Oracle® Containers for J2EE

Servlet Developer's Guide

10*g* (10.1.3.1.0)

B28959-01

October 2006

ORACLE®

Oracle Containers for J2EE Servlet Developer's Guide, 10*g* (10.1.3.1.0)

B28959-01

Primary Author: Alfred Franci

Contributing Author: Bonnie Vaughan, Brian Wright, Tim Smith

Contributors: Dana Singleterry, Olaf Heimburger, James Kirsch, Bryan Atsatt, Ashok Banerjee, Bill Bishop, Olivier Caudron, Cania Chung, Gerald Ingalls, 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, Tug Grall, Mike Lehmann, Steve Button

Contents

Preface xiii

Audience xiii
Documentation Accessibility xiii
Related Documentation xiv
Conventions xvi

1 Summary of What to Know About Servlets

Summary of Servlet and J2EE Technology 1-1
The Essence of Servlets 1-2
Why Use Servlets? 1-2
Servlet Lifecycle 1-3
JSP Pages and Other J2EE Component Types 1-4
Key Components and APIs of the Servlet Model 1-4
Key Methods of the Servlet Interface 1-4
Servlet Communication: Request and Response Objects 1-5
Key Methods of the HttpServletRequest Interface 1-6
Key Methods of the HttpServletResponse Interface 1-7
Servlet Execution in the Servlet Container 1-8
Servlet Configuration Objects 1-10
Obtaining a Servlet Configuration Object 1-10
Key Servlet Configuration Methods 1-10
Servlet Contexts: the Application Container 1-10
Servlet Context Basics 1-10
Obtaining a Servlet Context 1-11
Key Servlet Context Methods 1-11
What are Servlet Sessions (User Sessions) Used For? 1-12
Servlet Thread Models 1-12
Servlet Feature Table 1-13

2 Deploying and Invoking Servlets

Initial Considerations and OC4J Scenarios 2-1
A Brief Overview of OC4J Administration 2-1
OC4J in a Standalone Versus Oracle Application Server Environment 2-2
OC4J and Oracle Application Server Administration Tools 2-3
Summary of URL Components 2-3

Deploying a Web Application to OC4J 2-7
Application Structure 2-7
Summary of General Steps to Deploy a WAR File 2-8
Summary of General Steps to Deploy an EAR File 2-9
Invoking a Servlet in OC4J 2-10
Invoking a Servlet in a Standalone OC4J Environment 2-10
Invoking a Servlet by Class Name During OC4J Development 2-11
Invoking a Servlet in an Oracle Application Server Environment 2-12
Deploying and Invoking the Simple Servlet Example 2-12
Deploy the Servlet Example as a WAR File 2-12
Create the web.xml File 2-12
Create the WAR File 2-13
Deploy the WAR File and Bind the Web Application 2-13
Deploy the Servlet Example as an EAR File 2-14
Create the web.xml File and WAR file 2-14
Create the application.xml File 2-14
Create the EAR File 2-14
Deploy the EAR File and Bind the Contained Web Application 2-15
Invoke the Servlet Example 2-15
Preloading Servlets 2-16

3 Understanding and Using Servlet Sessions

Overview of Session Tracking 3-1
Session Objects 3-1
Session IDs 3-2
Cookies and Persistent Session Data 3-2
When to Use Cookies Versus Session Attributes 3-2
Using Session Tracking in OC4J 3-2
Configuring Session Tracking and Enabling or Disabling Cookies in OC4J 3-3
How OC4J Can Use Cookies for Session Tracking 3-3
Using URL Rewriting for Session Tracking 3-4
Session Tracking Through Secured Connections 3-4
Using a Session Object in Your Servlet 3-5
Summary of HttpSession Methods 3-5
Adding and Retrieving Session Attributes 3-7
Session Object Example 3-7
Using Cookies in Your Servlet 3-10
Configuring Cookies 3-11
Summary of Cookie Methods 3-11
Retrieving, Displaying, and Adding Cookies 3-12
Cookie Example 3-13
Canceling a Session 3-15
Using a Timeout to Cancel a Session 3-15
Explicitly Canceling a Session 3-16

4 Understanding and Using Servlet Filters

Overview of How Filters Work 4-1

How the Servlet Container Invokes Filters 4-1
Typical Filter Actions 4-3
Standard Filter Interfaces 4-3
Methods of the Filter Interface 4-3
Method of the FilterChain Interface 4-4
Methods of the FilterConfig Interface 4-4
Implementing and Configuring Filters 4-5
Implement the Filter Code 4-5
Configure the Filter 4-6
Construction of the Filter Chain 4-7
Simple Filter Example 4-7
Write the Simple Filter Code 4-7
Write the Target JSP Page 4-8
Configure the Simple Filter 4-8
Package the Simple Filter Example 4-8
Invoke the Simple Filter Example 4-9
Filtering Forward or Include Targets 4-9
The web.xml <dispatcher> Element 4-9
Configuring Filters for Forward or Include Targets 4-10
Using a Filter to Wrap and Alter the Request or Response 4-10
Response Filter Example 4-11
Write the Custom Output Stream Code 4-11
Write the Response Wrapper Code 4-12
Write the Base Filter Code 4-12
Write the Response Filter Code 4-13
Write the Target HTML Page 4-13
Configure the Response Filter 4-14
Package the Response Filter Example 4-14
Invoke the Response Filter Example 4-14
Form Authentication Filter 4-15

5 Understanding and Using Event Listeners

Overview of How Event Listeners Work 5-1
Event Listener Interfaces 5-2
ServletContextListener Methods, ServletContextEvent Class 5-2
ServletContextAttributeListener Methods, ServletContextAttributeEvent Class 5-3
HttpSessionListener Methods, HttpSessionEvent Class 5-3
HttpSessionAttributeListener Methods, HttpSessionBindingEvent Class 5-4
HttpSessionActivationListener Methods 5-4
HttpSessionBindingListener Methods 5-5
ServletRequestListener Methods, ServletRequestEvent Class 5-5
ServletRequestAttributeListener Methods, ServletRequestAttributeEvent Class 5-6
Implementing and Configuring Event Listeners 5-6
Implement the Listener Code 5-6
Configure the Listener 5-7
Physical File Required for Welcome File 5-8
Session Lifecycle Listener Example 5-8

Write the JSP Welcome Page 5-9
Write the Session Creation Servlet 5-9
Write the Session Invalidation Servlet 5-10
Write the Session Lifecycle Listener Code 5-10
Configure the Session Lifecycle Listener Example 5-11
Package the Session Lifecycle Listener Example 5-12
Invoke the Session Lifecycle Listener Example 5-12

6 Developing Servlets

Writing a Basic Servlet 6-1
When to Implement Methods of the Servlet Interface 6-2
When to Override the init() Method 6-2
When to Override the doGet() or doPost() Method 6-3
When to Override the doPut() Method 6-3
When to Override the doDelete() Method 6-3
When to Override the getServletInfo() Method 6-3
When to Override the destroy() Method 6-3
Setting Up the Response 6-4
Step-by-Step Through a Simple Servlet 6-4
Simple Servlet Example 6-5
Write the Sample Code 6-6
Compile the Sample Code 6-6
Using HTML Forms and Request Parameters 6-7
Using an HTML Form for User Input 6-7
Displaying Request Parameter Data Specified in User Input 6-8
Complete Example Using a Form and Request Parameters 6-8
Using the POST Method for URL Security 6-10
Calling Information Methods of the Request Object 6-11
Complete Example Retrieving Request Information 6-11
Dispatching to Other Servlets Through Includes and Forwards 6-12
Basics of Includes and Forwards 6-13
Why Use Includes and Forwards? 6-13
Step-by-Step Through the Include or Forward Process 6-14
Complete Example of a Servlet Include 6-14
When to Use Filters for Pre-Processing and Post-Processing 6-16
When to Use Event Listeners for Servlet Notification 6-17
How to Display the Stack Trace 6-18
Migrating an Application from Apache Tomcat to OC4J 6-18
Pointers for Migrating from Tomcat to OC4J 6-19
Introduction 6-19
Migration Approach for Servlets 6-19
Migrating a Simple Servlet 6-20
Migrating a WAR File 6-20
Migrating an Exploded Web Application 6-21
Tips From the Field 6-21
JNDI Lookups in Tomcat and OC4J 6-22
Tomcat-to-OC4J JSP Compilation Issues 6-23

Tomcat-to-OC4J Clustering Issues 6-24
Basic Configuration in Tomcat and OC4J 6-24
Network Considerations in Tomcat and OC4J 6-24
State Persistence Mechanisms in Tomcat and OC4J 6-25
Replication Algorithms in Tomcat and OC4J 6-26
State Replication Transmission 6-26
Application Design in Tomcat and OC4 6-26
Load Balancing in Tomcat and OC4J 6-26

7 Using Annotations for Services and Resource References

Overview of How Annotations Work 7-1
Annotations and Injection 7-2
Annotations in OC4J 7-3
EJB Annotation 7-3
Resource Annotation 7-4
Resources Annotation 7-4
PostConstruct Annotation 7-5
PreDestroy Annotation 7-5
PersistenceUnit(s) Annotation 7-5
PersistenceContext(s) Annotation 7-6
WebServiceRef Annotation 7-6
DeclaresRoles Annotation 7-7
RunAs Annotation 7-7
Annotation Rules and Guidelines 7-7
How Annotations Affect Performance with Servlet Version 2.5 7-8
Annotation Example 7-8

8 Using JDBC or Enterprise JavaBeans

Using JDBC in Servlets 8-1
Why Use JDBC? 8-1
Configuring a Data Source and Resource Reference 8-2
Configure the Data Source 8-2
Configure the Resource Reference 8-3
Implementing JDBC Calls 8-3
Database Query Servlet Example 8-4
Configure the Data Source for the Query Servlet 8-5
Write the HTML Welcome Page 8-5
Write the Query Servlet 8-5
Configure the Servlet and JNDI Resource Reference 8-7
Package the Query Example 8-7
Invoke the Query Example 8-8
TopLink Servlet Examples 8-8
Overview of Enterprise JavaBeans 8-8
Why Use Enterprise JavaBeans? 8-9
EJB Support in OC4J and Oracle Application Server 8-9
Servlet-EJB Lookup Scenarios 8-9

EJB Local Interfaces Versus Remote Interfaces 8-10
Using the Remote Flag for Remote Lookup within the Same Application 8-11

9 Best Practices and Performance

Best Practices for Sessions 9-1
Best Practices for Security 9-1
Considerations for Thread Models 9-2
Custom Thread Pool 9-3
Best Practices for Performance 9-4
Monitoring Performance 9-5
Oracle Application Server Dynamic Monitoring Service 9-5

A Web Module Administration

Application Server Control Console Top-Level Web Module Pages A-1
How to Get to a Web Module Home Page A-1
Summary of Top-Level Web Module Pages A-2
Application Server Control Web Module Configuration Pages A-3
Configuration Properties Page A-3
Deployment Descriptor Viewing Pages A-5
Servlet Mappings Page A-5
Filter Mappings Page A-6
Resource Reference Mappings Page A-7
EJB Reference Mappings Page A-8
Environment Entry Mappings Page A-8
Resource Reference Lookup Context Page A-9
Summary of Web Module MBeans and Administration A-10
General Overview of OC4J MBean Administration A-10
Summary of OC4J Web Module MBeans A-11

B Web Module Configuration Files

Overview of Web Application Configuration Files B-1
Standard web.xml Configuration File B-1
Oracle global-web-application.xml Configuration File B-2
Oracle orion-web.xml Configuration File B-3
Summary of Relationship Between Web Application Configuration Files B-3
Hierarchy of orion-web.xml and global-web-application.xml B-3
Elements and Attributes of orion-web.xml, global-web-application.xml B-4
<access-mask> B-4
<classpath> B-5
<context-attribute> B-6
<context-param-mapping> B-7
<ejb-ref-mapping> B-7
<env-entry-mapping> B-7
<expiration-setting> B-8
<group> B-8
<host-access> B-9

<ip-access> B-9
<jazn-web-app> B-10
<lookup-context> B-12
<mime-mappings> B-12
<ojsp-init> B-13
<orion-web-app> B-16
<request-tracker> B-22
<resource-env-ref-mapping> B-23
<resource-ref-mapping> B-23
<security-role-mapping> B-24
<service-ref-mapping> B-25
<servlet-chaining> B-25
<session-tracker> B-26
<session-tracking> B-27
<user> B-28
<virtual-directory> B-29
<web-app> B-29
<web-app-class-loader> B-30

C Third Party Licenses

ANTLR C-1
The ANTLR License C-1
Apache C-1
The Apache Software License C-2
Apache SOAP C-6
Apache SOAP License C-7

Index

Preface

This document is a developer's guide that 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 Containers for J2EE (OC4J).

The OC4J servlet container in Oracle Application Server 10*g* Release 3 (10.1.3.1.0) is a complete implementation of the Sun Microsystems *Java Servlet Specification, Version 2.4*.

This preface contains these topics:

- Audience
- Documentation Accessibility
- Related Documentation
- Conventions

Audience

The guide is intended for J2EE developers who are writing Web applications that use servlets and possibly JavaServer Pages (JSP) modules. 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 modules, refer to the *Oracle 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. Accessibility 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 more information, visit the Oracle Accessibility Program Web site at

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

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers 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.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documentation

For more information, see the following Oracle resources.

Additional OC4J documents:

- *Oracle Containers for J2EE Developer's Guide*

  This document discusses items of general interest to developers writing an application to run on OC4J—issues that are not specific to a particular container such as the servlet, EJB, or JSP container. (An example is class loading.)

- *Oracle Containers for J2EE Deployment Guide*

  This document covers information and procedures for deploying an application to an OC4J environment. This includes discussion of the deployment plan editor that comes with Oracle Enterprise Manager 10*g*.

- *Oracle Containers for J2EE Configuration and Administration Guide*

  This document discusses how to configure and administer applications for OC4J, including use of the Oracle Enterprise Manager 10*g* Application Server Control Console, use of standards-compliant MBeans provided with OC4J, and, where appropriate, direct use of OC4J-specific XML configuration files.

