for additional information about Oracle Application Server security and Oracle HTTP Server.

- *Oracle Application Server Security Guide* (including information about secure protocol between a client and Oracle HTTP Server)
- *Oracle Application Server Containers for J2EE Security Guide* (including an overview of SSL keys, certificates, and related concepts)
- *Oracle HTTP Server Administrator's Guide*

Use of Security Features

The following sections discuss how to use SSL features with OC4J and Oracle HTTP Server:

- Using Certificates with OC4J and Oracle HTTP Server
- Requesting Client Authentication

Using Certificates with OC4J and Oracle HTTP Server

The steps below are for using keys and certificates for SSL communication in OC4J. These are server-level steps, typically executed prior to deployment of an application that will require secure communication, perhaps when you first set up an Oracle Application Server instance.

Note that a *keystore* stores certificates, including the certificates of all trusted parties, for use by a program. Through its keystore, an entity such as OC4J (for example) can authenticate other parties, as well as authenticate itself to other parties. Oracle HTTP Server uses what is called a *wallet* for the same purpose.

In Java, a keystore is a `java.security.KeyStore` instance that you can create and manipulate using the `keytool` utility that is provided with the Sun Microsystems JDK. The underlying physical manifestation of this object is a file. Go to the following Web site for information about `keytool`:

`http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html`

The Oracle Wallet Manager has functionality for Oracle wallets that is equivalent to the functionality of `keytool` for keystores.

Here are the steps in using certificates between OC4J and Oracle HTTP Server:

1. Use `keytool` to generate a private key, public key, and unsigned certificate.You can place this information into either a new keystore or an existing keystore.
2. Obtain a signature for the certificate, using either of the following two approaches.

   Generate your own signature:

   a. Use `keytool` to "self-sign" the certificate. This is appropriate if your clients trust you as, in effect, your own certificate authority.

   Alternatively, obtain a signature from a recognized certificate authority:

   a. Using the certificate from Step 1, use `keytool` to generate a *certificate request*, which is a request to have the certificate signed by a certificate authority.

   b. Submit the certificate request to a certificate authority.

   c. Receive the signature from the certificate authority, and import it into the keystore, again using `keytool`. In the keystore, the signature is matched with the associated certificate.

   Note: Oracle Application Server includes Oracle Application Server Certificate Authority (OCA). OCA allows customers to create and issue certificates for themselves and their users, although these certificates would probably be unrecognized outside a customer's organization without prior arrangements. See the *Oracle Application Server Certificate Authority Administrator's Guide* for information about OCA.

The process for requesting and receiving signatures is up to the particular certificate authority you use. Because that is outside the scope and control of Oracle Application Server, the documentation does not cover it. You can go to the Web site of any certificate authority for information. (Any browser should have a list of trusted certificate authorities.) Here are the Web addresses for VeriSign, Inc. and Thawte, Inc., for example:

`http://www.verisign.com/`

`http://www.thawte.com/`

For SSL communication between OC4J and Oracle HTTP Server, execute the preceding steps for Oracle HTTP Server, but use a wallet and Oracle Wallet Manager instead of a keystore and the `keytool` utility. See the *Oracle Application Server Security Guide* for information about wallets and the Oracle Wallet Manager.

In addition to steps 1 and 2 above, execute the following steps as necessary:

1. **If the OC4J certificate is signed by an entity that Oracle HTTP Server does not yet trust**, obtain the certificate of the entity and import it into Oracle HTTP Server. The specifics depend on whether the OC4J certificate in question is self-signed, as follows.

   If OC4J has a self-signed certificate (essentially, Oracle HTTP Server does not yet trust OC4J):

   a. From OC4J, use `keytool` to export the OC4J certificate. This step places the certificate into a file that is accessible to Oracle HTTP Server.

   b. From Oracle HTTP Server, use Oracle Wallet Manager to import the OC4J certificate.

Alternatively, if OC4J has a certificate that is signed by another entity (that Oracle HTTP Server does not yet trust):

a. Obtain the certificate of the entity in any appropriate way, such as by exporting it from the entity. The exact steps vary widely, depending on the entity.

b. From Oracle HTTP Server, use Oracle Wallet Manager to import the certificate of the entity.

2. **If the Oracle HTTP Server certificate is signed by an entity that OC4J does not yet trust, and OC4J is in a mode of operation that requires client authentication** (as "Requesting Client Authentication" on page 2-37 discusses):

   a. Obtain the certificate of the entity in any appropriate way, such as by exporting it from the entity. The exact steps vary widely, depending on the entity.

   b. From OC4J, use `keytool` to import the certificate of the entity.

Note: During communications over SSL between Oracle HTTP Server and OC4J, all data on the communications channel between the two is encrypted. The following steps are executed: 1) The OC4J certificate chain is authenticated to Oracle HTTP Server during establishment of the encrypted channel. 2) Optionally, if OC4J is in client-authentication mode, Oracle HTTP Server is authenticated to OC4J. This process also occurs during establishment of the encrypted channel. 3) Any further communication after this initial exchange will be encrypted.

Example: Creating an SSL Certificate and Generating Your Own Signature This example corresponds to Step 2 above, in the mode where you generate your own signature by using `keytool` to self-sign the certificate.

First, create a keystore with an RSA private / public keypair, using the `keytool` command. The following example (in which `%` is the system prompt) uses the RSA keypair algorithm to generate a keystore to reside in a file named `mykeystore`, which has a password of `123456` and is valid for 21 days:

```
% keytool -genkey -keyalg "RSA" -keystore mykeystore -storepass 123456 -validity 21
```

Note the following:

- The `keystore` option specifies the name of the file in which the keys are stored.
- The `storepass` option sets the password for protecting the keystore.
- The `validity` option sets the number of days for which the certificate is valid.

The `keytool` prompts you for more information, as follows:

```
What is your first and last name?
  [Unknown]:  Test User
What is the name of your organizational unit?
  [Unknown]:  Support
What is the name of your organization?
  [Unknown]:  Oracle
What is the name of your City or Locality?
  [Unknown]:  Redwood Shores
What is the name of your State or Province?
  [Unknown]:  CA
```

```
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Test User, OU=Support, O=Oracle, L=Reading, ST=Berkshire, C=GB> correct?
  [no]:  yes

Enter key password for <mykey>
        (RETURN if same as keystore password):
```

Note: To determine your two-letter country code, use the ISO country code list at the following URL:

`http://www.bcpl.net/~jspath/isocodes.html`

The `mykeystore` file is created in the current directory. The default alias of the key is `mykey`.

Requesting Client Authentication

OC4J supports a *client authentication* mode in which the server explicitly requests authentication from the client before the server communicates with the client. In an Oracle Application Server environment, Oracle HTTP Server acts as the client to OC4J.

For client authentication, Oracle HTTP Server must have its own certificate and authenticates itself by sending a certificate and a certificate chain that ends with a root certificate. You can configure OC4J to accept only root certificates from a specified list in establishing a chain of trust back to a client.

A certificate that OC4J trusts is called a *trust point*. In the certificate chain from Oracle HTTP Server, the trust point is the first certificate OC4J encounters that matches one in its own keystore. There are three ways to establish trust:

- The client certificate is in the keystore.
- One of the intermediate CA certificates in the certificate chain from Oracle HTTP Server is in the keystore.
- The root CA certificate in the certificate chain from Oracle HTTP Server is in the keystore.

OC4J verifies that the entire certificate chain, up to and including the trust point, is valid to prevent any forged certificates.

If you request client authentication with the `needs-client-auth` attribute, perform the following steps. See "OC4J Configuration Steps for SSL" on page 2-38 for how to configure this attribute.

1. Decide which of the certificates in the chain from Oracle HTTP Server is to be your trust point. Ensure that you either have control over the issuance of certificates using this trust point or that you trust the certificate authority as an issuer.
2. Import the intermediate or root certificate in the server keystore as a trust point for authentication of the client certificate.

   Note: If you do not want OC4J to accept certain trust points, make sure these trust points are not in the keystore.

3. Execute the steps to create the client certificate (documented in "Using Certificates with OC4J and Oracle HTTP Server" on page 2-34). The client certificate includes

the intermediate or root certificate that is installed in the server. If you wish to trust another certificate authority, obtain a certificate from that authority.

4. Save the certificate in a file on Oracle HTTP Server.
5. Provide the certificate for the Oracle HTTP Server initiation of the secure AJP connection.

Configuration of Oracle HTTP Server and OC4J for SSL

For secure communication between Oracle HTTP Server and OC4J, configuration steps are required at each end, as the following sections discuss:

- Oracle HTTP Server Configuration Steps for SSL
- OC4J Configuration Steps for SSL

Oracle HTTP Server Configuration Steps for SSL

In Oracle HTTP Server, verify proper SSL settings in `mod_oc4j.conf` for secure communication. SSL must be enabled, with a wallet file and password specified, as follows:

```
Oc4jEnableSSL on
Oc4jSSLWalletFile wallet_path
Oc4jSSLWalletPassword pwd
```

The *`wallet_path`* value is a directory path to the wallet file, without a file name. (The wallet file name is already known.) The *`pwd`* value is the wallet password.

For more information about the `mod_oc4j.conf` file, see the *Oracle HTTP Server Administrator's Guide*.

OC4J Configuration Steps for SSL

In the `default-web-site.xml` file (or other Web site XML file, as appropriate), you must specify appropriate SSL settings under the `<web-site>` element.

1. Turn on the `secure` flag to specify secure communication, as follows:

```
<web-site ... secure="true" ... >
   ...
</web-site>
```

   Setting `secure="true"` specifies that the AJP protocol should use an SSL socket.

2. Use the `<ssl-config>` subelement and its `keystore` and `keystore-password` attributes to specify the path and password for the keystore, as follows:

```
<web-site ... secure="true" ... >
   ...
   <ssl-config keystore="path_and_file" keystore-password="pwd" />
</web-site>
```

   The `<ssl-config>` element is required whenever the `secure` flag is set to `"true"`.

   The *`path_and_file`* value can indicate either an absolute or relative directory path and includes the file name. A relative path is relative to the location of the Web site XML file.

3. Optionally, to specify that client authentication is required, turn on the `needs-client-auth` flag. This is an attribute of the `<ssl-config>` element.

```
<web-site ... secure="true" ... >
   ...
   <ssl-config keystore="path_and_file" keystore-password="pwd"
               needs-client-auth="true" />
</web-site>
```

   This step sets up a mode in which OC4J accepts or rejects a client entity, such as Oracle HTTP Server, for secure communication, depending on its identity. The `needs-client-auth` flag instructs OC4J to request the client certificate chain upon connection. If OC4J recognizes the root certificate of the client, then the client is accepted.

   The keystore that is specified in the `<ssl-config>` element must contain the certificates of any clients that are authorized to connect to OC4J through secure AJP and SSL.

Here is an example that sets up secure communication with client authentication:

```
<web-site display-name="OC4J Web Site" protocol="ajp13" secure="true" >
   <default-web-app application="default" name="defaultWebApp" root="/j2ee" />
   <access-log path="../log/default-web-access.log" />
   <ssl-config keystore="../keystore" keystore-password="welcome"
            needs-client-auth="true" />
</web-site>
```

Only the portions in bold are specific to security. The protocol value is always `"ajp13"` for communication through Oracle HTTP Server, whether or not you use secure communication. A protocol value of `ajp13` with `secure="false"` indicates AJP protocol; `ajp13` with `secure="true"` indicates secure AJP protocol.

For more information about elements and attributes of the `<web-site>` and `<ssl-config>` elements, see "Element Descriptions for Web Site XML Files" on page 6-20.

Also see "Requesting Client Authentication" on page 2-37 for related information.

SSL Common Problems and Solutions

This section discusses some common SSL errors and their causes and remedies, followed by a brief discussion of general SSL debugging.

SSL Common Errors

The following errors may occur when using SSL certificates:

Keytool Error: java.security.cert.CertificateException: Unsupported encoding

Cause: There is trailing white space, which the `keytool` utility does not allow.

Action: Delete all trailing white space. If the error still occurs, add a newline in your certificate reply file.

Keytool Error: KeyPairGenerator not available

Cause: You are probably using the `keytool` utility from an older JDK.

Action: Use the `keytool` utility from the latest JDK on your system. To ensure that you are using the latest JDK, specify the full path for this JDK.

Keytool Error: Failed to establish chain from reply

Cause: The `keytool` utility cannot locate the root CA certificates in your keystore, and therefore cannot build the certificate chain from your server key to the trusted root certificate authority.

Action: Execute the following command:

```
keytool -keystore keystore -import -alias cacert -file cacert.cer
(keytool -keystore keystore -import -alias intercert -file inter.cer)
```

If you use an intermediate CA `keytool` utility, then execute this command:

```
keystore keystore -genkey -keyalg RSA -alias serverkey
keytool -keystore keystore -certreq -file my.host.com.csr
```

Get the certificate from the Certificate Signing Request (CSR), then execute the following command:

```
keytool -keystore keystore -import -file my.host.com.cer -alias serverkey
```

No available certificate corresponds to the SSL cipher suites that are enabled

Cause: Something is wrong with your certificate.

Action: Determine and rectify the problem.

General SSL Debugging

While you are developing in OC4J standalone, you can display verbose debug information from the Java Secure Socket Extension (JSSE) implementation. To get a list of options, start OC4J as follows:

```
java -Djavax.net.debug=help -jar oc4j.jar
```

Start it as follows to enable full verbosity:

```
java -Djavax.net.debug=all -jar oc4j.jar
```

This will display the browser request header, server HTTP header, server HTTP body, content length (before and after encryption), and SSL version.

Additional Security Considerations

In addition to the SSL functionality discussed previously, the following are considerations for the security of your Web application running in the OC4J servlet container:

- In the `global-web-application.xml` file or `orion-web.xml` file, use the `<jazn-web-app>` subelement of `<orion-web-app>` to configure the OracleAS JAAS Provider and Single Sign-On (SSO) properties for servlet execution. These features must be set appropriately in order to invoke a servlet under the privileges of a particular security subject. This element is described under "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1.
- 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 servlet specification. OC4J also offers related support through the `global-web-application.xml` file `<security-role-mapping>` element. See "Configuration for global-web-application.xml and orion-web.xml" on page 6-1 for details about elements and attributes of `global-web-application.xml`.

- Invocation by class name should be considered only in a development environment, because there is a significant security risk when users are allowed to invoke servlets in this way.

  Invocation by class name can bypass standard security constraints unless this is specifically addressed in the `web.xml` file. In addition, 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 security issues, particularly in a production environment, you can disable servlet invocation by class name in either of two ways:

  – Set the system property `http.webdir.enable` to a value of `false`. This setting results in any `servlet-webdir` setting being ignored.

  – Set a `servlet-webdir` value of `""` (empty quotes), either through `global-web-application.xml` or `orion-web.xml`.

  (Invocation by class name is described in "Servlet Invocation by Class Name During OC4J Development" on page 2-22, including additional information about `servlet-webdir` settings.)

  The following configuration in `orion-web.xml`, for example, would disable invocation by class name:

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

- To guard against the guessing or "hacking" of session ID numbers for destructive purposes, OC4J uses `java.security.SecureRandom` functionality to generate random session ID numbers.

3 Servlet Filters and Event Listeners

This chapter describes the following servlet features:

- Servlet Filters
- Event Listeners

Servlet Filters

Servlet filters are used for preprocessing Web application requests and postprocessing responses, as described in the following sections:

- Overview of Servlet Filters
- How the Servlet Container Invokes Filters
- Filtering of Forward or Include Targets
- 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. In this scenario, the servlet must process the request and generate as much of the response as the application requires.

But there are many cases in which 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, may 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 nonsecure 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 one in which you want to apply preprocessing or postprocessing to requests or 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:

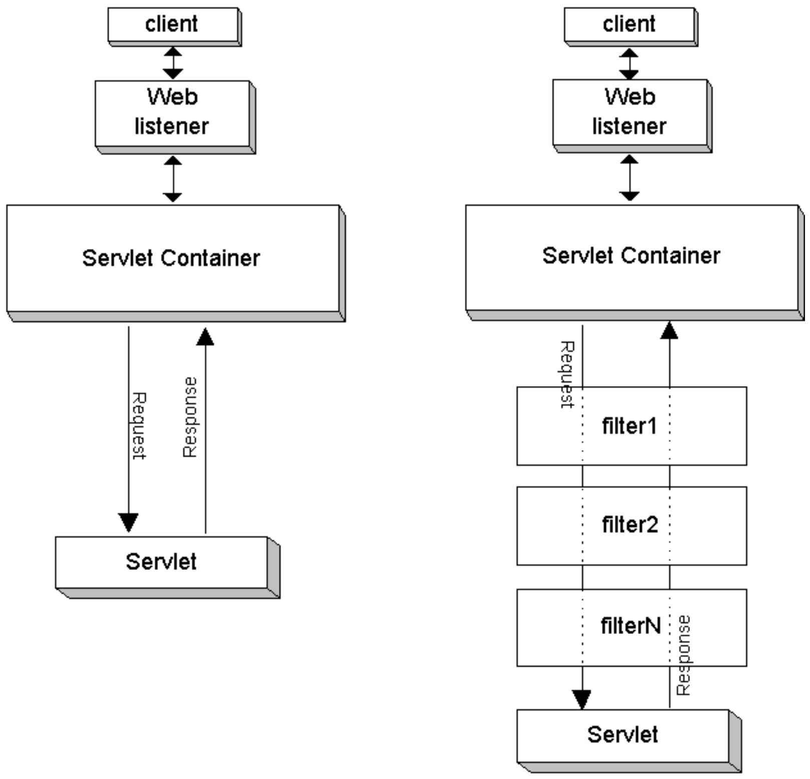
- `init()`
- `destroy()`

- `doFilter()`

How the Servlet Container Invokes Filters

Figure 3–1 shows how the servlet container invokes filters. On the left is a scenario in which 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 and after it has responded. The `web.xml` file specifies which servlets cause the container to invoke the filters.

Figure 3–1 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.

Note: Be careful in coordinating any use of multiple filters, in case of possible overlap in functionality or in what the filters are overwriting.

Filtering of Forward or Include Targets

In the OC4J 10.1.2 implementation, when a servlet is filtered, any servlets that are forwarded to or included from the filtered servlet are *not* filtered by default. You can change this behavior, however, through the following environment setting:

```
oracle.j2ee.filter.on.dispatch=true
```

This flag is set to `false` by default.

Note: This flag is a temporary mechanism for the current release. Future releases will adhere to version 2.4 of the servlet specification, which directs that servlets that are forwarded to or included from a filtered servlet are not filtered by default. But in compliance with the specification, this behavior will be configurable through the `web.xml` file.

See "Servlet Includes and Forwards" on page 2-10 for general information about including and forwarding.

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 `MyGenericFilter`, can be extended by other filters.

Generic Filter Here is the generic filter code. Assume that this generic filter is part of a package `com.example.filter` and set up a corresponding directory structure.

This is an elementary example of an empty (or "pass-through") filter and could be used as a template.

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

public class MyGenericFilter implements javax.servlet.Filter {
  public FilterConfig filterConfig;                                //1

  public void doFilter(final ServletRequest request,               //2
                       final ServletResponse response,
                       FilterChain chain)
      throws java.io.IOException, javax.servlet.ServletException {
    chain.doFilter(request,response);                              //3
  }

  public void init(final FilterConfig filterConfig) {              //4
    this.filterConfig = filterConfig;
  }

  public void destroy() {                                          //5
  }
}
```

Save this code in a file called `MyGenericFilter.java` in the package directory. The numbered code notes refer to the following:

1. This code declares a variable to save the filter configuration object.
2. The `doFilter()` method contains the code that implements the filter.
3. In the generic case, just call the filter chain.

4. The `init()` method saves the filter configuration in a variable.
5. The `destroy()` method can be overridden to accomplish any required finalization.

Filter Code: HelloWorldFilter.java This filter overrides the `doFilter()` method of the `MyGenericFilter` class above. It prints a message on the console when it is called on entrance, then 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 MyGenericFilter {
  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 root directory 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, use the OC4J standalone default Web application. Configure the filter in the `web.xml` file in the default Web application `/WEB-INF` directory (`j2ee/home/default-web-app/WEB-INF`, by default).

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 -->
```

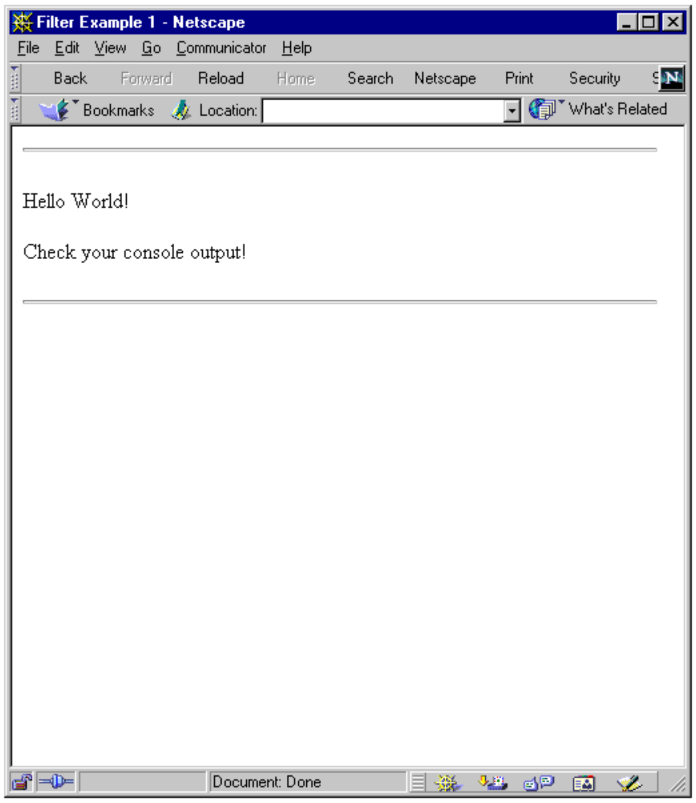
The `<filter>` element defines the name of the filter and the Java class that implements the filter. The `<filter-mapping>` element defines the URL pattern that specifies which targets the `<filter-name>` element should apply to. In this simple example, the filter applies to only one target: the JSP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

Running Example 1 Invoke `filter.jsp` from your Web browser. The console output should look like this:

```
hostname% Entering Filter
Exiting HelloWorldFilter
```

The Web browser output is similar to that shown in Figure 3–2, which follows.

Figure 3–2 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

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

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

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

Filter Code: MessageFilter.java Following is the code to implement the `message` filter example. Note that it uses the `MyGenericFilter` class from "Filter Example 1" on page 3-3.

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

public class MessageFilter extends MyGenericFilter {
  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:

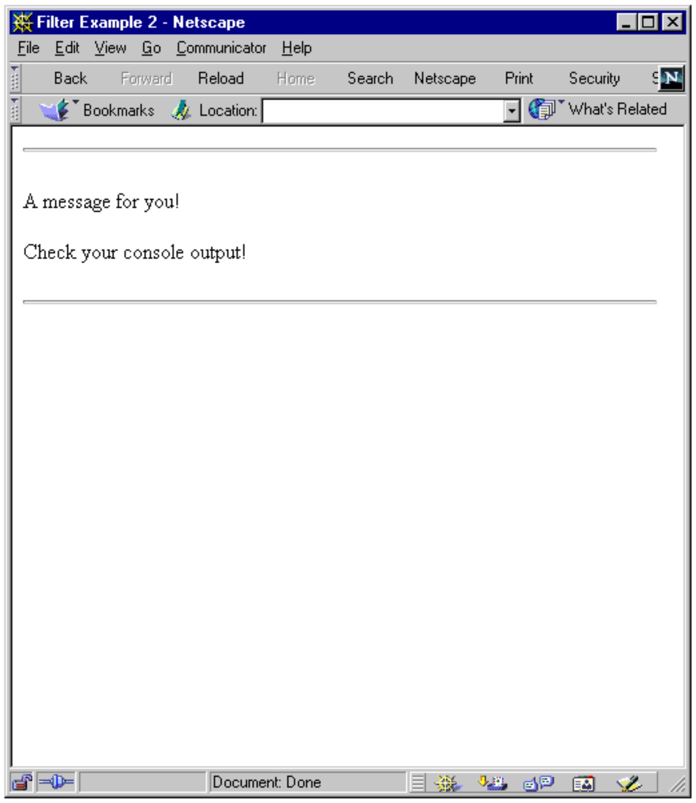
```
<HTML>
<HEAD>
<TITLE>Lesson 2</TITLE>
</HEAD>
<BODY>
<HR>
<P><%=request.getAttribute("message")%></P>
<P>Check your console output!</P>
<HR>
</BODY>
</HTML>
```

Running Example 2 Verify 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 is similar to 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 output is similar to that shown in Figure 3–3, which follows.

Figure 3–3 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.

Create three new classes to implement this example:

- `FilterServletOutputStream`: This class is a new implementation of `ServletOutputStream` for response wrappers.

- `GenericResponseWrapper`: This class is a basic implementation of the response wrapper interface.
- `PrePostFilter`: This class 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 (`j2ee/home/default-web-app` by default), 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 that was generated.

The wrapper must implement some utility methods, such as to retrieve the type and 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 (`j2ee/home/default-web-app` by default), and compile it:

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

Writing the Filter This filter adds content to the response after that target is invoked. This filter extends the filter from "Generic Filter" on page 3-3.

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 MyGenericFilter {

  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 (`j2ee/home/default-web-app` by default), and compile it:

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

As in the previous examples, create a simple JSP page:

```
<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>
```

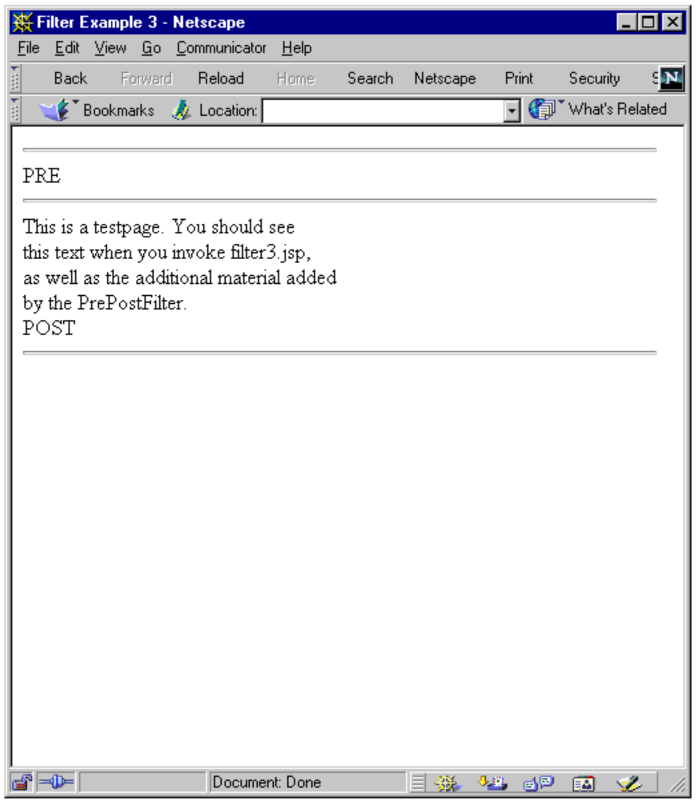
Save this JSP code in `filter3.jsp` in the root directory of the default Web application.

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

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

Running Example 3 Invoke the servlet in your Web browser. You will see a page that looks similar to the page in Figure 3–4, which follows.

Figure 3–4 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 specification includes 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. The following sections describe servlet event listeners:

- 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-level (application-level) event

This event involves resources or state held at the level of the application servlet context object.

- Session-level event

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

Each of these two levels has 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 3–1 summarizes the four categories and the associated interfaces.

Table 3–1 Event Listener Categories and Interfaces

Event Category	Event Descriptions	Java Interface
Servlet context lifecycle changes	Servlet context creation, at which point the first request can be serviced 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 comprising 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 may 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 it 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 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 are notified of servlet context lifecycle events, and the `MyLoggingModule` listener is also notified of session lifecycle events. For servlet context lifecycle events, the `MyConnectionManager` listener is 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; the listener classes themselves 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. The servlet specification discusses this.

Event Listener Methods and Related Classes

This section contains event listener methods that 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 method to retrieve the servlet context object that was created or is about to be destroyed, from which you can obtain information as desired. See "Introduction to Servlet Contexts" on page 1-6 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 method this to get the name of the attribute that was added, removed, or replaced.

- `Object getValue()`

  Use this method 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 method 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 1-5 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 method to get the name of the attribute that was added, removed, or replaced.

- `Object getValue()`

  Use this method 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 method 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, in which the servlets and listener class are declared.

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

http://www.oracle.com/technology/tech/java/oc4j/htdocs/oc4j-how-to.html

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

http://www.oracle.com/technology/

Memberships are free of charge.

Welcome Page: index.jsp

Here is the welcome page, the user interface that enables you to invoke the session-creation servlet by clicking the **Create New Session** link, or to invoke the session-destruction servlet by clicking 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>
```

Note: No new session object is created if you click the **Create New Session** link again after having already created a session from the same client, unless the session has reached a timeout limit or you have explicitly destroyed it in the meantime.

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, the servlet container automatically calls its methods, 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 section shows the listener class. Its `sessionCreated()` method is called by the servlet container when an HTTP session is created, which occurs when you click 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 click the **Destroy Current Session** link. 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
{
   /* A listener class must have a zero-argument constructor: */
   public SessionLifeCycleEventExample()
   {
   }

   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 click the **Create New Session** link in `index.jsp`. Its invocation results in the servlet container creating a request object and associated session object. 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 click 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();
      }
   }
```

4 JDBC and EJB Calls from Servlets

Dynamic Web applications typically access a database to provide content. This chapter, consisting of the following sections, shows how servlets can use JDBC—the Java standard for database connectivity—and Enterprise JavaBeans—used for secure, transactional server-side processing:

Use of JDBC in Servlets

A servlet can access a database using a JDBC driver. The recommended way to use JDBC is to employ an OC4J data source to get the database connection. See *Oracle Application Server Containers for J2EE Services Guide* for information about OC4J data sources. For more information about JDBC, see the *Oracle Database 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 Human Resources (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.

Note: For simplicity, this example makes the following assumptions:

- A database is installed and accessible through `localhost` at port 1521.
- You are using OC4J standalone and the OC4J default Web application, with the default context root of "`/`".

HTML Form

The Web browser accesses a form in a page served through the Web listener. Copy the following HTML into a file, `EmpInfo.html`:

```
<html>

<head>
<title>Query the Employees Table</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>
</form>

</body>
</html>
```

Then save this file in the root directory of the OC4J default Web application (`j2ee/home/default-web-app` by default).

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 construct the database query. Much of the code in this servlet consists of the JDBC statements required to connect to the database server and retrieve and process the query rows.

This servlet makes use of the `init()` method to perform 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. (Verify an appropriate service name is used for the URL.)

```
<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="hr"
        password="hr"
        url="jdbc:oracle:thin:@localhost:1521/myservice"
        inactivity-timeout="30"
/>
```

It is advisable to use only the `ejb-location` JNDI name in the JNDI lookup for an emulated data source. See the *Oracle Application Server Containers for J2EE Services Guide* for more information about data sources.

This example also assumes the following data source definition in the `web.xml` file:

```
  <resource-ref>
```

```
      <res-auth>Container</res-auth>
      <res-ref-name>jdbc/OracleDS</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
   </resource-ref>
```

Here is the servlet code:

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;  // for JNDI
import javax.sql.*;     // extended JDBC interfaces (such as data sources)
import java.sql.*;      // standard JDBC interfaces
import java.io.*;

public class GetEmpInfo extends HttpServlet {

  DataSource ds = null;
  Connection conn = null;

  public void init() throws ServletException {
    try {
      InitialContext ic = new InitialContext();  // JNDI initial context
      ds = (DataSource) ic.lookup("jdbc/OracleDS"); // JNDI lookup
      conn = ds.getConnection();  // database connection through data source
    }
    catch (SQLException se) {
      throw new ServletException(se);
    }
    catch (NamingException ne) {
      throw new ServletException(ne);
    }
  }

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

/* Get the user-specified WHERE clause from the HTTP request, then   */
/* construct the SQL query.                                          */
    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>");

/* Create a JDBC statement object, execute the query, and set up     */
/* HTML table formatting for the output.                             */
    try {
      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>");

/* Loop through the results. Use the ResultSet getString() and       */
/* getInt() methods to retrieve the individual data items.           */
        int count=0;
        while (rs.next()) {
         count++;
         out.println("<tr><td>" + rs.getString(1) + "</td><td>" +rs.getInt(2) +
                    "</td></tr>");

        }
         out.println("</table>");
         out.println("<h3>" + count + " rows retrieved</h3>");

      rs.close();
      stmt.close();
    }
    catch (SQLException se) {
      se.printStackTrace(out);
    }

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

  public void destroy() {
    try {
      conn.close();
    }
    catch (SQLException se) {
      se.printStackTrace();
    }
  }
}
```

Deployment and Testing of the Database Query Servlet

To deploy this example, save the HTML file in the root directory of the OC4J default Web application (`j2ee/home/default-web-app` by default) 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
```

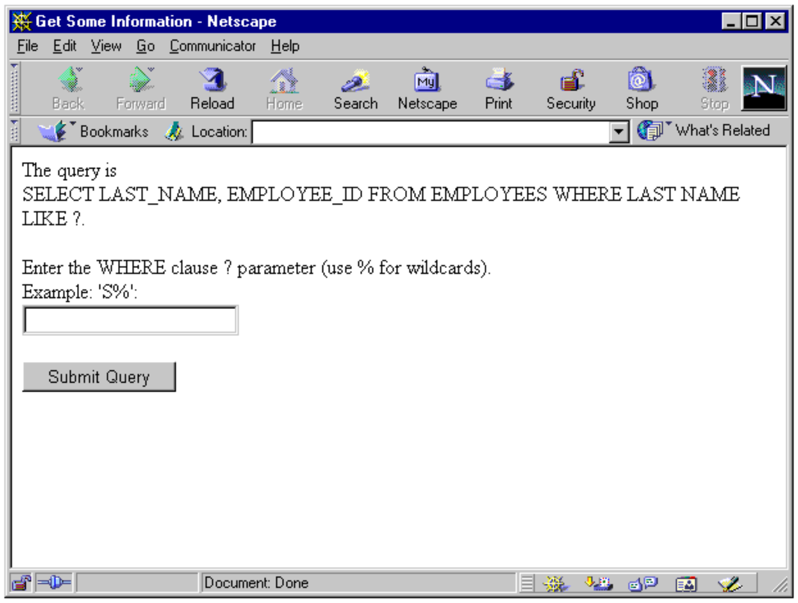
This assumes "`/`" is the context path of the OC4J standalone default Web application.

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

Note: For general information about invoking servlets in OC4J, see "Servlet Invocation" on page 2-19.

When you invoke `EmpInfo.html`, you will see browser output similar to that shown in Figure 4–1, which follows.

Figure 4–1 Employee Information Query

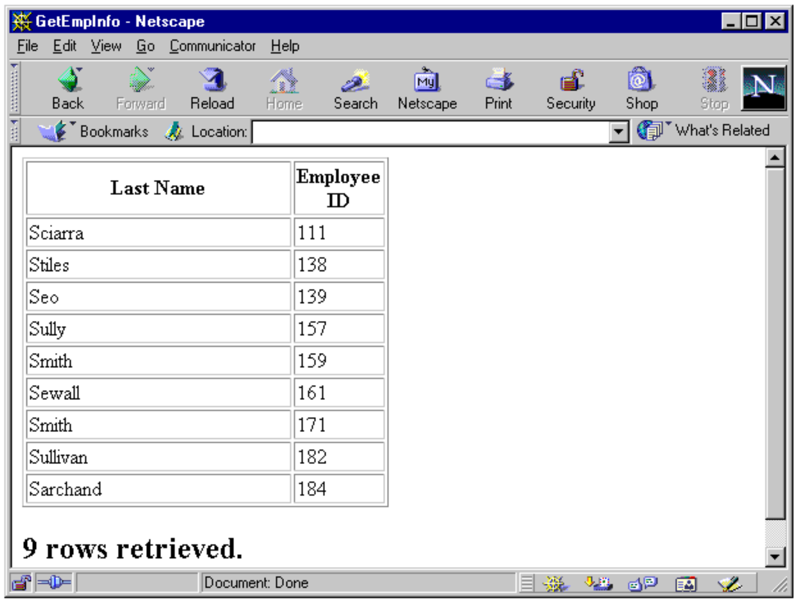

Get Some Information - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: What's Related
Submit Query
Document: Done

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 clicking **Submit Query** calls the `GetEmpInfo` servlet. The output looks like what is shown in Figure 4–2, which follows.

Figure 4–2 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 can call Enterprise JavaBeans 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.

The following sections discuss and provide examples for typical scenarios for the use of EJBs from servlets:

- Servlet-EJB Overview
- EJB Local Lookup
- EJB Remote Lookup within the Same Application
- EJB Remote Lookup Outside the Application

Important: Examples in this section assume you are using OC4J in standalone mode during development. This may affect the URL for a JNDI lookup, as compared to the URL in an Oracle Application Server environment, but otherwise has no effect on the servlet code.

Notes:

- For detailed information about EJB features and for servlet-EJB examples in an Oracle Application Server environment, refer to the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*.
- OC4J provides an EJB tag library to make accessing EJBs from JSP pages more convenient. See the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference* for information.

Servlet-EJB Overview

The following sections provide an overview of considerations for the use of EJBs from servlets:

- Servlet-EJB Scenarios
- EJB Local Interfaces Versus Remote Interfaces

Servlet-EJB Scenarios

The servlet-EJB examples in this chapter cover three scenarios:

- Local lookup: The servlet calls an EJB that is *co-located*, meaning it is in the same application and on the same host, running in the same JVM. The servlet and EJB would have been deployed in the same EAR file, or in EAR files with a parent/child relationship. The example uses EJB local interfaces, which were introduced in version 2.0 of the EJB specification. See "EJB Local Lookup" on page 4-8.
- Remote lookup within the same application: The servlet calls an EJB that is in the same application, but on a different host. (The same application is deployed to both hosts.) This requires EJB remote interfaces. This would be the case for a multitier application where the servlet and EJB are in the same application, but on different tiers. See "EJB Remote Lookup within the Same Application" on page 4-14.

- Remote lookup outside the application: The servlet calls an EJB that is not in the same application. This is a remote lookup and requires EJB remote interfaces. The EJB may be on the same host or on a different host, but is not running in the same JVM. See "EJB Remote Lookup Outside the Application" on page 4-19.

Servlet-EJB communications use JNDI for lookup and RMI for the EJB calls, over either ORMI (the Oracle implementation of RMI) or IIOP (the standard and interoperable Internet Inter-Orb Protocol). For the JNDI initial context factory, the examples in this document use the `ApplicationInitialContextFactory` class, which supports EJB references in `web.xml`, and the `RMIInitialContextFactory` class, which does not. Depending on the situation, another possibility is `ApplicationClientInitialContextFactory`, which supports EJB references in the `application-client.xml` file. For more information about the use of JNDI and RMI with EJBs, refer to the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*.

A remote lookup requires a JNDI environment to be set up, including the URL and a user name and password. This setup is typically in the servlet code, as shown in "EJB Remote Lookup Outside the Application" on page 4-19, but for a lookup in the same application it can be in the `rmi.xml` file instead.

Remote lookup within the same application on different hosts also requires proper setting of the `remote` flag in the `orion-application.xml` file for your application on each host, as shown in "Use of the Remote Flag" on page 4-14.

As in any application where EJBs are used, there must be an entry for each EJB in the `ejb-jar.xml` file.

Note: The examples here consider only an ORMI scenario. For information about using IIOP, see the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*.

EJB Local Interfaces Versus Remote Interfaces

In version 1.1 of the EJB specification, an EJB always had a remote interface extending the `javax.ejb.EJBObject` interface, and a home interface extending the `javax.ejb.EJBHome` interface. In this model, all EJBs are defined as remote objects, adding unnecessary overhead to EJB calls in situations where the servlet or other calling module is co-located with the EJB.

Note: The OC4J `copy-by-value` attribute (of the `<session-deployment>` element of the `orion-ejb-jar.xml` file) is also related to avoiding unnecessary overhead, specifying whether to copy all incoming and outgoing parameters in EJB calls. See the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for information.

Version 2.0 of the EJB specification added support for *local interfaces* for co-located lookups. In this case, the EJB has a local interface that extends the `javax.ejb.EJBLocalObject` interface, in contrast to having a remote interface. In addition, a local home interface that extends the `javax.ejb.EJBLocalHome` interface is specified, in contrast to having a home interface.

Any lookup involving EJB remote interfaces uses RMI and has additional overhead such as for security. RMI and other overhead are eliminated when you use local interfaces.

Notes:

- An EJB can have both local and remote interfaces. The examples in this section use either local interfaces or remote interfaces, but not both.
- The term *local lookup* in this document refers to a co-located lookup, in the same JVM. Do not confuse "local lookup" with "local interfaces". Although local interfaces are typically used in any local lookup, there may be situations in which remote interfaces are used instead. (Local lookups had to be performed this way prior to version 2.0 of the EJB specification.)

EJB Local Lookup

This section presents an example of a single servlet, `HelloServlet`, that calls a single co-located EJB, `HelloBean`, using local interfaces. This is the simplest servlet-EJB scenario.

Here are the key steps of the servlet code:

1. Import the EJB package for access to the bean home and remote interfaces. Also note the imports of `javax.naming` for JNDI, and `javax.rmi` for RMI.
2. Print the message from the servlet.
3. Create an output string, with an error default.
4. Use JNDI to look up the EJB local home interface.
5. Create the EJB local object from the local home.
6. Invoke the `helloWorld()` method on the local object, which puts the EJB output message in a Java string.
7. Print the message from the EJB.

The following sections cover all aspects of the sample:

- Servlet-EJB Application Code for Local Lookup
- Configuration and Deployment for Local Lookup
- Invocation of the Servlet-EJB Application

For further discussion and another complete example of using local interfaces, see the OC4J How-To document at the following location:

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

Servlet-EJB Application Code for Local Lookup

This section has code for a servlet that calls a co-located EJB, using local interfaces. This includes servlet code, EJB code, and EJB interface code. Note the bold passages in particular.

Servlet Code: HelloServlet This section has the servlet code. For a local lookup, the default JNDI context is used.

By default, this servlet uses `ApplicationInitialContextFactory` for the JNDI initial context factory. Therefore, the `web.xml` file is searched for EJB references. The `java:comp` syntax for the JNDI lookup indicates there is a reference defined within the application for the EJB, in this case in the `web.xml` file.

See *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for information about JNDI initial context factory classes.

```
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
import javax.rmi.*;          // for RMI, including PortableRemoteObject
import javax.ejb.CreateException;

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 local home interface.
    try {
      InitialContext ic = new InitialContext();
      HelloLocalHome hlh = (HelloLocalHome)ic.lookup
                           ("java:comp/env/ejb/HelloBean");

      // Step 5: Create the EJB local interface object.
      HelloLocal hl = (HelloLocal)hlh.create();
       // Step 6: Invoke the helloWorld() method on the local object.
       s = hl.helloWorld();
     } catch (NamingException ne) {
         System.out.println("Could not locate the bean.");
     } catch (CreateException ce) {
         System.out.println("Could not create the bean.");
     } catch (Exception e) {
         // Unexpected exception; send back to client for now.
         throw new ServletException(e);
     }
     // Step 7: Print the message from the EJB.
     out.println("<br>" + s);
     out.println("</html>");
  }
}
```

EJB Code: HelloBean Stateful Session Bean The EJB, as shown here, implements a single method, `helloWorld()`, that returns a greeting to the caller. The local home and local EJB interface code is also shown below.

```
package myEjb;
```

```
import javax.ejb.*;

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

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

EJB Interface Code: Local Home and Local Interfaces Here is the code for the local home interface:

```
package myEjb;

import javax.ejb.EJBLocalHome;
import javax.ejb.CreateException;

public interface HelloLocalHome extends EJBLocalHome
{
  public HelloLocal create () throws CreateException;
}
```

Here is the code for the local interface:

```
package myEjb;

import javax.ejb.EJBLocalObject;

public interface HelloLocal extends EJBLocalObject
{
  public String helloWorld ();
}
```

Configuration and Deployment for Local Lookup

This section discusses the deployment steps and configuration for the Servlet-EJB local lookup sample application. In the descriptor files, note the bold passages in particular. To deploy this application, you will need an EAR file that contains the following:

- A WAR (Web archive) file that includes the servlet code and `web.xml` Web descriptor
- An EJB JAR archive file that includes the EJB code and `ejb-jar.xml` EJB descriptor
- The `application.xml` application-level descriptor

See Chapter 5, "Deployment and Configuration Overview", for an overview of deployment to OC4J. See the *Oracle Application Server Containers for J2EE User's Guide* for detailed information.

Web Descriptor and Archive Create a standard `web.xml` Web descriptor as follows. Note the `<ejb-local-ref>` element and its `<local-home>` and `<local>` subelements for the use of local interfaces.

```
<?xml version="1.0"?>
```

```
<!DOCTYPE WEB-APP PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.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-local-ref>
    <ejb-ref-name>ejb/HelloBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>myEjb.HelloLocalHome</local-home>
    <local>myEjb.HelloLocal</local>
  </ejb-local-ref>
</web-app>
```

Next, create the standard J2EE directory structure for Web application deployment, then move the `web.xml` Web deployment descriptor and the compiled servlet class file into the structure. After you create and populate the directory structure, create a WAR file named `myapp-web.war` (for example) to contain the files. Here are the WAR file contents:

```
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
```

(The JAR utility automatically creates the `MANIFEST.MF` file.)

EJB Descriptor and Archive Create a standard `ejb-jar.xml` EJB descriptor as follows. Note that the `<ejb-ref-name>` value in the `web.xml` file above corresponds to the `<ejb-name>` value here. In this example, they use the same name, which is a good practice but is not required. The Web tier can specify any reference name, which is independent of the JNDI name.

Also note the `<local-home>` and `<local>` elements, for the use of local interfaces. These must be the same entries as in the `web.xml` file.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <description>Hello Bean</description>
      <ejb-name>ejb/HelloBean</ejb-name>
      <local-home>myEjb.HelloLocalHome</local-home>
      <local>myEjb.HelloLocal</local>
      <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 named `myapp-ejb.jar` (for example) with the standard J2EE structure to hold the EJB components. Here are the JAR file contents:

```
META-INF/
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
myEjb/
myEjb/HelloBean.class
myEjb/HelloLocalHome.class
myEjb/HelloLocal.class
```

(The JAR utility automatically creates the `MANIFEST.MF` file.)

Application-Level Descriptor To deploy the application, create a standard 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
 1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">

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

The `<context-root>` element is required, but in an OC4J standalone environment it is ignored. The context path is actually specified through the `root` attribute of the appropriate `<web-app>` element in `http-web-site.xml`, as shown in "Deployment Configuration" below. For consistency, and to avoid confusion, use the same setting for `<context-root>` as for the `<web-app>` `root` attribute.

Finally, create an EAR file named `myapp.ear` (for example) with the standard J2EE structure to hold the application components. Here are the EAR file contents:

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

(The JAR utility automatically creates the `MANIFEST.MF` file.)

Deployment Configuration To deploy the application, the following entry is added to the `server.xml` file in the OC4J configuration files directory, specifying the appropriate path information:

```
<application
   name="myapp"
   path="your_path/lib/myapp.ear"
/>
```

If you use the `admin.jar -deploy` option to deploy the application, this entry is made automatically. (See "Using admin.jar to Deploy the EAR File" on page 5-27.)

The next step is to bind the Web module to a Web site. You will need the following entry in the Web site XML file (typically `http-web-site.xml` in OC4J standalone) in the OC4J configuration files directory:

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

If you use the `admin.jar -bindWebApp` option after deploying the application, this entry is made automatically. (See "Using admin.jar to Bind the Web Application" on page 5-28.)

Invocation of the Servlet-EJB Application

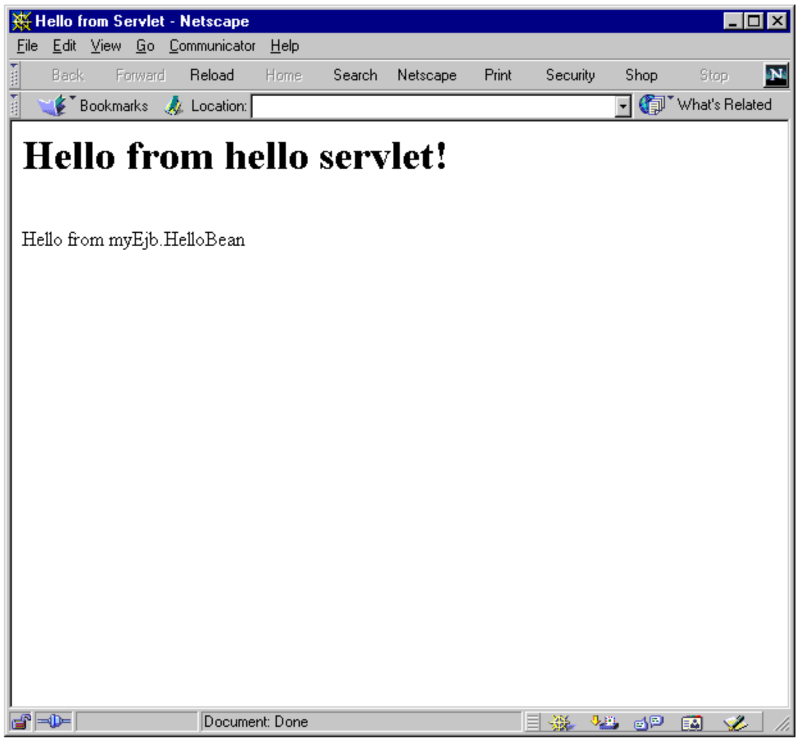
According to the configuration of this example, you can invoke the servlet as follows, assuming a host `myhost`. By default, OC4J standalone uses port 8888.

```
http://myhost:8888/myapp/DoubleHello
```

The context path, `myapp`, is according to the relevant `<web-app>` element `root` setting in the Web site XML file. The servlet path, `DoubleHello`, is according to the relevant `<url-pattern>` element in the `web.xml` file.

Figure 4–3, which follows, shows the application output to a Web browser. 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 4–3   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 Remote Lookup within the Same Application

This section adapts the preceding `HelloServlet`/`HelloBean` example for remote lookup within the same application, where the servlet and EJB are in the same application but on different tiers. The discussion highlights use of the `orion-application.xml` file `remote` flag, which determines where EJBs are deployed and searched for, and necessary changes to the code and descriptor files.

In this example, the default `ApplicationInitialContextFactory` is used for the JNDI context, as in the preceding local lookup example. An alternative would be to use `RMIInitialContextFactory`, as discussed in the next example, "EJB Remote Lookup Outside the Application" on page 4-19.

Use of the Remote Flag

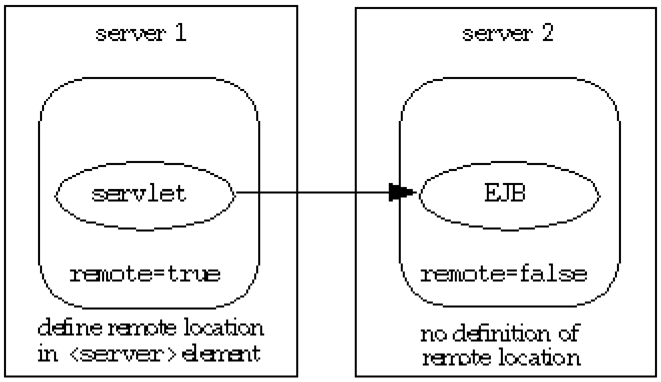
In OC4J, to perform a remote EJB lookup within the same application but on different tiers (where the same application has been deployed to both tiers), you must set the EJB `remote` flag appropriately on each tier. When this flag is set to "`true`", beans will be looked up on a remote server instead of the EJB service being used on the local server.

The `remote` flag is an attribute in the `<ejb-module>` subelement of an `<orion-application>` element in the `orion-application.xml` file. The default setting is `remote="false"`. Here is an example of setting it to "`true`":

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

The suggested steps are illustrated in Figure 4–4 and described below.

Figure 4–4 Setup for Remote Lookup within Application


server 1
servlet
remote=true
define remote location in <server> element
server 2
EJB
remote=false
no definition of remote location

1. Deploy the application EAR file to both servers, with a `remote` flag value of "`false`". If you provide an `orion-application.xml` file, it is suggested that it either have the `remote` flag explicitly set to `"false"`, or no `remote` flag setting at all, in which case its value is "`false`" by default. If you do not provide `orion-application.xml`, OC4J generates the file automatically with the `remote` flag disabled.
2. Set `remote="true"` in the `orion-application.xml` file for your application on server 1, the servlet tier. Given this setting, the servlet will not look for EJBs on server 1.
3. Ensure that `remote="false"` in the `orion-application.xml` file for your application on server 2, the EJB tier. Given this setting, the servlet will look for EJBs on server 2.
4. Use a `<server>` element in the `rmi.xml` file on server 1 to specify the remote host, server 2, where the servlet will look for the EJB. This includes the host name and port as well as a user name and password for authentication:

   ```
   <rmi-server ... >
   ...
      <server host="remote_host" port="remote_port" username="user_name"
              password="password" />
   ...
   </rmi-server>
   ```

   If there are `<server>` elements for multiple remote servers, the OC4J container will search all of them for the target EJB.

   Note: In the `rmi.xml` configuration, use the default administrative user name for the remote host, and the administrative password set up on the remote host through the OC4J `-install` option. This avoids possible JAZN configuration issues. See "Setting Up an Administrative User and Password" on page 5-24.

5. Ensure that your deployment and configuration file changes are picked up by OC4J on each server. You can accomplish this (on each server) in any of the following ways:

   – If the `check-for-updates` flag is enabled

   – By using the `admin.jar -updateConfig` option

   – By restarting the server

   See "Key OC4J Flags for Development" on page 2-3 for information about `check-for-updates` and `-updateConfig`.

> **Note:** Remember that this discussion assumes an OC4J standalone environment during development. In an Oracle Application Server environment, any configuration must be through Enterprise Manager or the `dcmctl` command-line utility. If updating `orion-application.xml` is not feasible after deployment, you would have to create and deploy two separate EAR files, one with an `orion-application.xml` file with `remote="true"` and one with an `orion-application.xml` file with `remote="false"`.
>
> Servlet EJB calls in an Oracle Application Server environment are discussed in the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*.

Servlet-EJB Application Code for Remote Lookup in the Same Application

This section has code for a servlet-EJB sample using remote lookup of an EJB component within the same application. This includes servlet code, EJB code, and EJB interface code. Note the bold passages in particular.

Servlet Code: HelloServlet This section contains the servlet code. In this example, the code and configuration are fundamentally the same as in the local lookup example, with two exceptions:

- This example uses remote interfaces instead of local interfaces.
- This example uses the `javax.rmi.PortableRemoteObject.narrow()` static method to ensure that objects can be cast to the desired type. This was not required in the local lookup example, but is mandatory for any remote lookup.

Again, the default `ApplicationInitialContextFactory` is used for the JNDI initial context factory, the `java:comp` syntax is used for the lookup, and the `web.xml` file is searched for EJB references.

This example assumes that the `rmi.xml` file on the servlet tier has been set up to specify the host, port, user name, and password for the remote lookup, as shown in the preceding section, "Use of the Remote Flag". With this assumption, there is no need to set up a JNDI context (URL, user name, and password) in the servlet code.

```
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
import javax.rmi.*;         // for RMI, including PortableRemoteObject
import javax.ejb.CreateException;
import java.rmi.RemoteException;

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 {
     InitialContext ic = new InitialContext();
     Object homeObject = ic.lookup("java:comp/env/ejb/HelloBean");
     HelloHome hh = (HelloHome)
        PortableRemoteObject.narrow(homeObject,HelloHome.class);

      // Step 5: Create the EJB local interface object.
      HelloRemote hr = (HelloRemote)
       PortableRemoteObject.narrow(hh.create(),HelloRemote.class);
      // Step 6: Invoke the helloWorld() method on the local object.
      s = hr.helloWorld();
     } catch (NamingException ne) {
         System.out.println("Could not locate the bean.");
     } catch (CreateException ce) {
         System.out.println("Could not create the bean.");
     } catch (RemoteException ce) {
         System.out.println("Error during execution of remote call.");
     } catch (Exception e) {
         // Unexpected exception; send back to client for now.
         throw new ServletException(e);
     }
    // Step 7: Print the message from the EJB.
    out.println("<br>" + s);
    out.println("</html>");
  }
}
```

EJB Code: HelloBean Stateful Session Bean The EJB code for a remote lookup within the same application is similar to that for a local lookup, but adds a `RemoteException`. The home and remote EJB interface code is also shown below.

```
package myEjb;

import javax.ejb.*;

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

  public void ejbCreate () throws 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. Extend `EJBHome` (instead of `EJBLocalHome` as when local interfaces are used). Also, a `RemoteException` is added.

```
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. Extend `EJBObject` (instead of `EJBLocalObject` as when local interfaces are used). As with the home interface above, the use of `RemoteException` is added.

```
package myEjb;

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

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

Configuration for Remote Lookup in the Same Application

This section highlights `web.xml` and `ejb-jar.xml` entries to use with remote interfaces. You can compare the highlighted passages to parallel passages in the local interface example.

Notes:

- The `web.xml` and `ejb-jar.xml` files are the same for deployment to each host.
- The `server.xml` entry on each host is the same as for the local lookup example, as shown in "Deployment Configuration" on page 4-13. This is handled automatically if you use the `admin.jar -deploy` option to deploy the application. The `http-web-site.xml` entry on the servlet tier is the same as for the local lookup example, but is not applicable on the EJB tier.

The `remote` flag must also be set appropriately in `orion-application.xml` on each host, as discussed in "Use of the Remote Flag" on page 4-14.

Web Descriptor The contents of `web.xml` for this example are as follows. Note the `<ejb-ref>` element and its `<home>` and `<remote>` subelements, for use of remote interfaces.

```
<?xml version="1.0"?>
<!DOCTYPE WEB-APP PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.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>
```

EJB Descriptor For this example, the contents of `ejb-jar.xml` are as follows. Note the `<home>` and `<remote>` elements, for the use of remote interfaces.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.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>
```

EJB Remote Lookup Outside the Application

This section adapts the preceding `HelloServlet`/`HelloBean` example for remote lookup to a different application (deployed to a different OC4J instance), highlighting necessary changes to the code and descriptor files.

Instead of using the default JNDI initial context factory, `ApplicationInitialContextFactory`, this example uses `RMIInitialContextFactory`.

The `remote` flag discussed in the preceding section, "EJB Remote Lookup within the Same Application" on page 4-14, is not relevant.

Servlet-EJB Application Code for Remote Lookup Outside the Application

This section has code for a servlet-EJB sample using remote lookup outside the application. This includes servlet code, EJB code, and EJB interface code. Note the bold passages in particular.

Servlet Code: HelloServlet This section contains the servlet code. In this scenario, the specification of URL, user, and password must be in the servlet code. (In the example of a remote lookup within the same application, there is an assumption that this information is specified in the `rmi.xml` file.) A step is added to the servlet code here to set up the JNDI environment for the lookup. In this code, the following are static fields of the `javax.naming.Context` interface, which is implemented by the `javax.naming.InitialContext` class:

- The `INITIAL_CONTEXT_FACTORY` setting specifies the initial context factory to use, `RMIInitialContextFactory` in this case.
- The `SECURITY_PRINCIPAL` setting specifies the identity of the principal (user name) for authenticating the caller to the service.
- The `SECURITY_CREDENTIALS` setting specifies the password of the principal for authenticating the caller to the service.
- The `PROVIDER_URL` setting specifies the URL, or a comma-delimited list of URLs, for the lookup. The information after the port number corresponds to the application name as defined in the `server.xml` file, "`myapp`" in this example.

When `RMIInitialContextFactory` is used, there is no `java:comp` syntax in the JNDI lookup of the remote EJB component you wish to connect to, and the lookup must use the EJB name as specified in the `ejb-jar.xml` file. The `web.xml` file is not accessed, so any EJB references there will be ignored for the lookup.

```
package myServlet;

// Step 1: Import the EJB package.
import myEjb.*;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;       // for JNDI
import javax.rmi.*;          // for RMI, including PortableRemoteObject
import javax.ejb.CreateException;
import java.rmi.RemoteException

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 2.5: Set up JNDI properties for remote call
    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, "welcome");
    env.put(Context.PROVIDER_URL, "ormi://myhost:port/myapp");

    // 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 {
      InitialContext ic = new InitialContext(env);
      Object homeObject = ic.lookup("ejb/HelloBean");
      HelloHome hh = (HelloHome)
        PortableRemoteObject.narrow(homeObject, HelloHome.class);

      // Step 5: Create the EJB remote interface.
      HelloRemote hr = (HelloRemote)
       PortableRemoteObject.narrow(hh.create(), HelloRemote.class);
       // Step 6: Invoke the helloWorld() method on the remote object.
       s = hr.helloWorld();
     } catch (NamingException ne) {
         System.out.println("Could not locate the bean.");
     } catch (CreateException ce) {
         System.out.println("Could not create the bean.");
     } catch (RemoteException ce) {
         System.out.println("Error during execution of remote call.");
     } catch (Exception e) {
         // Unexpected exception; send back to client for now.
         throw new ServletException(e);
     }
    // Step 7: Print the message from the EJB.
    out.println("<br>" + s);
    out.println("</html>");
  }
}
```

Notes:

- In the JNDI properties setup, use the default administrative user name for the remote host, and the administrative password set up on the remote host through the OC4J `-install` option. This avoids possible JAZN configuration issues. See "Setting Up an Administrative User and Password" on page 5-24.
- For an Oracle Application Server environment, because of OPMN dynamic port assignments, use "`opmn:ormi://...`" syntax instead of "`ormi://...`" syntax for the ORMI URL.
- In OC4J standalone cluster mode, use "`lookup:ormi://...`" syntax.

EJB Code: HelloBean Stateful Session Bean The EJB code for a remote lookup outside the application, including the bean code and the interface code, is identical to that for a remote lookup within the application, including the use of `RemoteException`.

```
package myEjb;

import javax.ejb.*;

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

  public void ejbCreate () throws 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;
}
```

Configuration and Deployment for Remote Lookup Outside the Application

This section highlights `ejb-jar.xml` entries that are specific to remote lookup. These entries are identical to those for remote lookup within the application. You can compare the highlighted passages to parallel passages in the other examples.

Because the servlet uses `RMIInitialContextFactory` for the JNDI initial context factory, the `web.xml` file is not relevant.

Note: The `ejb-jar.xml` file is the same for deployment to each host.

The `server.xml` entry on the local host is identical to that for the local lookup example, as shown in "Deployment Configuration" on page 4-13. This is handled automatically if you use the `admin.jar -deploy` option to deploy the application. The `server.xml` file on the remote host is configured as appropriate for the remote application. The `http-web-site.xml` entry on the local host is identical to that for the local lookup example, but is not applicable on the remote host.

EJB Descriptor and Archive The contents of `ejb-jar.xml` are as follows. Note the `<home>` and `<remote>` elements, for use of remote interfaces.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
 1.12//EN" "http://java.sun.com/dtd/ejb-jar_2_0.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>
```

Deployment Notes for Remote Lookup Outside the Application Complete the following steps:

1. To deploy the remote EJBs, place them in a separate EAR file, and deploy them to the appropriate OC4J server. The server you deploy to is reflected in the `PROVIDER_URL` in the servlet code.
2. Ensure that the remote and home interfaces are available to the calling servlet. For simplicity, you can make the whole EJB JAR file available in either of the following ways:
   - Place it in the `/WEB-INF/lib` directory of the WAR file.
   - Place it anywhere, as desired, and point to it through a `<library>` element in the `orion-application.xml` file of the application.

5 Deployment and Configuration Overview

This chapter provides an overview of OC4J configuration, packaging, and deployment for servlet developers, primarily in an OC4J standalone environment. It includes the following sections:

General Overview of OC4J Deployment and Configuration

Because this is a developer's guide, much of it is intended for use in an OC4J standalone environment, which is convenient for application development stages. OC4J standalone comprises a single OC4J instance outside the Oracle Application Server environment. Most of this chapter is specific to configuration and deployment in a standalone environment, where you are developing on the same system you are deploying to.

When your application is ready for enterprise use, you can deploy it to the Oracle Application Server environment. This chapter gives an overview of deployment and configuration in Oracle Application Server, and Chapter 7, "Configuration with Enterprise Manager", offers additional information . Your primary information source for using OC4J in an Oracle Application Server environment, however, should be the *Oracle Application Server Containers for J2EE User's Guide*.

The following sections provide some overview:

Overview: OC4J Standalone Versus the Oracle Application Server Environment

Many OC4J features discussed in this manual, particularly in this chapter, are for use in OC4J standalone only, during development. In Oracle Application Server, which offers enterprise management features for large-scale production environments, it is critical to maintain controls to prevent actions that may compromise the server during operation. Because this is not such a concern while you are developing in a standalone environment, there are fewer restrictions on what you can or should do in OC4J standalone.

OC4J standalone provides the `admin.jar` command-line utility for deploying, configuring, and managing applications. It is also possible, especially for early testing, to manually deploy files and manually update configuration files. In particular, for initial testing, you can use the OC4J default Web application for individual servlet files, JSP pages, and dependency classes.

Note: Key `admin.jar` commands are discussed under "Deployment Scenarios to OC4J Standalone" on page 5-23.

For initial considerations when using OC4J standalone for development, including use of the OC4J `development` flag to trigger automatic recompilation and reloading of modified servlets, see "OC4J Standalone for Development" on page 2-1.

In an enterprise production environment, OC4J is contained within Oracle Application Server, which takes over management of the J2EE enterprise systems. Oracle Application Server can oversee multiple clustered OC4J processes and is managed through the Oracle Enterprise Manager 10*g*. 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 manually updating configuration files, because this will conflict with the management provided by Enterprise Manager.

Table 5–1 summarizes OC4J deployment and configuration features, comparing OC4J standalone to OC4J in Oracle Application Server.

Table 5–1 OC4J in Standalone Versus Oracle Application Server: Deployment

Feature	OC4J Standalone	OC4J in Oracle Application Server
Deployment vehicle	Use `admin.jar` or manually place files.	Use Enterprise Manager or `dcmctl`.
Configuration vehicle	Use `admin.jar` or manually update files.	Use Enterprise Manager or `dcmctl`. Do *not* manually update files.
Deployment packaging	Use an EAR file, a WAR file, loose files in a J2EE application directory structure, or loose files in a Web application directory structure.	Use an EAR file or a WAR file.
Default J2EE application or J2EE application wrapper	The OC4J default J2EE application is available to contain independent WAR files. You do not have to create an EAR file for simple Web applications.	When you deploy an independent WAR file, OC4J automatically creates a J2EE application and an EAR file to wrap it.
Default Web application	The OC4J default Web application allows deployment of servlets through placement of files under the default root directory. No configuration is required.	Not applicable.

Table 5–1 (Cont.) OC4J in Standalone Versus Oracle Application Server: Deployment

Feature	OC4J Standalone	OC4J in Oracle Application Server
Automatic reloading of application or Web application	Modify `application.xml`, `web.xml`, servlet code under `/WEB-INF/classes`, or JAR file under `/WEB-INF/lib`.	Not applicable / appropriate.
Automatic recompilation and reloading	Use `development="true"` (or set JSP `main_mode` to "`recompile`" for JSP pages).	Not applicable / appropriate.

Table 5–2 summarizes OC4J features and practices relating to Web sites, again comparing OC4J standalone to OC4J in Oracle Application Server.

Table 5–2 OC4J in Standalone Versus Oracle Application Server: Web Sites

Feature	OC4J Standalone	OC4J in Oracle Application Server
Web server	OC4J	Oracle HTTP Server
Protocol	HTTP / HTTPS	AJP / secure AJP
Default port to invoke Web applications	8888	7777 (for Oracle HTTP Server and Oracle Application Server Web Cache)
Web site XML file	`http-web-site.xml`	`default-web-site.xml`

Note: In Oracle Application Server, use Enterprise Manager *or* `dcmctl`, but do not attempt to use both simultaneously to target the same OC4J instance or instances, or do not use both for different parts of the same deployment.