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

  This document 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 Containers for J2EE JSP Tag Libraries and Utilities Reference*

This document 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 Containers for J2EE Services Guide*

  This document 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 Containers for J2EE Security Guide*

  This document (not to be confused with the *Oracle Application Server 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 Containers for J2EE Enterprise JavaBeans Developer's Guide*

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

- *Oracle Containers for J2EE Resource Adapter Administrator's Guide*

  This document provides an overview of J2EE Connector Architecture features and describes how to configure and monitor resource adapters in OC4J.

Oracle Application Server Web services documents:

- *Oracle Application Server Web Services Developer's Guide*

  This document describes Web services development and configuration in OC4J and Oracle Application Server.

- *Oracle Application Server Advanced Web Services Developer's Guide*

  This document describes topics beyond basic Web service assembly. For example, it describes how to diagnose common interoperability problems, how to enable Web service management features (such as reliability, auditing, and logging), and how to use custom serialization of Java value types.

  This document also describes how to employ the Web Service Invocation Framework (WSIF), the Web Service Provider API, message attachments, and management features (reliability, logging, and auditing). It also describes alternative Web service strategies, such as using JMS as a transport mechanism.

- *Oracle Application Server Web Services Security Guide*

  This describes Web services security and configuration in OC4J and Oracle Application Server.

Java-related documents for Oracle Database:

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

Additional Oracle Application Server documents:

- *Oracle Application Server Administrator's Guide*
- *Oracle Application Server Performance Guide*
- *Oracle HTTP Server Administrator's Guide*
- *Oracle Process Manager and Notification Server Administrator's Guide*

Oracle Enterprise Manager 10*g* Application Server Control online help topics, available through the Application Server Control Console.

The following Oracle Technology Network Web site for Java servlets and JavaServer Pages modules is also available:

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

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

`http://java.sun.com/products/servlet/download.html#specs`

Resources from Sun Microsystems:

- Web site for Java servlet technology:

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

- Web site for JavaServer Pages technology:

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

- J2EE 1.4 Javadoc, including the servlet packages `javax.servlet` and `javax.servlet.http`:

  `http://java.sun.com/j2ee/1.4/docs/api/index.html`

- *Java™ Platform, Enterprise Edition Specification Version 5* (Java EE 5 specification):

  `http://java.sun.com/javaee/5`

- *Enterprise JavaBeans™ Specification, Version 3.0* (EJB specification):

  `http://java.sun.com/products/ejb`

- *Web Services for J2EE 1.2* (Web Services specification):

  `http://jcp.org/en/jsr/detail?id=109`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Summary of What to Know About Servlets

Oracle 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, resource adapter archive (RAR) files for resource adapters, and Java archive (JAR) files for any Enterprise JavaBeans (EJB) and application client modules in the application.

With Oracle Application Server 10*g* Release 3 (10.1.3.1.0), OC4J complies with *Java 2 Platform Enterprise Edition Specification, v1.4*, including full compliance with the Sun Microsystems *Java Servlet Specification, Version 2.4* in the OC4J servlet container. (Any mention of the servlet specification in this manual refers to this version unless otherwise noted.)

> **Note:** Servlets in this release require HTTP/1.1 and Java 2 Standard Edition (J2SE) 1.3 or higher.

This chapter, containing the following sections, provides an overview of servlet technology and concludes with a summary of servlet features:

- Summary of Servlet and J2EE Technology
- Key Components and APIs of the Servlet Model
- Servlet Feature Table

You can find the servlet specification at the following location:

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

> **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/index.html`

Summary of Servlet and J2EE Technology

The following sections offer a brief introduction to servlet and other J2EE technology:

- The Essence of Servlets

- Why Use Servlets?
- Servlet Lifecycle
- 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.

The Essence of Servlets

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 Enterprise JavaBeans (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 can 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.

Why Use 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 each 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 "Preloading Servlets" on page 2-16.
- 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.

Servlet Lifecycle

Servlets have a predictable and manageable lifecycle:

- When the servlet is loaded, its configuration details are read from the standard `web.xml` Web module configuration file. 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" on page 1-12.
- Client requests invoke the central `service()` method of the servlet, which then delegates the request to `doGet()` (for HTTP `GET` requests), `doPost()` (for HTTP `POST` requests), or some other 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 Chapter 4, "Understanding and Using Servlet Filters" for more information.
- A servlet can forward requests to other servlets or include output from other servlets. See "Dispatching to Other Servlets Through Includes and Forwards" on page 6-12.
- 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.

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. 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.

Servlets and JSP pages have a particularly close correspondence. Both servlets and JSP pages are referred to as *Web components*, and both are configured through the standard `web.xml` file. A JSP page implementation class, created by the JSP container during translation of a JSP page, is actually a servlet and implements the `javax.servlet.Servlet` interface, as does any servlet. JSP pages and servlets can be used seamlessly together in creating Web applications.

Servlets or JSP pages often call EJBs to perform further processing. A typical J2EE application uses servlets or JSP pages for the user interface and for initial processing of user requests, then calls EJBs to perform business logic and database access.

> **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:

- *Oracle Containers for J2EE Support for JavaServer Pages Developer's Guide*
- *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide*

Key Components and APIs of the Servlet Model

This section summarizes important components and programming interfaces of the servlet model, covering the following:

- Key Methods of the Servlet Interface
- Servlet Communication: Request and Response Objects
- Servlet Execution in the Servlet Container
- Servlet Contexts: the Application Container
- Servlet Configuration Objects
- What are Servlet Sessions (User Sessions) Used For?
- Servlet Thread Models

For complete information about APIs mentioned here, see the Sun Microsystems Javadoc for the `javax.servlet` and `javax.servlet.http` packages at the following location:

`http://java.sun.com/j2ee/1.4/docs/api/index.html`

Key Methods of the Servlet Interface

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 remove a servlet instance from service.

For Web applications, you can implement the `Servlet` interface by extending the `javax.servlet.http.HttpServlet` abstract class. This class, intended for HTTP servlets suitable for a Web site, extends the `javax.servlet.GenericServlet` class, which implements the `Servlet` interface.

The `HttpServlet` class includes the following methods, which are called by the OC4J servlet container (discussed later in this chapter) during servlet execution, as appropriate. You can write code to override any of them, as desired, to provide functionality in your servlet. See "When to Implement Methods of the Servlet Interface" on page 6-2.

- `void init(ServletConfig config)`

  Initializes the servlet, preparing it to serve requests. This takes a servlet configuration object, described in "Servlet Configuration Objects" on page 1-10, as input. Implement code here for any special startup requirements of your servlet.

- `void destroy()`

  Removes the servlet from service. Implement code here for any special shutdown requirements of your servlet, such as releasing resources.

- `void doGet(HttpServletRequest req, HttpServletResponse resp)`

  Implement code here to execute an HTTP `GET` request. HTTP request and response objects are described in the next section, "Servlet Communication: Request and Response Objects"

- `void doPost(HttpServletRequest req, HttpServletResponse resp)`

  Implement code here to execute an HTTP `POST` request.

- `void doPut(HttpServletRequest req, HttpServletResponse resp)`

  Implement code here to execute an HTTP `PUT` request.

- ```
  void doDelete(HttpServletRequest req,
                HttpServletResponse resp)
  ```

  Executes an HTTP `DELETE` request.

- `String getServletInfo()`

  Retrieves information about the servlet, such as author and release date.

Also be aware of the following methods:

- ```
  void service(HttpServletRequest req,
               HttpServletResponse resp)
  ```

  This is the central method of a servlet. It receives HTTP requests and, by default, dispatches them to the appropriate `do`*`XXX`*`()` methods that you have defined. There is typically no need to override this method.

- `ServletConfig getServletConfig()`

  Retrieves the servlet configuration object, which contains initialization and startup parameters.

Servlet Communication: Request and Response Objects

Servlet methods mentioned in the preceding section that handle HTTP operations—`doGet()`, `doPost()`, `doPut()`, and `doDelete()`—take as input an

HTTP request object (an instance of a class that implements the `javax.servlet.http.HttpServletRequest` interface, which extends the `javax.servlet.ServletRequest` interface) and an HTTP response object (an instance of a class that implements the `javax.servlet.http.HttpServletResponse` interface, which extends the `javax.servlet.ServletResponse` interface).

The request object 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 response object provides HTTP-specific functionality in sending the response, such as specifying the content length and MIME type and providing the output stream.

Key Methods of the HttpServletRequest Interface

This section summarizes methods of interest for HTTP request objects. See "Using HTML Forms and Request Parameters" on page 6-7 for examples of some of their uses.

The following methods are defined in the `HttpServletRequest` interface:

- `HttpSession getSession()`

  Returns (first creating, if necessary) an object representing the client session associated with this request. See "What are Servlet Sessions (User Sessions) Used For?" on page 1-12. You can optionally input the Boolean `true` or `false` to specify whether you want to create a new session if one does not already exist.

- `Cookie[] getCookies()`

  Returns an array of cookie objects for the cookies, used for session tracking, that were sent with this request. See "Using Cookies in Your Servlet" on page 3-10.

- `java.lang.StringBuffer getRequestURL()`

  Recreates the URL that was used for this request.

- `String getContextPath()`

  Returns the context path portion of the URL for this request. This corresponds to the root path of the servlet context for the Web application. See "Summary of URL Components" on page 2-3.

- `String getServletPath()`

  Returns the servlet path portion of the URL for this request. This is the portion that results in the invocation of the particular servlet being requested, according to configuration in the standard `web.xml` file.

- `String getRequestURI()`

  Returns a portion of the URL for this request, after the host and port and up to the query string (if any). This is typically the context path and servlet path.

- `String getQueryString()`

  Returns the query string appended to the URL (if applicable), following the "?" delimiter.

- `String getMethod()`

  Returns the HTTP method used with this request, such as `GET`, `POST`, or `PUT`.

- `String getProtocol()`

  Returns the protocol (typically HTTP) and version being used.

The following methods are inherited from the `ServletRequest` interface:

- `String getParameter(String name)`

  Returns a string indicating the value of the request parameter specified by name (or `null` if not found).

- `java.util.Enumeration getParameterNames()`

  Returns an enumeration object containing strings that indicate the names of all request parameters for this request.

- `javax.servlet.ServletInputStream getInputStream()`

  Retrieves the body of the request in binary format.

- `java.io.BufferedReader getReader()`

  Retrieves the body of the request in character format.

- `String getContentType()`

  Returns a string indicating the MIME type for the body of this request (or `null` if the MIME type is unknown).

- `void setCharacterEncoding(String charset)`

  Overrides the character encoding (MIME character set) that would otherwise be used for interpreting the body and parameters of this request.

- `String getCharacterEncoding()`

  Returns a string indicating the character encoding used for interpreting the body and parameters of this request.

- `RequestDispatcher getRequestDispatcher(String path)`

  Returns a *request dispatcher*, which is used as a wrapper for the resource at the specified path. See "Dispatching to Other Servlets Through Includes and Forwards" on page 6-12.

Key Methods of the HttpServletResponse Interface

This section summarizes methods of interest for HTTP response objects. See "Setting Up the Response" on page 6-4 for further information.

The following methods are defined in the `HttpServletResponse` interface:

- `void sendRedirect(String location)`

  For redirects, specify the alternative location (URL) to which the client is being redirected. (One reason for redirection might be load balancing, for example. Or perhaps a document has moved from one URL to another.)

- `String encodeURL(String url)`

  For session tracking, when cookies are disabled, this is used in URL rewriting to encode the specified URL with an ID for the session. It returns the encoded URL. See "Using URL Rewriting for Session Tracking" on page 3-4.

- `String encodeRedirectURL(String url)`

  For redirects, this is equivalent to `encodeURL()`.

- `void addCookie(javax.servlet.http.Cookie cookie)`

  For session tracking, when cookies are enabled, this adds the specified cookie to the response. See "Using Cookies in Your Servlet" on page 3-10.

- `void sendError(int code)`
  `void sendError(int code, String msg)`

  Send an error response, with a specified integer error code, to the client. You can optionally specify a descriptive message as well.

The following methods are inherited from the `ServletResponse` interface:

- `javax.servlet.ServletOutputStream getOutputStream()`

  Returns a stream object that can be used for writing binary data into the response to the client.

- `java.io.PrintWriter getWriter()`

  Returns a print writer object that can be used for writing character data into the response to the client.

- `void setContentType(String type)`

  Specify a MIME type for the body of this response. You can optionally also specify a character encoding (MIME character set). For example, `"text/html;charset=UTF-8"`.

- `String getContentType()`

  Returns a string indicating the MIME type for the body of this response. This method also returns the character encoding (MIME character set) if one had been specified.

- `void setCharacterEncoding(String charset)`

  Specify a character encoding for the body of this response. Alternatively, you can set a character encoding through the `setContentType()` method. A setting in `setCharacterEncoding()` overrides any character encoding set through `setContentType()`.

- `String getCharacterEncoding()`

  Returns a string indicating the character encoding for the body of this response.

Servlet Execution in 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.

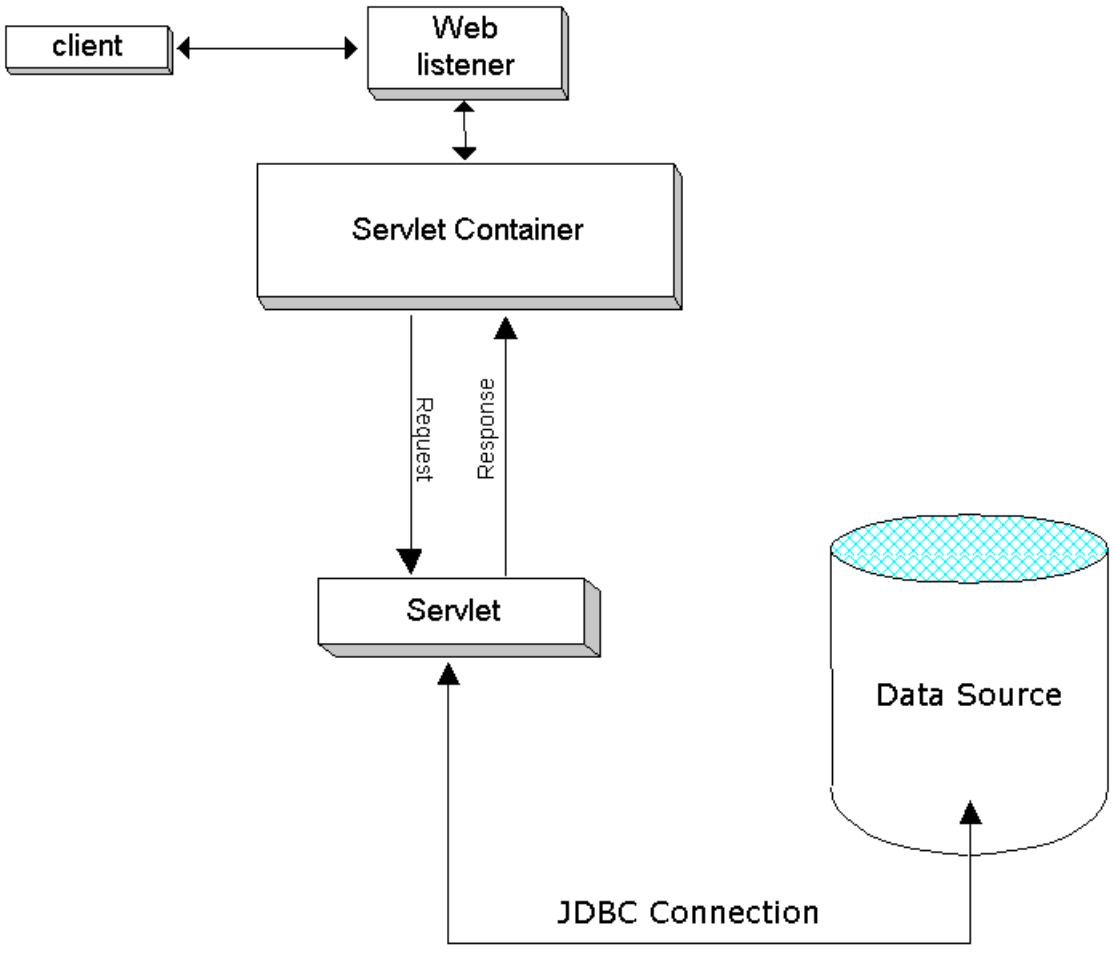
The servlet container provides the servlet with easy access to properties of the HTTP request, such as its headers and parameters. When a servlet is called, such as when it is specified by URL, the Web server passes the HTTP request to the servlet container. The container, in turn, passes the request 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, implemented 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.

Figure 1–1 Servlets and the Servlet Container



A servlet can use J2EE persistence with Oracle TopLink, a Java object-to-relational persistence architecture that provides a mechanism for storing Java objects and Enterprise Java Beans (EJBs) in relational databases and for converting between Java Objects and XML documents (JAXB). For information about using TopLink to integrate

persistence and object-transformation into your applications, see Oracle TopLink Developer's Guide.

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.

Obtaining a Servlet Configuration Object

A servlet can retrieve a servlet configuration object through 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 servlet `init()` method takes a `ServletConfig` object as input, so if you override the `init()` method, the servlet will have access to a servlet configuration object that the servlet container creates and passes during servlet execution.

Key Servlet Configuration Methods

The `ServletConfig` interface specifies the following methods:

- `ServletContext getServletContext()`

  Retrieves a servlet context for the application. See the following section, "Servlet Contexts: the Application Container".

- `String getServletName()`

  Retrieves the name of the servlet.

- `Enumeration getInitParameterNames()`

  Retrieves 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)`

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

Servlet Contexts: the Application Container

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 the 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.

Since 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.)

Obtaining a Servlet Context

A servlet can retrieve a servlet context through the `getServletContext()` method of a servlet configuration object. See "Servlet Configuration Objects" on page 1-10.

Key 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 the following:

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

  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)`

  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()`

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

- `void removeAttribute(String attrname)`

  Removes the specified attribute from the servlet context.

- `String getInitParameter(String name)`

  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()`

  Returns a `java.util.Enumeration` instance containing the names of the initialization parameters of the servlet context.

- `RequestDispatcher getRequestDispatcher(String path)`

Returns a *request dispatcher*, which acts as a wrapper for the resource located at the specified path. See "Dispatching to Other Servlets Through Includes and Forwards" on page 6-12.

- `RequestDispatcher getNamedDispatcher(String name)`

  Returns a request dispatcher that acts as a wrapper for the specified servlet.

- `String getRealPath(String path)`

  Returns the real path, as a string, for the specified virtual path.

- `URL getResource(String path)`

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

- `String getServerInfo()`

  Returns the name and version of the servlet container.

- `String getServletContextName()`

  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.

What are Servlet Sessions (User Sessions) Used For?

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 is a mechanism to identify user sessions and appropriately tie all of a user's requests to his or her session. This process is typically performed using cookies or URL rewriting.

In the standard servlet API, each user session is represented by an instance of a class that implements the `javax.servlet.http.HttpSession` interface.

See Chapter 3, "Understanding and Using Servlet Sessions" for details.

Servlet Thread Models

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 central `service()` method of the servlet.

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. See "Considerations for Thread Models" on page 9-2 for information.

Servlet Feature Table

Table 1–1 summarizes servlet development features described in this document (some of which have already been discussed), with cross-references for information. Features added in the servlet 2.4 specification are noted.

Table 1–1 Servlet Features in the Current Release

Feature	Information
Request and response objects	"Servlet Communication: Request and Response Objects" on page 1-5
Servlet container	"Servlet Execution in the Servlet Container" on page 1-8
Servlet configuration objects	"Servlet Configuration Objects" on page 1-10
Servlet contexts	"Servlet Contexts: the Application Container" on page 1-10
Sessions	Introduction in "What are Servlet Sessions (User Sessions) Used For?" on page 1-12; details in Chapter 3, "Understanding and Using Servlet Sessions"
Includes and forwards	"Dispatching to Other Servlets Through Includes and Forwards" on page 6-12
Servlet filters	Introduction in "When to Use Filters for Pre-Processing and Post-Processing" on page 6-16; details in Chapter 4, "Understanding and Using Servlet Filters" **Note**: The ability to use filters with include or forward targets was added in the servlet 2.4 specification.
Event listeners	Introduction in "When to Use Event Listeners for Servlet Notification" on page 6-17; details in Chapter 5, "Understanding and Using Event Listeners" **Note**: Support for request listeners (as opposed to servlet context or session listeners) was added in the servlet 2.4 specification.
Use of JDBC and data sources	"Using JDBC in Servlets" on page 8-1
Use of EJBs	"Overview of Enterprise JavaBeans" on page 8-8

2

Deploying and Invoking Servlets

After being deployed, a servlet is invoked by OC4J 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-forward or request-include mechanism, or from a remote object on a server.

A servlet is requested through its URL mapping, which is according to how the servlet is configured and deployed, with a portion of the URL (the servlet path) being specified in the standard `web.xml` file, and another portion (the context path) being determined either during deployment or according to the standard `application.xml` file, depending on how you deploy.

The following sections cover servlet deployment and invocation:

- Initial Considerations and OC4J Scenarios
- Summary of URL Components
- Deploying a Web Application to OC4J
- Invoking a Servlet in OC4J
- Deploying and Invoking the Simple Servlet Example
- Preloading Servlets

Initial Considerations and OC4J Scenarios

Before discussing deployment and invocation of servlets in OC4J, we summarize some initial considerations and scenarios:

- A Brief Overview of OC4J Administration
- OC4J in a Standalone Versus Oracle Application Server Environment
- OC4J and Oracle Application Server Administration Tools

A Brief Overview of OC4J Administration

OC4J supports the following standards for deploying and managing applications in a J2EE environment:

- *Java Management Extensions (JMX) 1.2* specification allows standard interfaces to be created for managing resources, such as services and applications, in a J2EE environment. The OC4J implementation of JMX provides a user interface that you can use to completely manage an OC4J server and applications running within it.
- *Java 2 Platform, Enterprise Edition Management Specification* (JSR-77) allows objects known as *MBeans* (managed beans) to be created for runtime management of

applications in a J2EE environment. In OC4J, you can directly access MBeans through a System MBean Browser in Oracle Enterprise Manager 10*g*, but many of their properties are exposed in a more user-friendly way through other features of Enterprise Manager.

- *Java 2 Enterprise Edition Deployment API Specification* (JSR-88) defines a standard API for configuring and deploying J2EE applications and modules into a J2EE-compatible environment. The OC4J implementation includes the ability to create or edit a *deployment plan* containing the OC4J-specific configuration data needed to deploy a component into OC4J.

A deployment plan is a client-side aggregation of all the configuration data needed to deploy an archive into OC4J. You can edit a deployment plan during deployment, using the deployment plan editor.

The OC4J deployment plan editor and System MBean Browser are exposed through Oracle Enterprise Manager 10*g* Application Server Control, referred to as *Application Server Control*. The user interface for this is the *Application Server Control Console*. Additionally, for convenience, many parameters corresponding to MBeans properties, including key properties relating to Web modules, are exposed through other pages of the Application Server Control Console.

In general, avoid direct manipulation of OC4J MBeans or OC4J-specific XML configuration files where possible. The XML files are updated automatically by OC4J when you use the Application Server Control Console. For this reason, this document contains relatively few examples of OC4J-specific XML configuration, although there is reference information in Appendix B, "Web Module Configuration Files". There may be deployment situations, however, where an `orion-web.xml` property is not exposed through the Application Server Control Console. In these situations, directly manipulating the XML file may be the only option.

For general information about OC4J deployment, configuration, and administration, refer to the *Oracle Containers for J2EE Deployment Guide*and the *Oracle Containers for J2EE Configuration and Administration Guide*. For more information about Application Server Control, you can also refer to the introduction to administration tools in the *Oracle Application Server Administrator's Guide*.

OC4J in a Standalone Versus Oracle Application Server Environment

During development, it is typical to use OC4J by itself, outside an Oracle Application Server environment. We refer to this as *standalone* OC4J (or, sometimes, as *unmanaged* OC4J). In this scenario, OC4J can use its own Web listener and is not managed by any external Oracle Application Server processes.

By contrast, a full Oracle Application Server environment (sometimes referred to as *managed* OC4J), includes the use of Oracle HTTP Server as the Web listener, and the Oracle Process Manager and Notification Server (OPMN) to manage the environment.

See the *Oracle Containers for J2EE Configuration and Administration Guide* for additional information about Oracle Application Server versus standalone environments and about the use of Oracle HTTP Server and OPMN with OC4J.

See the *Oracle HTTP Server Administrator's Guide* for general information about Oracle HTTP Server and the related `mod_oc4j` module. (Connection to the OC4J servlet container from Oracle HTTP Server is through this module.) See the *Oracle Process Manager and Notification Server Administrator's Guide* for general information about OPMN.

OC4J and Oracle Application Server Administration Tools

In either an Oracle Application Server or standalone environment, you can deploy, bind, configure, and administer your J2EE applications in OC4J through the Application Server Control, introduced in "A Brief Overview of OC4J Administration" on page 2-1. This is generally the preferred way to manage your applications, and is therefore emphasized in this document. You can deploy an application through the Application Server Control Console "Deploy" feature in the Applications tab that is accessible from the OC4J Home page. Application Server Control Console pages for Web module configuration are discussed in Appendix A, "Web Module Administration".

In standalone OC4J, you also have the option of using the command-line OC4J `admin_client.jar` tool to deploy and bind your J2EE applications.

Alternatively, if you use the Oracle JDeveloper tool to develop your application, you can use it to deploy and bind the application as well.

Also, in some cases and particularly during development, it may be necessary to configure aspects of an OC4J application through direct manipulation of OC4J-specific XML files. For this reason, reference documentation for these files is included in the OC4J documentation set. Elements and attributes of the `global-web-application.xml` (global) and `orion-web.xml` (application-level) OC4J-specific Web module configuration files are documented in Appendix B, "Web Module Configuration Files".

See the *Oracle Containers for J2EE Deployment Guide* and *Oracle Containers for J2EE Configuration and Administration Guide* for general information about using the Application Server Control Console or `admin_client.jar` tool to deploy and manage your applications. There is also extensive online help for the Application Server Control Console.

Summary of URL Components

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

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

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

`protocol``://`*`host`*`:`*`port`*`/`*`contextpath`*`/`*`servletpath`*`/`*`pathinfo`*`?`*`param1`*`=`*`value1`*`...`

Table 2–1 describes the components of the generic construct.

Table 2–1 URL Components

Component	Description
Protocol	The network protocol to be used when invoking the Web application. General examples are HTTP, HTTPS, FTP, or ORMI (for EJBs). In a standalone environment, OC4J typically uses HTTP protocol directly through its own Web listener. In an Oracle Application Server environment, Oracle HTTP Server is the Web listener and it uses AJP (Apache JServ Protocol) to communicate to OC4J, although AJP is invisible to the end user. Protocol for an OC4J Web site is reflected in the `protocol` attribute of the `<web-site>` element in the Web site XML file, such as (typically) `default-web-site.xml`. Use `protocol="http"` for HTTP or `protocol="ajp13"` for AJP. (These should be set appropriately by default.)
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 server 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. The port number for an OC4J Web site is reflected in the `port` attribute of the `<web-site>` element in the Web site XML file. For standalone OC4J, this is typically the `default-web-site.xml` file and the default port is 8888. For an Oracle Application Server environment, this is typically the `default-web-site.xml` file, but depending on the `port` setting, the actual port number may be determined by OPMN. For each port, there must be one associated protocol, according to the `<web-site>` element `protocol` attribute. Refer to the *Oracle Containers for J2EE Configuration and Administration Guide* for general information about OC4J Web site configuration and Web site XML files.