Overview of OC4J Deployment Scenarios

OC4J supports the standard J2EE application structure and deployment vehicles. This includes the use of an EAR file to deploy a complete J2EE application, which may include Web modules, EJB modules, application client modules, and resource adapter modules (used for connector factories). There can be zero or more of each type of module. It is also possible to use an independent WAR file to deploy an independent Web application. (For a complete application that includes a Web module, the WAR file is included inside the EAR file.) For more information about these features, refer to the J2EE specification and servlet specification, available at the following Web sites:

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

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

Before deploying a J2EE application, you must complete the following steps:

1. Create all components of the application, such as static HTML files, servlets, JSP pages, and EJBs.
2. Create J2EE descriptors, such as `application.xml` and `web.xml`, and, as desired, create OC4J descriptors, such as `orion-application.xml` and `orion-web.xml`. If you do not create the OC4J descriptors, they will be generated automatically during deployment of a J2EE application, which is sufficient if you do not need anything beyond default settings.

3. Package the application components and descriptors according to the J2EE application structure. If you provide `orion-application.xml`, place it with `application.xml`. If you provide `orion-web.xml`, place it with `web.xml`. Although it is possible to deploy loose files into the appropriate directory structure, it is more typical to deploy applications in EAR or WAR files. See "Application Packaging" on page 5-20.

> **Note:** In Oracle Application Server, deployment and configuration through Enterprise Manager results in the appropriate OC4J configuration files being created or updated automatically.

After you have packaged your application, there are several scenarios for deployment, as discussed later in this chapter.

In an OC4J standalone environment, your options include the following:

- Deploy an EAR file. See "Deploying an EAR File to OC4J Standalone" on page 5-27.
- Manually deploy files for a complete J2EE application into an application directory structure. See "Deploying Files into a J2EE Application Structure on OC4J Standalone" on page 5-32.
- Deploy an independent WAR file. See "Deploying an Independent WAR File to OC4J Standalone" on page 5-32.
- Manually deploy files for an independent Web application into a Web application directory structure. See "Deploying Files into a Web Application Directory Structure on OC4J Standalone" on page 5-34. This includes the option of conveniently deploying servlets or JSP pages into the OC4J default Web application during initial testing. (See "OC4J Default Application and Default Web Application" on page 5-25.)

In an Oracle Application Server environment, use Enterprise Manager to deploy and configure your applications. See "Overview of OC4J Deployment and Configuration in Oracle Application Server" on page 5-39.

> **Note:** You can also use an IDE, such as Oracle JDeveloper, for developing, packaging, deploying, and configuring your application. See "Oracle JDeveloper Support for Servlet Development" on page 2-19 for an introduction and overview.

Using Oracle Deployment Tools Versus Expert Modes

This discussion considers two modes of operation when deploying and configuring applications for OC4J. One can be called *supported client mode*, using tools provided by Oracle for either OC4J standalone or Oracle Application Server. The other, used during development and testing phases and in OC4J standalone only, can be called *expert mode*. In expert mode, you are manipulating files directly—manually placing EAR files, WAR files, or loose files on the system, and manually updating configuration descriptors. These modes are summarized in Table 5–3.

Table 5–3 Tools or Vehicles for Supported Client Mode and Expert Mode

Mode	Tool or Vehicle for OC4J Standalone	Tool or Vehicle for OC4J in Oracle Application Server
Supported client mode	`admin.jar`	Enterprise Manager or `dcmctl`
Expert mode	Direct manipulation of server files	Not applicable / appropriate

In expert mode, you are operating outside the safeguards and constraints of the OC4J and Oracle Application Server tools.

In OC4J standalone, Oracle generally assumes that you will either deploy with `admin.jar` and not manually update files, or you will manually update files and not deploy with `admin.jar`. Crossing between these scenarios may cause unpredictable results.

In Oracle Application Server, do *not* try to directly manipulate server files. The Oracle Application Server Distributed Configuration Management subsystem (DCM) maintains a repository of configuration information. This repository—not configuration files on the file system—contains the true configuration settings. Enterprise Manager and the `dcmctl` command-line tool work in concert with DCM so that the configuration repository is properly updated when you use either of these tools.

Overview of Configuration Files

There are several categories and levels of configuration files—both OC4J-specific and J2EE-standard, and both at the global level and the application level—for configuring OC4J and an OC4J application.

While developing and testing your application in an OC4J standalone environment, you can manipulate these configuration files either manually or through the `admin.jar` utility, which OC4J provides. For example, the `admin.jar -deploy` command automatically updates `server.xml` to add the specified J2EE application to OC4J, and the `admin.jar -bindWebApp` command automatically updates the specified Web site XML file to bind the specified Web module to the Web site.

Important: In an Oracle Application Server environment, nearly all configuration is accomplished through Oracle Enterprise Manager 10*g*. Do *not* directly manipulate the configuration files discussed here. Doing so would undermine enterprise management and cause undesirable results.

Key `admin.jar` commands are discussed later in this chapter, where applicable. See the *Oracle Application Server Containers for J2EE Stand Alone User's Guide* for further details about the `admin.jar` utility.

The following sections introduce OC4J and application configuration files:

- Introduction to OC4J and J2EE Configuration Files
- OC4J Top-Level Server Configuration File: server.xml
- OC4J and J2EE Application Descriptors
- OC4J and J2EE Web Descriptors
- OC4J Web Site Descriptors

- Example: Mappings to and from Web Site Descriptors

Refer to the following for additional information:

- For the use of Enterprise Manager to configure OC4J in an Oracle Application Server environment, see the *Oracle Application Server Containers for J2EE User's Guide*.
- For the use of Enterprise Manager to configure servlets and Web modules in particular, see Chapter 7, "Configuration with Enterprise Manager", later in this manual.
- To configure OC4J standalone, see the *Oracle Application Server Containers for J2EE Stand Alone User's Guide*, available with the OC4J download from the Oracle Technology Network.

Introduction to OC4J and J2EE Configuration Files

You can divide the OC4J and J2EE configuration files into five categories:

- Server configuration (OC4J-specific), with the overall top-level OC4J configuration file and server-level configuration files for security, data sources, RMI, JMS, and load balancing
- Global configuration (OC4J-specific), with a global application descriptor, a global Web descriptor, and a global descriptor for resource adapters
- Web site configuration (OC4J-specific)
- J2EE application-level configuration, with standard J2EE application, Web, EJB, application-client, and resource adapter (connector factory) descriptors
- OC4J-specific application-level configuration, with OC4J application, Web, EJB, application-client, and resource adapter (connector factory) descriptors

The global files can affect anything running on the OC4J server and can establish defaults for both J2EE and OC4J-specific features at the application level and the Web-site level. The J2EE files can override any defaults for standard J2EE features and establish additional J2EE-standard settings. The OC4J-specific application-level files can override defaults in the corresponding global files, override settings in the corresponding J2EE files, and add OC4J-specific features and settings.

Of particular interest to servlet developers are the top-level OC4J server configuration file (`server.xml`), the application descriptors (OC4J global, J2EE application-level, and OC4J application-level), the Web descriptors (OC4J global, J2EE application-level, and OC4J application-level), and the OC4J Web site descriptors. Each of these topics is discussed in more detail shortly.

Server-level and global configuration files are located in the `j2ee/home/config` directory by default. OC4J looks there for the `server.xml` file. The `server.xml` file specifies the locations of the other server-level and global files (by default, the same directory). In OC4J standalone, the configuration file directory is configurable through the `java -config` command-line option.

The following discussion summarizes the preceding five configuration file categories.

Summary of Server, Global, and Web Site Configuration Files

The `server.xml` file is located in the `j2ee/home/config` directory and specifies locations of the other server configuration files and the Web site configuration files, which are also in `j2ee/home/config` by default.

Server Configuration The files in this category are OC4J-specific, configuring different aspects of the OC4J server.

- `server.xml`: This parent file for all OC4J configuration includes elements pointing to other server-level and global configuration files, all Web sites on the server, and all applications on the server (including the OC4J default application). See "OC4J Top-Level Server Configuration File: server.xml" below.
- `jazn.xml`: This configuration file for Oracle Application Server Java Authentication and Authorization Service (JAAS) Provider (OracleAS JAAS Provider) specifies the directory path to the server-level `jazn-data.xml` file.
- `jazn-data.xml`: This server-level JAAS file contains user name and role information for the XML-based provider. There is also an application-level version. This file is not used if OracleAS JAAS Provider uses the LDAP-based provider instead.
- `data-sources.xml`: This file contains data source definitions for database connections.
- `rmi.xml`: This file contains configuration features for remote method invocation.
- `jms.xml`: This file contains configuration features for the Java Message Service.

OC4J Global Configuration Files in this category are OC4J-specific, defining settings for OC4J global features such as the default application and determining default settings for corresponding application-level configuration files.

- `application.xml`: This file is the OC4J-specific global application descriptor. See "OC4J and J2EE Application Descriptors" on page 5-11.

> **Note:** Do not confuse the OC4J global application descriptor with the J2EE-standard application-level descriptor `application.xml`. They are both used to define Web modules and have other similar features, but the OC4J global application descriptor uses an OC4J-specific DTD.

- `global-web-application.xml`: This file is the OC4J-specific global Web descriptor. See "OC4J and J2EE Web Descriptors" on page 5-15.
- `oc4j-connectors.xml`: This file is the OC4J-specific global descriptor for resource adapters (for connector factories).

Web Site Configuration Each Web site that is recognized by the server has a Web site XML file to configure it. There is just one Web site in Oracle Application Server. There is typically one Web site in OC4J standalone, but you can use a second Web site for "shared" applications, such as those in which some communication is through HTTP and some through HTTPS. It is also possible for there to be no Web sites, if the OC4J instance is not used for Web modules. See "OC4J Web Site Descriptors" on page 5-18.

- `default-web-site.xml`: This file is the default Web site descriptor in an Oracle Application Server environment.
- `http-web-site.xml`: This file is the default Web site descriptor in an OC4J standalone environment.
- Additional Web site XML files: Create a separate Web site XML file, named as desired, for any additional Web site.

Summary of Application-Level Configuration Files

J2EE configuration files are included within the standard application structure. If you include OC4J application-level configuration files in your EAR or WAR file, they also go within the application structure. See "J2EE Application Structure" on page 5-21. Upon deployment of a J2EE application, the OC4J files are either copied (if you included them) or generated (if you did not include them) in the deployment directory, typically under `j2ee/home/application-deployments`.

J2EE Application-Level Configuration Files in this category are all application-level and are defined by the J2EE specification.

- `application.xml`: This file is the J2EE-standard application descriptor. See "OC4J and J2EE Application Descriptors" on page 5-11.

> **Note:** Do not confuse the J2EE application descriptor with the OC4J global application descriptor `application.xml`. They are both used to define Web modules and have other similar features, but have separate and distinct DTDs.

- `web.xml`: This file is the J2EE-standard Web descriptor. See "OC4J and J2EE Web Descriptors" on page 5-15.
- `ejb-jar.xml`: This file is the J2EE-standard EJB descriptor.
- `application-client.xml`: This file is the J2EE-standard descriptor for application clients.
- `ra.xml`: This file is the J2EE-standard descriptor for resource adapters (for connector factories).

OC4J Application-Level Configuration Files in this category are OC4J-specific at the application level. They configure OC4J-specific functionality to complement standard functionality from the corresponding J2EE descriptor, and override default settings from the corresponding server-level or global descriptor.

- `orion-application.xml`: This file is the OC4J-specific application descriptor. See "OC4J and J2EE Application Descriptors" on page 5-11.
- `orion-web.xml`: This file is the OC4J-specific Web descriptor. See "OC4J and J2EE Web Descriptors" on page 5-15.
- `orion-ejb-jar.xml`: This file is the OC4J-specific EJB descriptor.
- `jazn-data.xml`: This application-level JAAS file contains user name and role information for the XML-based provider. There is also a server-level version. This file is not used if OracleAS JAAS Provider uses the LDAP-based provider instead.
- `orion-application-client.xml`: This file is the OC4J-specific descriptor for application clients.
- `oc4j-ra.xml`: This file is the OC4J-specific descriptor for resource adapters (for connector factories).

Additional Information See the following documents for more information about the preceding descriptors:

- *Oracle Application Server Containers for J2EE User's Guide* for information about `server.xml` and load balancing

- *Oracle Application Server Containers for J2EE Services Guide* for information about data sources, RMI, JMS, and resource adapters in OC4J, and related descriptors
- *Oracle Application Server Containers for J2EE Security Guide* for information about security and JAAS in OC4J, and related descriptors
- *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for information about EJB development and the J2EE and OC4J EJB descriptors

OC4J Top-Level Server Configuration File: server.xml

The OC4J `server.xml` file, located in the OC4J `j2ee/home/config` directory by default, is the starting point for configuration of the OC4J server and all J2EE applications, Web applications, and Web sites for the server. In particular, note the following:

- The attributes of the top-level `<application-server>` element specify, among other things, the target directory for deployed EAR files and where they are unpacked (determined by the `application-directory` setting), and the target directory for copied or generated OC4J descriptors (determined by the `deployment-directory` setting).

Note: A key `<application-server>` attribute in the OC4J 10.1.2 implementation is `check-for-updates`, to automatically check OC4J XML configuration files for updates. See "Key OC4J Flags for Development" on page 2-3.

- The `<global-application>` element specifies the OC4J global application, otherwise known as the default application, which, by default, is the parent of all other applications. The `name` attribute defines its name; the `path` attribute specifies what to use as the OC4J global application descriptor. See "OC4J Default Application and Default Web Application" on page 5-25 for a discussion of the OC4J default application.
- The `<global-web-app-config>` element, through its `path` attribute, specifies what to use as the OC4J global Web application descriptor.
- There is a `<web-site>` element for each Web site recognized by the server, with the `path` attribute specifying what to use as the corresponding Web site XML file. OC4J comes with one such element already configured.
- There is an `<application>` element for each J2EE application deployed to the server. The `name` attribute specifies the desired J2EE application name. The `path` attribute reflects where the EAR file is deployed and unpacked, or where application files exist that have already been unpacked (or were manually placed). In either case, the `name` attribute is typically the same as the EAR file name without the `.ear` extension. In the first case, the `path` attribute specifies the full path to the EAR file, including the EAR file name. In the second case, the path attribute specifies the top-level directory of the extracted files.
- The `<rmi-config>` element, through its `path` attribute, specifies what to use as the OC4J RMI descriptor.
- The `<jms-config>` element, through its `path` attribute, specifies what to use as the OC4J JMS descriptor.

Note: In Oracle Application Server, port settings in the RMI descriptor (`rmi.xml` by default) and JMS descriptor (`jms.xml` by default) are overridden.

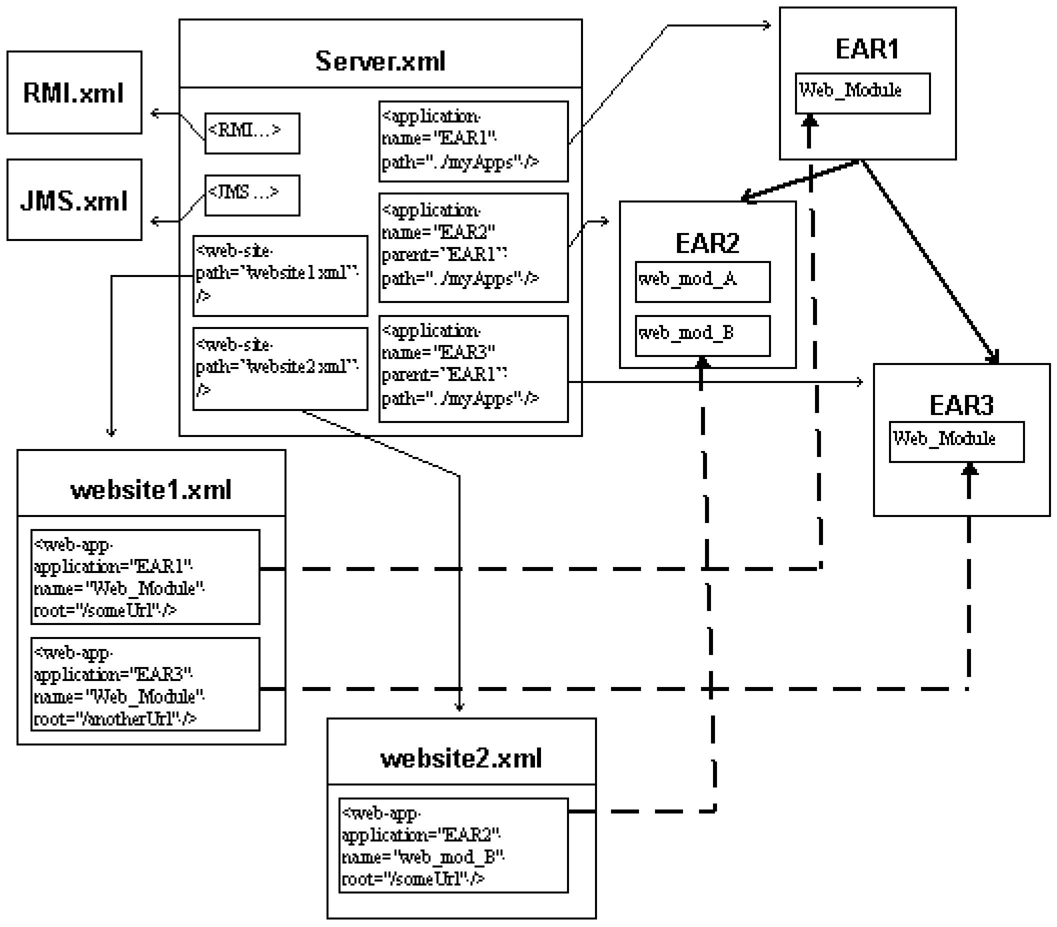
The `server.xml` file is discussed in detail in the *Oracle Application Server Containers for J2EE User's Guide*, but an example is also provided here:

```
<?xml version="1.0"?>
<!DOCTYPE application-server PUBLIC "-//Evermind//DTD Orion
 Application-server//EN"
 "http://xmlns.oracle.com/ias/dtds/application-server.dtd">

<application-server application-directory="../applications"
                    deployment-directory="../application-deployments"
                    connector-directory="../connectors"
>
   <rmi-config path="./rmi.xml" />
   <jms-config path="./jms.xml" />
   <log>
      <file path="../log/server.log" />
   </log>
   <transaction-config timeout="250000" />
   <global-application name="default" path="application.xml" />
   <application name="petstore" path="../applications/petstore.ear" ... />
   <global-web-app-config path="global-web-application.xml" />
   <web-site default="true" path="./default-web-site.xml" />
   <web-site path="../myconfig/my-web-site.xml" />
   <cluster id="-1406559522" />
</application-server>
```

Figure 5–1, which follows, illustrates the mappings between the `server.xml` file, other XML files including Web site XML files, and J2EE EAR files. Note that in this figure, the `<application>` elements in `server.xml` point to top-level directories of extracted EAR files instead of pointing to intact EAR files.

Figure 5–1 Mappings from the server.xml File



RMI.xml
JMS.xml
Server.xml
<RMI...>
<JMS ...>
<web-site path="website1.xml" />
<web-site path="website2.xml" />
<application name="EAR1" path="./myApps"/>
<application name="EAR2" parent="EAR1" path="./myApps"/>
<application name="EAR3" parent="EAR1" path="./myApps"/>
EAR1
Web_Module
EAR2
web_mod_A
web_mod_B
EAR3
Web_Module
website1.xml
<web-app application="EAR1" name="Web_Module" root="/someUrl"/>
<web-app application="EAR3" name="Web_Module" root="/anotherUrl"/>
website2.xml
<web-app application="EAR2" name="web_mod_B" root="/someUrl"/>

OC4J and J2EE Application Descriptors

An *application descriptor* specifies the components of a J2EE application, such as EJB and Web modules, and can specify additional configuration for the application as well.

OC4J uses three categories of application descriptors. The following sections discuss each of them, then summarize their relationships to each other:

- Standard J2EE Application Descriptors
- OC4J Global Application Descriptor
- OC4J-Specific Application Descriptors
- Summary of Relationships Between Application Descriptors

The `server.xml` file points to the application descriptor of each application on OC4J, either directly or indirectly. In the case of a typical J2EE application, `server.xml` points to the EAR file (or extracted EAR top-level directory) and, therefore, to the `application.xml` file that the EAR file contains. In the case of the OC4J global application, the `server.xml` file points directly to the OC4J global application descriptor.

See the *Oracle Application Server Containers for J2EE User's Guide* for more information about application descriptors in OC4J.

Standard J2EE Application Descriptors

The J2EE standard defines the concept and DTD of an application descriptor, called `application.xml`, that you must include in the `/META-INF` directory of the EAR file of a J2EE application. The application descriptor acts as a manifest of the modules contained in the application, possibly with additional configuration information as well, and in some development environments can be created automatically for you. See the J2EE specification for more information.

Here is an example for an application with an EJB module, a Web module, and a client module:

```
<?xml version="1.0" ?>
<!DOCTYPE application (View Source for full doctype...)>
<application>
  <display-name>stateful, application:</display-name>
  <description>
    A sample J2EE application that uses a remote stateful session
    bean to call a local entity bean.
  </description>
  <module>
    <ejb>stateful-ejb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>stateful-web.war</web-uri>
      <context-root>/stateful</context-root>
    </web>
  </module>
  <module>
    <java>stateful-client.jar</java>
  </module>
</application>
```

OC4J Global Application Descriptor

The OC4J-specific global application descriptor is defined by an OC4J-specific DTD, `orion-application.dtd`. This is the descriptor for the OC4J global application, as specified by the `<global-application>` element in the `server.xml` file. This element specifies the global application name, `default` by default, and the global application descriptor name, `application.xml` by default and usually located in the same directory as `server.xml`.

The OC4J global application is usually referred to as the "default application" and, by default, is the parent application of all other applications in the OC4J instance.

Note: The standard J2EE application descriptor and the OC4J global application descriptor are both named `application.xml`, despite being defined by different DTDs. Do not confuse the two. The standard `application.xml` file is not applicable to the OC4J default application.

Following is an abbreviated sample `application.xml` file for the OC4J default application. Note that it specifies the name of a Web application, `defaultWebApp`, which by convention is bound to one or more Web sites as the default Web application.

Do not confuse the OC4J default J2EE application with the default Web application that it contains. See "OC4J Default Application and Default Web Application" on page 5-25 for related information.

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE orion-application PUBLIC "-//Evermind//DTD J2EE Application runtime
 1.2//EN" "http://xmlns.oracle.com/ias/dtds/orion-application.dtd">

<!-- The global application config that is the parent of all the other
  applications in this server. -->

<orion-application autocreate-tables="true"
                   default-data-source="jdbc/OracleDS">
  <web-module id="defaultWebApp" path="../../home/default-web-app" />
  <connectors path="./oc4j-connectors.xml"/>

  <!-- Path to the libraries that are installed on this server.
  These will accesible for the servlets, EJBs etc -->
  <library path="../../home/lib" />
  <!-- Path to the taglib directory that is shared
        among different applications. -->
  <library path="../../home/jsp/lib/taglib" />

  <log>
    <file path="../log/global-application.log" />
  </log>

  <data-sources path="data-sources.xml" />
  <namespace-access>
    <read-access>
      <namespace-resource root="">
        <security-role-mapping>
          <group name="administrators" />
        </security-role-mapping>
      </namespace-resource>
    </read-access>
    <write-access>
      <namespace-resource root="">
        <security-role-mapping>
          <group name="administrators" />
        </security-role-mapping>
      </namespace-resource>
    </write-access>
  </namespace-access>
</orion-application>
```

OC4J-Specific Application Descriptors

In addition to the standard application descriptor, `application.xml`, there is an OC4J-specific application-level application descriptor, `orion-application.xml`. This descriptor is defined by the same DTD as the OC4J global application descriptor. You can provide an `orion-application.xml` file as well as an `application.xml` file, also in the `/META-INF` directory of your EAR file. The `orion-application.xml` file can add OC4J-specific configuration.

Including an `orion-application.xml` file in your EAR file is optional. If you include it, OC4J copies it into the deployment directory during deployment (under `j2ee/home/application-deployments` by default). Otherwise, OC4J generates one for you in the deployment directory, using default settings from the OC4J global application descriptor (assuming the OC4J default application is the parent application, as is the case by default) and the `application.xml` file in the EAR file. See "J2EE Application Structure" on page 5-21 for information about where `orion-application.xml` fits in the application structure.

Note: When OC4J copies `orion-application.xml`, it may make changes to the file but these changes are transparent. For example, an attribute setting that specifies the default value may be ignored or removed.

In most circumstances, you should use `orion-application.xml` only to define OC4J-specific configuration such as security role mappings. Also note that if OC4J generates the file, it creates `<web-module>` elements to reflect the modules specified in the J2EE `application.xml` file.

The following example shows some OC4J-specific configuration and defines the same EJB, Web, and client modules as defined in the example of the standard `application.xml` file in "Standard J2EE Application Descriptors" on page 5-12:

```
<?xml version="1.0"?>
<!DOCTYPE orion-application PUBLIC "-//Evermind//DTD J2EE Application runtime
 1.2//EN" "http://xmlns.oracle.com/ias/dtds/orion-application.dtd">

<orion-application default-data-source="jdbc/OracleDS">
  <ejb-module remote="false" path="stateful-ejb.jar" />
  <web-module id="stateful-web" path="stateful-web.war" />
  <client-module path="stateful-client.jar" auto-start="false" />
  <persistence path="persistence" />
  <log>
    <file path="application.log" />
  </log>
  <namespace-access>
    <read-access>
      <namespace-resource root="">
        <security-role-mapping name="&lt;jndi-user-role&gt;">
          <group name="users" />
        </security-role-mapping>
      </namespace-resource>
    </read-access>
    <write-access>
      <namespace-resource root="">
        <security-role-mapping name="&lt;jndi-user-role&gt;">
          <group name="users" />
        </security-role-mapping>
      </namespace-resource>
    </write-access>
  </namespace-access>
</orion-application>
```

Summary of Relationships Between Application Descriptors

To summarize the relationship between J2EE application descriptors, the OC4J global application descriptor, and OC4J application-level application descriptors:

- For a typical J2EE application, the key application descriptor is the standard J2EE application descriptor, `application.xml`. This file acts as a manifest for the modules of a J2EE application and must be placed in the `/META-INF` directory of the J2EE application EAR file.
- If you want to deploy a standalone WAR file (rather than a WAR file within an EAR file), you can use the OC4J default application, or global application, as the containing application. (See "OC4J Default Application and Default Web Application" on page 5-25.) In this case, the OC4J global application descriptor,

also called `application.xml` but defined by an OC4J-specific DTD, becomes relevant because no J2EE standard `application.xml` file is associated with a standalone WAR file.

- You can optionally include an `orion-application.xml` descriptor for additional OC4J configuration, such as for security role mappings. The `orion-application.xml` file may also specify additional modules, beyond those specified in the J2EE `application.xml` file, and can even override modules specified in `application.xml` (though this is not advisable). The `orion-application.xml` file would also be in the `/META-INF` directory of the EAR file. If you do not include this file, it is created automatically during deployment, using defaults from the OC4J global application descriptor (assuming the default application is the parent of your application, which is true by default). The `orion-application.xml` descriptors are defined according to the same DTD as the OC4J global application descriptor.

OC4J and J2EE Web Descriptors

A *Web descriptor* specifies and configures a set of J2EE Web components: static pages, servlets, and JSP pages. The Web components may together form an independent Web application and be deployed in a standalone WAR file. More typically, however, they will form just part of an overall J2EE application, being deployed in a WAR file within the EAR file of the J2EE application.

OC4J uses three categories of Web descriptors. The following sections discuss each of them and summarize the relationships between them:

- Standard J2EE Web Descriptors
- OC4J Global Web Application Descriptor
- OC4J-Specific Web Descriptors
- Summary of Relationships Between Web Descriptors

Standard J2EE Web Descriptors

The servlet specification defines the concept and DTD of a Web descriptor, called `web.xml`, that you must include in the `/WEB-INF` directory of the associated WAR file. The `web.xml` file specifies and configures the Web components of the WAR file, as well as other components, such as EJBs, that the Web components may call. See the servlet specification for more information.

Here is a sample `web.xml` file specifying, among other things, a servlet, the servlet mapping, and a local EJB lookup:

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <display-name>stateful, web-app:</display-name>
   <description>no description</description>
   <welcome-file-list>
      <welcome-file>index.html</welcome-file>
   </welcome-file-list>

   <ejb-local-ref>
      <ejb-ref-name>CartBean</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <local-home>cart.CartHome</local-home>
      <local>cart.Cart</local>
   </ejb-local-ref>

   <servlet>
      <servlet-name>cart</servlet-name>
      <servlet-class>cart.CartServlet</servlet-class>
      <init-param>
         <param-name>param1</param-name>
         <param-value>1</param-value>
      </init-param>
   </servlet>
   <servlet-mapping>
      <servlet-name>cart</servlet-name>
      <url-pattern>/cart</url-pattern>
   </servlet-mapping>
   <security-role>
      <role-name>users</role-name>
   </security-role>
</web-app>
```

OC4J Global Web Application Descriptor

The `server.xml` file, through its `<global-web-app-config>` element, specifies the OC4J global Web application descriptor. It is typically `global-web-application.xml`, in the same directory as `server.xml`. This descriptor defines default behavior for Web applications in OC4J.

The global Web application descriptor is defined by the DTD `orion-web.dtd`. This is the same DTD as for the application-level OC4J-specific Web descriptor, `orion-web.xml`, described in the next section, "OC4J-Specific Web Descriptors".

The `orion-web.dtd` is a superset of the standard DTD for `web.xml`. There is a `<web-app>` subelement of the `<orion-web-app>` top-level element in `orion-web.dtd`, which has the same specification as the top-level `<web-app>` element of `web.xml`. There are also many other subelements of `<orion-web-app>` for specifying and configuring OC4J-specific features.

For any default settings you specify within the `<web-app>` element in `global-web-application.xml`, you can add to or, optionally, override these settings through `<web-app>` settings in `web.xml`. You can then add to or, optionally, override the resulting settings through `<web-app>` settings in `orion-web.xml`.

Note: Avoid using the `<web-app>` element in `global-web-application.xml` or `orion-web.xml`. Because it is customary to look in `web.xml` for any `<web-app>` entries, having such entries elsewhere could be confusing and may cause difficulty during troubleshooting.

For any default settings you specify outside the `<web-app>` element in `global-web-application.xml`, you can add to or, optionally, override these settings through parallel settings in `orion-web.xml`.

For detailed information about the elements and attributes of the OC4J global Web application descriptor, including the DTD and a hierarchical representation, see "Configuration for global-web-application.xml and orion-web.xml" on page 6-1.

See "Sample global-web-application.xml Settings" on page 6-19 for an abbreviated sample `global-web-application.xml` file.

OC4J-Specific Web Descriptors

In addition to the standard Web descriptor, `web.xml`, and the OC4J global Web application descriptor, `global-web-application.xml` (which establishes default behavior), there is an OC4J-specific application-level Web descriptor, `orion-web.xml`.

The `orion-web.xml` descriptor is defined by the DTD `orion-web.dtd`. This is the same DTD as for the global Web application descriptor that was described in the previous section, "OC4J Global Web Application Descriptor".

You can provide an `orion-web.xml` file as well as the `web.xml` file, also in the `/WEB-INF` directory of your WAR file. Use `orion-web.xml` to add to or, optionally, override any default settings in `global-web-application.xml`, as well as to add to or override any settings in `web.xml`.

Including an `orion-web.xml` file in your WAR file (inside the EAR file) is optional. If you include it, OC4J copies it into the deployment directory during deployment (under the `j2ee/home/application-deployments` directory by default). Otherwise, OC4J generates `orion-web.xml` for you in the deployment directory, using default settings from `global-web-application.xml`. Additionally, some `web.xml` settings will influence the generation of `orion-web.xml`. For example, `<resource-ref>` entries in `web.xml` will result in corresponding `<resource-ref-mapping>` entries in `orion-web.xml`. See "J2EE Application Structure" on page 5-21 for information about where `orion-web.xml` fits in the application structure.

Note: When OC4J copies `orion-web.xml`, it may make changes to the file but these changes are transparent. For example, an attribute setting that specifies the default value may be ignored or removed.

For detailed information about the elements and attributes of the OC4J-specific Web descriptor, including the DTD and a hierarchical representation, see "Configuration for global-web-application.xml and orion-web.xml" on page 6-1.

A sample `orion-web.xml` file follows:

```
<?xml version="1.0" ?>
<!DOCTYPE orion-web-app (View Source for full doctype...)>
<orion-web-app jsp-cache-directory="./persistence" temporary-directory="./temp"
              servlet-webdir="/servlet/" default-buffer-size="2048"
              development="false" directory-browsing="deny"
              file-modification-check-interval="1000" jsp-timeout="0 (never)">
  <ejb-ref-mapping name="EmployeeBean" />
  <security-role-mapping name="users">
     <group name="users" />
  </security-role-mapping>
  <!--
  <web-app>
      There are no <web-app> entries in this sample.
  </web-app>
  -->
</orion-web-app>
```

Summary of Relationships Between Web Descriptors

You can think of the relationship between `global-web-application.xml`, `web.xml`, and `orion-web.xml` as follows:

1. The `global-web-application.xml` file establishes defaults for any Web application in OC4J.
2. The `web.xml` file overlays anything defined in the `<web-app>` element of `global-web-application.xml`, adding to and possibly overriding any Web components and other settings defined there.
3. The `orion-web.xml` file overlays everything, adding to and possibly overriding any settings from `global-web-application.xml` and `web.xml`.

OC4J Web Site Descriptors

Each Web site in OC4J is defined and configured through a Web site XML file. The key functions of a Web site XML file are the following:

- It binds specified Web modules to the Web site, identifying each Web module to bind, the J2EE application it belongs to, and the context path portion of the URL to use in accessing it.
- It defines key settings for the Web site, such as the host name, port number, and protocol. The protocol setting should indicate AJP (Apache JServ Protocol) in an Oracle Application Server environment, and HTTP in a standalone environment.

The `server.xml` file indicates the number of Web sites that OC4J recognizes, by including a `<web-site>` element for each site. Each of these elements specifies the path and file name for the corresponding Web site XML file, as in the following sample `server.xml` entries:

```
...
<web-site path="./default-web-site.xml" />
<web-site path="mydir/my-web-site.xml" />
...
```

In Oracle Application Server, there is just one Web site. In OC4J standalone, there is typically one Web site, but you can use a second Web site for "shared" applications, such as where some communication is through HTTP and some through HTTPS. (For information about shared applications, see the description of the `<web-app>` element `shared` attribute in "Element Descriptions for Web Site XML Files" on page 6-20.)

A Web site XML file contains a `<web-app>` element for each Web module to bind to the Web site. At a minimum, each `<web-app>` element has the following:

- An `application` attribute to specify the name of the J2EE application to which the Web module belongs (the same as the EAR file name without the `.ear` extension)
- A `name` attribute to specify the name of the Web module (the same as the WAR file name without the `.war` extension)
- A `root` attribute to specify the context path on this Web site to which the Web module is to be bound

There is also a `<default-web-app>` element for the default Web application. The default Web application is useful in OC4J standalone during development, as discussed in "OC4J Default Application and Default Web Application" on page 5-25. In Oracle Application Server, it is used for some system-level functionality but is not otherwise meaningful. See "OC4J Default Web Application in Oracle Application Server" on page 5-40.

Important:

- The `root` setting overrides the setting of the `<context-root>` element for this Web module in the `application.xml` descriptor for the containing J2EE application. The `<context-root>` element is required in `application.xml`, but is not used by OC4J. See "Example: Mappings to and from Web Site Descriptors", which follows.
- A `root` setting of "`/`" overrides the OC4J default Web application. This setting (or a null setting, which is converted to "`/`") is not allowed by the `admin.jar` utility when binding a Web application to the Web site.

By default, OC4J comes configured with one Web site XML file: `http-web-site.xml` in OC4J standalone, or `default-web-site.xml` in Oracle Application Server.

See "Configuration for Web Site XML Files" on page 6-20 for detailed information about the elements and attributes of Web site XML files, including the DTD and a hierarchical representation.

See "Sample default-web-site.xml File" on page 6-30 for an example, in this case a sample `default-web-site.xml` file for an Oracle Application Server environment.

Example: Mappings to and from Web Site Descriptors

This example shows how an entry in `server.xml` points to a Web site descriptor (Web site XML file), and how a `<web-app>` element in the Web site XML file points to a Web module. The `<web-app>` element binds the Web module to the Web site. The Web module is defined in the `application.xml` file (also shown) of the containing J2EE application.

The `server.xml` file includes an `<application>` element for the relevant J2EE application (which contains the desired Web module) and includes a `<web-site>` element specifying the Web site XML file for the desired Web site:

```
<application-server ... >
   ...
   <application name="myear" path="../myapps/myear.ear" />
   ...
   <web-site path="my-web-site.xml" />
   ...
</application-server>
```

The Web site XML file, `my-web-site.xml`, configures the Web site and has a `<web-app>` element that specifies the J2EE application that contains the Web module, the name of the Web module itself, and the root context path for accessing the Web module:

```
...
<web-site protocol="http" port="8888" display-name="My Web Site"
          host="[ALL]" log-request-info="false" secure="false">
  ...
  <web-app application="myear" name="mywebmod1" root="/someUrl"
           load-on-startup="false" max-inactivity-time="no shutdown"
           shared="false" />
  ...
</web-site>
```

See "Element Descriptions for Web Site XML Files" on page 6-20 for information about the `<web-site>` and `<web-app>` attributes shown here.

Note: For a Web application (WAR file) that is deployed to the OC4J default application instead of being deployed within an EAR file, the `<web-app>` element `application` attribute indicates the name of the OC4J default application (`default` by default) instead of indicating an EAR file name.

See "OC4J Default Application and Default Web Application" on page 5-25 for general information about the default application.

The J2EE `application.xml` file in `myear.ear` specifies the Web module:

```
<application ... >
  ...
  <module>
    <web>
      <web-uri>mywebmod1.war</web-uri>
      <context-root>/someUrl</context-root>
    </web>
  </module>
  ...
</application>
```

Notes:

- The value of the `root` attribute of the `<web-app>` element in `my-web-site.xml` overrides the value of the `<context-root>` element in `application.xml`. As a convention, though, use the same setting in both places.
- A Web application deployed to the OC4J default application is defined in the OC4J global application descriptor.
- In an Oracle Application Server environment, the `default-web-site.xml` file, by default, sets up a Web site that accesses OC4J through the Oracle HTTP Server and AJP (Apache JServ protocol), using a protocol setting of "`ajp13`" and a port setting of "`0`". However, OPMN, the Oracle Process Management and Notification system, overrides this port setting.
- In an OC4J standalone environment, the `http-web-site.xml` file, by default, sets up a Web site that accesses the OC4J listener directly, using a protocol setting of "`http`" and a port setting of "`8888`".

Application Packaging

OC4J supports standard J2EE archive files for deployment, including WAR files for Web modules and EAR files for overall J2EE applications. The following sections review the structure of these files:

- J2EE Application Structure
- EAR File and WAR File Structures

J2EE Application Structure

This section reviews the standard J2EE application structure, which you can use as your development structure as appropriate. This discussion also shows the relative locations of optional OC4J-specific descriptors. If you do not include the OC4J-specific descriptors in your deployment, OC4J generates them for you when you deploy a J2EE application, using values from corresponding OC4J global descriptors and J2EE descriptors as defaults.

```
J2EEAppName/

   META-INF/
      application.xml
      orion-application.xml (optional)

   EJBModuleName/
      (EJB classes, according to package)
      META-INF/
         ejb-jar.xml
         orion-ejb-jar.xml (optional)

   WebModuleName/
      (static files, such as index.html)
      (JSP pages)
      WEB-INF/
         web.xml
         orion-web.xml (optional)
         classes/
            (servlet classes, according to package)
         lib/
            (JAR files for dependency classes)

   ClientModuleName/
      (client classes, according to package)
      META-INF/
         application-client.xml
         orion-application-client.xml

   ResourceAdapterModuleName/
      META-INF/
         ra.xml
      (JAR files for required classes)
      (required static files or other files)
```

The Web portion is marked in bold type. This portion reflects the structure of WAR files used to deploy Web modules. At the top level are static pages (such as `index.html`), JSP pages, and the `/WEB-INF` directory.

Notes:

- This structure is defined in the J2EE specification and related specifications. The J2EE specification is at the following location:

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

- See "OC4J and J2EE Application Descriptors" on page 5-11 for an overview of `application.xml` and `orion-application.xml`.
- See "OC4J and J2EE Web Descriptors" on page 5-15 for an overview of `web.xml` and `orion-web.xml`.
- See the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for information about `ejb-jar.xml` and `orion-ejb-jar.xml`.
- See the *Oracle Application Server Containers for J2EE User's Guide* for information about `application-client.xml` and `orion-application-client.xml`.

EAR File and WAR File Structures

In J2EE, a WAR file is typically contained within an EAR file. In the example in the preceding section, the EAR file, `J2EEAppName.ear`, would have its `/META-INF` directory at the top level, along with Web module WAR files, EJB module JAR files, client application JAR files, and resource adapter RAR files (zero or more of each, as applicable):

```
META-INF/
   application.xml
   orion-application.xml (optional)
EJBModuleName.jar
WebModuleName.war
ClientModuleName.jar
ResourceAdapterModuleName.rar
```

Sample EAR and WAR File

This example shows the structure of the archive files for a simple Web application. The EAR file contains a WAR file, which contains a single servlet.

Following are the contents of `utility.ear`. If there were EJB, client application, or resource adapter modules, the associated JAR files would be at the same level as the WAR file. Optionally, you could also include an `orion-application.xml` file in the `/META-INF` directory. Instead, in this example, one would be generated by OC4J during deployment.

```
META-INF/MANIFEST.MF
META-INF/application.xml
utility_web.war
```

Here are the contents of `utility_web.war`. Optionally, you could also include an `orion-web.xml` file in the `/WEB-INF` directory. Instead, in this example, one would be generated by OC4J during deployment.

```
META-INF/MANIFEST.MF
WEB-INF/classes/TestStatusServlet.class
```

```
WEB-INF/web.xml
index.html
```

Notes:

- This document assumes you have some J2EE development experience and a means of creating EAR and WAR files, either by using the JAR utility directly, or through an IDE such as Oracle JDeveloper, or by using the `ant` utility and a `build.xml` file. See the following site for information about `ant`:

`http://ant.apache.org`

- The `MANIFEST.MF` files are created automatically by the JAR utility.

Deployment Scenarios to OC4J Standalone

This section reviews some preliminary considerations and then discusses several scenarios for deployment to OC4J standalone. The primary deployment scenario is to use the `admin.jar` utility after you package your application in an EAR file. The EAR file optionally contains Web module WAR files, EJB module JAR files, client application JAR files, and resource adapter RAR files (zero or more of each). See "Application Packaging" on page 5-20 for more information about structure and packaging.

The use of EAR files for OC4J deployment, and features of the `admin.jar` utility, are covered extensively in the *Oracle Application Server Containers for J2EE Stand Alone User's Guide*. Key features are discussed here.

In addition, this section considers alternative deployment scenarios that you may find useful during development, such as manually creating and populating a J2EE application structure or deploying an independent WAR file into the OC4J default application.

Note: In these alternative deployment scenarios, in which you manually place and update files, you are considered to be in "expert mode". You are operating outside the safeguards and constraints of the OC4J and Oracle Application Server tools. See "Using Oracle Deployment Tools Versus Expert Modes" on page 5-4.

This section includes the following subjects:

- Setting Up an Administrative User and Password
- Starting and Stopping OC4J Standalone
- OC4J Default Application and Default Web Application
- Deploying an EAR File to OC4J Standalone
- Deploying Files into a J2EE Application Structure on OC4J Standalone
- Deploying an Independent WAR File to OC4J Standalone
- Deploying Files into a Web Application Directory Structure on OC4J Standalone

- Application Undeployment or Redeployment in OC4J Standalone

Setting Up an Administrative User and Password

Before using the `admin.jar` utility to deploy an application in OC4J standalone, you must have a user with administrative privileges.

The `j2ee/home/config/jazn-data.xml` file determines security privileges for user accounts. By default, there is a user `admin` with administrative privileges, as specified in the following sample `jazn-data.xml` entry:

```
<role>
   <name>administrators</name>
   <display-name>Realm Admin Role</display-name>
   <description>Administrative role for this realm.</description>
   <members>
      <member>
         <type>user</type>
         <name>admin</name>
      </member>
   </members>
</role>
```

For the default administrative user `admin`, the default password is `welcome`, as in the following sample `jazn-data.xml` entry:

```
<users>
...
   <user>
      <name>admin</name>
      <display-name>OC4J Administrator</display-name>
      <description>OC4J Administrator</description>
      <credentials>!welcome</credentials>
   </user>
...
</users>
```

(The file is automatically rewritten later to obfuscate the specified password.) See the *Oracle Application Server Containers for J2EE Security Guide* for more information about the `jazn-data.xml` file, especially regarding the `<credentials>` element.

Important: You cannot use the `admin.jar` utility in an Oracle Application Server environment. It is for use in OC4J standalone only.

Note: If you are still using the deprecated `principals.xml` file for security, the administrative account password is determined through the OC4J `-install` command:

```
% java -jar oc4j.jar -install
```

(Assume `%` is the system prompt and `j2ee/home` is your current directory.) You will be prompted for the desired password.

Starting and Stopping OC4J Standalone

This section provides a quick review of how to start and stop OC4J standalone. Assume `%` is the system prompt and `j2ee/home` is your current directory.

Issue the following command to start OC4J:

```
% java -jar oc4j.jar [options]
```

See the *Oracle Application Server Containers for J2EE Stand Alone User's Guide* for a discussion of OC4J command-line options.

Issue the following command to stop OC4J:

```
% java -jar admin.jar ormi://oc4j_host:oc4j_ormi_port
       adminuser adminpassword -shutdown
```

For the `admin.jar -shutdown` command, note the following:

- In OC4J standalone, you can get the OC4J ORMI port number from the `j2ee/home/config/rmi.xml` file, where there will be an entry such as the following:

  ```
  <rmi-server port="23791" host="[ALL]">
  ```

- See the previous section, "Setting Up an Administrative User and Password", for information about *`adminuser`* and *`adminpassword`*.

OC4J Default Application and Default Web Application

The following sections discuss features and configuration of the OC4J default application and default Web application. Some of the OC4J standalone deployment scenarios described later will use these features.

- Use of the Default Application and Default Web Application
- Configuration of the Default Application and Default Web Application

Note: Use of the default application and default Web application for deployment during testing is a useful OC4J convenience feature, but is considered to be an expert mode because you are manually placing application files and sometimes manually updating configuration files. See "Using Oracle Deployment Tools Versus Expert Modes" on page 5-4.

Use of the Default Application and Default Web Application

OC4J is installed with a default configuration that includes a *default application* (also known as the *global application*). The default application is, by default, the parent of all other J2EE applications in OC4J.

In OC4J, a Web application must be contained within a parent J2EE application. Usually, a WAR file is deployed within an EAR file that defines the parent J2EE application. If you want to deploy an independent WAR file, you can deploy to the OC4J default application instead. By default, the OC4J `server.xml` file specifies the location and name of the global application descriptor that defines the default application.

In a typical OC4J installation, the default application contains a *default Web application*. The name and root directory path of the default Web application are specified in the global application descriptor, and the default Web application is bound to a Web site through the `http-web-site.xml` file for OC4J standalone (`default-web-site.xml` in Oracle Application Server). In OC4J standalone, the default context path for the default Web application is "`/`".

Also by default in OC4J standalone, the root directory of the default Web application is `j2ee/home/default-web-app`. To deploy to the default Web application, place your JSP pages and class files under this directory in the standard Web application directory structure: static pages and JSP pages at the top level, servlet classes under `j2ee/home/default-web-app/WEB-INF/classes`, and library JAR files in `j2ee/home/default-web-app/WEB-INF/lib`. Also see "Using a Web Application Directory Structure in the Default Web Application" on page 5-34.

> **Note:** The default Web application, in addition to being invoked by use of the context path "`/`", is invoked if the context path mapping of any requested URL fails. This occurs if a requested URL has no matching context path in any `<web-app>` element `root` attribute in the Web site XML file.

Configuration of the Default Application and Default Web Application

This section details the default configurations for the OC4J default application and default Web application.

The server.xml Configuration for Default Application In `server.xml`, the `<global-application>` element specifies the OC4J default application. The `name` attribute specifies its name, and the `path` attribute specifies what to use as the OC4J global application descriptor:

```
<application-server ... >
   ...
   <global-application name="default" path="application.xml" />
   ...
</application-server>
```

The application.xml Configuration for Default Web Application The specified descriptor for the default application (or global application), `application.xml`, specifies the name and root directory path of the default Web application, which is contained in the default application:

```
<orion-application ... >
  ...
  <web-module id="defaultWebApp" path="../../home/default-web-app" />
  ...
</orion-application>
```

To deploy to the default Web application, place your files under this directory according to the standard Web application structure.

Binding of Default Web Application in Web Site XML File By default, the default Web application is bound to a Web site in the `http-web-site.xml` file for OC4J standalone (`default-web-site.xml` in Oracle Application Server).

Although most OC4J Web applications are bound to a Web site through `<web-app>` subelements of a `<web-site>` element in the Web site XML file, the default Web application is instead configured through the `<default-web-app>` subelement of `<web-site>`.

In OC4J standalone, the default context path of the default Web application is "/", without a `root` attribute being required. Here is an example:

```
<web-site ... >
...
   <default-web-app application="default" name="defaultWebApp"
                    load-on-startup="true" shared="false" />
...
</web-site>
```

See "Configuration for Web Site XML Files" on page 6-20 for detailed information about elements and attributes of Web site XML files.

Note: The `<default-web-app>` element is required in any Web site XML file.

Deploying an EAR File to OC4J Standalone

The following sections describe the process of deploying an EAR file in OC4J standalone using the `admin.jar` utility.

This discussion assumes that if you modify your code, you would then repackage it and redeploy it.

Using admin.jar to Deploy the EAR File

After you have packaged your application into an EAR file, you can use the OC4J `admin.jar` utility to deploy it, using the following syntax:

```
% java -jar admin.jar ormi://oc4j_host:oc4j_ormi_port
                      adminuser adminpassword
                      -deploy -file path/filename.ear
                      -deploymentName appname
```

This command uses RMI to communicate with OC4J. Note the following:

- See "Starting and Stopping OC4J Standalone" on page 5-25 for information about the ORMI port.
- See "Setting Up an Administrative User and Password" on page 5-24 for information about *`adminuser`* and *`adminpassword`*.
- For `-file`, specify the path to the EAR file, including the file name.
- For `-deploymentName`, specify the desired application name, by convention the same as the EAR file name without the `.ear` extension.

Note: During development, assuming you develop and run on the same system, you will deploy locally. However, `admin.jar` is also capable of deploying remotely.

By default, a deployment results in the following:

- The EAR file is copied to the `j2ee/home/applications` directory. This directory is set as the default through the `application-directory` attribute of the `<application-server>` element in the `server.xml` file.
- The EAR file is unpacked beneath the `j2ee/home/applications` directory.

- The OC4J-specific descriptors—at a minimum, `orion-application.xml` and `orion-web.xml` for a Web application in an EAR file—are copied or generated under the `j2ee/home/application-deployments` directory. This directory is set as the default through the `deployment-directory` attribute of the `<application-server>` element in the `server.xml` file. These descriptors are copied from the EAR file if they exist there; otherwise, OC4J generates them.
- An `<application>` element is added to the `server.xml` file. This element specifies the application name, according to the `-deploymentName` setting in `admin.jar`, and specifies the path to where the EAR file was deployed, `j2ee/home/applications` by default.

See "Sample Deployment" on page 5-29 for an example.

Note: The target directories are configurable. See the *Oracle Application Server Containers for J2EE Stand Alone User's Guide* for additional information about `admin.jar`, including the `-targetPath` and `-deploymentDirectory` options.

Using admin.jar to Bind the Web Application

After you have deployed your application, you can use the OC4J `admin.jar` utility to bind the associated Web application to a Web site:

```
% java -jar admin.jar ormi://oc4j_host:oc4j_ormi_port
                      adminuser adminpassword
                      -bindWebApp appname webappname websitename contextpath
```

As with the `-deploy` command, the `-bindWebApp` command uses RMI to communicate with OC4J. Note the following:

- See "Starting and Stopping OC4J Standalone" on page 5-25 for information about the ORMI port.
- See "Setting Up an Administrative User and Password" on page 5-24 for information about *`adminuser`* and *`adminpassword`*.
- The *`appname`* is the application name, according to the `-deploymentName` setting when you deployed it.
- The *`webappname`* is the name of the Web application. This is the WAR file name without the `.war` extension.
- The *`websitename`* is indicated by the Web site XML file name for the desired site, without the `.xml` extension (for example, `http-web-site` in OC4J standalone).
- Specify the desired context path portion of the URL for invoking the Web application.

Note: A context path setting of "`/`", which overrides the OC4J default Web application, is disallowed by the `admin.jar` utility when binding a Web application to the Web site. A setting of null, which is converted to "`/`", is also disallowed.

As a result of this command, a `<web-app>` element is added to the specified Web site XML file, indicating the application name, the Web application name, and the context path.

See the next section, "Sample Deployment", for an example.

Sample Deployment

This example illustrates the result of deploying the `utility.ear` file shown in "EAR File and WAR File Structures" on page 5-22, then binding its Web application. Here are the `admin.jar` commands, assuming `%` is the system prompt, `j2ee/home` is the current directory, and the EAR file is in `j2ee/home/demo`:

```
% java -jar admin.jar ormi://myhost:23791 admin welcome
  -deploy -file demo/utility.ear -deploymentName utility

% java -jar admin.jar ormi://myhost:23791 admin welcome
  -bindWebApp utility utility_web http-web-site /utilroot
```

Note the following:

- The OC4J host name in this example is `myhost`; the port setting is 23791 in `j2ee/home/config/rmi.xml`.
- In this example, the administrative account name is `admin` and the password is `welcome`.
- The Web application name within `utility.ear` is `utility_web`, based on the WAR file name, `utility_web.war`.
- The `-bindWebApp` command is to bind `utility_web` to the Web site defined by `j2ee/home/config/http-web-site.xml`. This assumes the following entry is in the `server.xml` file, as is the case by default:

  ```
  <web-site path="http-web-site.xml" />
  ```

- The desired context path portion of the URL to invoke the Web application is `"/utilroot"`.

The `-deploy` command results in the following entry in `server.xml`, as a subelement of the top-level `<application-server>` element:

```
<application name="utility" path="../applications/utility.ear"
             auto-start="true" />
```

The `auto-start` attribute specifies whether this application should be automatically restarted each time OC4J is restarted.

The `-bindWebApp` command results in the following entry in `http-web-site.xml`, as a subelement of the top-level `<web-site>` element:

```
<web-app application="utility" name="utility_web" root="/utilroot"
 load-on-startup="false" max-inactivity-time="no shutdown" shared="false" />
```

(See "Element Descriptions for Web Site XML Files" on page 6-20 for information about the `load-on-startup`, `max-inactivity-time`, and `shared` attributes.)

Notes:

- Remember that the value of the `root` attribute of the `<web-app>` element in `http-web-site.xml` overrides the value of the `<context-root>` element in `application.xml`. As a convention, though, use the same value in both places.
- Information about the resulting `server.xml` and `http-web-site.xml` entries is provided as informative background. You should not have any reason to update these files manually when you use `admin.jar`.

After the deployment of `utility.ear`, the directory structure for key files is as follows, assuming default settings for the target directories:

```
j2ee/home/
   application-deployments/
      utility/
         orion-application.xml
         utility_web/
            orion-web.xml
   applications/
      utility.ear
      utility/
         utility_web.war
         META-INF/
            application.xml
         utility_web/
            index.html
            META-INF/
            WEB-INF/
               web.xml
               classes/
                  TestStatusServlet.class
```

The `server.xml` and `http-web-site.xml` files are in the `j2ee/home/config` directory.

If `orion-application.xml` and `orion-web.xml` exist in the EAR file, they are copied from there into the directories shown above. Otherwise, OC4J generates them into the directories shown above, using default settings from the corresponding OC4J global descriptors and J2EE descriptors.

Descriptors for Sample Deployment

The deployment in the preceding section uses the following descriptors. Passages of particular interest to servlet developers are marked in bold type.

The application.xml File The standard `application.xml` descriptor is supplied by the developer. Some IDEs, such as Oracle JDeveloper, will create this for you.

```
<?xml version="1.0" ?>
<!DOCTYPE application (View Source for full doctype...)>
<application>
  <display-name>Web Services Demo</display-name>
  <module>
    <web>
      <web-uri>utility_web.war</web-uri>
      <context-root>/j2ee/utility</context-root>
    </web>
  </module>
</application>
```

Remember that the `<context-root>` element here is overridden by the `root` attribute of the `<web-app>` element in the Web site XML file.

The web.xml File The standard `web.xml` descriptor is supplied by the developer.

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (View Source for full doctype...)>
<web-app>
  <display-name>Web Services Example</display-name>
  <description>A few examples of web service publication</description>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>TestStatus</servlet-name>
    <servlet-class>TestStatusServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestStatus</servlet-name>
    <url-pattern>/TestStatusServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

The orion-application.xml File Because the `orion-application.xml` descriptor is not included in the EAR file in this example, it is generated by OC4J. Most of the file is omitted here, but note that the `<web-module>` element mirrors the entry in the `application.xml` file.

```
<?xml version="1.0" ?>
<!DOCTYPE orion-application (View Source for full doctype...)>
<orion-application deployment-version="10.1.2.0.0"
                   default-data-source="jdbc/OracleDS"
                   treat-zero-as-null="true" autocreate-tables="true"
                   autodelete-tables="false">
  ...
  <web-module id="utility_web" path="utility_web.war" />
  ...
</orion-application>
```

The orion-web.xml File Because the `orion-web.xml` descriptor is not included in the WAR file (within the EAR file) in this example, it is generated by OC4J. It is not shown here because there are no entries specific to the example.

Invoking the Sample Application

Given the information for the sample deployment in the preceding sections, in which `/utilroot` is specified as the context path in the `admin.jar -bindWebApp` command and `/TestStatusServlet` is specified as the servlet path in `web.xml`, you invoke the application as follows:

```
http://host:port/utilroot/TestStatusServlet
```

Deploying Files into a J2EE Application Structure on OC4J Standalone

Instead of deploying an EAR file, you can manually deploy the file structure and then update the `server.xml` and Web site XML files. This is an expert mode. (See "Using Oracle Deployment Tools Versus Expert Modes" on page 5-4.)

Look again at "Sample Deployment" on page 5-29, but assume that you manually create the `j2ee/home/applications/utility` directory and manually populate the application directory structure underneath:

```
j2ee/home/applications/utility/
   META-INF/
      application.xml
   utility_web/
      index.html
      META-INF/
      WEB-INF/
         web.xml
         classes/
            TestStatusServlet.class
```

Further assume that you update the `server.xml` file as follows:

```
<application name="utility" path="../applications/utility"
             auto-start="true" />
```

And you update the `http-web-site.xml` file as follows:

```
<web-app application="utility" name="utility_web" root="/utilroot"
 load-on-startup="false" max-inactivity-time="no shutdown" shared="false" />
```

Note that the `path` attribute in the `<application>` element in `server.xml` is "`../applications/utility`" instead of "`../applications/utility.ear`". This is because there is no EAR file, just the directory structure under the `utility` directory.

By default in OC4J standalone, when you update `server.xml`, OC4J detects the change, deploys the application, and copies or generates the `orion-web.xml` file and `orion-application.xml` file under the `application-deployments` directory as follows:

```
j2ee/home/
   application-deployments/
      utility/
         orion-application.xml
         utility_web/
            orion-web.xml
```

However, if OC4J update-checking is disabled, you must manually inform OC4J of your configuration updates, using the `admin.jar -updateConfig` option. Checking is enabled through the `server.xml` `check-for-updates` flag. See "Key OC4J Flags for Development" on page 2-3.

Deploying an Independent WAR File to OC4J Standalone

"Deploying an EAR File to OC4J Standalone" on page 5-27 discusses using the `admin.jar` utility to deploy an EAR file, including a WAR file within an EAR file for a Web module. For convenience during testing, it is also possible to manually deploy an independent WAR file, using the OC4J default application as the containing application. (A Web application must always be part of a parent J2EE application in

OC4J.) This is an expert mode. See "Using Oracle Deployment Tools Versus Expert Modes" on page 5-4.

See "OC4J Default Application and Default Web Application" on page 5-25 for background information.

Use the following steps to deploy an independent WAR file to the OC4J default application:

1. Place your WAR file in the desired directory.
2. Update the OC4J global `application.xml` file to add a `<web-module>` element to specify the Web application name and the location of the WAR file.
3. Update the appropriate Web site XML file, typically `http-web-site.xml` in OC4J standalone, to add a `<web-app>` element to bind the Web application to the Web site.

The following example illustrates deployment of a WAR file to the default application.

Note: In Oracle Application Server, all applications are deployed in EAR files. If you use the Application Server Control Console Deploy Web Application Page in Enterprise Manager, which prompts you for a WAR file, then an EAR file is transparently created to contain the WAR file.

Example: Web Application Name in OC4J Default Application

This example shows entries in `server.xml`, the Web site XML file, and the OC4J global `application.xml` file for a Web application, `mywebmod1`, within the OC4J default application. Note the following:

- The `path` attribute of the `<web-site>` element in `server.xml` specifies the path and name of the Web site XML file, `http-web-site.xml` in this example.
- The `name` attribute of the `<global-application>` element in `server.xml` specifies the name of the OC4J default application and corresponds to the `application` attribute of the `<web-app>` element in `http-web-site.xml`.
- The `path` attribute of the `<global-application>` element in `server.xml` points to the OC4J global `application.xml` file.
- The `name` attribute of the `<web-app>` element in `http-web-site.xml` indicates a Web application, `mywebmod1`, within the OC4J default application and corresponds to the `id` attribute of a `<web-module>` element in the OC4J global `application.xml` file. Both of these attributes typically correspond to the WAR file name without the `.war` extension.
- The global `application.xml` file must specify the name and location of the WAR file, through the `path` attribute of the `<web-module>` element, because there is no containing EAR file.

The following entries are in the `server.xml` file (no changes required):

```
<application-server ... >
   ...
   <global-application name="default" path="application.xml" />
   ...
   <web-site path="http-web-site.xml" />
   ...
</application-server>
```

Place the bold entry into the Web site XML file, `http-web-site.xml`:

```
<web-site protocol="http" port="8888" display-name="HTTP Web Site"
          host="[ALL]" log-request-info="false" secure="false">
  ...
  <web-app application="default" name="mywebmod1" root="/someUrl"
           load-on-startup="false" max-inactivity-time="no shutdown"
           shared="false" />
  ...
</web-site>
```

(See "Element Descriptions for Web Site XML Files" on page 6-20 for information about the `<web-site>` and `<web-app>` attributes shown here.)

Place the bold entry into the OC4J global `application.xml` file:

```
<orion-application ... >
  ...
  <web-module id="mywebmod1" path="../myhome/mywebmod1.war" />
  ...
</orion-application>
```

Note: By default in OC4J standalone, editing the global `application.xml` file automatically results in the WAR file being unpacked beneath the directory in which you placed it. The `orion-web.xml` file, if you included one, is copied from the WAR file to the deployment directory (under `j2ee/home/application-deployments` by default). If you did not include `orion-web.xml`, one is generated for you in the deployment directory.

However, automatic detection of configuration changes depends on the `server.xml` `check-for-updates` flag, which is set to "`true`" by default. If this flag is disabled, you can trigger a one-time check through the `admin.jar -updateConfig` option. See "Key OC4J Flags for Development" on page 2-3.

Deploying Files into a Web Application Directory Structure on OC4J Standalone

The previous section discusses how to deploy an independent WAR file to the OC4J default application. Alternatively, you can manually set up the J2EE Web application directory structure instead of using a WAR file. Again, this involves the OC4J default application. The simplest way to do this is to also use the OC4J default Web application, but you can optionally define a new Web application under the default application. See "OC4J Default Application and Default Web Application" on page 5-25 for background information.

Each of these scenarios is an expert mode. (See "Using Oracle Deployment Tools Versus Expert Modes" on page 5-4.) The following sections discuss them in detail:

- Using a Web Application Directory Structure in the Default Web Application
- Using a Web Application Directory Structure in an Alternative Web Application

Using a Web Application Directory Structure in the Default Web Application

By default, OC4J is configured with a default application and with a default Web application contained in the default application. To use the default Web application for your test files, put them in the standard Web application directory structure under the j2ee/home/default-web-app directory (the default directory for the default Web application).

Perform the following steps:

1. Check relevant configuration in the server.xml file. First, you should see the following entry to define the name and specify the application descriptor for the default application:

   ```
   <global-application name="default" path="application.xml" />
   ```

   You should also see the following entry to specify the Web site XML file:

   ```
   <web-site path="./http-web-site.xml" />
   ```

2. Look in the OC4J global application.xml descriptor. You should see the following entry to define the default Web application name and specify its root directory:

   ```
   <web-module id="defaultWebApp" path="../../home/default-web-app" />
   ```

3. Look in the Web site XML file indicated in server.xml (http-web-site.xml, by default, in OC4J standalone). You should see the following entry to define defaultWebApp as the default Web application for the Web site. (As noted earlier, "/" is the context path for the default Web application in OC4J standalone, without the necessity of a root attribute in the <default-web-app> element.)

   ```
   <default-web-app application="default" name="defaultWebApp" />
   ```

4. Given this configuration, deploy your files as follows:

   ```
   j2ee/home/default-web-app/
      index.html
      WEB-INF/
         web.xml
         classes/
            TestServlet.class
   ```

   No further action is necessary before you invoke the servlet.

Using a Web Application Directory Structure in an Alternative Web Application

In the previous section, the OC4J default Web application, defaultWebApp, is used to deploy a Web application directory structure. Alternatively, you can define some other Web application that will also be contained in the OC4J default application. This is useful if you want functionality similar to that of the default Web application, but deploying to a clean directory.

Here are the steps to define a Web application, myDefaultWebApp, within the default application:

1. Add a <web-module> element to the OC4J global application.xml file. This defines the name and specifies the root directory of a new OC4J Web application. The entry for the OC4J default Web application is also shown for comparison:

   ```
   <web-module id="defaultWebApp" path="../../home/default-web-app" />
   <web-module id="myDefaultWebApp" path="../../home/my-default-web-app" />
   ```

2. Add a <web-app> element to the Web site XML file, http-web-site.xml. This ties the Web application to a context path. The <default-web-app> element for the OC4J default Web application is also shown for comparison:

   ```
   <default-web-app application="default" name="defaultWebApp" />
   ```

```
<web-app application="default" name="myDefaultWebApp" root="/mydefroot" />
```

3. Given this configuration, deploy your files as follows. No further action is necessary before you invoke the servlet.

```
j2ee/home/my-default-web-app/
   index.html
   WEB-INF/
      web.xml
      classes/
         TestServlet.class
```

Application Undeployment or Redeployment in OC4J Standalone

During testing, you will presumably have to modify and redeploy your application. The following sections describe undeployment and redeployment features for OC4J standalone:

- Using admin.jar to Undeploy an Application
- Using admin.jar to Redeploy an Application
- Manually Redeploying a WAR File
- Triggering Application Redeployment after File Manipulation

Using admin.jar to Undeploy an Application

If you are finished using an application, you can use `admin.jar` to undeploy it as follows:

```
% java -jar admin.jar ormi://oc4j_host:oc4j_ormi_port
                      adminuser adminpassword
                      -undeploy appname
```

This removes associated entries in `server.xml` and the Web site XML file, as well as removing all directories and files that were created and copied. See "Starting and Stopping OC4J Standalone" on page 5-25 for information about the ORMI port. See "Setting Up an Administrative User and Password" on page 5-24 for information about *`adminuser`* and *`adminpassword`*.