Table 2–1 (Cont.) URL Components

Component	Description
Context path (sometimes referred to as context root)	The designated root path for the servlet context. When you deploy an EAR file using the Application Server Control Console, this is according to a `<context-root>` element in the standard `application.xml` file within the EAR file, as shown in "Create the application.xml File" on page 2-14. When you deploy a WAR file using the Application Server Control Console, you can specify the context path during deployment. When you deploy an EAR file using `admin_client.jar`, you specify the context path when you bind any Web module that is part of the application. (See the *Oracle Containers for J2EE Deployment Guide* for information.) In OC4J, the specified context path is reflected in the setting of the `root` attribute of the `<web-app>` element (a subelement of `<web-site>`) for the applicable Web module in the Web site XML file. (Each context is associated with a directory path in the server file system.) The `<web-app>` element also reflects the J2EE application name (and EAR file name) you specify during deployment, through its `application` attribute, and the Web module name (and WAR file name) you specify, 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" />` When you deploy a WAR file by itself, it is associated with the OC4J default J2EE application.

Servlet path	The designated path, beyond the context path, for the particular servlet you want to invoke. Specify the servlet path through standard mappings in the Web module `web.xml` file.
	
	A servlet class is mapped to a servlet name of your choosing 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>`

Table 2–1 (Cont.) URL Components

Component	Description
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.

Consider the following sample URL:

`http://www.example.com:`*`port`*`/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.

As shown in this example, the context path, servlet path, and any path information can all be "compound" components, with one or more forward-slashes between parts. 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:

- 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).
- You can retrieve the context path, servlet path, and path information through the `getContextPath()`, `getServletPath()`, and `getPathInfo()` methods of the HTTP request object.

Deploying a Web Application to OC4J

In OC4J, according to the J2EE specification, you deploy a J2EE application as an EAR file. The EAR file contents include zero or more WAR files for Web applications (combinations of servlets and JSP pages) that are part of the overall J2EE application.

If you want to deploy a Web application only, you can either package the WAR file inside an EAR file, effectively defining a J2EE wrapper application, or you can deploy the WAR file directly.

The following sections, after a review of standard application structure, discuss the general steps for each approach:

- Application Structure
- Summary of General Steps to Deploy a WAR File
- Summary of General Steps to Deploy an EAR File

Note that this discussion does not go into detail—it is intended only as a summary. See the *Oracle Containers for J2EE Deployment Guide* for specific information and procedures for deploying to OC4J.

Application Structure

The standard Web application structure, as specified in the *Java Servlet Specification*, is as follows:

```
root_directory/
   Static files (for example, index.html)
   JSP pages
   WEB-INF/
      web.xml
      classes/
         servlet classes (directory substructure according to Java package)
      lib/
         JAR files (libraries and dependency classes)
```

This structure is reflected in the structure of a standard WAR file, used to deploy a Web application. The standard `web.xml` file, also specified in the *Java Servlet Specification*, is where you configure servlets and JSP pages (among other things). See "Summary of General Steps to Deploy a WAR File", which follows shortly, for additional information about WAR files and `web.xml`.

> **Note:** For OC4J-specific Web module settings, you can include an `orion-web.xml` under `/WEB-INF` along with the `web.xml` file. Alternatively, you can let OC4J create the `orion-web.xml` file automatically and use the Application Server Control Console for OC4J-specific settings (which are then reflected in `orion-web.xml`).

The standard J2EE application structure, as specified in the *Java 2 Enterprise Edition Specification*, is a superset of the Web application structure, as follows:

```
root_directory/
   META-INF/
      application.xml
   WebModule/
      Static files (for example, index.html)
      JSP pages
      WEB-INF/
         web.xml
         classes/
            servlet classes (directory substructure according to Java package)
         lib/
            JAR files (libraries and dependency classes)
   EJBModule/...
   ClientModule/...
   ResourceAdapterModule/...
```

This structure is reflected in the structure of a standard EAR file, used to deploy a J2EE application and the WAR and other archive files that it contains. The standard `application.xml` file, also specified in the *Java 2 Enterprise Edition Specification*, is where you configure a J2EE application and its modules, including Web modules. The EAR file contains any WAR files, EJB JAR files, application client JAR files, and resource adapter RAR files containing modules for the application. See "Summary of General Steps to Deploy an EAR File", which follows shortly, for additional information about EAR files and `application.xml`.

> **Note:** For OC4J-specific J2EE application settings, it is permissible to include an `orion-application.xml` file under `/META-INF` along with the standard `application.xml` file. But it is more typical to let OC4J create the `orion-application.xml` file automatically and to use the Application Server Control Console for OC4J-specific settings (which are then reflected in `orion-application.xml`). See the *Oracle Containers for J2EE Developer's Guide* for information about the `orion-application.xml` file.

Summary of General Steps to Deploy a WAR File

To deploy a Web application directly as a WAR file (as opposed to putting the WAR file into a J2EE EAR file), use the following general steps.

1. Create a standard `web.xml` file to configure the Web application. A `web.xml` file is required within a WAR file. Within the top-level `<web-app>` element, use `<servlet>` and `<servlet-mapping>` subelements to configure servlets and JSP pages.

   Map a servlet class to a URL servlet path by using the `<servlet-name>` and `<servlet-class>` subelements of a `<servlet>` element, and the `<servlet-name>` and `<url-pattern>` subelements of a `<servlet-mapping>`

element. The name is of your choosing but should be logical; its purpose is simply to map the servlet class to the servlet path.

```
<web-app>
...
  <servlet>
    <servlet-name>servletname</servlet-name>
    <servlet-class>package.Classname</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>servletname</servlet-name>
    <url-pattern>servletpath</url-pattern>
  </servlet-mapping>
...
</web-app>
```

2. Create a WAR file to contain your Web application components and `web.xml` file, from the root directory of a directory structure that parallels the Web application structure shown in "Application Structure" on page 2-7.
3. Deploy the WAR file to OC4J. If you use the Application Server Control Console "Deploy" feature (available from the Applications tab that is accessible from the OC4J Home page), this includes the opportunity to supply or create, and optionally edit, a standard JSR-88-compliant deployment plan.
4. Bind the Web application. This is the process that associates the Web application with an OC4J Web site, and associates a URL context path to use in accessing the Web application. When you use the Application Server Control Console to deploy a WAR file, binding is included in the deployment steps and you have the opportunity to specify the context path.

Note: You cannot configure parameters that would correspond to an `orion-application.xml` file when you deploy a standalone WAR file, because there can be no `orion-application.xml` file within a WAR file. You would have to package the WAR file in an EAR file to configure such parameters. See "Application Structure" on page 2-7 for related information.

Summary of General Steps to Deploy an EAR File

To deploy a Web application as a WAR file within an EAR file, as opposed to deploying the WAR file directly, use the following steps.

1. Create a `web.xml` file and WAR file for the Web application, as described in the preceding section, "Summary of General Steps to Deploy a WAR File".
2. Create a standard `application.xml` file, which configures a J2EE application. An `application.xml` file is required within an EAR file. In particular, map the Web application you are deploying to a URL context path—this is where Application Server Control obtains context path information, which is subsequently written to the Web site XML file.

   Within a `<web>` element, use a `<web-uri>` subelement to specify the WAR file name, together with a `<context-root>` subelement to specify the context path.

```
<application>
...
   <module>
      <web>
```

```
            <web-uri>warname.war</web-uri>
            <context-root>contextpath</context-root>
        </web>
    </module>
...
</application>
```

3. Create an EAR file to contain your application components and `application.xml` file, from the root directory of a directory structure that parallels the J2EE application structure shown in "Application Structure" on page 2-7.
4. Deploy the EAR file to OC4J. If you use the Application Server Control Console "Deploy" feature (available from the Applications tab that is accessible from the OC4J Home page), this includes the opportunity to supply or create, and optionally edit, a standard JSR-88-compliant deployment plan.
5. Bind any Web application to be invoked. This is the process that associates the Web application with an OC4J Web site, and associates a URL context path to use in accessing the Web application. When you use the Application Server Control Console to deploy an EAR file, binding of a Web application is included in the deployment steps, and the context path is according to the standard `application.xml` file that you provided in the EAR file, as noted in the preceding text.

Invoking a Servlet in OC4J

This section discusses how to invoke a servlet in a standalone OC4J environment versus an Oracle Application Server environment, and covers special OC4J features for invoking a servlet by class name in a development or testing scenario:

- Invoking a Servlet in a Standalone OC4J Environment
- Invoking a Servlet by Class Name During OC4J Development
- Invoking a Servlet in an Oracle Application Server Environment

Invoking a Servlet in a Standalone OC4J Environment

In a standalone OC4J environment, a Web site uses HTTP protocol directly through the OC4J Web listener, without going through the Oracle HTTP Server, and is configured according to settings in the `default-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 described in "Summary of URL Components" on page 2-3, which also discusses considerations in how the context path and servlet path are determined. By default in standalone OC4J, the port is 8888, which is used for many examples in this document (given that it is a developer's guide).

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

```
http://www.example.com:8888/mypath/myservlet
```

Invoking a Servlet by Class Name During OC4J Development

For a development or testing scenario in standalone OC4J, there is a convenient mechanism for invoking a servlet by class name. For security reasons, use this mechanism *only* while developing or testing your application.

With a `true` setting of the OC4J system property `http.webdir.enable` (where `false` is the default), 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. 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.)

OC4J will look for the class under the `/WEB-INF/classes` directory (in a subdirectory according to the package) or in a JAR file in the `/WEB-INF/lib` directory of the Web application associated with the context path.

Notes:

- You can set `servlet-webdir` (`servletWebdir`) through the Application Server Control deployment plan editor.
- You can use the invocation by class name mechanism with the OC4J default Web application by placing the class under the `j2ee/home/default-web-app/WEB-INF/classes` directory and using the context path of the default Web application ("/" by default) in the URL.

Generally speaking, for any given application in a scenario with the `true` setting of `http.webdir.enable`, 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 included 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:
  - Use the default `false` value of the system property `http.webdir.enable`. This results in any `servlet-webdir` setting being ignored. (See the *Oracle Containers for J2EE Configuration and Administration Guide* for general information about OC4J system properties.)
  - Set a `servlet-webdir` value of `""` (empty quotes), either through `global-web-application.xml` or `orion-web.xml`.

The `servlet-webdir` attribute is also discussed in "<orion-web-app>" on page B-16.

Assuming a context path of "`/mypath`" and a setting of `servlet-webdir="/servlet/"`, the following URL invokes the servlet `foo.bar.SessionServlet` by its class name:

```
http://www.example.com:8888/mypath/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 "Best Practices for Security" on page 9-1 for information.

Invoking a Servlet in an Oracle Application Server Environment

In an Oracle Application Server environment, OC4J is accessed through the Oracle HTTP Server, which uses AJP protocol to communicate to OC4J. (AJP is invisible to the end user.) A Web site is configured according to settings in the `default-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 described in "Summary of URL Components" on page 2-3, which also discusses considerations in how the port, context path, and servlet path are determined.

If "`/mypath`" is the context path and "`/myservlet`" is the servlet path, for example, you will invoke the servlet with a URL such as the following (with an appropriate port number):

```
http://www.example.com:port/mypath/myservlet
```

Deploying and Invoking the Simple Servlet Example

In this section, we deploy and invoke the servlet shown in "Simple Servlet Example" on page 6-5. First we deploy it directly as a WAR file, then as a WAR file within an EAR file.

Deploy the Servlet Example as a WAR File

In this section, we use the following steps, first outlined in "Summary of General Steps to Deploy a WAR File" on page 2-8, to deploy the simple servlet example directly as a WAR file:

1. Create the web.xml File
2. Create the WAR File
3. Deploy the WAR File and Bind the Web Application

See "Invoke the Servlet Example" on page 2-15 for execution of the servlet, and for how some of the deployment specifications are reflected in the URL of the servlet.

Create the web.xml File

Here is a `web.xml` file for the simple servlet example. The `<servlet-class>` element reflects the package name and class name specified in `HelloWorld.java`, shown in "Write the Sample Code" on page 6-6. The `<url-pattern>` element specifies `myhello` as the servlet path portion of the URL to use in invoking the servlet. The servlet name maps the class name to the servlet path.

```
<?xml version="1.0"?>
<!DOCTYPE web-app (doctype...)>
<web-app>
  <servlet>
    <servlet-name>hello</servlet-name>
```

```
    <servlet-class>mytest.HelloWorld</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>myhello</url-pattern>
  </servlet-mapping>
</web-app>
```

Create the WAR File

Here is the directory structure for the simple servlet example, according to the standard Web application structure:

```
root_directory/
   WEB-INF/
      web.xml
      classes/
         mytest/
            HelloWorld.class
            HelloWorld.java
```

The WAR file must reflect this structure. If you want to create the WAR file manually, you can use the JAR utility, with the following command, while your current directory is the root directory (assume % is the system prompt):

```
% jar cvf MyHelloWorld.war .
```

This generates the WAR file `MyHelloWorld.war`, with the following contents and structure (where the `Manifest.mf` file and `META-INF` directory are created automatically):

```
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/mytest/HelloWorld.class
WEB-INF/classes/mytest/HelloWorld.java
```

The WAR file name should reflect the key Web component name. In this case, it reflects the class name of the servlet example.

Deploy the WAR File and Bind the Web Application

Deploy the WAR file, typically through Application Server Control "Deploy" feature (available from the Applications tab, which is accessible from the OC4J Home page).

In the Application Server Control Console:

1. Specify the WAR file.
2. Instruct Application Server Control to create a new deployment plan.
3. Specify an application (Web application) name, which should reflect the WAR file name.
4. Specify the parent application (the OC4J default application by default).
5. Specify the Web site for binding the Web application, such as `default-web-site`.
6. Specify the context path (or use the default, derived from the WAR file name). For this example, specify "`/mycontext`".

Note: If you use `admin_client.jar` instead of the Application Server Control Console, deploying and binding are separate steps, and you specify the context path when you bind. See the *Oracle Containers for J2EE Deployment Guide* for information.

Deploy the Servlet Example as an EAR File

In this section, we use the following steps, first outlined in "Summary of General Steps to Deploy an EAR File" on page 2-9, to deploy the simple servlet example as a WAR file within an EAR file:

1. Create the web.xml File and WAR file.
2. Create the application.xml File
3. Create the EAR File
4. Deploy the EAR File and Bind the Contained Web Application

See "Invoke the Servlet Example" on page 2-15 for execution of the servlet, and for how some of the deployment specifications are reflected in the URL of the servlet.

Create the web.xml File and WAR file.

First, create the `web.xml` file and WAR file, as you would if you were deploying the WAR file directly. This is described in "Create the web.xml File" on page 2-12 and "Create the WAR File" on page 2-13.

Create the application.xml File

Here is an `application.xml` file for the simple servlet example, to use in deploying the servlet in a WAR file within an EAR file. When you use the Application Server Control Console to deploy, this is where the context path is determined; then it is subsequently written to the Web site XML file. The `<web-uri>` element indicates the name of the WAR file. The `<context-root>` element ties the Web application of the WAR file (in this case, the servlet example) to the desired URL context path, `/mycontext`.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<!DOCTYPE web-app (doctype...)>
<application>
   <module>
      <web>
         <web-uri>MyHelloWorld.war</web-uri>
         <context-root>/mycontext</context-root>
      </web>
   </module>
</application>
```

Note: When you use `admin_client.jar` to deploy an EAR file and bind a contained Web application, you specify the context path directly when you bind the Web application.