To undeploy the `utility.ear` application shown in "Sample Deployment" on page 5-29, for example:

```
% java -jar admin.jar ormi://myhost:23791 admin welcome -undeploy utility
```

Note: There is no need to undeploy an application before redeploying it. The `admin.jar -undeploy` option is for permanent removal.

Using admin.jar to Redeploy an Application

As when you initially deploy an application to OC4J standalone, use the `admin.jar -deploy` command to redeploy it. There is no difference to the user. The utility automatically effectively undeploys it first so that you will have a clean start. See "Deploying an EAR File to OC4J Standalone" on page 5-27 for information about the `-deploy` command.

Before redeploying, you must repackage your EAR file to pick up any updated files. If you updated OC4J descriptors on the server, such as `orion-web.xml` and `orion-application.xml`, and want to keep the changes, then you must include these in the EAR file as well. Previously copied or generated OC4J descriptors are lost if there has been any update to the EAR file.

Manually Redeploying a WAR File

Deploying an independent WAR file to the OC4J default application is described in "Deploying an Independent WAR File to OC4J Standalone" on page 5-32. If you want to redeploy, you must reverse the steps you took to deploy, repackage the WAR file, and repeat the deployment steps. This is an expert mode. (See "Using Oracle Deployment Tools Versus Expert Modes" on page 5-4.)

To summarize:

1. Comment out your updates to the global `application.xml` file and the `http-web-site.xml` file (or other Web site XML file).
2. Update or repackage the WAR file with your updates. If there is anything in `orion-web.xml` that you want to save, then include it in the WAR file.
3. Place the new WAR file back in the original target directory, overwriting the original if it is still there.
4. Uncomment the updates in `application.xml` and `http-web-site.xml`. Updating `application.xml` causes OC4J to unpack the WAR file and copy or generate the `orion-web.xml` file as during the initial deployment.

Triggering Application Redeployment after File Manipulation

Depending on OC4J polling, which is enabled by default in OC4J standalone, there are several ways to trigger a redeployment of your application when you modify files in place on the server. (This is an expert mode. See "Using Oracle Deployment Tools Versus Expert Modes" on page 5-4.)

- Modify servlet class files under `/WEB-INF/classes`.

  If you update a servlet `.class` file under `/WEB-INF/classes`, then upon the next request, the servlet and its dependency classes are reloaded and the Web application is redeployed, regardless of whether OC4J polling is enabled.

  Notes:

  - A servlet and its dependency classes are reloaded immediately, instead of upon next request, if the servlet is set to be preloaded. This is according to `load-on-startup` settings. See "Servlet Preloading" on page 2-6.
  - Changing a servlet class file in a directory location specified in a `<classpath>` element in `global-web-application.xml` or `orion-web.xml` has the same effect as changing a servlet class file in `/WEB-INF/classes`. However, changing a JAR file or a dependency class (such as a JavaBean) in a `<classpath>` location has no effect. See the description of the `<classpath>` element in "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1.

- Modify standard descriptors (anything that changes the timestamp).

  If you modify `web.xml` while polling is enabled, the Web application is redeployed the next time the OC4J task manager runs, which by default is once each second. Servlets and dependency classes in the Web application are reloaded upon the next request.

If you modify `application.xml` while polling is enabled, the J2EE application is redeployed. Servlets and dependency classes in the Web application are reloaded upon the next request.

- Modify library JAR files under `/WEB-INF/lib`.

  If you modify a JAR file in `/WEB-INF/lib` while polling is enabled, the Web application is redeployed the next time the OC4J task manager runs. Servlets and dependency classes in the Web application are reloaded upon the next request.

- Set the OC4J `development` flag to "`true`".

  The `development` flag is an attribute of the `<orion-web-app>` element in `global-web-application.xml` and `orion-web.xml`. If `development` is set to `"true"`, then the OC4J server checks a particular directory (the application `/WEB-INF/classes` directory by default) for updates to servlet source files. If a source file has changed since the last request, then OC4J will, upon the next request, recompile the servlet, redeploy the Web application, and reload the servlet and any dependency classes. See the description of `development` under "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1 for further information.

- For JSP applications, set the JSP `main_mode` flag to "`recompile`".

  See the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information.

- Stop and restart OC4J.

  See "Starting and Stopping OC4J Standalone" on page 5-25.

In OC4J standalone, polling is controlled through the `server.xml` `check-for-updates` flag, which is set to "`true`" by default. Alternatively, you can use the `admin.jar -updateConfig` option to trigger one-time polling. See "Key OC4J Flags for Development" on page 2-3.

Notes: Be aware of a few points in any of the preceding scenarios:

- In this discussion, "redeployment" of a Web application refers to the process in which OC4J removes the Web application from its execution space, removes the classloader that was associated with execution of the Web application, reparses `web.xml` and `orion-web.xml`, and reinitializes servlet listeners, filters, and mappings.
- To ensure a clean start, shut down and restart OC4J after the redeployment. See "Starting and Stopping OC4J Standalone" on page 5-25.
- Redeployment does not significantly affect OC4J descriptors such as `orion-application.xml` and `orion-web.xml` in the server deployment directory. After you trigger reloading, the previously copied or generated files will keep any nondefault settings that you have specified.

OC4J Deployment in Oracle Application Server

This section considers deployment and redeployment scenarios to OC4J in an Oracle Application Server environment.

In Oracle Application Server, you must use either Enterprise Manager or parallel commands in the `dcmctl` command-line utility for starting, stopping, configuring, and deploying applications. Both these tools are coordinated with the Oracle Application Server Distributed Configuration Management subsystem (DCM). You cannot use the OC4J standalone utility `admin.jar` for managing OC4J instances in an Oracle Application Server instance. In addition, do *not* manually update configuration files in Oracle Application Server. (See "Using Oracle Deployment Tools Versus Expert Modes" on page 5-4.)

Note: In Oracle Application Server, use either Enterprise Manager *or* `dcmctl`. Do not attempt to use both simultaneously to target the same OC4J instance or instances, and do not use both for different parts of the same deployment.

The following sections are included here:

- Overview of OC4J Deployment and Configuration in Oracle Application Server
- OC4J Default Web Application in Oracle Application Server
- Application Undeployment and Redeployment in Oracle Application Server

Overview of OC4J Deployment and Configuration in Oracle Application Server

The Enterprise Manager pages for deploying and configuring Web modules are discussed in Chapter 7, "Configuration with Enterprise Manager". See the *Oracle Application Server Containers for J2EE User's Guide* for further information about using Enterprise Manager or the `dcmctl` command-line utility with OC4J.

The deployment scenarios discussed earlier in this chapter, using `admin.jar` or a manual deployment of application files, do *not* apply in an Oracle Application Server environment. Enterprise Manager includes pages for deploying an EAR file or a WAR file, as described in "Application Server Control Console Deploy Application (EAR) Page" on page 7-3 and "Application Server Control Console Deploy Web Application (WAR) Page" on page 7-5. In Oracle Application Server, do *not* manually deploy EAR or WAR files, or deploy loose files, as described in some of the scenarios for OC4J standalone.

In Oracle Application Server, copying EAR or WAR files, unpacking these files into a directory structure, and copying or generating the OC4J descriptors (such as `orion-web.xml` and `orion-application.xml`) are generally handled automatically and transparently through Enterprise Manager.

Deployment to Oracle Application Server through Enterprise Manager or `dcmctl` automatically registers Web applications with Oracle HTTP Server and results in a new mount point in the `mod_oc4j.conf file`. A URL mapping you specify in Enterprise Manager, such as "`/mypath/myapp`", defines the mount point. Mount points determine which URL requests are routed from Oracle HTTP Server to OC4J for processing. In this case, any URL request starting with "`/mypath/myapp`" (after the host and port) is handed off to OC4J.

Notes:

- In Oracle Application Server, all applications are deployed in EAR files. If you use the Application Server Control Console Deploy Web Application Page in Enterprise Manager, which prompts you for a WAR file, an EAR file is transparently created to contain the WAR file.
- Do *not* specify a context path of "`/`" when deploying to OC4J.

Several Enterprise Manager pages, also described in Chapter 7, are available for configuring servlet or Web site parameters. Manipulating settings in these pages results in appropriate configuration updates being made automatically. The configuration files discussed earlier in this chapter are used by OC4J in Oracle Application Server, but this is largely transparent and there are additional logistics to consider:

- The Oracle Process Management and Notification subsystem (OPMN) dynamically overrides some of the settings in the configuration files, as well as some system properties and environment variables.
- The DCM subsystem maintains a repository of configuration information. This repository, rather than the configuration files, contains the true configuration settings.

For these reasons, it is imperative that you not attempt to update configuration manually in Oracle Application Server.

If for some reason you *must* modify configuration files without going through Enterprise Manager, you must run a `dcmctl` update command to inform DCM of the changes. This will affect *all* instances of OC4J managed by DCM and should be avoided.

OPMN and DCM basics are covered in the *Oracle Application Server Administrator's Guide*. The `dcmctl` tool is documented in the *Oracle Application Server Administrator's Guide* as well. Also see the *Oracle Enterprise Manager Concepts* for further information about Enterprise Manager.

OC4J Default Web Application in Oracle Application Server

The default Web application in OC4J standalone is of potential use during development and is discussed in "OC4J Default Application and Default Web Application" on page 5-25. OC4J has a default Web application in an Oracle Application Server environment as well, but it is not for developer use.

In Oracle Application Server, there is just one Web site, and the root namespace is owned by Oracle HTTP Server, not OC4J. The concept of an OC4J default Web application in an Oracle Application Server environment is not sensible in the way that it is for OC4J standalone, in which OC4J itself owns any Web sites. Furthermore, in OC4J standalone, the default Web application is used by manipulating files manually, which is not appropriate in Oracle Application Server.

In Oracle Application Server, as noted elsewhere, routing from Oracle HTTP Server to OC4J is accomplished through mount points in the `mod_oc4j.conf` file. Each time you deploy an application to OC4J in Oracle Application Server (as described in "Application Server Control Console Deploy Application (EAR) Page" on page 7-3 and "Application Server Control Console Deploy Web Application (WAR) Page" on page 7-5), the URL that you specify as the context path results in the specification of that URL as another mount point.

One default OC4J mount point and context path, `/j2ee`, with a default Web application, `defaultWebApp`, exists for OC4J system use only. For example, if a request has a URL pattern that matches an OC4J mount point and, therefore, results in routing to OC4J, but the specified Web application cannot be found, then OC4J uses this default Web application to print an error message. This context path and default Web application are specified in the `<default-web-app>` element in `default-web-site.xml`. This element is required, but is not of direct use to developers.

Application Undeployment and Redeployment in Oracle Application Server

Oracle Enterprise Manager 10*g* includes features to undeploy or redeploy an application. The following sections introduce these features:

- Using Enterprise Manager to Undeploy an Application
- Using Enterprise Manager to Redeploy an Application

Note: Using Enterprise Manager in Oracle Application Server, you can undeploy or redeploy an application through the Application Server Control Console OC4J Applications Page by selecting it from the applications list and then clicking the appropriate button. In either case, if you initially deployed a standalone WAR file, it was automatically wrapped in an EAR file during the deployment process. Therefore, it appears in the applications list.

Using Enterprise Manager to Undeploy an Application

If you are finished using a J2EE application in Oracle Application Server, you can undeploy it through the Application Server Control Console OC4J Applications Page in Enterprise Manager. Select the application from the applications list (using the corresponding radio button) and click the **Undeploy** button. This process removes all directories and files that were created and copied, and updates the server configuration appropriately.

See "Application Server Control Console OC4J Applications Page" on page 7-3 to see what the page looks like and for further information.

Note: There is no need to undeploy an application before redeploying it. The Enterprise Manager `Undeploy` feature is for permanent removal.

Using Enterprise Manager to Redeploy an Application

You can redeploy a J2EE application in Oracle Application Server through the Application Server Control Console OC4J Applications Page in Enterprise Manager. Select the application from the applications list (using the corresponding radio button) and click the **Redeploy** button. You will be prompted for the path to the EAR file.

See "Application Server Control Console OC4J Applications Page" on page 7-3 to see what the page looks like and for further information.

During redeployment, if you have not changed the EAR file since the previous deployment, then server configuration settings are maintained from the previous deployment. This is particularly relevant for settings in the OC4J descriptors, such as `orion-application.xml` and `orion-web.xml`, given that standard configuration, such as through `application.xml` and `web.xml`, is presumably in your EAR file anyway.

If you have changed the EAR file, however, then the previous server configuration is discarded. It is replaced with information from the EAR file, such as from the OC4J descriptors (if present) and applicable default values. In this scenario, if you have made OC4J-specific configuration changes on the server, then you should make the same changes to the OC4J descriptors in the EAR file, in order to keep those changes.

After redeployment, you can check the OC4J servlet and Web site configuration pages, described in Chapter 7, "Configuration with Enterprise Manager", to verify whether desired configurations settings have been maintained.

6 Configuration File Descriptions

This chapter describes the elements and attributes of OC4J configuration files for servlets and Web sites. It includes the following sections:

- Configuration for global-web-application.xml and orion-web.xml
- Configuration for Web Site XML Files

Note: The detailed discussion in this chapter regarding configuration files and their elements and attributes assumes an OC4J standalone development environment. In an Oracle Application Server environment using Enterprise Manager, configuration is through Application Server Control Console 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 Oracle Application Server, see Chapter 7, "Configuration with Enterprise Manager".

Configuration for global-web-application.xml and orion-web.xml

The following sections provide detailed information about the `global-web-application.xml` and `orion-web.xml` configuration files:

- Element Descriptions for global-web-application.xml and orion-web.xml
- DTD for global-web-application.xml and orion-web.xml
- Hierarchical Representation of global-web-application.xml and orion-web.xml
- Sample global-web-application.xml Settings

For an overview of these files, see "OC4J and J2EE Web Descriptors" on page 5-15.

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

This section describes the elements and attributes of the `global-web-application.xml` and `orion-web.xml` files.

The element descriptions in this section apply to either `global-web-application.xml` or an application-specific `orion-web.xml` configuration file. The `global-web-application.xml` file configures the global application and sets defaults; the `orion-web.xml` file can override these defaults for a particular application deployment, as appropriate.

<orion-web-app ... >

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

Notes:

- The `autoreload-jsp-pages` and `autoreload-jsp-beans` attributes of the `<orion-web-app>` element are not currently supported by the OC4J JSP container. You can use the JSP `main_mode` configuration parameter for functionality equivalent to that of `autoreload-jsp-pages`. See the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about this parameter.
- The `<servlet-filter>` subelement and the `document-root`, `get-locale-from-user`, `internationalize-resources`, and `default-mime-type` attributes are no longer supported.

Subelements of `<orion-web-app>`:

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

Attributes of `<orion-web-app>`:

- `default-buffer-size`: Specifies the default size of the output buffer for servlet responses, in bytes. The default is `"2048"`.

Note: The `default-buffer-size` attribute does not affect JSP buffer size.

- `default-charset`: This attribute specifies the ISO character set to use by default. The default is `"iso-8859-1"`.
- `deployment-version`: This attribute specifies 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 attribute is a convenience flag during development. If `development` is set to `"true"`, then the OC4J server checks a particular directory for updates to servlet source files. If a source file has changed since the last request, then OC4J will, upon the next request, recompile the servlet, redeploy the Web application, and reload the servlet and any dependency classes.

  The directory is determined by the setting of the `source-directory` attribute (described next). Supported values for `development` are `"true"` and `"false"` (default).

  > **Note:** The OC4J JSP container does not currently support the `development` flag. It is for servlets only. Use the JSP `main_mode` flag for similar functionality for JSP pages, as documented in the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide*. Features of the old Orion JSP container that relate to the `development` flag do not apply to the OC4J JSP container.

- `source-directory`: For situations in which the `development` attribute is set to `"true"`, the `source-directory` setting specifies where to look for servlet source files to auto-compile. The default is `"/WEB-INF/src"` if it exists, otherwise `"/WEB-INF/classes"`.
- `directory-browsing`: Specifies whether to allow directory browsing for a URL that ends in "`/`". Supported values are `"allow"` and `"deny"` (default).

  Assume the following circumstances:

  – There is no `index.html` file in the application root directory.

  – There is no welcome file defined in the `web.xml` file.

  If `directory-browsing` is set to "`allow`" under these circumstances, then a URL ending in "`/`" results in the contents of the corresponding directory being displayed in the user's browser.

  If `directory-browsing` is set to "`deny`" under these circumstances, then a URL ending in "`/`" results in an error indicating that the directory contents cannot be displayed.

  If there *is* a defined welcome file or there is an `index.html` file in the application root directory, then the contents of that file are displayed, regardless of the `directory-browsing` setting.
- `file-modification-check-interval`: This attribute applies to static files such as HTML files and is the amount of time, in milliseconds, for which a file-modification check is valid. Within that time period since the last check, further checks are not necessary. Zero or a negative number specifies that a check always occurs. The default is `"1000"`. For performance reasons, a very large value ("`1000000`", for example) is recommended in a production environment.
- `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 in which some pages are called infrequently. The default value is 0 (zero), for no timeout.

- `jsp-cache-directory`: This attribute specifies the JSP cache directory, which is used as a base directory for output files from the JSP translator. It is also used as a base directory for application-level TLD caching. The default value is "`./persistence`", relative to the deployment directory of the application.
- `jsp-cache-tlds`: This flag indicates whether persistent TLD caching is enabled for JSP pages. TLD caching is implemented both at a global level, for TLD files in "well-known" tag library locations, and at an application level, for TLD files under the `WEB-INF` directory. Use a "`true`" or "`on`" setting, which is the default, to search for TLD files among all application files. A setting of "`standard`" searches for TLD files only in `/WEB-INF` and subdirectories other than `/WEB-INF/classes` or `/WEB-INF/lib`. A setting of "`false`" or "`off`" disables this feature. Well-known locations are according to the `jsp-taglib-locations` attribute.
- `jsp-taglib-locations`: If persistent TLD caching is enabled for JSP pages (through the `jsp-cache-tlds` attribute), you can use `jsp-taglib-locations` to specify a semicolon-delimited list of one or more directories to use as "well-known" locations. Tag library JAR files can be placed in these locations for sharing across multiple JSP pages and Web applications, and for TLD caching.

  You can specify any combination of absolute directory paths or relative directory paths. Relative paths would be under *`ORACLE_HOME`* if *`ORACLE_HOME`* is defined, or under the current directory (from which the OC4J process was started) if *`ORACLE_HOME`* is not defined. The default value is as follows:

  – *`ORACLE_HOME`*`/j2ee/home/jsp/lib/taglib/` if *`ORACLE_HOME`* is defined.

  or:

  – `./jsp/lib/taglib` if *`ORACLE_HOME`* is not defined.

  Important: Use the `jsp-taglib-locations` attribute only in `global-web-application.xml`, not in `orion-web.xml`.

- `simple-jsp-mapping`: Set this flag to "`true`" if "`*.jsp`" is mapped to *only* the `oracle.jsp.runtimev2.JspServlet` front-end JSP servlet in the `<servlet>` elements of any Web descriptors affecting your application (`global-web-application.xml`, `web.xml`, and `orion-web.xml`). This allows performance improvements for JSP pages. The default setting is "`false`".
- `enable-jsp-dispatcher-shortcut`: A "`true`" setting, which is the case by default, results in significant performance improvements by the OC4J JSP container, especially in conjunction with a "`true`" setting for the `simple-jsp-mapping` attribute. This is particularly true for JSP pages with numerous `jsp:include` statements. Use of the "`true`" setting assumes, however, that if you define JSP files with `<jsp-file>` elements in `web.xml`, then you have corresponding `<url-pattern>` specifications for those files.

  Note: Processing related to the `jsp-print-null`, `jsp-timeout`, `jsp-cache-directory`, `jsp-cache-tlds`, `jsp-taglib-locations`, `simple-jsp-mapping`, and `enable-jsp-dispatcher-shortcut` attributes are handled by the OC4J JSP container. For more information about these attributes and related features, see the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide.*

- `persistence-path`: Indicates where to store servlet `HttpSession` objects for persistence across server restarts or application redeployments. Specify a relative path, which will be relative to an OC4J temporary storage area under the `application-deployments` directory. There is no default value. If no value is defined, then there is no persistence of session objects across restarts or redeployments.

  Session objects must be serializable (directly or indirectly implementing the `java.io.Serializable` interface) or remoteable (directly or indirectly implementing the `java.rmi.Remote` interface) for this feature to work.

  The `persistence-path` attribute is ignored if OC4J clustering is enabled, according to the `<cluster-config>` subelement of the `<orion-web-app>` element.

- `servlet-webdir`: Specifies the path for invoking a servlet by class name. Anything appearing after this path in a URL is assumed to be a class name, including the package, as appropriate.

  This feature is typically for use in an OC4J standalone environment during development and testing. For deployment, use the standard `web.xml` mechanisms for defining the context path and servlet path.

  Here is an example of servlet invocation by class name, assuming a setting of `servlet-webdir="/servlet/"`:

  ```
  http://www.example.com:8888/servlet/foo.bar.SessionServlet
  ```

  Important:

  - Any `servlet-webdir` setting that starts with a slash ("/") enables invocation by class name. This presents a significant security risk and should not be used in a production environment. You can disable invocation by class name with a setting of `servlet-webdir=""` (empty quotes) or by setting the OC4J system property `http.webdir.enable` to a value of `false`.
  - The `servlet-webdir` attribute for an application takes its default value from `global-web-application.xml` if there is a setting there. If there is no setting in `global-web-application.xml`, then the default value is `""`.

  Also see "Servlet Invocation by Class Name During OC4J Development" on page 2-22 and "Additional Security Considerations" on page 2-40.

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

  A servlet may use a temporary directory, for example, to write information to disk as a user is entering data in a form (perhaps for interim or short-term storage before the information is written to a database).

  The specified directory can then be recalled from the servlet context, where it is available through the attribute `javax.servlet.context.tempdir`, as in the following example.

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

A `java.io.File` object is returned, from which you can obtain directory information and contents.

<classpath ... >

Use this element to inform OC4J of additional code locations for Web application classloading—either library files or locations for individual class files.

Attribute of `<classpath>`:

- `path`: You can specify one or more locations, separated by commas or semicolons, where a location can be either of the following:
  - The complete path to a JAR or ZIP file, including the file name
  - A directory path

  In either case, you can use an absolute path or a path that is relative to the configuration file location (`global-web-application.xml` or `orion-web.xml`, as applicable).

  If you specify a directory path, the classloader recognizes only individual class files in the specified directory, not JAR or ZIP files (unless those are specified separately).

  For example, assume the following setting in `orion-web.xml`:

  ```
  <classpath path=/abc/def/lib1.jar,/abc/def/zip1.jar,/abc/def,mydir />
  ```

  The classloader recognizes the following:
  - The `lib1.jar` and `zip1.jar` libraries (but no other libraries in `/abc/def`)
  - Any class files in `/abc/def`
  - Any class files in `mydir`, relative to the location of `orion-web.xml`

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

In `orion-web.xml`, this element 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 of `<context-param-mapping>`:

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

<mime-mappings ... >

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

Attribute of `<mime-mappings>`:

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

<virtual-directory ... >

This element adds a virtual directory mapping for static content, working in a way that is conceptually similar to symbolic links on a UNIX system, for example. The virtual directory enables you to make the contents of the real document root directory available to the application without physically residing in the Web application WAR file. This would be useful, for example, to link an enterprise-wide error page into multiple WAR files.

Attributes of `<virtual-directory>`:

- `real-path`: This is a real path, such as `/usr/local/realpath` in 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 of `<access-mask>`:

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

Attribute of `<access-mask>`:

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

Attributes of `<host-access>`:

- `domain`: Specifies the host or domain.
- `mode`: Specifies whether to allow or deny access from 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 from which to allow or deny access.

Attributes of `<ip-access>`:

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

<cluster-config ... >

Use this element if, and only if, you want to use OC4J clustering. Remove it or comment it out otherwise. Clustered applications have their HTTP session data replicated between clusters in the cluster island. Objects in the HTTP session data must be serializable (directly or indirectly implementing the `java.io.Serializable` interface) or remoteable (directly or indirectly implementing the `java.rmi.Remote` interface) for the session replication to work.

See the *Oracle Application Server Performance Guide* for general information about clustering.

Attributes of `<cluster-config>`:

- `host`: This is the multicast host/IP for transmitting and receiving cluster data. The default is `"230.230.0.1"`.
- `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 is called after the current servlet. This is known as *servlet chaining*, for filtering or transforming certain kinds of output.

Note: Servlet chaining is an older mechanism with essentially the same functionality as standard servlet filtering, which was introduced in version 2.3 of the servlet specification. Use servlet filtering instead. See "Servlet Filters" on page 3-1.

Attributes of `<servlet-chaining>`:

- `mime-type`: Specifies the MIME type to trigger the chaining, such as `"text/html"`.
- `servlet-name`: Specifies 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 a request tracker. Request trackers are useful for logging information, for example.

You must define any request trackers in `orion-web.xml`, not `global-web-application.xml`, because a `<request-tracker>` element points to a servlet defined within the same application.

A request tracker is invoked for each separate request sent from a browser to the server, at the time that the corresponding response is committed (immediately before the response is actually sent).

There can be multiple request trackers, each one defined in a separate `<request-tracker>` element.

Attribute of `<request-tracker>`:

- `servlet-name`: Specifies the servlet to invoke. You can specify either the servlet name or the class name, according to the corresponding `<servlet-name>` or `<servlet-class>` element (both of which are subelements of a `<servlet>` element) in the `web.xml` file.

<session-tracking ... >

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

Notes:

- 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.
- OC4J does not support auto-encoding, in which session IDs are automatically encoded into the URL by the servlet container. This process is nonstandard and expensive. Therefore, OC4J does not support the `<session-tracking>` attributes `autoencode-urls` and `autoencode-absolute-urls`. Also see "Session Tracking in OC4J" on page 2-27.

For general information about servlet sessions, see "Servlet Sessions" on page 2-25.

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

Subelement of `<session-tracking>`:

```
<session-tracker>
```

Attributes of `<session-tracking>`:

- `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`: Specifies the desired domain for cookies. You can use this attribute to track a single client or user over multiple Web sites. The setting must start with a period ("`.`"). For example:

  ```
  <session-tracking cookie-domain=".us.oracle.com" />
  ```

  In this case, the same cookie is used and reused when the user visits any site that matches the "`.us.oracle.com`" domain pattern, such as `webserv1.us.oracle.com` or `webserv2.us.oracle.com`.

  The domain specification must consist of at least two elements, such as "`.us.oracle.com`" or "`.oracle.com`". A setting of "`.com`", for example, is illegal.

  Here are two scenarios in which cookie domain functionality is useful:

  – You can use it to share session state between nodes of a Web application running on different hosts.

  – In OC4J standalone, you can use it for a shared application, when `shared="true"` in a `<web-app>` element in the Web site XML file. In such an application, some requests go through a secure port and some go through a nonsecure port, with each port denoting a separate Web site. You would want the same cookie used regardless of which port is being used. In this scenario, using `cookie-domain` is unnecessary, however, if you use the default ports of 80 for HTTP and 443 for HTTPS. The client would already recognize these as different ports of the same Web site, and only a single cookie would be used.

- `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 a session tracker. Session trackers are useful for logging information, for example.

You must define any session trackers in `orion-web.xml`, not `global-web-application.xml`, because a `<session-tracker>` element points to a servlet defined within the same application.

A session tracker is invoked as soon as a session is created; specifically, at the same time as the invocation of the `sessionCreated()` method of the HTTP session listener (an instance of a class implementing the `javax.servlet.http.HttpSessionListener` interface).

There can be multiple session trackers, each one defined in a separate `<session-tracker>` element.

Attribute of `<session-tracker>`:

- `servlet-name`: Specifies the servlet to invoke. You can specify either the servlet name or the class name, according to the corresponding `<servlet-name>` or `<servlet-class>` element (both of which are subelements of a `<servlet>` element) in the `web.xml` file.

<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 of `<resource-ref-mapping>`:

```
<lookup-context>
```

Attributes of `<resource-ref-mapping>`:

- `location`: Specifies the JNDI location from which to look up the resource. For example:

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

- `name`: Specifies 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 JNDI context (`javax.naming.Context` instance) 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 JNDI context implementation supplied by the resource vendor, or, if none exists, write an implementation that negotiates with the vendor software.

Subelement of `<lookup-context>`:

```
<context-attribute>
```

Attribute of `<lookup-context>`:

- `location`: Specifies the name to look for in the "foreign" (such as third-party) JNDI 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", such as third-party, JNDI context.

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

Attributes of `<context-attribute>`:

- `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 of `<env-entry-mapping>`:

- `name`: Specifies 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.

See the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for additional information about the `<security-role-mapping>` element in OC4J configuration files.

Subelements of `<security-role-mapping>`:

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

Attributes of `<security-role-mapping>`:

- `impliesAll`: Specifies whether this mapping implies all users. Supported values are `"true"` or `"false"` (default).
- `name`: Specifies 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 `jazn-data.xml` (or other valid user managers), then OC4J maps 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 may 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 of `<group>`:

- `name`: Specifies 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 of `<user>`:

- `name`: Specifies 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 of `<ejb-ref-mapping>`:

- `location`: Specifies the JNDI location from which to look up the EJB home.
- `name`: Specifies 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; that is, how long before the resources would expire in the browser. (The browser reloads an expired resource upon the next request for it.) This is useful for caching policies, such as for not reloading images as frequently as documents.

Attributes of `<expiration-setting>`:

- `expires`: Specifies the number of seconds before expiration, or `"never"` for no expiration. The default setting is "`0`" (zero), for immediate expiration.
- `url-pattern`: Specifies the URL pattern that the expiration applies to, such as in the following example:

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

<jazn-web-app ... >

Use this element to configure the OracleAS JAAS Provider and Single Sign-On (SSO) properties for servlet execution. You must set these features appropriately to invoke a servlet under the privileges of a particular security subject.

Attributes of `<jazn-web-app>`:

- `auth-method`: Supported values are "`BASIC`" (for basic J2EE authentication, the default) and "`SSO`". Use "`SSO`" to employ Oracle Application Server Single Sign-On for HTTP client authentication. Use "`BASIC`" mode if your application uses a custom `LoginModule` instance.
- `runas-mode`: Set `runas-mode` to "`true`" to invoke the servlet using the privileges of a particular *subject*. A subject is defined by an instance of the `javax.security.auth.Subject` class and includes a set of facts regarding a single entity, such as a person. Such facts include identities and security-related attributes, such as passwords and cryptographic keys.

  With the default `runas-mode="false"` setting, `doasprivileged-mode` is ignored.
- `doasprivileged-mode`: Assuming `runas-mode="true"`, use the default "`true`" setting of `doasprivileged-mode` to use privileges of a particular subject without being limited by the access-control restrictions of the server.

  Values of `runas-mode="true"` and `doasprivileged-mode="true"` result in use of the static `Subject.doAsPrivileged()` method when the servlet is invoked. Values of `runas-mode="true"` and `doasprivileged-mode="false"` result in use of the static `Subject.doAs()` method. In either case, the JAAS Provider passes in the `Subject` instance in the method call.

  When the `doAsPrivileged()` method is used, the JAAS Provider invokes the method with a null `java.security.AccessControlContext` instance. This is to start the action freshly and execute the servlet without the restrictions of the current server `AccessControlContext` instance. When the `doAs()` method is used, an `AccessControlContext` instance is retrieved from the current thread (from the server).

For additional information about JAAS and the features described for this element, see the *Oracle Application Server Containers for J2EE Security Guide*. You can also refer to Sun Microsystems documentation at the following location:

`http://java.sun.com/j2se/1.4.1/docs/guide/security/jaas/JAASRefGuide.html`

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

Use this element for classloading instructions. See "Loading WAR File Classes Before System Classes in OC4J" on page 2-9 for additional information.

Attributes of `<web-app-class-loader>`:

- `search-local-classes-first`: Set this to "`true`" to search and load WAR file classes before system classes. The default setting is "`false`".
- `include-war-manifest-class-path`: Set this to "`false`" to *not* include the classpath 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 setting is "`true`".

Notes:

- If both attributes are set to "`true`", the overall classpath is constructed so that classes physically residing in the WAR file are loaded prior to any classes from the WAR file manifest classpath. So you can assume that in the event of any conflict, classes physically residing in the WAR file will take precedence.
- To comply with the servlet specification, `search-local-classes-first` functionality cannot be used in loading classes in `java.*` or `javax.*` packages.

<authenticate-on-dispatch ... >

Use this element to disable OC4J authentication of forward or include targets.

Attributes of `<authenticate-on-dispatch>`:

- `value`: Set this to "`false`" to disable authentication of forward or include targets, which complies with the servlet specification. The default value is "`true`" to protect against security violations for applications developed against previous OC4J versions.

<web-app ... >

This element is used as in the standard `web.xml` file; see the servlet 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`.

DTD for global-web-application.xml and orion-web.xml

This section provides the OC4J-specific portion of the DTD for the `global-web-application.xml` and `orion-web.xml` files in the OC4J 10.1.2 implementation. This does not include the DTD portion for the standard `<web-app>` element of the `web.xml` file. (The DTD for `global-web-application.xml` and `orion-web.xml` is a superset of the standard `web.xml` DTD.)

```
<!ENTITY % CHARSET "CDATA">

<!ENTITY % WEBPATH "CDATA">

<!ENTITY % NUMBER "CDATA">

<!ENTITY % HOST "CDATA">

<!ENTITY % PATH "CDATA">

<!ENTITY % CLASSNAME "CDATA">

<!-- A group that this security-role-mapping implies. Ie all the members of the
 specified group are included in this role. -->
<!ELEMENT group (#PCDATA)>
<!ATTLIST group name CDATA #IMPLIED
>

<!-- An attribute sent to the context. The only mandatory attribute in JNDI is
 the 'java.naming.factory.initial' which is the classname of the context factory
 implementation. -->
<!ELEMENT context-attribute (#PCDATA)>
<!ATTLIST context-attribute name CDATA #IMPLIED
value CDATA #IMPLIED
>

<!-- Defines the relative/absolute path to a file containing mime-mappings to
 use. -->
<!ELEMENT mime-mappings (#PCDATA)>
<!ATTLIST mime-mappings path CDATA #IMPLIED
>

<!-- Specifies a codebase where classes used by this application (such as
 servlets/beans) can be found. -->
<!ELEMENT classpath (#PCDATA)>
<!ATTLIST classpath path CDATA #REQUIRED
>

<!-- The specification of an optional javax.naming.Context implementation used
 for retrieving the resource. This is useful when hooking up with 3rd party
 modules, such as a 3rd party JMS server for instance. Either use the context
 implementation supplied by the resource vendor or if none exists write an
 implementation which in turn negotiates with the vendor software. -->
<!ELEMENT lookup-context (context-attribute+)>
<!ATTLIST lookup-context location CDATA #IMPLIED
>

<!-- Specifies a servlet to use as request-tracker; request-trackers are invoked
 for every request and are useful for logging purposes, for example -->
<!ELEMENT request-tracker (#PCDATA)>
<!ATTLIST request-tracker servlet-name CDATA #IMPLIED
>

<!-- The resource-ref element is used for the declaration of a reference to
 an external resource such as a datasource, JMS queue, mail session or similar.
 The resource-ref-mapping ties this to a JNDI-location when deploying. -->
<!ELEMENT resource-ref-mapping (lookup-context?)>
<!ATTLIST resource-ref-mapping location CDATA #IMPLIED
name CDATA #REQUIRED
>

<!-- Tag that is defined if the application is to be clustered. Clustered
 applications have their ServletContext and session data
 shared between the apps in the cluster, the values have to be either
 Serializable or be remote RMI-objects (implement java.rmi.Remote). -->
<!ELEMENT cluster-config (#PCDATA)>
<!ATTLIST cluster-config host %HOST; "230.0.0.1"
id CDATA "based on local IP"
port %NUMBER; "9127"
>

<!-- Specifies an optional access-mask for this application, hostnames and
 ip/subnets can be used to filter out allowed clients of this application. -->
<!ELEMENT access-mask (host-access*, ip-access*)>
<!ATTLIST access-mask default (allow|deny) "allow"
>

<!-- Overrides the value of an env-entry in the assembly descriptor. It is used
 to keep the .ear (assembly) clean from deployment-specific values. The body is
 the value. -->
```

```
<!ELEMENT env-entry-mapping (#PCDATA)>
<!ATTLIST env-entry-mapping name CDATA #IMPLIED
>

<!-- Specifies the Expires setting for a given set of resources, useful for
 caching policies (for instance for browsers not to reload images as frequently
 as documents). -->
<!ELEMENT expiration-setting (#PCDATA)>
<!ATTLIST expiration-setting expires CDATA #IMPLIED
url-pattern CDATA #IMPLIED
>

<!-- Overrides the value of a context-param in the assembly descriptor. It is
 used to keep the .ear (assembly) clean from deployment-specific values. The
 body is the value. -->
<!ELEMENT context-param-mapping (#PCDATA)>
<!ATTLIST context-param-mapping name CDATA #IMPLIED
>

<!-- Session-tracking settings for this application. -->
<!ELEMENT session-tracking (session-tracker*)>
<!ATTLIST session-tracking autoencode-absolute-urls (true|false) "false"
autoencode-urls (true|false) "true"
autojoin-session (true|false) "false"
cookie-domain CDATA #IMPLIED
cookie-max-age %NUMBER; "in memory only"
cookies (enabled|disabled) "enabled"
>

<!-- A user that this security-role-mapping implies. -->
<!ELEMENT user (#PCDATA)>
<!ATTLIST user name CDATA #IMPLIED
>

<!-- Adds a virtual directory mapping, used to include files that doesnt
 physically reside below the document root among the web-exposed files. -->
<!ELEMENT virtual-directory (#PCDATA)>
<!ATTLIST virtual-directory real-path %PATH; #IMPLIED
virtual-path %PATH; #IMPLIED
>

<!-- Specifies an ip/netmask who is allowed access. -->
<!ELEMENT ip-access (#PCDATA)>
<!ATTLIST ip-access ip CDATA #REQUIRED
mode (allow|deny) #REQUIRED
netmask CDATA #IMPLIED
>

<!-- Specifies a servlet to use as chainer for a specified mime-type. Useful to
 filter/transform certain kinds of output. -->
<!ELEMENT servlet-chaining (#PCDATA)>
<!ATTLIST servlet-chaining mime-type CDATA #IMPLIED
servlet-name CDATA #IMPLIED
>

<!-- Specifies a domain or netmask who is allowed access. -->
<!ELEMENT host-access (#PCDATA)>
<!ATTLIST host-access domain CDATA #REQUIRED
mode (allow|deny) #REQUIRED
>
```

```
<!-- The ejb-ref element is used for the declaration of a reference to
 another enterprise bean's home. The ejb-ref-mapping ties this to JNDI-location
 when deploying. -->
<!ELEMENT ejb-ref-mapping (#PCDATA)>
<!ATTLIST ejb-ref-mapping location CDATA #IMPLIED
name CDATA #REQUIRED
>

<!-- The runtime mapping (to groups and users) of a role. Maps to a
 security-role of the same name in the assembly descriptor. -->
<!ELEMENT security-role-mapping (group*, user*)>
<!ATTLIST security-role-mapping impliesAll CDATA #IMPLIED
name CDATA #IMPLIED
>

<!-- Specifies a servlet to use as session-tracker; session-trackers are invoked
 as soon as a session is created and are useful for logging purposes, for
 example -->
<!ELEMENT session-tracker (#PCDATA)>
<!ATTLIST session-tracker servlet-name CDATA #IMPLIED
>

<!-- JAZN configuration -->
<!ELEMENT jazn-web-app (#PCDATA)>
<!ATTLIST jazn-web-app auth-method CDATA #IMPLIED
runas-mode (true | false) "false"
doasprivileged-mode (true | false) "true"
>

<!-- Web-app classloader configuration -->
<!ELEMENT web-app-class-loader EMPTY>
<!ATTLIST web-app-class-loader
search-local-classes-first (true | false) "false"
include-war-manifest-class-path (true | false) "true"
>

<!-- Authentication of forward/include targets -->
<!ELEMENT authenticate-on-dispatch EMPTY>
<!ATTLIST authenticate-on-dispatch
value (true | false) "true"
>

<!-- This file contains the orion-specific configuration for a web-application.
 The path to the file is located at
 ORION_HOME/application-deployments/deploymentName/warname(.war)/orion-web.xml
 or (web-app-root/)WEB-INF/orion-web.xml if no deployment-directory is specified
 in server.xml. -->
<!ELEMENT orion-web-app (  classpath*,  context-param-mapping*,  mime-mappings*,
  virtual-directory*,  access-mask?,  cluster-config?,  servlet-chaining*,
  request-tracker*,  session-tracking?,  resource-ref-mapping*,
  security-role-mapping*,  env-entry-mapping*,  ejb-ref-mapping*,
  expiration-setting*,  web-app?, jazn-web-app?, web-app-class-loader?,
  authenticate-on-dispatch? )>
<!ATTLIST orion-web-app autoreload-jsp-beans (true|false) "true"
autoreload-jsp-pages (true|false) "true"
default-buffer-size CDATA "2048"
default-charset %CHARSET; "iso-8859-1"
deployment-version CDATA #IMPLIED
development (true|false) "false"
```

```
directory-browsing (allow|deny) "deny"
file-modification-check-interval %NUMBER; "1000"
jsp-cache-directory CDATA #IMPLIED
jsp-cache-tlds (true|on|standard|false|off) "true"
jsp-taglib-locations CDATA #IMPLIED
jsp-print-null (true|false) "true"
jsp-timeout %NUMBER; "0 (never)"
simple-jsp-mapping (true|false) "false"
enable-jsp-dispatcher-shortcut (true|false) "true"
persistence-path CDATA #IMPLIED
servlet-webdir %PATH; "/servlet/"
source-directory CDATA #IMPLIED
temporary-directory CDATA #IMPLIED
>
```

Hierarchical Representation of global-web-application.xml and orion-web.xml

This section contains a representation of the hierarchy of the `global-web-application.xml` and `orion-web.xml` files.

Note: For simplicity of presentation, end-tags are omitted.

```
<orion-web-app default-buffer-size="..." default-charset="..."
               deployment-version="..." development="..."
               source-directory="..." directory-browsing="..."
               file-modification-check-interval="..."
               jsp-print-null="..." jsp-timeout="..." jsp-cache-directory="..."
               jsp-cache-tlds="..." jsp-taglib-locations="..."
               simple-jsp-mapping="..." enable-jsp-dispatcher-shortcut="..."
               persistence-path="..." servlet-webdir="..."
               temporary-directory="...">
     <classpath path="...">
     <context-param-mapping name="...">
     <mime-mappings path="...">
     <virtual-directory real-path="..." virtual-path="...">
     <access-mask default="...">
          <host-access domain="..." mode="...">
          <ip-access ip="..." netmask="..." mode="...">
     <cluster-config host="..." id="..." port="...">
     <servlet-chaining mime-type="..." servlet-name="...">
     <request-tracker servlet-name="...">
     <session-tracking autojoin-session="..." cookies="..."
                       cookie-domain="..." cookie-max-age="...">
          <session-tracker servlet-name="...">
     <resource-ref-mapping location="..." name="...">
          <lookup-context location="...">
               <context-attribute name="..." value="...">
     <env-entry-mapping name="...">
     <security-role-mapping impliesAll="..." name="...">
          <group name="...">
          <user name="...">
     <ejb-ref-mapping location="..." name="...">
     <expiration-setting expires="..." url-pattern="...">
     <jazn-web-app auth-method="..." runas-mode="..."
                   doasprivileged-mode="...">
     <web-app-class-loader search-local-classes-first="..."
                           include-war-manifest-class-path="...">
     <authenticate-on-dispatch value="...">
```

```
      <web-app>   AS IN STANDARD WEB.XML
```

Sample global-web-application.xml Settings

This is an abbreviated example of a default `global-web-application.xml` file, showing some `<orion-web-app>` attribute settings, mime-mapping settings, and the setup and mapping of the JSP and RMI front-end servlets (all possibly 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"
   jsp-timeout="0"
   jsp-taglib-locations="./jsp/lib/taglib"
>

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

   <web-app>

      <servlet>
         <servlet-name>jsp</servlet-name>
         <servlet-class>oracle.jsp.runtimev2.JspServlet</servlet-class>
         <load-on-startup>0</load-on-startup>
         <!-- you can disable page scope listener if you
              don't need this function. -->
         <init-param>
            <param-name>check_page_scope</param-name>
            <param-value>true</param-value>
         </init-param>
         <!-- you can set main_mode to "justrun" to speed up
              JSP dispatching, if you don't need to recompile
              your JSP anymore. You can always switch your
              main_mode. Please see our doc for details -->
         <!--
         <init-param>
            <param-name>main_mode</param-name>
            <param-value>justrun</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>
         <servlet-name>rmi</servlet-name>
         <servlet-class>
```

```
            com.evermind.server.rmi.RMIHttpTunnelServlet
         </servlet-class>
      </servlet>
      <servlet-mapping>
         <servlet-name>rmi</servlet-name>
         <url-pattern>/*.tunnelrmi</url-pattern>
      </servlet-mapping>

   </web-app>
</orion-web-app>
```

Configuration for Web Site XML Files

The following sections provide detailed information about Web site XML configuration files, including `default-web-site.xml`, for an Oracle Application Server environment, and `http-web-site.xml`, for an OC4J standalone environment:

- Element Descriptions for Web Site XML Files
- DTD for Web Site XML Files
- Hierarchical Representation of Web Site XML Files
- Sample default-web-site.xml File

For an overview of these files, see "OC4J Web Site Descriptors" on page 5-18.

Element Descriptions for Web Site XML Files

The element descriptions in this section apply to any OC4J Web site XML file, including `default-web-site.xml` (Oracle Application Server) and `http-web-site.xml` (OC4J standalone).

<web-site ... >

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

Subelements of `<web-site>`:

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

Attributes of `<web-site>`:

- `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 Oracle Application Server. 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 *Oracle Application Server Performance Guide* for general information about clustering.

- `display-name`: You can use this attribute to specify a user-friendly or informal Web site name.
- `host`: Specifies the host for this Web site, as either a DNS host name or an IP address. If a server is a "multi-home" machine (having multiple IP addresses), you can use the "`[ALL]`" setting to listen to all IP addresses. They would all be combined into this single Web site.
- `log-request-info`: Specifies whether to write information about the incoming request into the Web site log if an error occurs. Supported values are `"true"` and `"false"` (default). The Web site log is enabled through either the `<access-log>` or `<odl-access-log>` element, described later in this section. ("OC4J Logging" on page 2-14 provides additional information about enabling logs, including the Web site log.)
- `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 Secure Socket Layer (SSL) functionality. Supported values are `"true"` and `"false"` (default). For a protocol setting of "`ajp13`" (used in an Oracle Application Server environment), a "`true`" setting results in secure AJP protocol between Oracle HTTP Server and OC4J. For a protocol setting of "`http`" (used in OC4J standalone), a "`true`" setting results in HTTPS protocol between the client and OC4J.

  Also note that a `secure="true"` setting requires that you use the `<ssl-config>` element (a subelement under the `<web-site>` element) to specify the keystore path and password. This element is documented later in this section.

  Note: SSL and HTTPS features are also available through Oracle HTTP Server for communication between Oracle HTTP Server and the client. For information, see *Oracle Application Server Security Guide*.

- `protocol`: Specifies the protocol that the Web site is using. Possible values are `"http"` and `"ajp13"` (for AJP, the default). In a production environment with Oracle Application Server, you should use only the "`ajp13`" setting. The AJP protocol is for use with Oracle HTTP Server and `mod_oc4j`. Note that each protocol must have a corresponding port, and each port must have a corresponding protocol.

  The "`http`" setting is for OC4J standalone.

  To use either an "`ajp13`" or "`http`" setting in secure mode (SSL), you must set the `secure` flag to "`true`" and use the `<ssl-config>` subelement to specify the keystore path and password. This element is documented later in this section.
- `port`: Specifies the port number for this Web site. Each port must have a corresponding protocol, and each protocol must have a corresponding port. In OC4J standalone, a `port` setting of 8888 is used by default for direct access to the OC4J listener, but you can change this as desired.

  In an Oracle Application Server environment, this port setting is overridden by OPMN, the Oracle Process Management and Notification system. Oracle Application Server uses port 7777 by default for access through Oracle HTTP Server with Oracle Application Server Web Cache enabled.

Important: In a UNIX environment, port numbers less than 1024 require root privileges for access. Also note that if there is no port specification from the client browser, port 80 is assumed for HTTP protocol and port 443 for HTTPS.

- `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 significant 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 element specifies a perceived front-end host and port of this Web site as seen by HTTP clients. When the site is behind 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 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 of `<frontend>`:

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

<web-app ... >

This element binds a particular Web module to this Web site. It specifies the name of a J2EE application archive (EAR file name minus the `.ear` extension) from the `server.xml` file, and the name of a Web module within the J2EE application. The Web module is defined in the J2EE `application.xml` file in the application EAR file (or possibly in the `orion-application.xml` file in the EAR file). The Web module is bound at the location specified by the `<web-app>` element `root` attribute.

Note: It is possible to deploy a WAR file by itself, instead of within an EAR file. In OC4J standalone, such Web applications are added to the OC4J default application. (In OC4J, there must always be a parent application of some sort.) See "OC4J Default Application and Default Web Application" on page 5-25 for more information.

In this scenario, the Web site XML file `<web-app>` element specifies the name of the default application rather than the name of a J2EE application archive. More details are provided in the attribute descriptions and examples that follow.

Mapping to and from Web site XML files, particularly with respect to the `application` and `name` attributes, is shown in examples elsewhere in this document. See "Example: Mappings to and from Web Site Descriptors" on page 5-19 (for a typical scenario of deploying a WAR file within an EAR file) and "Deploying an Independent WAR File to OC4J Standalone" on page 5-32 (for the scenario of deploying a WAR file by itself to the OC4J default application).

Attributes of `<web-app>`:

- `application`: Specifies the J2EE application archive name, which is the EAR file name without the `.ear` extension, and which corresponds to the `name` attribute of an `<application>` element in the `server.xml` file.

  Notes: If you deploy a WAR file by itself in OC4J standalone, using the OC4J default application as the parent, then the `application` attribute instead reflects the name of the default application, according to the `<global-application>` element in the `server.xml` file.

- `load-on-startup`: This is an optional attribute to specify whether this Web module 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 2-6 for more information.

- `max-inactivity-time`: This is an optional integer attribute to specify the number of minutes of inactivity after which OC4J will shut down the Web module. By default, a Web module is never shut down due to inactivity.

- `name`: Specifies the name of a Web module within the specified J2EE application, and corresponds to the `<web-uri>` value (without the `.war` extension) of a `<web>` subelement of a `<module>` element in the J2EE `application.xml` file. The J2EE `application.xml` file is in the EAR file.

Notes:

- If you deploy a WAR file by itself in OC4J standalone, using the OC4J default application as the parent, then the `name` attribute instead reflects the value of the `id` attribute of a `<web-module>` element in the OC4J global `application.xml` file. This is the `application.xml` file for the OC4J default application, but be aware that it is not a standard J2EE file; it is OC4J-specific. Also note that the `id` attribute, as with the `name` attribute of the `<web-app>` element, does not have the `.war` extension.
- An application can also have an `orion-application.xml` file in the EAR file, with `<web-module>` elements that define additional Web modules, or even override Web modules defined in the J2EE `application.xml` file (although overriding is *not* advised). The `name` attribute can reflect the `id` value of a `<web-module>` element in `orion-application.xml`, instead of reflecting a `<web-uri>` value in the J2EE `application.xml` file.
- The `orion-application.xml` file uses the same DTD as the OC4J global `application.xml` file; namely, `orion-application.dtd`.

- `root`: Specifies the path to which the Web module is to be bound, which defines the context path portion of the URL used to invoke the module. For example, if the Web module `CatalogApp` at Web site `www.example.com` is bound to the `root` setting `"/catalog"`, then it can be invoked as follows:

```
http://www.example.com/catalog
```

Important:

- The `root` attribute overrides the `<context-root>` value of the corresponding `<web>` element in the J2EE `application.xml` file. Even though the `<context-root>` element is mandatory in an `application.xml` file, its value is not used by OC4J.
- Specifying a `root` setting of "`/`" will override the OC4J default Web application. This setting or a null setting is not allowed by the `admin.jar` utility when binding a Web application to the Web site.

- `shared`: Allows sharing of a published Web module between Web sites, when a Web site is defined by a particular pairing of a protocol and a port. 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. An example is to share a Web application in OC4J standalone between an HTTP site and an HTTPS site at the same context path, when SSL is required for some but not all the communications. (Performance is improved by encrypting only sensitive information, rather than all information.)

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 may be necessary to work around issues such as SSL session timeouts not being properly supported in some browsers.

Important: Use `shared="true"` only in OC4J standalone.

<default-web-app ... >

This element creates a reference to the default Web application of this Web site. For users, it is meaningful only in an OC4J standalone environment. See "OC4J Default Application and Default Web Application" on page 5-25 for more information.

In an Oracle Application Server environment, the OC4J default Web application has system-level functionality but is not otherwise meaningful. See "OC4J Default Web Application in Oracle Application Server" on page 5-40.

The `<default-web-app>` element uses the same attributes as the `<web-app>` element described immediately preceding, but note that 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 module and associated `web-application.xml` file. User applications are reached at */username/* from the server root.

Attributes of `<user-web-apps>`:

- `max-inactivity-time`: Optional integer attribute to specify the number of minutes of inactivity after which OC4J will shut down the Web module. By default, a Web module is never shut down due to inactivity.
- `path`: Specifies a path to specify the local directory of the user application, including a wildcard for the user name. The default path setting in UNIX, for example, is `"/home/`*`username`*`"`, where *`username`* is replaced by the particular user name.

<access-log ... >

Use this element to enable text-based access logging for this Web site and to specify information about the access log, including the path, file name, and what information is included. The log file is where incoming requests (each access of the Web site) are logged.

Alternatively, use the `<odl-access-log>` element (described immediately following) for ODL logging. See "Oracle Diagnostic Logging Versus Text-Based Logging" on page 2-16 for information about ODL.

Note: Do not use both `<access-log>` and `<odl-access-log>`; you can use only one type of logging. (The last element in the Web site XML file would take precedence, but do not count on this behavior.)

Attributes of `<access-log>`:

- `format`: Specifies 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 results 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
  ```

  In this example, the user is null, the time is in brackets (as specified in the `format` setting), the request is in single-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. This can be an absolute path or a path relative to the `j2ee/home/config` directory. The default setting in `default-web-site.xml` is the following:

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

  Note: Note the difference between the `path` attribute of `<access-log>`, which specifies a path and file name, and the `path` attribute of `<odl-access-log>`, which specifies a path only. (ODL log file names are fixed.)

- `split`: Specifies how often to begin a new access log. Supported values are `"none"` (never, which is the default), `"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 it uses, refer to the Sun Microsystems Javadoc at the following location:

  http://java.sun.com/j2se/1.4.2/docs/api/

  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 are named such as in the following example:

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

<odl-access-log ... >

Use this element to enable ODL-based access logging for the Web site and to specify information about the access logs, including the path, and maximum values for the size of each file and the total size of all files in the log directory. The log files are where incoming requests (each access of the Web site) are logged.

Alternatively, use the `<access-log>` element (described immediately preceding) for text-based logging.

See "Oracle Diagnostic Logging Versus Text-Based Logging" on page 2-16 for information about ODL.

> **Note:** Do not use both `<access-log>` and `<odl-access-log>`; you can use only one type of logging or the other. (The last element in the Web site XML file would take precedence, but do not count on this behavior.)

Attributes of `<odl-access-log>`:

- `path`: Specifies the path to the access log directory. This can be an absolute path or a path relative to the `j2ee/home/config` directory. For example:

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

  The initial log file name in this directory is `log1.xml`. As the maximum file size (specified by the `max-file-size` attribute) is reached, subsequent log files are named `log2.xml`, `log3.xml`, and so on.

  > **Note:** Note the difference between the `path` attribute of `<access-log>`, which specifies a path and file name, and the `path` attribute of `<odl-access-log>`, which specifies a path only. (ODL log file names are fixed.)

- `max-file-size`: Specifies the maximum size of each log file, in kilobytes.
- `max-directory-size`: Specifies the maximum total size, in kilobytes, of all log files in the directory specified in the `path` attribute.

<ssl-config ... >

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

See "Servlet Security" on page 2-33 for related information.

Subelement of `<ssl-config>`:

```
<property>
```

Attributes of `<ssl-config>`:

- `keystore`: A relative or absolute path to the keystore database (a binary file) used by this Web site to store certificates and keys for the user base in this installation. The path value includes the file name. A relative path is relative to the location of the Web site XML file.

  A keystore is a `java.security.KeyStore` instance and can be created and maintained using the `keytool` utility, provided with the Sun Microsystems JDK.

- `keystore-password`: The required password to open the keystore.
- `needs-client-auth`: Indicates whether the entity that is a client to OC4J, such as Oracle HTTP Server, must submit a certificate for authorization so it can communicate with OC4J. Supported values are `"true"` for "client authentication" (certificate required), and `"false"` (default, no certificate required).
- `provider`: You can use this attribute to specify a provider if you are using JSSE (Java Secure Socket Extension). By default, OC4J usually employs the Sun Microsystems implementation of SSL, using an instance of the following for the provider:

  ```
  com.sun.net.ssl.internal.ssl.Provider
  ```

  However, OC4J employs the Oracle SSL implementation in some cases, such as for SOAP and `http_client`.
- `factory`: If you are not using JSSE, use the `factory` attribute to specify an implementation of `SSLServerSocketFactory`. The default setting is:

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

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

<property ... >

Use `<property>` subelements of the `<ssl-config>` element to pass parameters to a third-party `SSLServerSocketFactory` implementation, if applicable.

Attributes of `<property>`:

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

DTD for Web Site XML Files

This section provides the DTD for Web site XML configuration files, including `default-web-site.xml` and `http-web-site.xml`, in the OC4J 10.1.2 implementation.

```
<!ENTITY % WEBPATH "CDATA">

<!ENTITY % NUMBER "CDATA">

<!ENTITY % HOST "CDATA">

<!ENTITY % BOOLEAN "true|false">

<!ENTITY % PATH "CDATA">

<!-- When enabled user dirs/apps will be supported. Each user has his own
 private web-application (and connected web-application.xml file).
 The user apps are reached at /~username/ from the server root. -->
<!ELEMENT user-web-apps (#PCDATA)>
<!ATTLIST user-web-apps max-inactivity-time CDATA "no shutdown"
path %PATH; #IMPLIED
>

<!-- Reference to the default <a class="link"
```

```
 href="web.xml.html">web-application</a> of this site. This application will be
 bound to the root of the site. -->
<!ELEMENT default-web-app (#PCDATA)>
<!ATTLIST default-web-app application CDATA #IMPLIED
load-on-startup (true|false) "true"
max-inactivity-time %NUMBER; #IMPLIED
name CDATA #IMPLIED
root %WEBPATH; #IMPLIED
shared (true|false) "false"
>

<!-- A short description of this web-site. -->
<!ELEMENT description (#PCDATA)>

<!-- Relative/absolute path to the access-log for this site, this is where
 incoming requests will be logged. -->
<!ELEMENT access-log (#PCDATA)>
<!ATTLIST access-log format CDATA "$ip - $user - [$time] '$request' $status
 $size"
path CDATA #IMPLIED
split (none|hour|day|week|month) "none"
suffix CDATA #IMPLIED
>

<!-- An ODL formated log file. The max-file-size is the maximum number of
 kilobytes a single log file is allowed to grow to. The max-directory-size is
 the maximum number of kilobytes that the directory is allowed to contain. -->
<!ELEMENT odl-access-log (#PCDATA)>
<!ATTLIST odl-access-log path CDATA #REQUIRED max-file-size CDATA #IMPLIED
 max-directory-size CDATA #IMPLIED>

<!-- Reference to a <a class="link" href="web.xml.html">web-application</a>.
 This application will be bound at the location specified by the 'root'
 attribute. -->
<!ELEMENT web-app (#PCDATA)>
<!ATTLIST web-app application CDATA #IMPLIED
load-on-startup (true|false) "false"
max-inactivity-time %NUMBER; "no shutdown"
name CDATA #IMPLIED
root %WEBPATH; #IMPLIED
shared (true|false) "false"
>

<!-- A configuration parameter. -->
<!ELEMENT property (#PCDATA)>
<!ATTLIST property name CDATA #IMPLIED
value CDATA #IMPLIED
>

<!-- Specifies SSL-configuration settings. These settings are used if
secure="true" is specified on the site.
If a 3rd party SSLServerSocketFactory implementation is used then x property
tags can be defined to send arbitary arguments to the factory. -->
<!ELEMENT ssl-config (property*)>
<!ATTLIST ssl-config factory CDATA
 "com.evermind.server.JSSESSLServerSocketFactory"
keystore CDATA #IMPLIED
keystore-password CDATA #IMPLIED
needs-client-auth (true|false) "false"
provider CDATA #IMPLIED
```

```
>

<!-- The frontend tag describes which IP, port, and so on that HTTP clients
 perceive this site to be. This is needed when acting behind a load balancer or
 firewall in order to provide the correct info to web-app code when rewriting
 URLs -->
<!ELEMENT frontend (#PCDATA)>
<!ATTLIST frontend host CDATA #IMPLIED
port CDATA #IMPLIED
>

<!-- This file contains the configuration for a web-site. -->
<!ELEMENT web-site (description?, frontend?, default-web-app, web-app*,
 user-web-apps?, access-log?, odl-access-log?, ssl-config?)>
<!ATTLIST web-site cluster-island CDATA #IMPLIED
display-name CDATA #IMPLIED
protocol CDATA #IMPLIED
host %HOST; "[ALL]"
log-request-info (true|false) "false"
max-request-size CDATA #IMPLIED
port %NUMBER; "80"
secure (true|false) "false"
use-keep-alives CDATA #IMPLIED
virtual-hosts CDATA #IMPLIED
>
```

Hierarchical Representation of Web Site XML Files

This section contains a representation of the hierarchy of Web site XML configuration files, including `default-web-site.xml` and `http-web-site.xml`.

Note: For simplicity of presentation, end-tags are omitted.

```
<web-site cluster-island="..." display-name="..." host="..."
          log-request-info="..." max-request-size="..." secure="..."
          protocol="..." port="..." use-keep-alives="..."
          virtual-hosts="...">
     <description>
     <frontend host="..." port="...">
     <web-app application="..." load-on-startup="..."
             max-inactivity-time="..." name="..." root="..." shared="...">
     <default-web-app application="..." load-on-startup="..."
             max-inactivity-time="..." name="..." root="..." shared="...">
     <user-web-apps max-inactivity-time="..." path="...">
     <access-log format="..." path="..." split="..." suffix="...">
     <odl-access-log path="..." max-file-size="..." max-directory-size="...">
     <ssl-config keystore="..." keystore-password="..."
                needs-client-auth="..." provider="..." factory="...">
          <property name="..." value="...">
```

Sample default-web-site.xml File

This is a sample `default-web-site.xml` file, similar to the default file provided with OC4J for an Oracle Application Server environment:

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

```
<web-site host="myhost" port="0"  protocol="ajp13"
          display-name="Default Oracle Application Server Java WebSite"
          cluster-island="1" >

   <!-- 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="default" name="admin_web" root="/adminoc4j" />

   <!-- Access Log, where requests are logged to -->
   <access-log path="../log/default-web-access.log" />

   <!-- Uncomment this if you want to use ODL logging capabilities
   <odl-access-log path="../log/default-web-access" max-file-size="1000"
                   max-directory-size="10000"/>
   -->

</web-site>
```

7 Configuration with Enterprise Manager

In an Oracle Application Server environment, configuration of Web modules is performed through Oracle Enterprise Manager 10*g*. This chapter describes key features of Enterprise Manager for servlet and Web site configuration. It includes the following sections:

- Web Module Configuration in Oracle Enterprise Manager 10g
- Application Server Control Console Page Descriptions

Web Module Configuration in Oracle Enterprise Manager 10*g*

The direct use of `global-web-application.xml`, `orion-web.xml`, and `default-web-site.xml` elements and attributes, described in Chapter 6, is for development and deployment in an OC4J standalone environment. In an Oracle Application Server environment, such as for production deployment, use Enterprise Manager for Web module configuration and deployment.

Oracle Enterprise Manager 10*g* Application Server Control Console is the administration console for an Oracle Application Server instance. It enables you to monitor real-time performance, manage Oracle Application Server components and instances, and configure these components and instances. This includes any instances of OC4J. In particular, Application Server Control Console includes pages to configure servlets and Web sites. Application Server Control Console comes with your Oracle Application Server installation. Log in as the `ias_admin` user.

This chapter covers relevant Application Server Control Console pages for managing and configuring Web modules in an OC4J instance within Oracle Application Server. Some of the pages allow you to alter `global-web-application.xml`, `orion-web.xml`, and `default-web-site.xml` settings. Other pages display `web.xml` settings, which you can override through `orion-web.xml` settings.

Each page description notes the corresponding elements and attributes in `web.xml`, `orion-web.xml`/`global-web-application.xml`, or `default-web-site.xml`. The elements and attributes for `global-web-application.xml` or `orion-web.xml` are documented in "Element Descriptions for global-web-application.xml and orion-web.xml" on page 6-1. The `default-web-site.xml` elements and attributes are covered in "Element Descriptions for Web Site XML Files" on page 6-20. For information about `web.xml` elements, refer to the servlet specification.

See the *Oracle Application Server Containers for J2EE User's Guide* for additional information about using Enterprise Manager with OC4J.

Application Server Control Console Page Descriptions

The following sections discuss key Application Server Control Console pages in Enterprise Manager for servlet and Web site configuration and deployment:

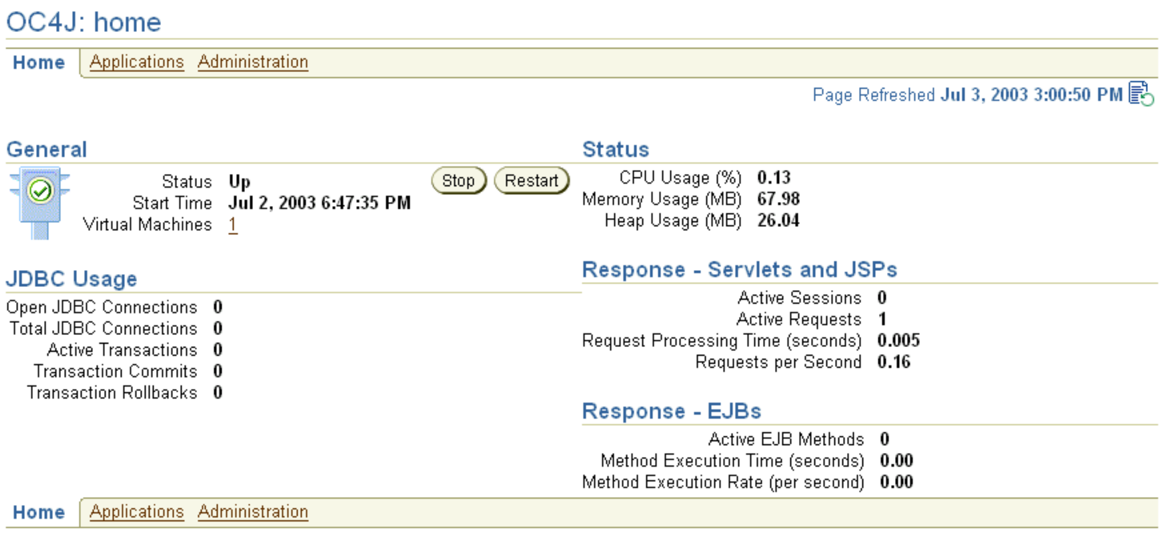
- Application Server Control Console OC4J Home Page
- Application Server Control Console OC4J Applications Page
- Application Server Control Console Deploy Application (EAR) Page
- Application Server Control Console Deploy Web Application (WAR) Page
- Application Server Control Console OC4J Administration Page
- Application Server Control Console Website Properties Page
- Application Server Control Console Web Module Page
- Application Server Control Console Web Module Properties Page
- Application Server Control Console Web Module Mappings Page
- Application Server Control Console Web Module Filtering and Chaining Page
- Application Server Control Console Web Module Environment Page
- Application Server Control Console Web Module Advanced Properties Page

Application Server Control Console OC4J Home Page

When you first access an Oracle Application Server instance through Application Server Control Console in Enterprise Manager, you reach the Oracle Application Server Instance Home Page. Through this page, you can drill down to any of the running OC4J instances by selecting the name of the instance (`home`, for example) in the System Components table. Application Server Control Console then displays the OC4J Home Page for that instance.