Create the EAR File

Before creating the EAR file, follow these steps to prepare your directory structure:

1. From your desired root directory, create a `META-INF` subdirectory.

2. Place the `application.xml` file in `META-INF`, according to the standard J2EE application structure.
3. Place the WAR file in the root directory.

This results in the following directory structure:

```
root_directory/
   META-INF/
      application.xml
   MyHelloWorld.war
```

If you want to create the EAR file manually, you can use the JAR utility, with the following command, while your current directory is the root directory (assume `%` is the system prompt):

```
% jar cvf MyHelloWorld.ear .
```

This generates the EAR file `MyHelloWorld.ear`, with the following contents and structure (where the `Manifest.mf` file is created automatically):

```
MyHelloWorld.war
META-INF/application.xml
META-INF/Manifest.mf
```

Deploy the EAR File and Bind the Contained Web Application

Deploy the EAR file, typically through the Application Server Control "Deploy" feature (available from the Applications tab, which is accessible from the OC4J Home page).

In the Application Server Control Console:

1. Specify the EAR file.
2. Instruct Application Server Control to create a new deployment plan.
3. Specify a J2EE application name, which should reflect the EAR file name.
4. Specify the parent application (the OC4J default application by default).
5. Specify the Web site for binding the Web application, such as `default-web-site`.

> **Note:** If you use `admin_client.jar` instead of the Application Server Control Console, deploying an EAR file and binding a contained Web application are separate steps. See the *Oracle Containers for J2EE Deployment Guide* for information.

Invoke the Servlet Example

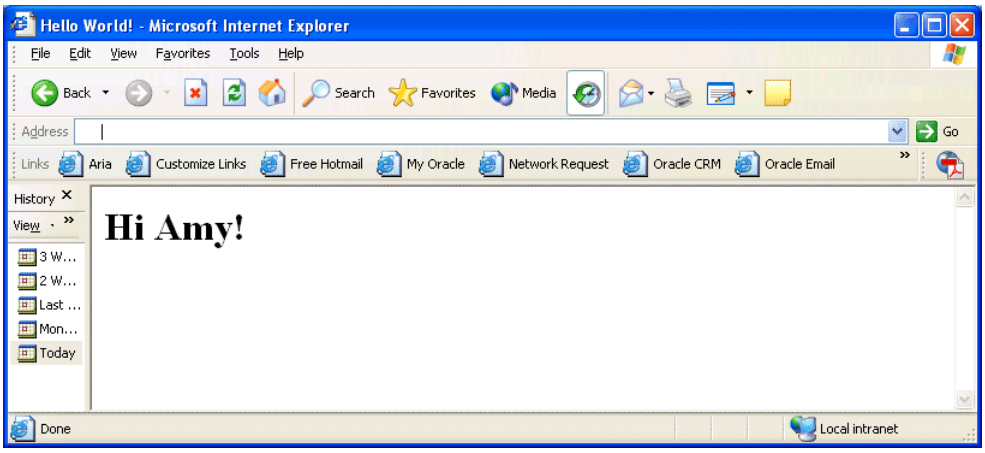
Given the WAR file deployment or EAR file deployment shown previously (both of which specify the same servlet path and context path), invoke the servlet as follows, specifying the appropriate host name. The URL here assumes an OC4J port number of 8888, which is the default for OC4J in a standalone environment.

```
http://www.example.com:8888/mycontext/myhello
```

The servlet path, `myhello`, is according to the `web.xml` file you provided during deployment. The context path, `/mycontext`, is according to either the `application.xml` file or according to your specification in binding the Web

application, as discussed earlier. ("Summary of URL Components" on page 2-3 includes discussion of context paths.)

Here is the output of the servlet:



Preloading Servlets

Typically, the servlet container instantiates and loads a servlet class when it is first requested, such as by direct request through the browser, or through an include or forward. However, if you want any servlets to start as soon as the server starts, you can arrange for them to be preloaded through settings in the `server.xml` file, the Web site XML file (such as `default-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 `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 the *Oracle Containers for J2EE Configuration and Administration Guide* for information about OC4J 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 2–2 explains the behavior of the `<load-on-startup>` element in `web.xml`.

Table 2–2 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.

Table 2–2 (Cont.) File web.xml <load-on-startup> Behavior

Value Range	Behavior
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*, described in detail in the *Oracle Containers for J2EE Developer's Guide*. 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.

3

Understanding and Using Servlet Sessions

Servlet sessions, used to track state information for users over multiple requests and responses, were introduced in "What are Servlet Sessions (User Sessions) Used For?" on page 1-12. The following sections provide details and examples, including use of session attributes and cookies:

- Overview of Session Tracking
- Using Session Tracking in OC4J
- Using a Session Object in Your Servlet
- Using Cookies in Your Servlet
- Canceling a Session

Overview of Session Tracking

The OC4J servlet container, in compliance 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. (Such a class is provided by the OC4J servlet container.) When a servlet creates an HTTP session object, the client interaction is considered to be stateful.

You can think of a session object as a dictionary that stores values (Java objects) together with their associated keys, or names (Java strings). Each name/value pair constitutes a session attribute.

The following subsections discuss session objects and other session-tracking features:

- Session Objects
- Session IDs
- Cookies and Persistent Session Data
- When to Use Cookies Versus Session Attributes

Session Objects

A servlet uses the `getSession()` method of the HTTP request object to retrieve or create an HTTP session object. 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. You can use attributes of the session object to store and retrieve data related to the user session, through the `setAttribute()` and `getAttribute()` methods. See "Using a Session Object in Your Servlet" on page 3-5.

You cannot use session objects to share data between applications, nor can you use them to share data between different clients of the same application. There is one HTTP session object for each client in each application.

Session IDs

In order to maintain a mapping between a particular session and the appropriate session object, so that it can properly access information for the current session, OC4J generates a unique session ID for each session. When a stateful servlet (essentially, a servlet that has created a session object) returns a response to the client, the session ID generated by OC4J is included in the response. If cookies are enabled, OC4J uses a session ID cookie to accomplish this. See "How OC4J Can Use Cookies for Session Tracking" on page 3-3.

If cookies are disabled, the session ID is communicated through URL rewriting, and you are required to call the `encodeURL()` method (or the `encodeRedirectURL()` method, for includes or forwards) of your HTTP response object. See "Using URL Rewriting for Session Tracking" on page 3-4.

Cookies and Persistent Session Data

To make session data persistent, you can store it in a database if you need the protection, transactional safety, and backup features that a database offers. For smaller amounts of data, where database functionality is not required, you can create and use your own cookies, or perhaps use the file system or other remote objects.

A cookie has a name and a single associated value, and is stored as an attribute of the HTTP request and response headers. See "Using Cookies in Your Servlet" on page 3-10.

When to Use Cookies Versus Session Attributes

Session data is typically stored and manipulated through attributes of the session object, but for small amounts of data, particularly information that you want to be persistent, you have the option of creating and using cookies instead. When to use cookies instead of session attributes is a matter of preference and intent, but consider the following:

- Cookies are communicated back and forth between the client and server in the HTTP requests and responses, while a session object remains in the server. It is obviously inefficient to move large amounts of data back and forth between the client and server, so cookies should be used only for small amounts of data.
- Because cookies travel back and forth over the network or Internet, they are generally less secure than session objects.

One philosophy, for example, might be to use session attributes for business-related data, and cookies for presentation-related data. A cookie can indicate information, such as information about the user, that tells the servlet what to display or how to display it, and does not tend to change between subsequent sessions of the same user.

Using Session Tracking in OC4J

The following sections describe key OC4J features for session tracking:

- Configuring Session Tracking and Enabling or Disabling Cookies in OC4J
- How OC4J Can Use Cookies for Session Tracking
- Using URL Rewriting for Session Tracking

- Session Tracking Through Secured Connections

Configuring Session Tracking and Enabling or Disabling Cookies in OC4J

OC4J performs session tracking according to settings in the `<session-tracking>` element of the `global-web-application.xml` or `orion-web.xml` file.

The servlet container first attempts to accomplish session tracking through cookies. If cookies are disabled, the server can maintain session tracking only through URL rewriting, through the `encodeURL()` method of the response object, or the `encodeRedirectURL()` method for forwards or includes. You must include the `encodeURL()` or `encodeRedirectURL()` calls in your servlet if cookies may be disabled. See "Using URL Rewriting for Session Tracking" on page 3-4.

Cookies are enabled by default, as reflected by the setting `cookies="enabled"` in `<session-tracking>`. A setting of `cookies="disabled"` disables cookies:

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

> **Note:** You can explicitly enable or disable cookies through the Application Server Control deployment plan editor `sessionTracking` property, as discussed in the *Oracle Containers for J2EE Deployment Guide*.

You can also designate one or more session tracker servlets, according to settings of `<session-tracker>` subelements of `<session-tracking>`. 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 that is 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 an HTTP session listener, which is an instance of a class that implements the `javax.servlet.http.HttpSessionListener` interface. See Chapter 5, "Understanding and Using Event Listeners" for information about session listeners.

Also see "<session-tracking>" on page B-27 and "<session-tracker>" on page B-26.

How OC4J Can Use Cookies for Session Tracking

If cookies are enabled, OC4J uses session tracking cookies as follows:

1. With the first response to a servlet after a session is created, the servlet container sends a cookie with a session ID 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 an HTTP `Set-Cookie` response header.
2. Upon each subsequent request from the same Web client session, the client sends the cookie back to the server as part of the request, in an HTTP Cookie request header, 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.

Your servlet code is not required to do anything to send this cookie; the container handles this automatically. Sending cookies back to the server is handled automatically by the Web browser.

Also see "Session Tracking Through Secured Connections" on page 3-4.

Using URL Rewriting for Session Tracking

An alternative to cookies is URL rewriting, through the `encodeURL()` method of the HTTP response object (or, equivalently, the `encodeRedirectURL()` method for includes or forwards). This mechanism allows OC4J to encode the session ID into a URL path if cookies are disabled. The following conditions must be met:

- The session must be valid.
- The session ID has not been received through cookies in previous exchanges with the client.
- The URL points to a location within the application.

To ensure that the servlet container can use URL rewriting, you must use `encodeURL()` whenever you write a URL to the output stream, instead of writing the URL directly. For example:

```
out.println("Click <a href=" + res.encodeURL(req.getRequestURL().toString()) +
            ">this link</a>");
out.println(" to access this page again.<br>");
```

OC4J uses the parameter `jsessionid` (in conformance with the servlet specification) to indicate the session ID in the URL path, as in the following example:

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

To comply with the servlet specification, calls to the `encodeURL()` and `encodeRedirectURL()` methods result in no action if cookies are enabled.

Notes:

- The `encodeURL()` method replaced the servlet 2.0 `encodeUrl()` method (note capitalization), which is deprecated.
- 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 resource-intensive process.

Session Tracking Through Secured Connections

When exchanges between OC4J and a client include sensitive information, the transmissions should occur over a secured connection. You can achieve this with HTTPS (transmitting the HTTP protocol over SSL sockets, as discussed in detail in the *Oracle Containers for J2EE Security Guide*). In this case, cookies or URL rewriting would not be appropriate for transmitting a session ID, given that the ID could be intercepted or spoofed. If the value of the session ID is compromised, the associated session state is vulnerable.

In this secured transmission situation, where HTTPS is used for all transmissions, OC4J stores the information needed to retrieve the session state directly into the SSL connection, as an attribute of the SSL session (functionality that is invisible to the user). This provides the greatest level of security for the session state, but also ties the lifetime of the session state to the lifetime of the SSL connection itself. If the SSL connection is dropped, the session state is lost.

It is also possible for an application to be shared between HTTP and HTTPS. (See "Making an Application Available on HTTP and HTTPS in Standalone OC4J" on page 3-5.) If an application is shared in this way, OC4J assumes that transmissions to it may or may not be over the SSL connection. The lack of a guaranteed SSL connection where session information can be stored results in OC4J falling back to cookies or URL rewriting for session tracking, as discussed previously.

Note: Some older browsers drop the SSL connection in certain circumstances, causing the subsequent and sometimes unexpected loss of session state. If this is a problem, you can work around it by specifying the application as shared to force the use of cookies or URL rewriting for session tracking. This is less secure, but may be the only workable alternative.

Making an Application Available on HTTP and HTTPS in Standalone OC4J

To make a single application available over two protocols in standalone OC4J, that application must be declared in two different Web site XML files, and marked as "shared". This feature is enabled if the `shared` attribute of the `<web-app>` element in each Web site XML file is set to "`true`". Setting an application as shared makes a single application deployment available over the protocols defined for each Web site it is declared within. To share between HTTP and HTTPS, one Web site would be secured, configured for HTTPS, and the other would be not secured, configured for HTTP.

See the *Oracle Containers for J2EE Configuration and Administration Guide* for additional information about shared applications in OC4J.

Using a Session Object in Your Servlet

This section shows you how to use attributes of a session object, first summarizing key methods of the `HttpSession` interface, then discussing the creation and retrieval of session attributes through getter and setter methods, and concluding with an example.