Figure 7–1 shows portions of the OC4J Home Page for the `home` instance.

Figure 7–1 Application Server Control Console OC4J Home Page



Page Refreshed Jul 3, 2003 3:00:50 PM

General

Status	Up
Start Time	Jul 2, 2003 6:47:35 PM
Virtual Machines	1

Status

CPU Usage (%)	0.13
Memory Usage (MB)	67.98
Heap Usage (MB)	26.04

JDBC Usage

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

Response - Servlets and JSPs

Active Sessions	0
Active Requests	1
Request Processing Time (seconds)	0.005
Requests per Second	0.16

Response - EJBs

Active EJB Methods	0
Method Execution Time (seconds)	0.00
Method Execution Rate (per second)	0.00

From the OC4J Home Page, you can do the following:

- Click **Applications** to access the Application Server Control Console OC4J Applications Page.
- Click **Administration** to access the Application Server Control Console OC4J Administration Page.

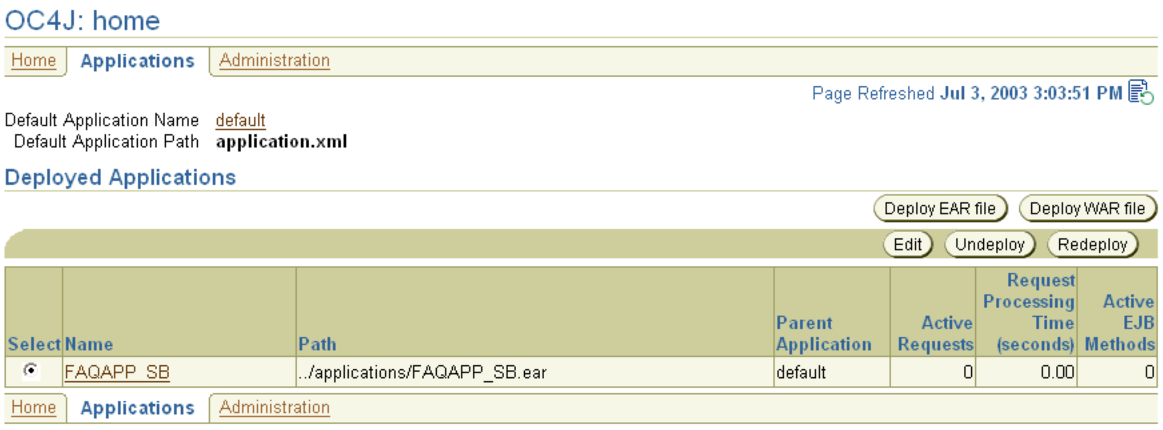
Application Server Control Console OC4J Applications Page

Figure 7–2 shows the OC4J Applications Page, which enables you to deploy applications. You can reach this page by clicking **Applications** from the OC4J Home Page.

In particular, relating to topics covered in this manual, note the following:

- Clicking the **Deploy EAR file** button accesses the Deploy Application Page.
- Clicking the **Deploy WAR file** button accesses the Deploy Web Application Page.

Figure 7–2 Application Server Control Console OC4J Applications Page



Page Refreshed **Jul 3, 2003 3:03:51 PM**

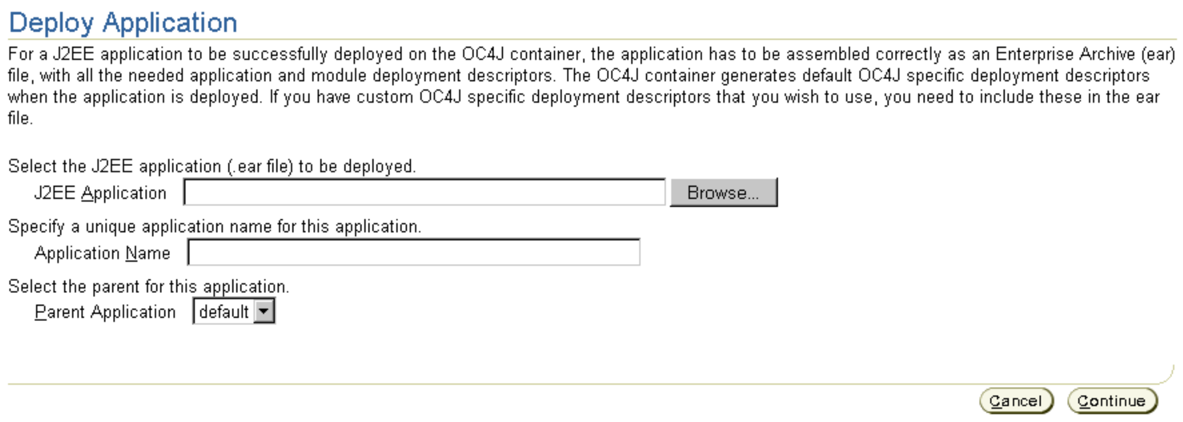
Default Application Name default
Default Application Path **application.xml**

Select	Name	Path	Parent Application	Active Requests	Request Processing Time (seconds)	Active EJB Methods
●	FAQAPP_SB	../applications/FAQAPP_SB.ear	default	0	0.00	0

Application Server Control Console Deploy Application (EAR) Page

Figure 7–3 shows the key portion of the Application Server Control Console Deploy Application Page, which is the page for deploying an EAR file. Drill down to this page from the Applications Page of an OC4J instance by clicking the **Deploy EAR file** button.

Figure 7–3 Application Server Control Console Deploy Application Page



Deploy Application

For a J2EE application to be successfully deployed on the OC4J container, the application has to be assembled correctly as an Enterprise Archive (ear) file, with all the needed application and module deployment descriptors. The OC4J container generates default OC4J specific deployment descriptors when the application is deployed. If you have custom OC4J specific deployment descriptors that you wish to use, you need to include these in the ear file.

Select the J2EE application (.ear file) to be deployed.

J2EE Application

Specify a unique application name for this application.

Application Name

Select the parent for this application.

Parent Application: default

In the Deploy Application Page, click the **Browse** button to select an EAR file to deploy, then specify the desired J2EE application name, which is typically the same as the EAR file name without the `.ear` extension. You can also specify a parent application, but it is typical to use the OC4J default application as the parent.

Deployment results in a new `<application>` element being entered in the `server.xml` file.

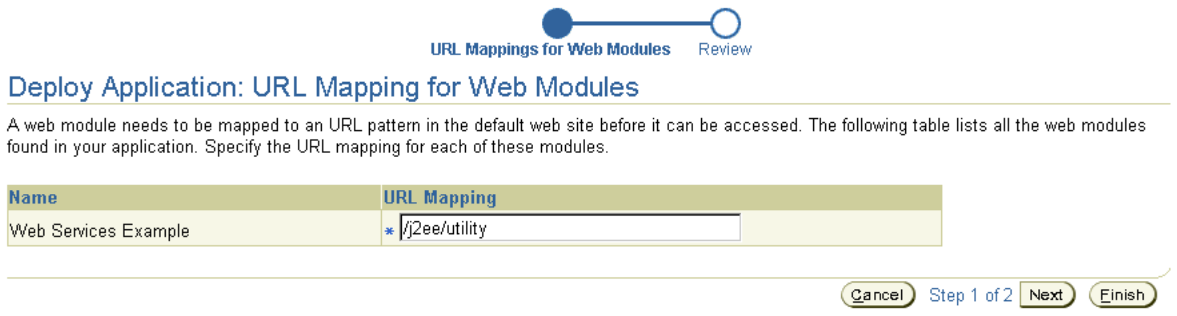
When you click the **Continue** button, the Deploy Application: URL Mapping for Web Modules Page appears. This page enables you to set a URL context path for the Web application that the J2EE application contains. Figure 7–4 shows this page, with the default context path for the Web application of a J2EE application named `utility`. Clicking the **Next** button enables you to review your entries and then deploy.

Specifying a URL context path results in an entry in the `default-web-site.xml` file to bind the Web application to the Web site. This is accomplished through a new `<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.

Note: In specifying the context path, the following forms are treated equivalently:

```
someUrl
/someUrl
/someUrl/
```

Figure 7–4 Application Server Control Console Deploy Application: URL Mapping Page



Deploy Application: URL Mapping for Web Modules

A web module needs to be mapped to an URL pattern in the default web site before it can be accessed. The following table lists all the web modules found in your application. Specify the URL mapping for each of these modules.

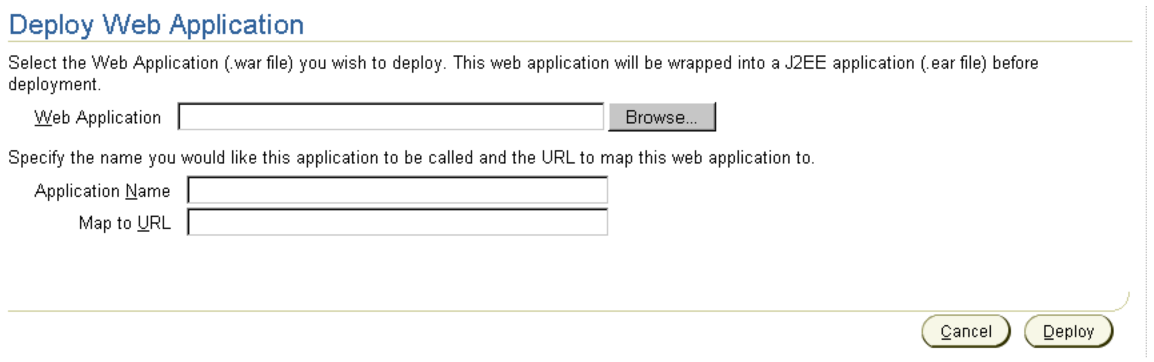
Name	URL Mapping
Web Services Example	* /j2ee/utility

Application Server Control Console Deploy Web Application (WAR) Page

Figure 7–5 contains the key portion of the Application Server Control Console Deploy Web Application Page, which is the page for deploying an independent WAR file. Drill down to this page from the Applications Page of an OC4J instance by clicking the **Deploy WAR file** button.

Note: When you deploy an independent WAR file, it is wrapped in an EAR file transparently.

Figure 7–5 Application Server Control Console 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 a desired J2EE application name along with a URL context path to map to the Web application. Transparently, a J2EE application by the specified application name is created to contain the Web application. In OC4J, any Web application must be contained in a J2EE application.

As with an EAR file, the deployment results in a new `<application>` element in the `server.xml` file. Additionally, to bind the Web application to the Web site, a new `<web-app>` subelement is added to the `<web-site>` element in the `default-web-site.xml` file. Finally, the `mod_oc4j.conf` configuration file for the Oracle HTTP Server `mod_oc4j` Apache mod is updated with appropriate mount points.

Note: In specifying the context path, the following forms are treated equivalently:

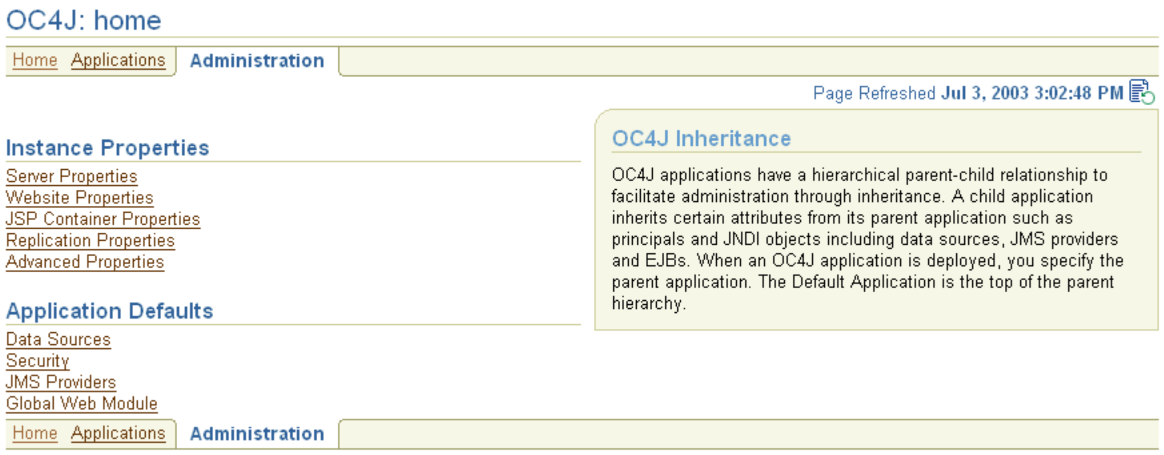
```
someUrl
/someUrl
/someUrl/
```

Application Server Control Console OC4J Administration Page

Figure 7–6 shows the OC4J Administration Page, which enables you to access OC4J instance properties. You can reach this page by clicking **Administration** from the OC4J Home Page.

Clicking **Website Properties** under Instance Properties accesses the Website Properties Page, through which you can access a variety of pages to update Web module properties.

Figure 7–6 Application Server Control Console OC4J Administration Page

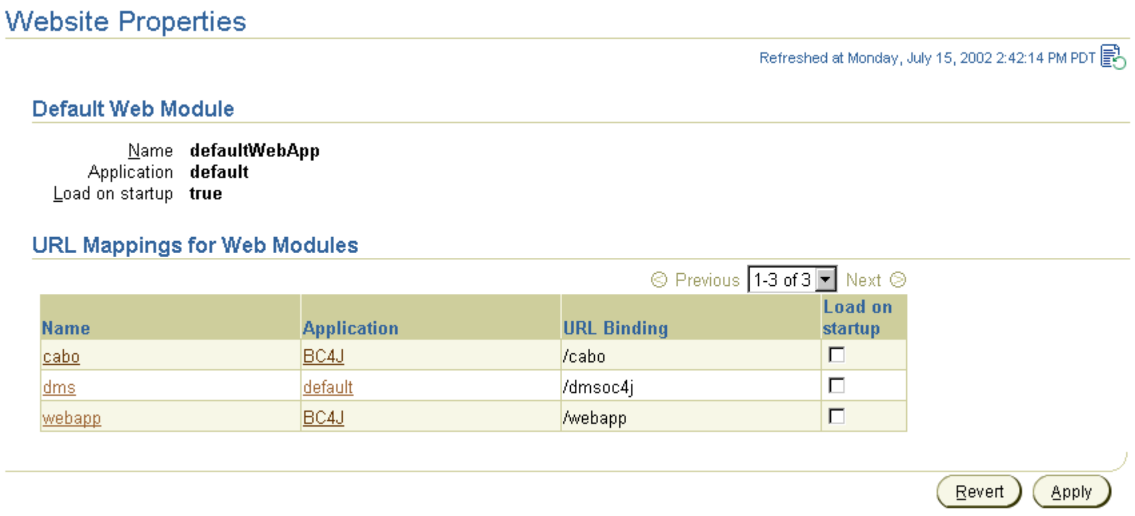


OC4J applications have a hierarchical parent-child relationship to facilitate administration through inheritance. A child application inherits certain attributes from its parent application such as principals and JNDI objects including data sources, JMS providers and EJBs. When an OC4J application is deployed, you specify the parent application. The Default Application is the top of the parent hierarchy.

Application Server Control Console Website Properties Page

Figure 7–7 shows the key portion of the Application Server Control Console Website Properties Page for an OC4J instance. Drill down to this page by clicking **Website Properties** under Instance Properties in the OC4J Administration Page.

Figure 7–7 Application Server Control Console Website Properties Page



Website Properties

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

Default Web Module

Name	**defaultWebApp**
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	☐

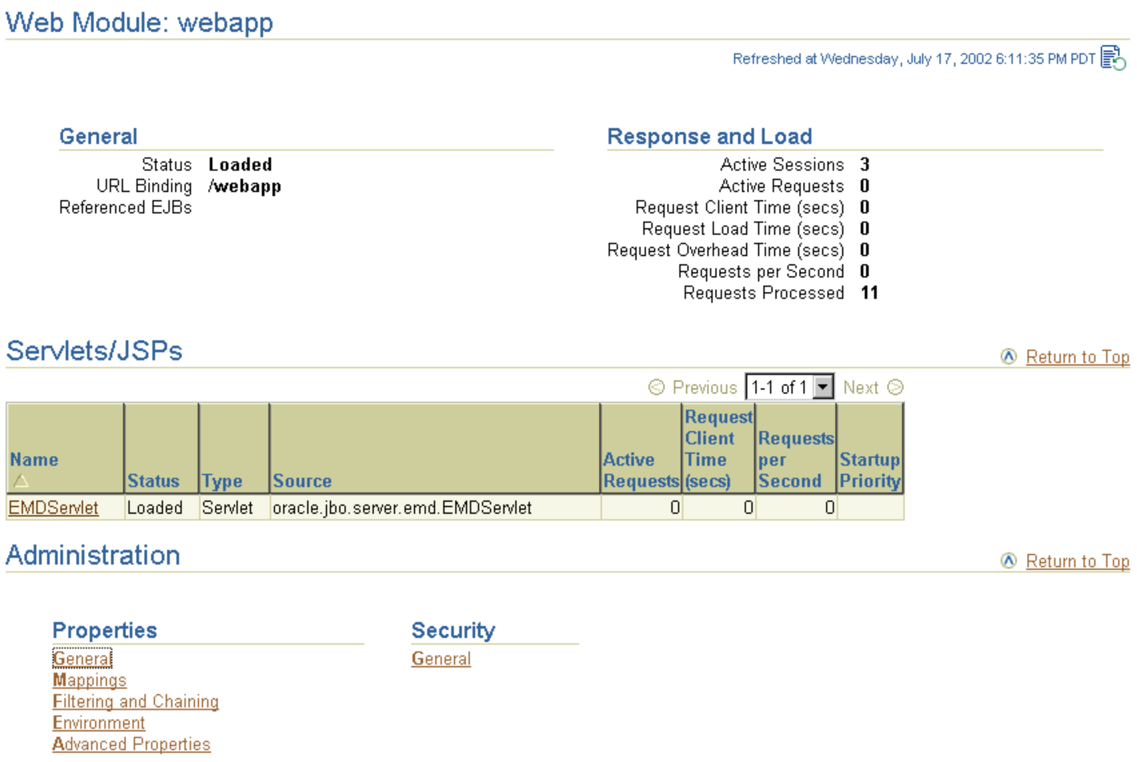
Among other things, this page enables you to specify whether each application should be loaded automatically when OC4J starts. (Otherwise, an application is not loaded until the first request for it.) This corresponds to the `load-on-startup` attribute of the appropriate `<web-app>` subelement of the `<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 2-6.)

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

Application Server Control Console Web Module Page

Figure 7–8 contains the key portion of the Application Server Control Console Web Module Page for the module `webapp`. Drill down to the Web Module Page for a particular module by clicking the module name in the Website Properties Page.

Figure 7–8 Application Server Control Console 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

Return to Top

Previous 1-1 of 1 Next

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

Return to Top

Properties

General
Mappings
Filtering and Chaining
Environment
Advanced Properties

Security

General

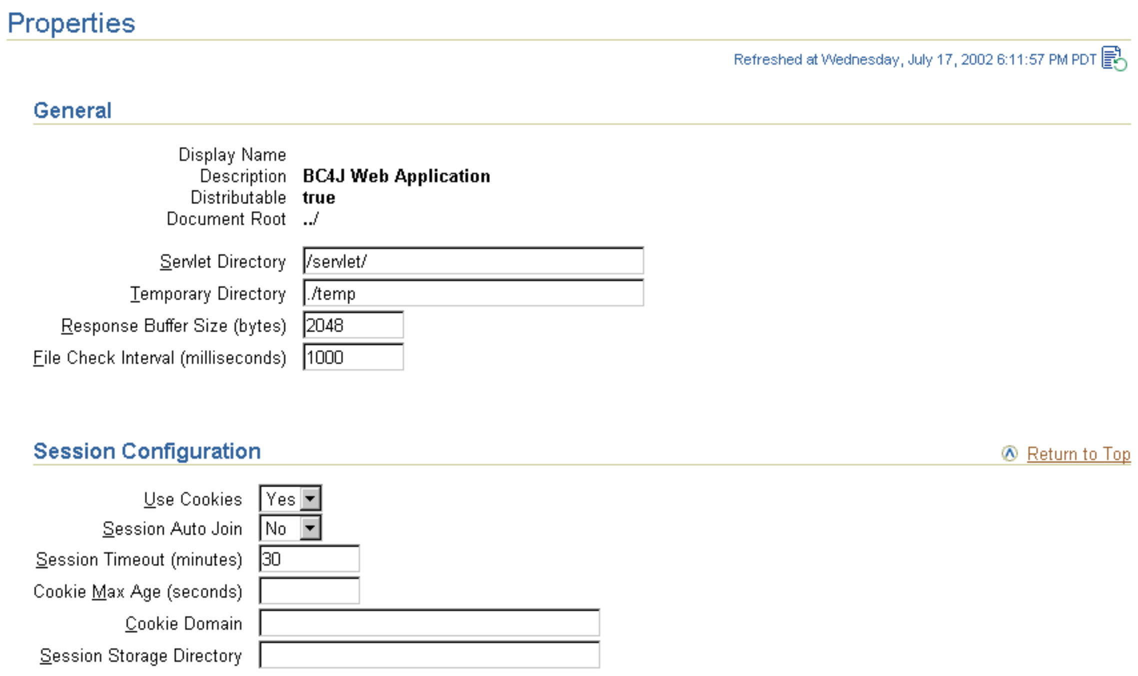
From the Web Module Page, you can access several categories of Web module properties through the following links, under Properties in the Administration section of the page:

- **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

Application Server Control Console Web Module Properties Page

Figure 7–9 and Figure 7–10 show portions of the Application Server Control Console Web Module Properties Page for a particular module. Drill down to this page by clicking **General** under Properties in the Administration section of the Web Module Page.

Figure 7–9 Application Server Control Console Web Module Properties Page (1 of 2)



Properties

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

General

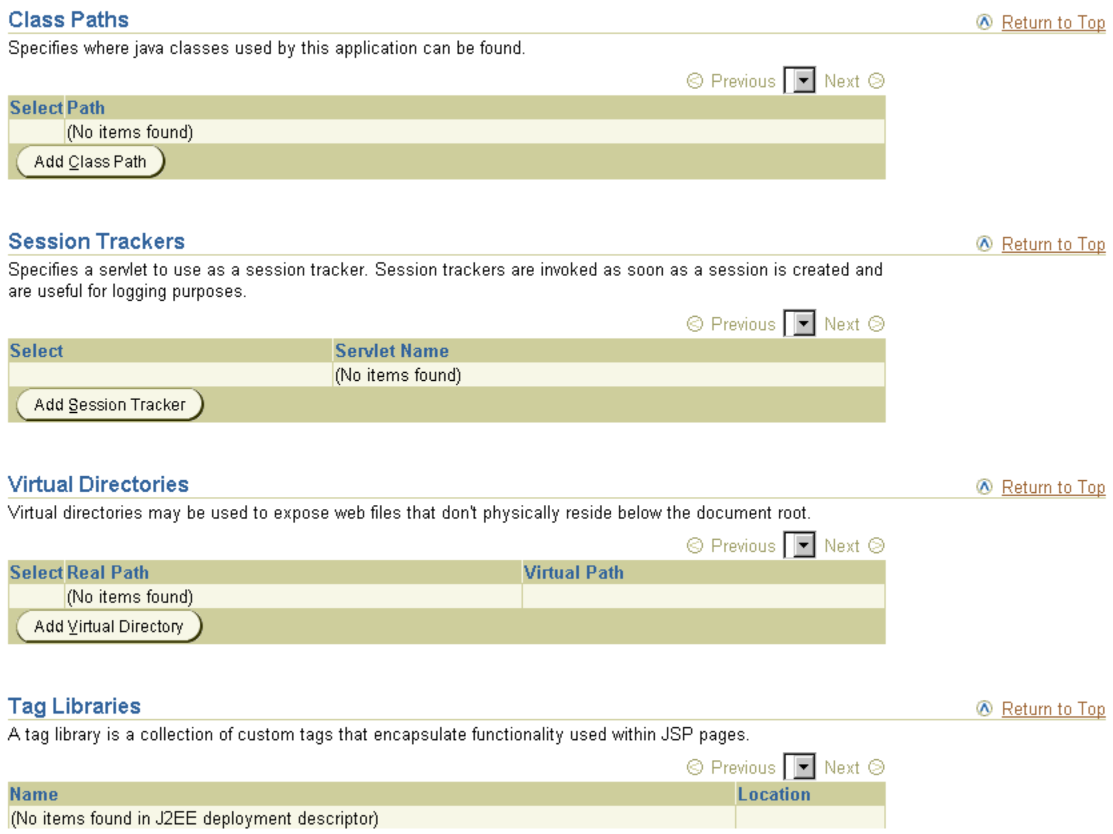
Display Name	
Description	**BC4J Web Application**
Distributable	**true**
Document Root	../
Servlet Directory	/servlet/
Temporary Directory	./temp
Response Buffer Size (bytes)	2048
File Check Interval (milliseconds)	1000

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 7–10 Application Server Control Console 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)	

These settings corrrespond to `orion-web.xml` elements 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.

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.

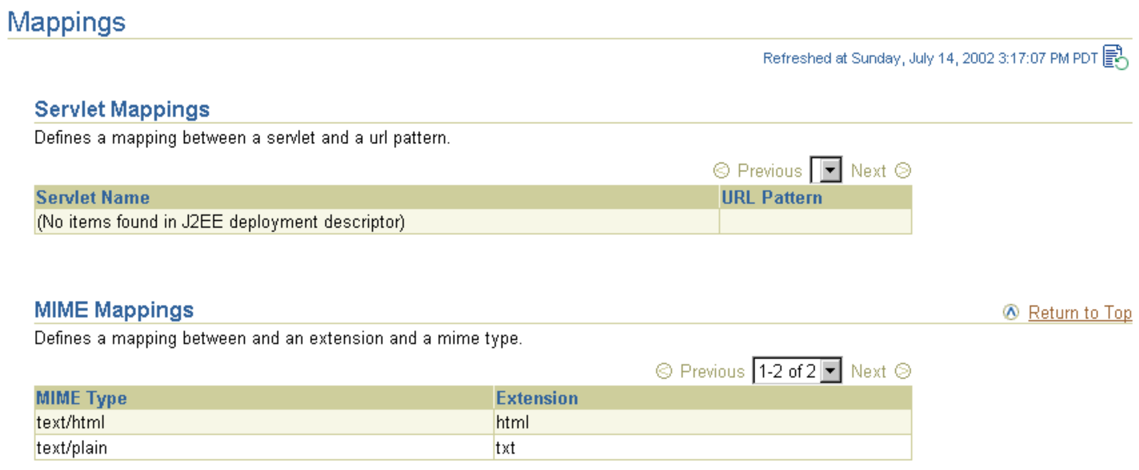
In the Tag Libraries section:

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

Application Server Control Console Web Module Mappings Page

Figure 7–11 and Figure 7–12 contain portions of the Application Server Control Console Web Module Mappings Page for a particular module. Drill down to this page by clicking **Mappings** under Properties in the Administration section of the Web Module Page.

Figure 7–11 Application Server Control Console 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.

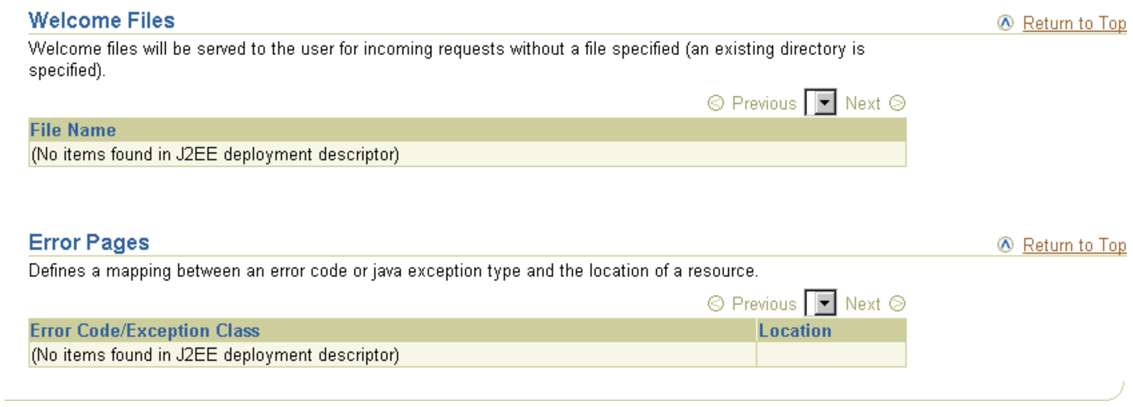
Servlet Name	URL Pattern
(No items found in J2EE deployment descriptor)	

MIME Mappings

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

MIME Type	Extension
text/html	html
text/plain	txt

Figure 7–12 Application Server Control Console Web Module Mappings Page (2 of 2)



Welcome Files

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

File Name
(No items found in J2EE deployment descriptor)

Error Pages

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

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

The following 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 "Application Server Control Console Web Module Advanced Properties Page" on page 7-14.

In the Servlet Mappings section:

- A servlet name or URL pattern specified here corresponds to the `<servlet-name>` or `<url-pattern>` subelement 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.

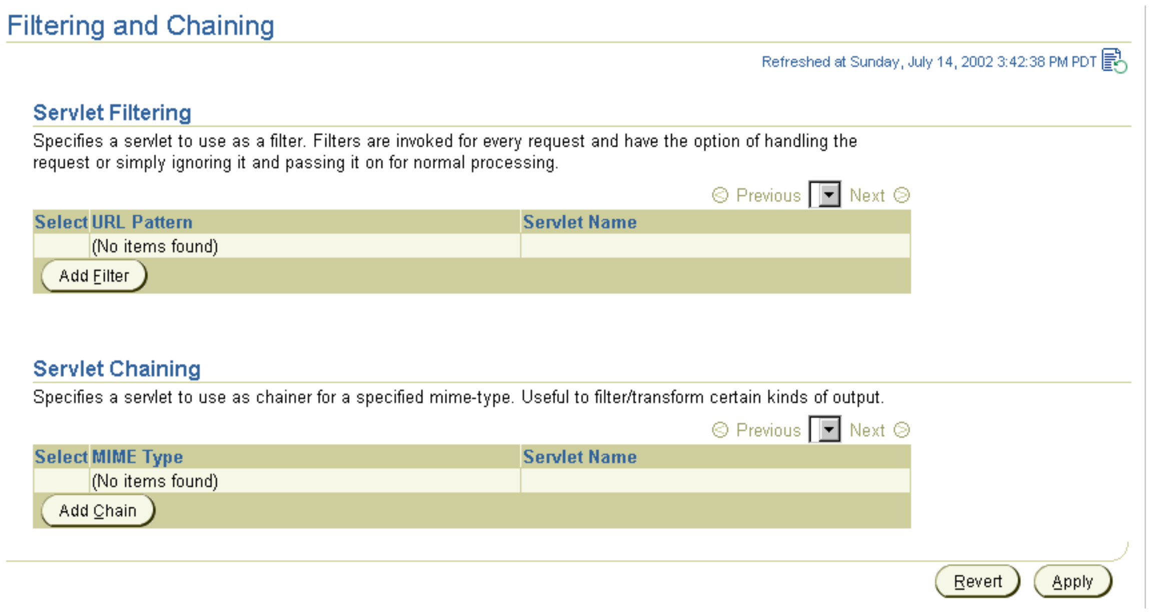
In the Error Pages section:

- 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.

Application Server Control Console Web Module Filtering and Chaining Page

Figure 7–13 displays the key portion of the Application Server Control Console Web Module Filtering and Chaining Page for a particular module. Drill down to this page by clicking **Filtering and Chaining** under Properties in the Administration section of the Web Module Page.

Figure 7–13 Application Server Control Console 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)	

These settings correspond to `orion-web.xml` elements as follows.

In the Servlet Filtering section:

- Adding a filter here is equivalent to setting the `<servlet-name>` or `<url-pattern>` subelement of a `<filter-mapping>` subelement under the `<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:

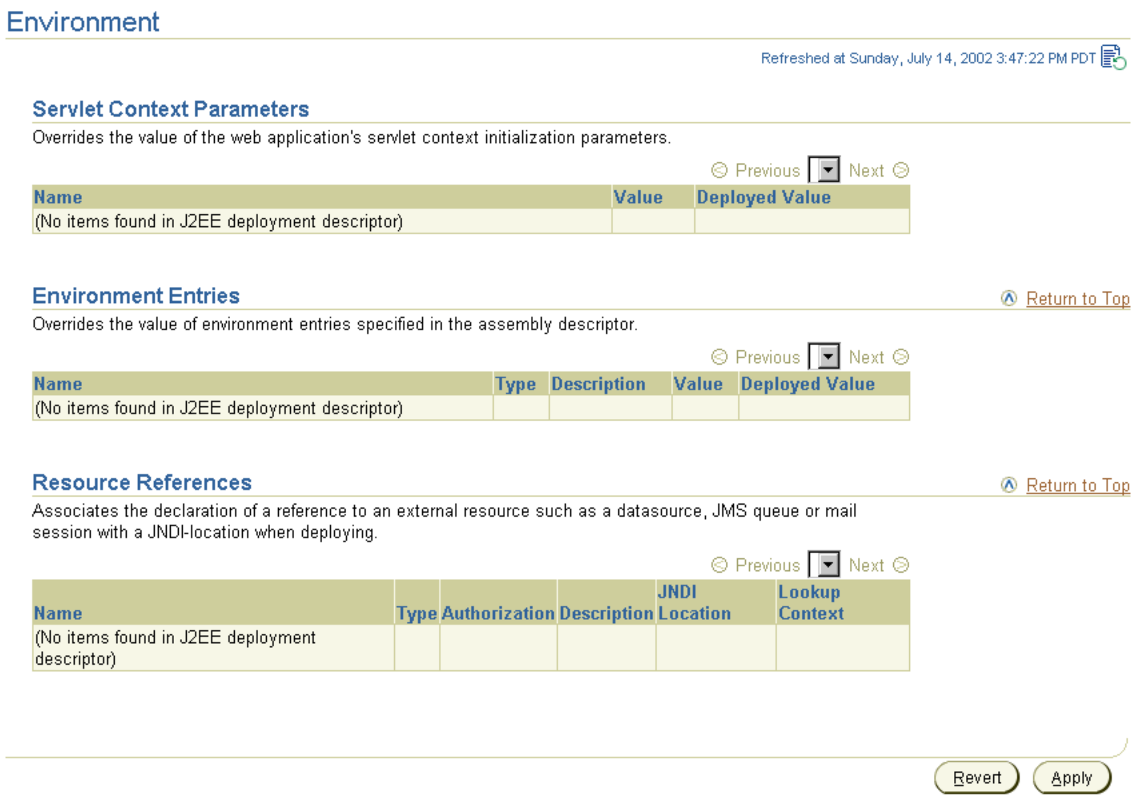
- Adding a chain here is equivalent to setting the `servlet-name` and `mime-type` attributes of a `<servlet-chaining>` subelement of the `<orion-web-app>` element. The servlet name you specify is tied to a servlet class through its standard configuration in the `web.xml` file.