Summary of HttpSession Methods

The servlet container uses HTTP session objects—instances of a class provided by OC4J that implements the `javax.servlet.http.HttpSession` interface—in managing data for user sessions. The `HttpSession` interface specifies the following public methods to handle session attributes:

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

  Add a session attribute, adding the specified object to the session object, under the specified name.

- `Enumeration getAttributeNames()`

  Retrieves a `java.util.Enumeration` object consisting of `String` objects for the names of all session attributes.

- `Object getAttribute(String name)`

Retrieves the value of the session attribute that has the specified name (or returns `null` if there is no match).

- `void removeAttribute(String name)`

  Removes the session attribute that has the specified name.

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

- `void setMaxInactiveInterval(int interval)`

  Set a session timeout interval, in seconds, as an integer, overriding any default in the servlet container or set through the `web.xml` `<session-timeout>` element. A negative value indicates no timeout. A value of 0 results in an immediate timeout. Also see "Canceling a Session" on page 3-15.

- `int getMaxInactiveInterval()`

  Retrieves a value indicating the session timeout interval, in seconds. If you had specified a timeout value through `setMaxInactiveInterval()`, that is the value that is returned. Otherwise, the value of the `web.xml` `<session-timeout>` element is returned, if that was specified. Otherwise, the servlet container default timeout is returned.

- `void invalidate()`

  Immediately invalidate the session, unbinding any objects from it.

There are also utility methods to retrieve a servlet context, and to retrieve information about session creation and access:

- `ServletContext getServletContext()`

  Retrieves the servlet context that the session belongs to.

- `boolean isNew()`

  Returns `true` within the request that created the session; returns `false` otherwise.

- `long getCreationTime()`

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

- `long getLastAccessedTime()`

  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.

Development Tip:

The `java.util.Date` class has a constructor that takes milliseconds since midnight, January 1, 1970, and converts that value into the date, hours, minutes, and seconds.

For complete information about `HttpSession` methods, refer to the Sun Microsystems Javadoc for the `javax.servlet.http` package, at the following location:

http://java.sun.com/j2ee/1.4/docs/api/index.html

Adding and Retrieving Session Attributes

Here are some typical steps in creating and displaying session attributes. These steps are shown in a complete example in "Session Object Example" on page 3-7.

1. Get the session object from the request:

```
HttpSession session = request.getSession();
```

2. Take user input to specify each attribute name and value. For this example, assume the servlet has a form that takes the name and value for an attribute and stores them in request parameters called `dataname` and `datavalue`. (The form is shown in the complete example.)

```
String dataName = request.getParameter("dataname");
String dataValue = request.getParameter("datavalue");
```

3. Use the specified name and value to set each attribute:

```
if (dataName != null && dataValue != null) {
    session.setAttribute(dataName, dataValue);
}
```

4. If you want to output all the attributes back to the browser, start by calling `getAttributeNames()` to get all the names:

```
Enumeration attributeNames = session.getAttributeNames();
```

5. Iterate through the `java.util.Enumeration` object to retrieve each name, then retrieve each corresponding value, and output the name and value. In this example, assume all the names and values can be handled as Java strings:

```
while (attributeNames.hasMoreElements()) {
   String name = attributeNames.nextElement().toString();
   String value = session.getAttribute(name).toString();
   out.println(name + " = " + value + "<br>");
}
```

Important: In a clustered application, any object used as a session attribute must implement the `java.io.Serializable` interface.

Note: In the `HttpSession` interface, the methods `setAttribute()`, `getAttributeNames()`, and `getAttribute()` replace `putValue()`, `getValueNames()`, and `getValue()`, which are deprecated.

Session Object Example

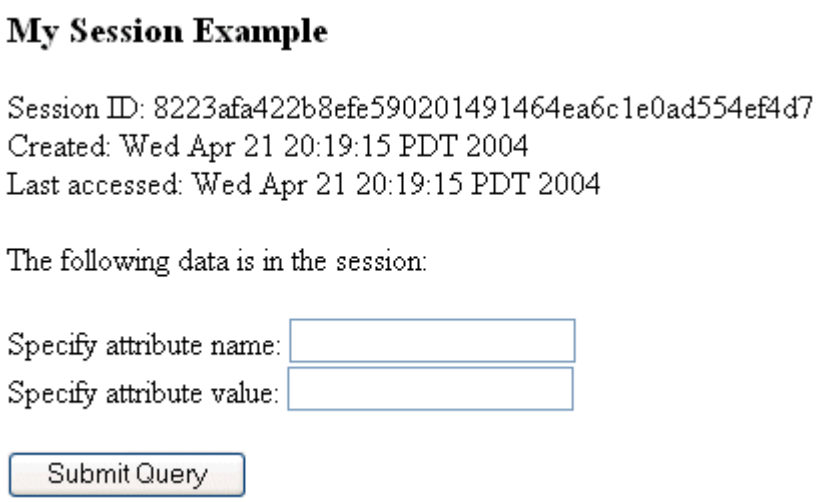
This example puts together the steps in "Adding and Retrieving Session Attributes" on page 3-7. There is also code for a form to take user input for attribute names and values, and code to display the session ID, creation time, and last access time (in this case, the time when the last attribute was added).

As in previous examples, the `doGet()` method is called through the `doPost()` method, for a `POST` request.

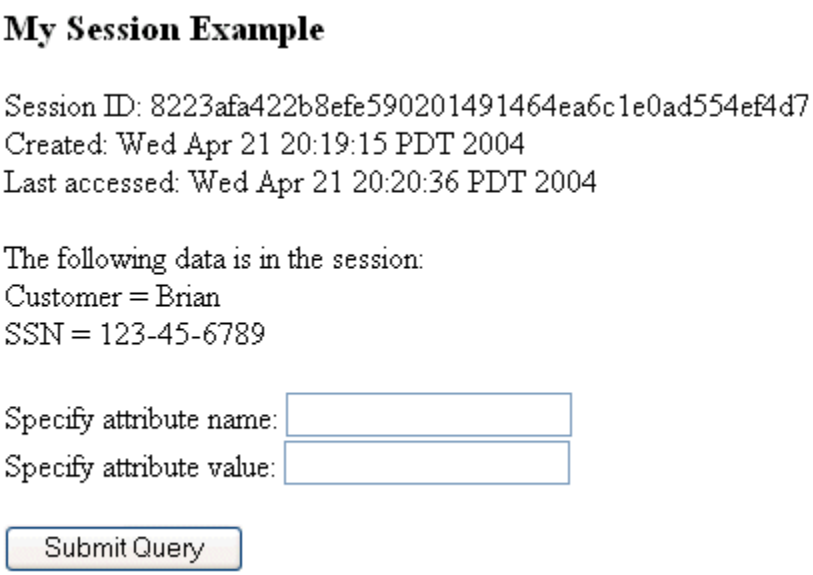
```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionExample extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body bgcolor=\"white\">");

        out.println("<h3>" + "My Session Example" + "</h3>");

/* Display session ID, creation time, and access time. */

        HttpSession session = request.getSession();
        out.println("Session ID: " + session.getId());
        out.println("<br>");
        out.println("Created: ");
        out.println(new Date(session.getCreationTime()) + "<br>");
        out.println("Last accessed: ");
        out.println(new Date(session.getLastAccessedTime()));

/* Set attribute, based on data from user (form is later in code). */

        String dataName = request.getParameter("dataname");
        String dataValue = request.getParameter("datavalue");
        if (dataName != null && dataValue != null) {
            session.setAttribute(dataName, dataValue);
        }

/* Display all attributes. */

        out.println("<P>");
        out.println("The following data is in the session: <br>");
        Enumeration attributeNames = session.getAttributeNames();
        while (attributeNames.hasMoreElements()) {
           String name = attributeNames.nextElement().toString();
           String value = session.getAttribute(name).toString();
           out.println(name + " = " + value + "<br>");
        }

/* Take user input for an attribute. */

        out.println("<P>");
        out.print("<form action=\"");
        out.print("SessionExample\" ");
        out.println("method=POST>");
        out.println("Specify attribute name: ");
        out.println("<input type=text size=20 name=dataname>");
        out.println("<br>");
        out.println("Specify attribute value: ");
        out.println("<input type=text size=20 name=datavalue>");
        out.println("<br><br>");
```

```
        out.println("<input type=submit>");
        out.println("</form>");

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

    }

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

}
```

When you first run the servlet, it displays something like the following:



Each time you enter a new attribute name and value and click **Submit Query**, the output is updated to show the new name/value pair. If you enter "Customer" and "Brian" for one attribute name and its value, and "SSN" and "123-45-6789" for another attribute name and its value, clicking **Submit Query** after each pair of entries, the output is updated as follows. Note that the last access time of the session object has also changed:



Using Cookies in Your Servlet

For small amounts of data that are to be persistent, you can create and use your own cookies, assuming cookies are enabled. Cookies are represented by instances of the `javax.servlet.http.Cookie` class, and are passed between client and server as attributes of the HTTP request and response headers.

A servlet can do the following with cookies:

- Create a cookie with specified name and value, using the `Cookie` class constructor
- Add a cookie to the response, using the `addCookie()` method of the response object.
- Retrieve cookies from the request, using the `getCookies()` method of the request object
- Get the name of a cookie, and get or set the value and other information of the cookie, using methods of the `Cookie` class (which are summarized shortly)

The following sections provide details, and a complete example, about how to use cookies:

- Configuring Cookies
- Summary of Cookie Methods
- Retrieving, Displaying, and Adding Cookies
- Cookie Example

Note: If cookies are enabled, OC4J automatically uses a cookie in keeping track of your session ID, as discussed in "How OC4J Can Use Cookies for Session Tracking" on page 3-3.

Configuring Cookies

Cookies are enabled by default. However, as described in "Configuring Session Tracking and Enabling or Disabling Cookies in OC4J" on page 3-3, you can explicitly enable them or disable them by setting `cookies="enabled"` or `cookies="disabled"` in the `<session-tracking>` element of the `global-web-application.xml` or `orion-web.xml` file.

The `<session-tracking>` element has additional cookie settings as well:

- `cookie-domain`: This is the desired domain for cookies. In general, this would be used 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 parts, such as "`.us.oracle.com`" or "`.oracle.com`". A setting of "`.com`", for example, is illegal.

  Cookie domain functionality can be used, for example, to share session state between nodes of a Web application running on different hosts.

  For a particular cookie, you can override the `cookie-domain` setting through the `setDomain()` method of the cookie.

- `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 then discarded, so that it is not persistent.

  For a particular cookie, you can override the `cookie-max-age` setting through the `setMaxAge()` method of the cookie. See the next section, "Summary of Cookie Methods", for additional information.

Also see "<session-tracking>" on page B-27.

> **Note:** You can configure cookies through attributes of the `sessionTracking` property in the Application Server Control deployment plan editor, as discussed in the *Oracle Containers for J2EE Deployment Guide*.

Summary of Cookie Methods

Use the `Cookie` constructor to create a new cookie:

- `Cookie(String name, String value)`

  Create a cookie with the specified name and value, as Java strings.

Use `Cookie` getter and setter methods, including the following, to specify or retrieve information about the cookie:

- `String getName()`

  Returns the name of the cookie.

- `void setValue(String value)`

Specify a new value for the cookie, as a Java string.

- `String getValue()`

  Returns the current value of the cookie, as a Java string.

- `void setDomain(String value)`

  By default, cookies are only returned to the server that sent them, but you can use this method to specify a domain, or Domain Name System zone, in which the cookie is visible. See "Configuring Cookies" on page 3-11 for information about using cookie domains in OC4J.

- `String getDomain()`

  Returns the domain in which the cookie is visible.

- `void setMaxAge(int maxAge)`

  Specify a maximum age for the cookie, in seconds, after which it will expire. A value of 0 causes the cookie to be deleted immediately. For a value of -1, the cookie will exist until browser shutdown, so will not be stored persistently. The default is -1, or as otherwise specified in your OC4J configuration. See "Configuring Cookies" on page 3-11.

- `int getMaxAge()`

  Returns the maximum age of the cookie before expiration, in seconds, or special values as described for `setMaxAge()`.

- `void setComment(String comment)`

  Create a comment to provide information about the cookie.

- `String getComment()`

  Returns a comment that provides information about the cookie (or `null` if there is no comment).

For complete information about `Cookie` methods, refer to the Sun Microsystems Javadoc for the `javax.servlet.http` package at the following location:

`http://java.sun.com/j2ee/1.4/docs/api/index.html`

Retrieving, Displaying, and Adding Cookies

Here are some typical steps in creating and displaying cookies. These steps are shown in a complete example in "Cookie Example" on page 3-13.

1. To retrieve an array of `javax.servlet.http.Cookie` objects containing the current set of cookies in the request, call the `getCookies()` method of the request object:

   ```
   Cookie[] cookies = request.getCookies();
   ```

2. To output the cookie names and values, iterate through the array (or notify the user if the array is empty):

   ```
   if (cookies != null && cookies.length > 0) {
      for (int i = 0; i < cookies.length; i++) {
         Cookie cookie = cookies[i];
         out.print("Cookie name: " + cookie.getName() + "<br>");
         out.println("Cookie value: " + cookie.getValue() + "<br><br>");
      }
   } else {
      out.println("There are no cookies.");
   ```

```
}
```

3. To create a new cookie, you can take user input to specify the name and value. For this example, assume the servlet has a form that takes the name and value and stores them in request parameters called `cookiename` and `cookievalue`. (The form is shown in the complete example.)

```
String cookieName = request.getParameter("cookiename");
String cookieValue = request.getParameter("cookievalue");
```

4. To add the new cookie to the response, first input the specified name and value to the `Cookie` constructor to create the cookie, then call the `addCookie()` method of the response object:

```
if (cookieName != null && cookieValue != null) {
   Cookie cookie = new Cookie(cookieName, cookieValue);
   response.addCookie(cookie);
}
```

Cookie Example

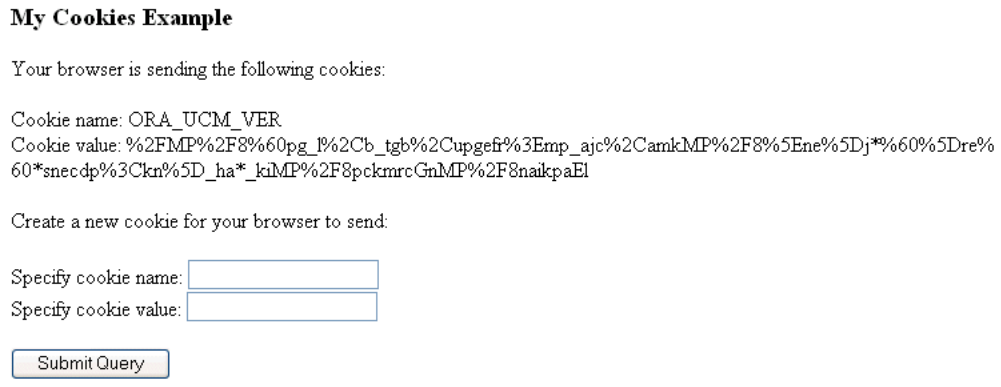
This example puts together the steps in "Retrieving, Displaying, and Adding Cookies" on page 3-12. There is also code for a form to take user input for cookie names and values.

As in previous examples, the `doGet()` method is called through the `doPost()` method, for a `POST` request.

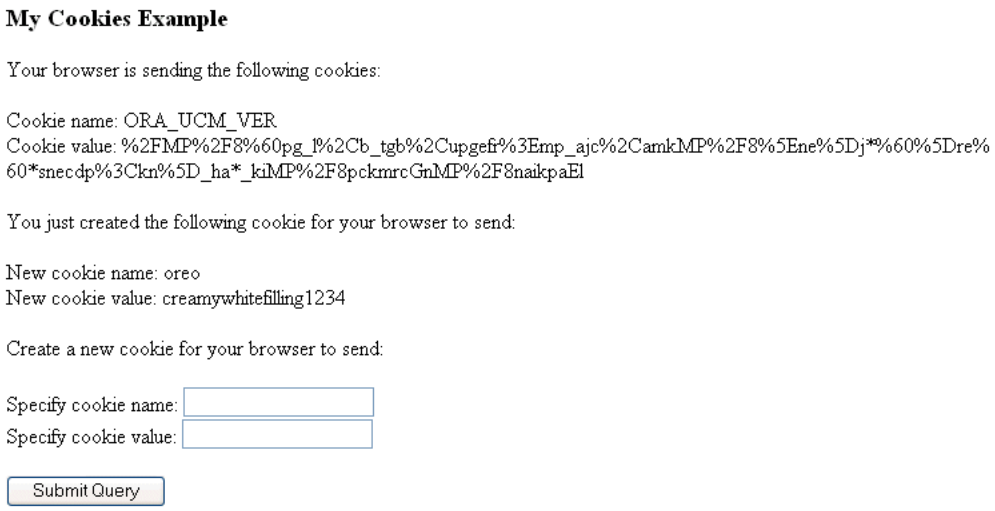
```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CookieExample extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body bgcolor=\"white\">");

        out.println("<h3>" + "My Cookies Example" + "</h3>");

/* Show cookies currently in request. */

        Cookie[] cookies = request.getCookies();
        if (cookies != null && cookies.length > 0) {
            out.println("Your browser is sending the following cookies:<br><br>");
            for (int i = 0; i < cookies.length; i++) {
                Cookie cookie = cookies[i];
                out.print("Cookie name: " + cookie.getName() + "<br>");
                out.println("Cookie value: " + cookie.getValue() + "<br><br>");
            }
        } else {
            out.println("There are no cookies.");
        }
```

```
/* Add new cookie that was just specified by user (form code below), and output
   back to the browser to confirm what the user just added.                  */

        String cookieName = request.getParameter("cookiename");
        String cookieValue = request.getParameter("cookievalue");
        if (cookieName != null && cookieValue != null) {
            Cookie cookie = new Cookie(cookieName, cookieValue);
            response.addCookie(cookie);
            out.println("<P>");
            out.println
      ("You just created the following cookie for your browser to send:<br><br>");
            out.print("New cookie name: " + cookieName + "<br>");
            out.print("New cookie value: " + cookieValue + "<br>");
        }

/* Form to prompt user for a cookie name and value. */

        out.println("<P>");
        out.println("Create a new cookie for your browser to send:<br>");
        out.print("<form action=\"");
        out.println("CookieExample\" method=POST>");
        out.print("Specify cookie name: ");
        out.println("<input type=text length=20 name=cookiename><br>");
        out.print("Specify cookie value: ");
        out.println("<input type=text length=20 name=cookievalue><br><br>");
        out.println("<input type=submit></form>");

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

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

}
```

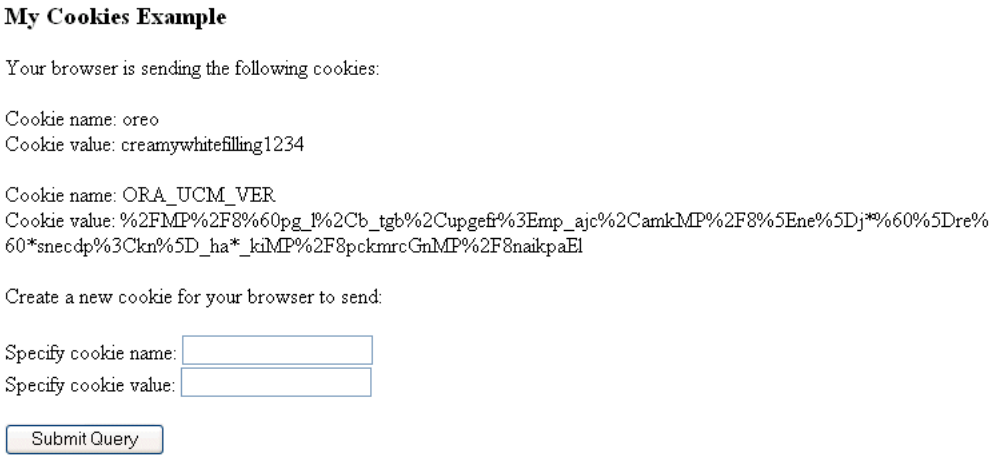
When you first run the servlet, it outputs the names and values of any cookies that already exist in the request, then prompts you to add a new cookie. In the following sample output, there is already one cookie:



As soon as you specify a cookie name and value and click **Submit Query**, the servlet confirms what you entered:



If you reload the servlet, you will see your new cookie included in the list of cookies currently in the request:



Canceling a Session

HTTP session objects generally persist for the duration of the server-side session; however, a session can be terminated explicitly by the servlet, or can be canceled by the servlet container after a certain expiration period.

Using a Timeout to Cancel a Session

The default session timeout for the OC4J server is 20 minutes. You can change the default for a specific application by setting the `<session-timeout>` subelement under the `<session-config>` element in `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.

To override the OC4J timeout or any `<session-timeout>` setting, for a particular servlet, you can use the `setMaxInactiveInterval()` method of the session object. This method is specified in the `HttpSession` interface. Specify an integer value, but note that with this method, you specify the timeout value in *seconds*, not minutes. Again, a negative value means there is no timeout.

The following example specifies a timeout of 10 minutes:

```
HttpSession session = request.getSession();
...
session.setMaxInactiveInterval(600);
```

The `getMaxInactiveInterval()` method returns an integer indicating the timeout interval, in seconds. This would be the value set in `setMaxInactiveInterval()`, if applicable. Otherwise, it would be the value specified in `<session-timeout>` (converted to seconds), if applicable. If neither `setMaxInactiveInterval()` nor `<session-timeout>` has been used, `getMaxInactiveInterval()` will return the OC4J default timeout value in seconds: 1200.

Explicitly Canceling a Session

A servlet can explicitly cancel a session by invoking the `invalidate()` method on the session object, as in the following example:

```
HttpSession session = request.getSession();
...
session.invalidate();
```

This immediately invalidates the session and unbinds any objects that were bound to it.

4

Understanding and Using 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.

Alternatively, you can use servlet filters to preprocess Web application requests and postprocess server responses. Filters were introduced in "When to Use Filters for Pre-Processing and Post-Processing" on page 6-16, and are described in the following sections:

- Overview of How Filters Work
- Standard Filter Interfaces
- Implementing and Configuring Filters
- Simple Filter Example
- Filtering Forward or Include Targets
- Using a Filter to Wrap and Alter the Request or Response
- Response Filter Example

Overview of How Filters Work

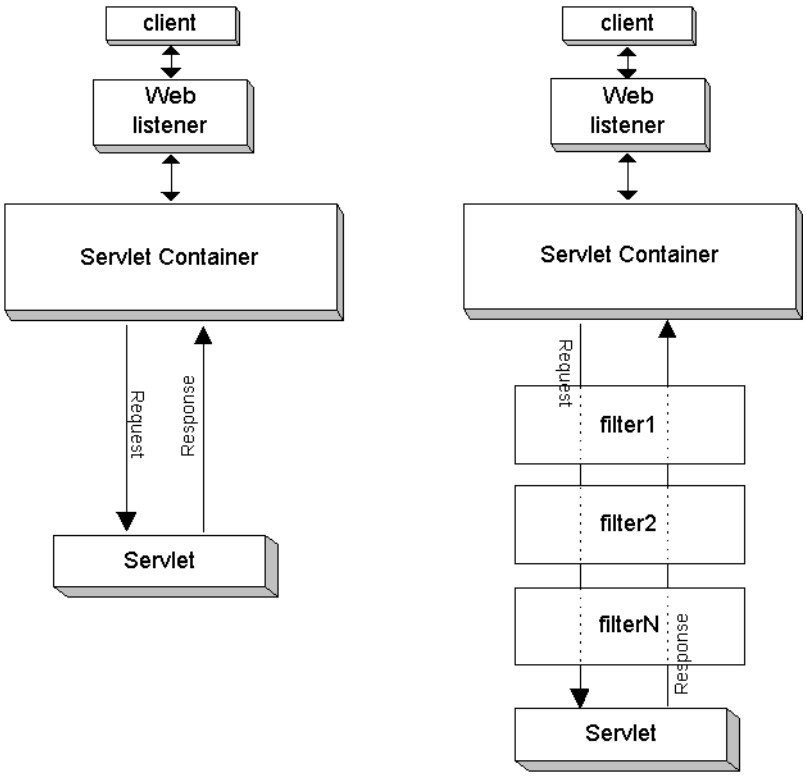
This section provides an overview of the following topics:

- How the Servlet Container Invokes Filters
- Typical Filter Actions

How the Servlet Container Invokes Filters

Figure 4–1 shows, on the left, a scenario in which no filters are configured for the servlet being requested. On the right, several filters (1, 2,..., N) have been configured.

Figure 4–1 Servlet Invocation with and without Filters



Each filter implements the `javax.servlet.Filter` interface, which includes a `doFilter()` method that takes as input a request and response pair along with a filter chain, which is an instance of a class (provided by the servlet container) that implements the `javax.servlet.FilterChain` interface. The filter chain reflects the order of the filters. The servlet container, based on the configuration order in the `web.xml` file, constructs the chain of filters for any servlet or other resource that has filters mapped to it. For each filter in the chain, the filter chain object passed to it represents the remaining filters to be called, in order, followed by the target servlet.

The `FilterChain` interface also specifies a `doFilter()` method, which takes a request and response pair as input and is used by each filter to invoke the next entity in the chain.

Also see "Standard Filter Interfaces" on page 4-3.

If there are two filters, for example, the key steps of this mechanism would be as follows:

1. The target servlet is requested. The container detects that there are two filters and creates the filter chain.
2. The first filter in the chain is invoked by its `doFilter()` method.
3. The first filter completes any preprocessing, then calls the `doFilter()` method of the filter chain. This results in the second filter being invoked by its `doFilter()` method.
4. The second filter completes any preprocessing, then calls the `doFilter()` method of the filter chain. This results in the target servlet being invoked by its `service()` method.

5. When the target servlet is finished, the chain `doFilter()` call in the second filter returns, and the second filter can do any postprocessing.
6. When the second filter is finished, the chain `doFilter()` call in the first filter returns, and the first filter can do any postprocessing.
7. When the first filter is finished, execution is complete.

None of the filters are aware of their order. Ordering is handled entirely through the filter chain, according to the order in which filters are configured in `web.xml`.

Typical Filter Actions

The following are among the possible actions of the `doFilter()` method of a filter:

- Create a wrapper for the request object to allow input filtering. Process the content or headers of the request wrapper as desired.
- Create a wrapper for the response object to allow output filtering. Process the content or headers of the response wrapper as desired.
- Pass the request and response pair (or wrappers) to the next entity in the chain, using the chain `doFilter()` method. Alternatively, to block request processing, do *not* call the chain `doFilter()` method.

Any processing you want to occur before the target resource is invoked must be prior to the chain `doFilter()` call. Any processing you want to occur after the completion of the target resource must be after the chain `doFilter()` call. This can include directly setting headers on the response.

Note that if you want to preprocess the request object or postprocess the response object, you cannot directly manipulate the original request or response object. You must use wrappers. When postprocessing a response, for example, the target servlet has already completed and the response could already be committed by the time a filter would have a chance to do anything with the response. You must pass a response wrapper instead of the original response in the chain `doFilter()` call. See "Using a Filter to Wrap and Alter the Request or Response" on page 4-10.

Standard Filter Interfaces

A servlet filter implements the `javax.servlet.Filter` interface. The main method of this interface, `doFilter()`, takes a `javax.servlet.FilterChain` instance, created by the servlet container to represent the entire chain of filters, as input. The initialization method of the `Filter` interface, `init()`, takes a filter configuration object, which is an instance of `javax.servlet.FilterConfig`, as input. This section briefly describes the methods specified in these interfaces.

For additional information about the interfaces and methods discussed here, refer to the Sun Microsystems Javadoc for the `javax.servlet` package, at:

`http://java.sun.com/j2ee/1.4/docs/api/index.html`

Methods of the Filter Interface

The `Filter` interface specifies the following methods to implement in your filters:

- `void init(FilterConfig filterConfig)`

  The servlet container calls `init()` as a filter is first instantiated and placed into service. This method takes a `javax.servlet.FilterConfig` instance as input, which the servlet container uses to pass information to the filter during the

initialization. Include any special initialization requirements in your implementation. Also see "Methods of the FilterConfig Interface" on page 4-4.