> **Note:** Servlet chaining is an older mechanism with essentially the same functionality as standard servlet filtering, which was introduced in version 2.3 of the servlet specification. It is advisable to use servlet filtering instead. See "Servlet Filters" on page 3-1.

Application Server Control Console Web Module Environment Page

Figure 7–14 shows the key portion of the Application Server Control Console Web Module Environment Page for a particular module. Drill down to this page by clicking **Environment** under Properties in the Administration section of the Web Module Page.

Figure 7–14 Application Server Control Console 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 includes 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.

These settings correspond to `web.xml` and `orion-web.xml` elements 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:

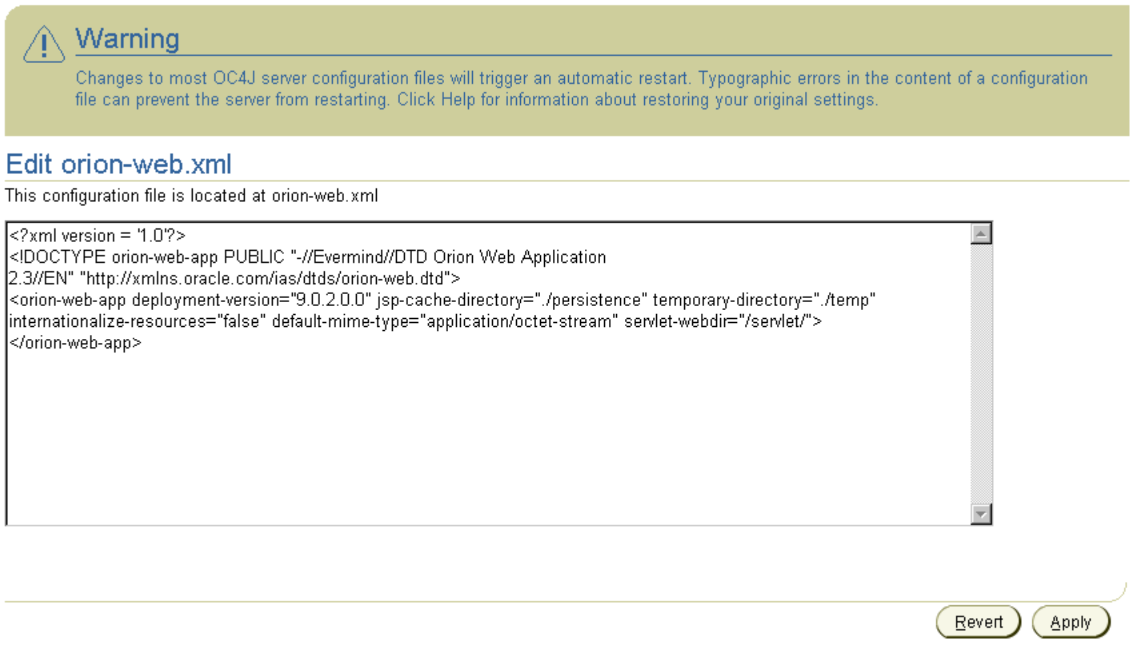
- This section displays a combination of `web.xml` and `orion-web.xml` settings. The name and type of a resource reference correspond to `<res-ref-name>` and `<res-type>` subelements under a `<resource-ref>` subelement of the `<web-app>` element in the `web.xml` file. The JNDI location and lookup context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

Application Server Control Console Web Module Advanced Properties Page

Figure 7–15 shows the key portion of the Application Server Control Console Web Module Advanced Properties Page for a particular module. Drill down to this page by clicking **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 Application Server Control Console Web module pages. In fact, you can make any `orion-web.xml` or `global-web-application.xml` entries through the Advanced Properties Page; however, use the previously described pages whenever possible because of their error handling and reporting features.

Figure 7–15 Application Server Control Console 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>
```

A Open Source Frameworks and Utilities

There are common open source frameworks and utilities that you can use with OC4J in Oracle Application Server 10*g* Release 2 (10.1.2). This appendix describes how to configure and use two of them in particular: Jakarta Struts 1.0.2 and Jakarta log4j 1.2.8.

The focus of this discussion is to assist you in configuring and using these open source utilities in the OC4J standalone environment. The following sections cover the details:

- Configuration and Use of Jakarta Struts in OC4J
- Configuration and Use of Jakarta log4j in OC4J

Important:

- The packaging and configuration instructions in this document are written for an OC4J standalone installation. In an Oracle Application Server installation, use the management tools provided, such as Enterprise Manager and the `dcmctl` command line utility, to accomplish the same tasks. Avoid manual modifications to configuration files in an Oracle Application Server environment.
- The open source utilities and frameworks discussed here are not supported directly by Oracle. In addition, there has been no formal testing or certification of these utilities and frameworks with the OC4J product. For assistance in employing these frameworks, use the regular forums available in the open source community.

Configuration and Use of Jakarta Struts in OC4J

The following sections cover steps for using Jakarta Struts in an OC4J standalone environment:

- Overview of Jakarta Struts
- Downloading the Struts Binary Distribution
- Unpacking the Struts Binary Distribution
- Installing and Accessing Struts Documentation
- Installing the Struts Sample Web Application
- Deploying Your Own Application with the Struts Framework

Note: Oracle JDeveloper includes a wizard that simplifies Struts usage.

Overview of Jakarta Struts

Jakarta Struts is an open source framework to assist with the development of Web applications using open standards such as Java servlets, JavaServer Pages, and XML. Struts supports a modular application development model based on the Model-View-Controller (MVC) pattern. With Struts, you can create an extensible development environment for your application, based on industry standards and proven design models.

The sections that follow describe how to install the Struts libraries, documentation, and sample applications in an OC4J standalone environment. This document does not cover how to build applications with Struts. See the user guide, installation guide, and other documentation on the official Struts Web site, including the following locations:

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

`http://jakarta.apache.org/struts/learning.html`

Note: Struts is part of the Apache Jakarta Project, sponsored by the Apache Software Foundation.

Downloading the Struts Binary Distribution

The Struts 1.0.2 distribution is available at the following location:

`http://jakarta.apache.org/builds/jakarta-struts/release/v1.0.2/`

Download the archive file from this location, choosing the appropriate format (ZIP file or compressed TAR file) for your platform, and save it to your local file system.

Note: You can deploy the Struts 1.1 Beta releases to OC4J using the same steps as in deploying the 1.0.2 release. Due to the increased functionality in Struts 1.1, additional library files are supplied with the distribution. The instructions given here generally apply to deploying an application using Struts 1.1, aside from the additional library files and tag libraries in Struts 1.1.

Unpacking the Struts Binary Distribution

Use the appropriate tool for your platform, such as WinZip or TAR, to unpack the archive file of the Struts 1.0.2 binary distribution that you downloaded. This creates the following directory structure:

```
jakarta-struts-1.0.2/INSTALL
jakarta-struts-1.0.2/LICENSE
jakarta-struts-1.0.2/README

jakarta-struts-1.0.2/lib/jdbc2_0-stdext.jar
jakarta-struts-1.0.2/lib/struts.jar
jakarta-struts-1.0.2/lib/struts.tld
jakarta-struts-1.0.2/lib/struts-bean.tld
jakarta-struts-1.0.2/lib/struts-config_1_0.dtd
```

```
jakarta-struts-1.0.2/lib/struts-form.tld
jakarta-struts-1.0.2/lib/struts-html.tld
jakarta-struts-1.0.2/lib/struts-logic.tld
jakarta-struts-1.0.2/lib/struts-template.tld
jakarta-struts-1.0.2/lib/web-app_2_2.dtd
jakarta-struts-1.0.2/lib/web-app_2_3.dtd

jakarta-struts-1.0.2/webapps/struts-blank.war
jakarta-struts-1.0.2/webapps/struts-documentation.war
jakarta-struts-1.0.2/webapps/struts-example.war
jakarta-struts-1.0.2/webapps/struts-exercise-taglib.war
jakarta-struts-1.0.2/webapps/struts-template.war
jakarta-struts-1.0.2/webapps/struts-upload.war
```

Installing and Accessing Struts Documentation

The Struts documentation is supplied as a Web application in a WAR file in the `webapps` directory of the Struts archive. Use the following steps to deploy the Struts documentation Web application to the OC4J default application.

Configuration files are in the `j2ee/home/config` directory.

1. In the OC4J global application descriptor, `application.xml`, add a new `<web-module>` element for the `struts-documentation.war` file. Place this element after any `<web-module>` elements already in the file.

   Specify the path to the directory in which the Struts binary distribution was extracted. Here is a sample entry:

   ```
   <orion-application ... >
      ...
      <web-module id="struts-documentation"
      path="your_path/jakarta-struts-1.0.2/webapps/struts-documentation.war" />
   ...
   </orion-application>
   ```

2. In the Web site XML file, `http-web-site.xml`, add a new `<web-app>` element to bind the documentation Web application to a URL context path. Place this element after any `<web-app>` elements already in the file. Here is a sample entry specifying `/struts/doc` as the URL context path for the Struts documentation:

   ```
   <web-site ... >
      ...
      <web-app application="default" name="struts-documentation"
               root="/struts/doc" />
      ...
   </web-site>
   ```

   Note the `application="default"` setting to use the OC4J default application. Any Web application deployed to OC4J must be contained in a J2EE application. Typically, this is accomplished by packaging the Web application WAR file inside a J2EE application EAR file. For convenience, however, you can use an OC4J default application in deploying a standalone WAR file, as in this case.

3. Start OC4J from the command line:

   ```
   % java -jar oc4j.jar
   ```

   You will see output similar to the following:

   ```
   Auto-unpacking /java/jakarta-struts-1.0.2/webapps/struts-documentation.war
   ... done.
   ```

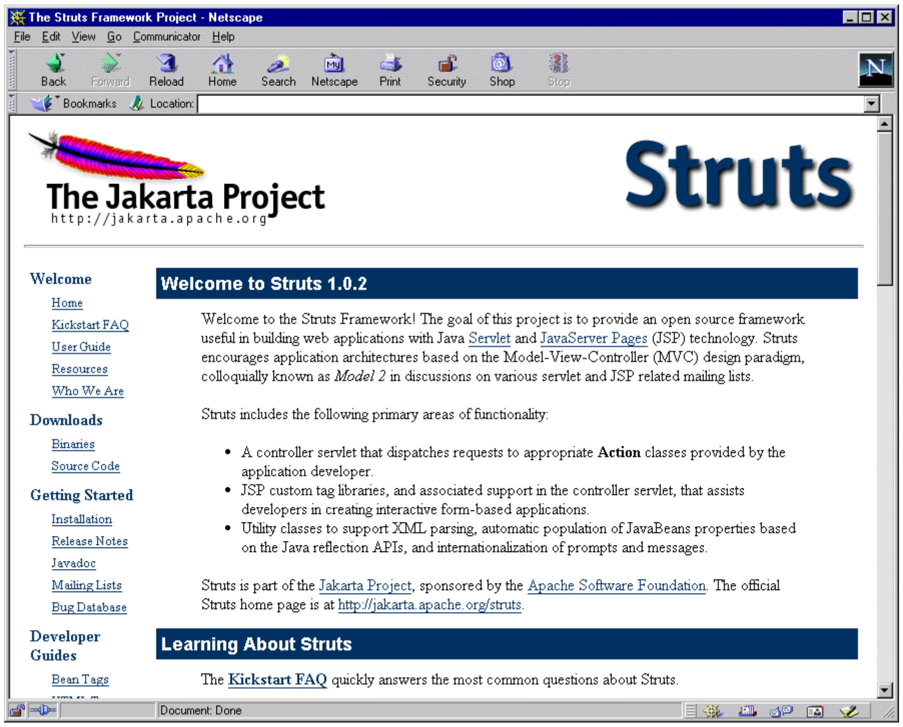
```
Oracle Application Server (10.1.2.0.0) Containers for J2EE initialized
```

Unpacking `struts-documentation.war` results in the creation and population of the `struts-documentation` directory and subdirectories under the `jakarta-struts-1.0.2/webapps` directory.

4. Access the documentation according to the URL context path you specified in `http-web-site.xml`:

   ```
   http://host:8888/struts/doc
   ```

   The Struts documentation welcome page appears, as in the following graphic:


The Struts Framework Project - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:
The Jakarta Project
http://jakarta.apache.org
Struts
Document: Done

Welcome
Home
Kickstart FAQ
User Guide
Resources
Who We Are
Downloads
Binaries
Source Code
Getting Started
Installation
Release Notes
Javadoc
Mailing Lists
Bug Database
Developer Guides
Bean Tags

Welcome to Struts 1.0.2

Welcome to the Struts Framework! The goal of this project is to provide an open source framework useful in building web applications with Java Servlet and JavaServer Pages (JSP) technology. Struts encourages application architectures based on the Model-View-Controller (MVC) design paradigm, colloquially known as *Model 2* in discussions on various servlet and JSP related mailing lists.

Struts includes the following primary areas of functionality:

- A controller servlet that dispatches requests to appropriate **Action** classes provided by the application developer.
- JSP custom tag libraries, and associated support in the controller servlet, that assists developers in creating interactive form-based applications.
- Utility classes to support XML parsing, automatic population of JavaBeans properties based on the Java reflection APIs, and internationalization of prompts and messages.

Struts is part of the Jakarta Project, sponsored by the Apache Software Foundation. The official Struts home page is at http://jakarta.apache.org/struts.

Learning About Struts

The **Kickstart FAQ** quickly answers the most common questions about Struts.

Installing the Struts Sample Web Application

The Struts binary distribution also provides a sample Web application in a WAR file in the `webapps` directory. As with the documentation Web application, you can deploy the Struts sample Web application to the OC4J default application. Use the following steps. Configuration files are in the `j2ee/home/config` directory.

1. In the OC4J global application descriptor, `application.xml`, add a new `<web-module>` element for the `struts-example.war` file. Specify the path to the directory in which the Struts binary distribution was extracted. Here is a sample entry:

   ```
   <web-module id="struts-example"
      path="your_path/jakarta-struts-1.0.2/webapps/struts-example.war" />
   ```

   Place this immediately after the `<web-module>` element you created for `struts-documentation.war`.

2. In the Web site XML file, `http-web-site.xml`, add a new `<web-app>` element to bind the sample Web application to a URL context path. Here is a sample entry

that specifies `/struts/example` as the URL context path for the Struts documentation:

```
<web-app application="default" name="struts-example"
         root="/struts/example" />
```

Place this immediately after the `<web-app>` element you created for the documentation Web application.

As with the documentation Web application, the `application="default"` setting uses the OC4J default application to contain the sample Web application.

3. Start OC4J from the command line:

```
% java -jar oc4j.jar
```

You will see output similar to the following:

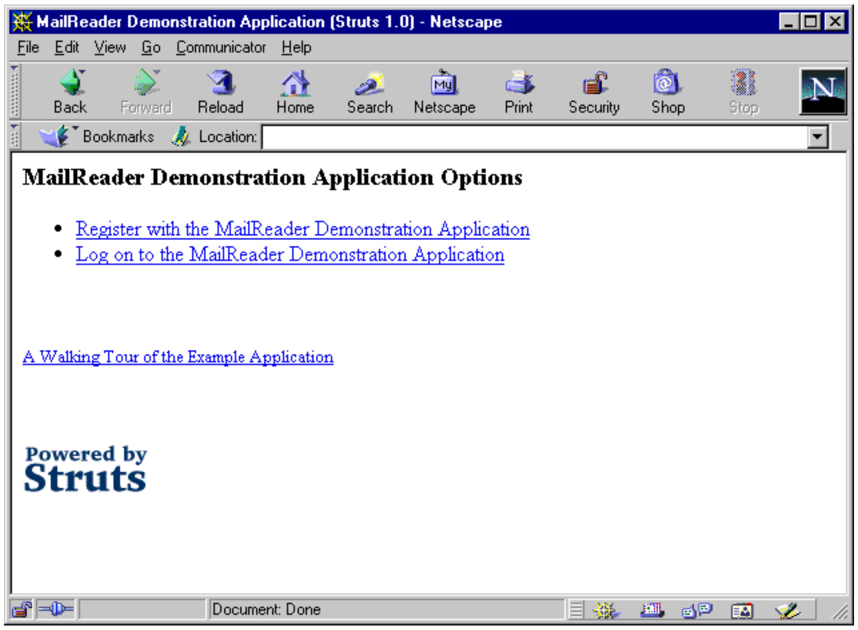
```
Auto-unpacking /java/jakarta-struts-1.0.2/webapps/struts-example.war
...done.
Oracle Application Server (10.1.2.0.0) Containers for J2EE initialized
```

Unpacking `struts-example.war` results in the creation and population of the `struts-example` directory and subdirectories under the `jakarta-struts-1.0.2/webapps` directory.

4. Access the sample Web application according to the URL context path you specified in `http-web-site.xml`:

```
http://host:8888/struts/example
```

The Struts sample application welcome page appears, as in the following graphic:


MailReader Demonstration Application (Struts 1.0) - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:

MailReader Demonstration Application Options

- Register with the MailReader Demonstration Application
- Log on to the MailReader Demonstration Application

A Walking Tour of the Example Application

Powered by Struts

Document: Done

Deploying Your Own Application with the Struts Framework

When deploying your own applications using the Struts framework, you must package the Struts library artifacts within your own WAR file and configure the standard `web.xml` deployment descriptor with the required entries for the Struts components. Your Web application will be constructed and packaged as a WAR file.

Note: A good example of a WAR file configured to use Struts is provided in the `webapps` folder of the Struts archive file as `struts-blank.war`. This example serves as a useful template when constructing your own Web applications.

---

1. Copy the Struts library from the Struts `lib` directory to the `/WEB-INF/lib` directory of your application. The following example is for a UNIX environment (from the directory in which you unpacked the archive file), where "`%`" is the system prompt:

   ```
   % cp jakarta-struts-1.0.2/lib/struts.jar web-inf/lib
   ```

2. Copy the Struts tag library descriptor files (all `.tld` files, for JSP tag libraries) from the Struts `lib` directory to your `/WEB-INF` directory:

   ```
   % cp jakarta-struts-1.0.2/lib/*.tld web-inf
   ```

Note: These steps, using a JSP 1.1 methodology, describe only one way to access JSP tag library descriptor files. Other options are available in a JSP 1.2 environment such as in OC4J. See the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information.

---

3. Add Struts servlet and tag library entries to the `web.xml` file.

   a. Add the servlet definition element for the Struts controller. (You can optionally specify an application-wide `MessageResource` file to use, the name and location of the Struts configuration file, and additional properties such as debugging levels.) The `<servlet>` element is a subelement of the top-level `<web-app>` element.

   ```
   <servlet>
     <servlet-name>action</servlet-name>
     <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
     <init-param>
       <param-name>application</param-name>
       <param-value>ApplicationResources</param-value>
     </init-param>
     <init-param>
       <param-name>config</param-name>
       <param-value>/WEB-INF/struts-config.xml</param-value>
     </init-param>
   </servlet>
   ```

   b. Add a servlet mapping element for the Struts controller servlet. This step maps the servlet name (mapped to the servlet class in the `<servlet>` element above) to a URL servlet path. The `<servlet-mapping>` element is a subelement of the top-level `<web-app>` element.

   ```
   <servlet-mapping>
     <servlet-name>action</servlet-name>
     <url-pattern>*.do</url-pattern>
   </servlet-mapping>
   ```

c. Add entries for the Struts tag libraries. These entries assume the TLD files were placed in the `/WEB-INF` directory as shown in Step 2. The `<taglib>` element is a subelement of the top-level `<web-app>` element.

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

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

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

You now have a Web application that can support the deployment of applications that utilize the Struts framework.

After the remainder of the Web application—including JSP pages, servlets, Struts components, and other files—has been added to the WAR file, the application can be deployed to OC4J.

Note: Additional steps are required to use Struts within your applications. You must also create the Action classes and other components used by Struts at runtime and make corresponding entries in the Struts configuration file, `struts-config.xml`. These points are not OC4J-specific and are beyond the scope of this document. See the learning guide on the Struts Web site for more information:

`http://jakarta.apache.org/struts/learning.html`

Configuration and Use of Jakarta log4j in OC4J

The following sections cover considerations for using Jakarta log4j in an OC4J standalone environment:

- Overview of Jakarta log4j
- Downloading the log4j Binary Distribution
- Unpacking the log4j Binary Distribution
- Installing the log4j Library
- Using log4j Configuration Files
- Enabling log4j Debug Mode

Overview of Jakarta log4j

The log4j framework is an open source project that provides an efficient and flexible API to support runtime logging operations for Java applications. It enables developers to insert log statements into their code, incorporating messages at different levels of alarm as desired. Log4j also enables system administrators to separately define the level of logging they wish to see from the application at runtime, without requiring changes to the supplied application code.

Features of log4j allow you to enable logging at runtime without having to modify the application binary file. Statements can remain in shipped code without incurring significant performance cost. Logging is controlled through a configuration file without requiring changes to the application binary.

The sections that follow describe how to install the log4j library and configure it for use with OC4J. Use of the extensive log4j API is not OC4J-specific, so is not covered in this document. See the documentation on the official log4j Web site, including the following locations:

http://jakarta.apache.org/log4j/docs/index.html

http://jakarta.apache.org/log4j/docs/documentation.html

Note: The log4j framework is part of the Apache Jakarta Project, sponsored by the Apache Software Foundation.

Downloading the log4j Binary Distribution

The log4j distribution is available at the following location:

http://jakarta.apache.org/log4j/docs/download.html

Download the archive file from this location, choosing the appropriate format (ZIP file or compressed TAR file) for your platform, and save it to your local file system.

Unpacking the log4j Binary Distribution

Use the appropriate tool for your platform, such as WinZip or TAR, to unpack the log4j archive file that you downloaded. This creates and populates the following directory structure:

```
jakarta-log4j-1.2.8/
   build/
   contrib/
      ...
   dist/
      classes/
         ...
      lib/
   docs/
      ...
   examples/
      ...
   src/
      ...
```

(Some of the directory structure is not shown; there are many further subdirectories.)

Installing the log4j Library

To enable J2EE applications to use log4j functionality, the log4j library must be made available by the classloaders of OC4J. You can accomplish this in several ways, depending on your specific operational requirements. For example, you can install the log4j library at a system or global application level, making it available to all applications deployed to the container. Alternatively, you can package the log4j library as a library of a specific application (or applications). Different approaches have different operating characteristics, such as the way in which the automatic loading of configuration files works. For more details about possible approaches and their advantages and disadvantages, refer to the log4j Web site and user mailing lists.

The following sections cover three techniques to make log4j available to OC4J:

- Use the log4j Library at a Global Application Level
- Package the log4j Library as a Web Application Library
- Package the log4j Library as a Shared Library for EJB and Web Applications

Use the log4j Library at a Global Application Level

To install the log4j library at a global application level in OC4J, copy the `log4j-1.2.8.jar` file from the log4j `lib` directory to the `j2ee/home/applib` directory. By default, a `<library>` element in the `j2ee/home/config/application.xml` global application descriptor makes this directory available for libraries that are to be shared between all applications deployed to the OC4J instance. At runtime, OC4J automatically loads all libraries in the `applib` directory. The following example is for a UNIX environment (from the directory in which you unpacked the archive file), where "%" is the system prompt:

```
% cp jakarta-log4j-1.2.8/dist/lib/log4j-1.2.8.jar j2ee/home/applib
```

Notes:

- Be aware of the overhead in using this approach. If you do not want the log4j library to always be loaded, do not use the `applib` directory.
- Do not use the `applib` directory for log4j in an Oracle Application Server environment. Oracle Enterprise Manager 10*g* also uses log4j, and placing your copy at the global application level may cause version conflicts for Enterprise Manager.

Package the log4j Library as a Web Application Library

To package the log4j library for a specific Web application, copy the `log4j-1.2.8.jar` file from the log4j `lib` directory into the `/WEB-INF/lib` directory of your Web application. At runtime, the servlet container makes the log4j library available to the Web application through a Web application classloader. The following example is for a UNIX environment (from the directory in which you unpacked the archive file), where "%" is the system prompt:

```
% cp jakarta-log4j-1.2.8/dist/lib/log4j-1.2.8.jar web-inf/lib
```

Package the log4j Library as a Shared Library for EJB and Web Applications

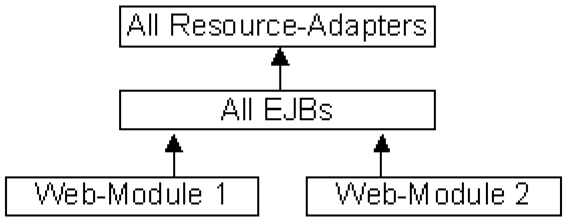
When you have an application that comprises EJB components and Web components that all use log4j, you can package the log4j library as a single shared library that both sets of components can use.

The J2EE classloading mechanism implies that a Web application deployed within the same EAR file as an EJB application has access to classes available in the EJB classloader. Making log4j a library of the EJB application also makes it a library of the Web application.

The EJB classloader, as well as the Web classloader, can access any libraries specified in the `Class-Path` attribute of the `META-INF/Manifest.mf` file of the EAR file. The library JAR files are loaded relative to the file (such as the EAR file) with the `Class-Path` entry, so they are stored in the same directory as that file. Using this facility, it is possible to place the log4j JAR file in the same directory as the EJB JAR file and reference it in the manifest file as a required library. This also makes the log4j library accessible to the Web applications inside the same EAR file, because they have visibility of the classes of the EJB components.

Figure A–1 illustrates the classloading hierarchy for a J2EE application.

Figure A–1 J2EE Classloading Hierarchy



Using log4j Configuration Files

The log4j framework enables you to control logging behavior through settings specified in an external configuration file, allowing you to make changes to the logging behavior without modifying application code.

There are three common ways to use the external configuration files. Each approach defines what the configuration files are named and how they are located by the J2EE application server at runtime.

The following sections describe the three approaches:

- Use the Default Files for Automatic log4j Configuration
- Use Alternative Files for Automatic log4j Configuration
- Programmatically Specify External Configuration Files

Use the Default Files for Automatic log4j Configuration

By default, log4j uses a configuration file named `log4j.properties` or `log4j.xml` to determine its logging behavior. It automatically attempts to load these files from the classloaders available to it at runtime. If it finds both files, then `log4j.xml` takes precedence.

To use an automatic configuration file, place it in a directory location that falls within the classpath used by OC4J. This includes, in order of loading precedence:

1. Global application level: `j2ee/home/applib`
2. Web application level: `/WEB-INF/classes`

Note: A log4j runtime is configured only once, using the automatic configuration files, when the first call is made to the `org.apache.log4j.Logger` class. If you install the log4j library at the global application level, by placing it in the `j2ee/home/applib` directory, then you can use only one automatic configuration file to define all the log levels, appenders, and other log4j properties for all the applications running on your server. If you install the log4j library separately for each Web application, in each `/WEB-INF/lib` directory, then the log4j logger is initialized separately for each Web application. This enables you to use separate automatic log4j configuration files for each Web application. Visit the following log4j Web site and see the log4j user mailing list for more information:

http://www.mail-archive.com/log4j-user@jakarta.apache.org/

Use Alternative Files for Automatic log4j Configuration

You can choose alternative file names instead of using the default names for automatic configuration of log4j. To do this, specify an additional runtime property when OC4J is started, as follows, where "%" is the system prompt and *`url`* designates the location of the configuration file to use:

```
% java -Dlog4j.configuration=url
```

If the specified value for the `log4j.configuration` property is a fully formed URL, log4j loads the URL directly and uses that as the configuration file.

If the specified value is not a correctly formed URL, log4j uses the specified value as the name of the configuration file to load from the classloaders it has available.

For example, assume OC4J is started as follows (where this is a single wraparound command line):

```
% java -Dlog4j.debug=true -Dlog4j.configuration=file:///d:\temp\foobar.xml
       -jar oc4j.jar
```

In this case, log4j tries to load the file `d:\temp\foobar.xml` as its configuration file.

As another example, assume OC4J is started as follows:

```
% java -Dlog4j.debug=true -Dlog4j.configuration=foobar.xml -jar oc4j.jar
```

In this case, log4j tries to load `foobar.xml` from the classloaders it has available. This works in the same manner as using the default automatic configuration file `log4j.xml`, but using the specified file name instead.

Note: This approach, although offering an additional level of flexibility, does require all external configuration files for all deployed applications to have the same name.

Programmatically Specify External Configuration Files

Instead of relying on the automatic configuration file loading mechanism, some applications use a programmatic approach to load the external configuration file. In this case, the path to the configuration file is supplied directly within the application code. This allows different file names to be used for each application. The log4j utility loads and parses the specified configuration file (either an XML document or a properties file) to determine required logging behavior.

Here is an example:

```
public void init(ServletContext context) throws ServletException
{
  // Load the barfoo.xml file as the log4j external configuration file.
  DOMConfigurator("barfoo.xml");
  logger = Logger.getLogger(Log4JExample.class);
}
```

In this case, log4j tries to load `barfoo.xml` from the same directory from which OC4J was started.

Using the programmatic approach provides the most flexibility to developers and system administrators. A configuration file can be of any arbitrary name and be loaded from any location. System administrators can still make changes to the logging behavior without requiring application code changes through the external configuration file.

To provide even further flexibility, and to avoid coding a specific name and location into an application, a useful technique is to supply the file name and location as parameters inside the standard `web.xml` deployment descriptor. The servlet or JSP page reads the values of the parameters specifying the location and name of the configuration file, and dynamically constructs the location from which to load the configuration file. This process allows system administrators to choose both the name and location of the configuration file to use.

Here is a sample `web.xml` entry specifying the name and location of the configuration file:

```
<context-param>
  <param-name>log4j-config-file</param-name>
  <param-value>/web-inf/classes/app2-log4j-config.xml</param-value>
</context-param>
```

The application reads the location value from the deployment descriptor, constructs a full path to the file on the local file system, and loads the file. Following is some sample code:

```
public void init(ServletContext context) throws ServletException
{
  /*
   * Read the path to the config file from the web.xml file,
   * should return something line /web-inf/xxx.xml or web-inf/classes/xxx.xml.
   */
  String configPath = context.getInitParameter("log4j-config-file");

  /*
   * This loads the file based on the base directory of the web application
   * as it is deployed on the application server.
   */
  String realPath = context.getRealPath(configPath);
   if(realPath!=null)
   DOMConfigurator.configure(realPath);
  _logger = Logger.getLogger(Log4JExample.class);
}
```

Note: It is a good practice to place files that define behavior, and that should not be accessible to clients from an HTTP request, directly into the `/WEB-INF` directory of the Web application. (Do not use a subdirectory of `/WEB-INF`.) This applies to `log4j.xml`, for example. The servlet specification requires contents of the `/WEB-INF` directory to be inaccessible to clients.

Enabling log4j Debug Mode

When deploying an application on OC4J that uses log4j and external configuration files, it is sometimes helpful to view how log4j is trying to find and load the requested configuration files. To facilitate this, log4j provides a debug mode that displays how it is loading (or attempting to load) its configuration files.

To turn on log4j debug mode, specify an additional runtime property when you start OC4J, as follows (where "%" is the system prompt):

```
% java -Dlog4j.debug=true -jar oc4j.jar
```

OC4J displays output similar to the following:

```
Oracle Application Server (10.1.2.0.0) Containers for J2EE initialized
log4j: Trying to find [log4j.xml] using context classloader [ClassLoader:
[[D:\myprojects\java\log4j\app1\webapp1\WEB-INF\classes],
[D:\myprojects\java\log4j\app1\webapp1\WEB-INF\lib/log4j-1.2.7.jar]]].
log4j: Using URL [file:/D:/myprojects/java/log4j/app1/webapp1/WEB-INF/classes/
log4j.xml] for automatic log4j configuration.
log4j: Preferred configurator class: org.apache.log4j.xml.DOMConfigurator
log4j: System property is :null
log4j: Standard DocumentBuilderFactory search succeded.
log4j: DocumentBuilderFactory is: oracle.xml.jaxp.JXDocumentBuilderFactory
log4j: URL to log4j.dtd is [classloader:/org/apache/log4j/xml/log4j.dtd].
log4j: debug attribute= "null".
log4j: Ignoring debug attribute.
log4j: Threshold ="null".
log4j: Level value for root is  [debug].
log4j: root level set to DEBUG
log4j: Class name: [org.apache.log4j.FileAppender]
log4j: Setting property [file] to [d:/temp/webapp1.out].
log4j: Setting property [append] to [false].
log4j: Parsing layout of class: "org.apache.log4j.PatternLayout"
log4j: Setting property [conversionPattern] to [%n%-5p %d{DD/MM/yyyy}
d{HH:mm:ss} [%-10c] [%r]%m%n].
log4j: setFile called: d:/temp/webapp1.out, false
log4j: setFile ended
log4j: Adding appender named [FileAppender] to category [root].
```

Note: You can also use the `debug` attribute of the `log4j:configuration` tag in an external configuration file to enable debug output. However, this does not display the loading operations that take place, so does not offer the best service for resolving problems in loading configuration files.

B
Third-Party Licenses

This appendix contains the Third-Party License for third-party products included with Oracle Application Server and discussed in this manual. Topics include:

- Apache HTTP Server

Apache HTTP Server

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

The Apache Software License

Index