- `void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)`

  This is where your filter processing occurs. Each time a target resource (such as a servlet or JSP page) is requested, where the target resource is mapped to a chain of one or more filters, the servlet container calls the `doFilter()` method of each filter in the chain, in order according to `web.xml` filter configurations. (See "Construction of the Filter Chain" on page 4-7.) Within the `doFilter()` processing of a filter, invoke the `doFilter()` method on the filter chain object that is passed in to the `doFilter()` method of the filter. (An exception to this is if you want to block request processing.) This is what leads to invocation of the next entity in the chain (either the next filter, or the target servlet if this is the last filter in the chain) after a filter has completed.

- `destroy()`: The servlet container calls `destroy()` after all execution of the filter has completed (all threads of the `doFilter()` method have completed, or a timeout has occurred) and the filter is being taken out of service. Include any special cleanup requirements in your implementation.

Method of the FilterChain Interface

The `FilterChain` interface specifies one method:

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

  Invoking this method, which you do from the `doFilter()` method of a filter, causes the next entity in the chain to be invoked—either the next filter, or the target resource (such as a servlet or JSP page) if this method is called from the last filter in the chain.

Methods of the FilterConfig Interface

The `FilterConfig` interface specifies the following methods, available for your optional use:

- `java.util.Enumeration getInitParameterNames()`

  You can set initialization parameters for a filter through `<init-param>` elements under the `<filter>` element in the `web.xml` file. (See "Configure the Filter" on page 4-6.) Then, in your filter, you can use the `getInitParameterNames()` method of the `FilterConfig` object, which is passed in through the `init()` method, to retrieve an `Enumeration` object of Java strings containing the names of the initialization parameters. (The `Enumeration` object is empty if there are no initialization parameters for the filter.)

- `String getInitParameter(String paramname)`

  After retrieving initialization parameter names, use `getInitParameter()` to retrieve the value of a specified parameter.

- `ServletContext getServletContext()`

  You can use this method to retrieve the servlet context associated with the requested servlet (which the filter is filtering).

- `String getFilterName()`

You can use this method to retrieve the name of the filter, according to the `<filter-name>` element in the `web.xml` file.

Implementing and Configuring Filters

This section shows the basic steps of implementing and configuring a filter. Steps such as these are included in a complete sample in "Simple Filter Example" on page 4-7.

There is a subsection describing construction of the filter chain, based on your filter configuration order in `web.xml`.

Implement the Filter Code

This section lists steps in implementing code for a servlet filter.

1. Create a class that implements the `javax.servlet.Filter` interface. For example:

```
public class TimerFilter implements javax.servlet.Filter { }
```

2. For initialization of your filter, implement the `init()` method, specified in the `Filter` interface. First, create or retrieve a `javax.servlet.FilterConfig` object, which `init()` takes as input. For example:

```
private FilterConfig filterConfig;
...
public void init(final FilterConfig filterConfig)
{
    this.filterConfig = filterConfig;
}
```

   In case you want any special initialization processing, see "Methods of the FilterConfig Interface" on page 4-4.

3. For your filter processing, implement the `doFilter()` method, specified in the `Filter` interface. This method takes a request object, a response object, and a `javax.servlet.FilterChain` object created by the servlet container. Implement whatever processing you want, and (typically) call the `doFilter()` method of the filter chain object to invoke the next entity in the chain. For example:

```
public void doFilter(ServletRequest request, ServletResponse response,
       FilterChain chain)
       throws java.io.IOException, javax.servlet.ServletException
{
    long start = System.currentTimeMillis();
    System.out.println("Milliseconds in: " + start);
    chain.doFilter(request, response);
    long end = System.currentTimeMillis();
    System.out.println("Milliseconds out: " + end);
}
```

   The first `println()` call is executed before the rest of the chain is invoked; the second `println()` call is executed afterward, when `chain.doFilter()` returns.

4. Implement the `destroy()` method, specified in the `Filter` interface, to clean up resources or do anything special before the filter is taken out of service. For example:

```
public void destroy()
```

```
{
    filterConfig = null;
}
```

Note: There are additional considerations in implementing a filter to alter the HTTP request or response. See "Using a Filter to Wrap and Alter the Request or Response" on page 4-10.

Configure the Filter

This section lists the steps in configuring a servlet filter. Do the following in `web.xml` for each filter:

1. Declare the filter through a `<filter>` element and its subelements, which maps the filter class (including package) to a filter name. For example:

   ```
   <filter>
       <filter-name>timer</filter-name>
       <filter-class>filter.TimerFilter</filter-class>
   </filter>
   ```

   You can optionally specify initialization parameters here, similarly to how you would for a servlet:

   ```
   <filter>
       <filter-name>timer</filter-name>
       <filter-class>filter.TimerFilter</filter-class>
       <init-param>
           <param-name>name</param-name>
           <param-value>value</param-value>
       <init-param>
   </filter>
   ```

2. Using a `<filter-mapping>` element and its subelements, map the filter name to a servlet name or URL pattern to associate the filter with the corresponding resource (such as a servlet or JSP page) or resources. For example, to have the filter invoked whenever the servlet of name `myservlet` is invoked:

   ```
   <filter-mapping>
       <filter-name>timer</filter-name>
       <servlet-name>myservlet</servlet-name>
   </filter-mapping>
   ```

   Or, to have the filter invoked whenever `sleepy.jsp` is requested, according to URL pattern:

   ```
   <filter-mapping>
       <filter-name>timer</filter-name>
       <url-pattern>/sleepy.jsp</url-pattern>
   </filter-mapping>
   ```

   Note that instead of specifying a particular resource in the `<url-pattern>` element, you can use wild card characters to match multiple resources, such as in the following example:

   ```
   <url-pattern>/mypath/*</url-pattern>
   ```

The filter name can be arbitrary, but preferably is meaningful. It is simply used as the linkage in mapping a filter class to a servlet name or URL pattern.

If you configure multiple filters that apply to a resource, they will be entered in the servlet chain according to their declaration order in `web.xml`, and they will be invoked in that order when the target servlet is requested. See the next section, "Construction of the Filter Chain".

> **Note:** There are additional steps to configure a filter for a forward or include target. See "Filtering Forward or Include Targets" on page 4-9.

Construction of the Filter Chain

When you declare and map filters in `web.xml`, the servlet container determines which filters apply to each servlet or other resource (such as a JSP page or static page) in the Web application. Then, for each servlet or resource, the servlet container builds a chain of applicable filters, according to your `web.xml` configuration order, as follows:

1. First, any filters that match a servlet or resource according to a `<url-pattern>` element are placed in the chain, in the order in which the filters are declared in `web.xml`.
2. Next, any filters that match a servlet or resource according to a `<servlet-name>` element are placed in the chain, with the first `<servlet-name>` match following the last `<url-pattern>` match.
3. Finally, the target servlet or other resource is placed at the end of the chain, following the last filter with a `<servlet-name>` match.

Simple Filter Example

This example shows a filter that is invoked when a JSP page is requested. The JSP page writes a line to the browser. The filter writes two lines to the OC4J console—one line before the JSP page runs, and one after.

Write the Simple Filter Code

Here is the code for the simple filter, `TimerFilter`. The `doFilter()` method writes two lines to the OC4J console, one before the target JSP page is executed and one after.

```
package filter;

import javax.servlet.*;

public class TimerFilter implements javax.servlet.Filter
{
    private FilterConfig filterConfig;

    public void doFilter(ServletRequest request, ServletResponse response,
           FilterChain chain)
           throws java.io.IOException, javax.servlet.ServletException
    {
        long start = System.currentTimeMillis();
        System.out.println("Milliseconds in: " + start);
        chain.doFilter(request, response);
        long end = System.currentTimeMillis();
        System.out.println("Milliseconds out: " + end);
    }

    public void init(final FilterConfig filterConfig)
    {
```

```
        this.filterConfig = filterConfig;
    }

    public void destroy()
    {
        filterConfig = null;
    }
}
```

Write the Target JSP Page

Here is the target JSP page, `sleepy.jsp`, which has a wait interval then outputs the wait time to the browser.

```
<%
  int sleeptime = 1000;
  Thread.sleep(sleeptime);
%>
<HTML>
<HEAD>
<TITLE>Sleepy JSP</TITLE>
</HEAD>
<BODY>
<HR>
<P><CENTER>Sleepy JSP slept for <%= sleeptime %> milliseconds!</CENTER></P>
<HR>
</BODY>
</HTML>
```

Configure the Simple Filter

Here is the configuration in `web.xml` that declares the simple filter and maps it to requests for `sleepy.jsp`:

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<web-app>
   <filter>
      <filter-name>timer</filter-name>
      <filter-class>filter.TimerFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>timer</filter-name>
      <url-pattern>/sleepy.jsp</url-pattern>
   </filter-mapping>
</web-app>
```

Package the Simple Filter Example

The WAR file for this example, which we name `filter.war`, has the following contents and structure:

```
sleepy.jsp
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/filter/TimerFilter.class
WEB-INF/classes/filter/TimerFilter.java
```

And the EAR file is as follows:

```
filter.war
```

```
META-INF/Manifest.mf
META-INF/application.xml
```

(The `Manifest.mf` files are created automatically by the JAR utility.)

Invoke the Simple Filter Example

For this example, assume that `application.xml` maps the context path `/myfilter` to `filter.war`. In this case, after deployment, you invoke `sleepy.jsp` as follows, and the filter is executed as a result:

```
http://host:port/myfilter/sleepy.jsp
```

The following is written to the browser:

```
Sleepy JSP slept for 1000 milliseconds!
```

In a sample execution, the following is written to the OC4J console (where you started OC4J, for example, if you are running OC4J in a standalone environment):

```
04/04/30 13:01:19 Milliseconds in: 1083355279136
04/04/30 13:01:20 Milliseconds out: 1083355280152
```

The "Milliseconds in" line is written before the JSP page is invoked. The "Milliseconds out" line is written after the JSP page is done and execution returns to the filter. In this example, there is a difference of 1016 milliseconds, mostly due to the 1000-millisecond wait in the JSP page.

Filtering Forward or Include Targets

You can configure a filter to act on forward or include targets in addition to, or instead of, acting on direct request targets. This is described in the following subsections:

- The web.xml <dispatcher> Element
- Configuring Filters for Forward or Include Targets

> **Note:** See "Dispatching to Other Servlets Through Includes and Forwards" on page 6-12 for general information about including and forwarding.

The web.xml <dispatcher> Element

Use the `<dispatcher>` subelement of `<filter-mapping>` in `web.xml` if you want to configure filters for forward or include targets. This element has four supported values:

- `INCLUDE`: Use this for the filter to be applied to any include targets matching a specified servlet name or with URLs matching a specified pattern.
- `FORWARD`: Use this for the filter to be applied to any forward targets matching a specified servlet name or with URLs matching a specified pattern.
- `REQUEST`: Use this in addition to an `INCLUDE` or `FORWARD` setting (one `<dispatcher>` element for each setting) for the filter to also be applied to direct request targets matching a specified servlet name or with URLs matching a specified pattern. (It is nonsensical to use the `REQUEST` value only. If you want the filter to apply only to direct requests, there is no need to use the `<dispatcher>` element.)

- ERROR: Use this for the filter to be applied under the error page mechanism.

See the following section, "Configuring Filters for Forward or Include Targets", for examples.

Configuring Filters for Forward or Include Targets

This section provides a few sample configurations to have a filter act on forward or include targets. We start with the filter declaration, followed by alternative filter mapping configurations:

```
<filter>
    <filter-name>myfilter</filter-name>
    <filter-class>mypackage.MyFilter</filter-class>
</filter>
```

To execute `MyFilter` to filter an include target named `includedservlet`:

```
<filter-mapping>
   <filter-name>myfilter</filter-name>
   <servlet-name>includedservlet</servlet-name>
   <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

Note that the `include()` call can come from any servlet (or other resource) in the application. Also note that `MyFilter` would not execute for a direct request of `includedservlet`, unless you have another `<dispatcher>` element with the value `REQUEST`.

To execute `MyFilter` to filter any servlet directly requested through a URL pattern "`/mypath/`", or to execute it to filter any forward target that is invoked through a URL pattern starting with "`/mypath/`":

```
<filter-mapping>
   <filter-name>myfilter</filter-name>
   <url-pattern>/mypath/*</url-pattern>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Using a Filter to Wrap and Alter the Request or Response

Particularly useful functions for a filter are to manipulate a request, or manipulate the response to a request. To manipulate a request or response, you must create a wrapper. You can use the following general steps:

1. To manipulate requests, create a class that extends the standard `javax.servlet.http.HttpServletRequestWrapper` class. This class will be your request wrapper, allowing you to modify a request as desired.
2. To manipulate responses, create a class that extends the standard `javax.servlet.http.HttpServletResponseWrapper` class. This class will be your response wrapper, allowing you to modify a response after the target servlet or other resource has delivered and possibly committed it.
3. Optionally create a class that extends the standard `javax.servlet.ServletOutputStream` class, if you want to add custom functionality to an output stream for the response.
4. Create a filter that uses instances of your custom classes to alter the request or response as desired.

The next section, "Response Filter Example", provides an example of a filter that alters the response.

Response Filter Example

This example employs an HTTP servlet response wrapper that uses a custom servlet output stream. This functionality allows the wrapper to manipulate the response data after the target HTML page is finished writing it out. Without using a wrapper, you cannot change the response data after the servlet output stream has been closed (essentially, after the servlet has committed the response). That is the reason for implementing a filter-specific extension to the `ServletOutputStream` class in this example.

This example uses the following custom classes:

- `GenericResponseWrapper`: Extends `HttpServletResponseWrapper` for custom functionality in manipulating an HTTP response.
- `FilterServletOutputStream`: Extends `ServletOutputStream` to provide custom functionality for use in the response wrapper.
- `MyGenericFilter`: This class is for a generic, empty ("pass-through") filter that is used as a base class.
- `PrePostFilter`: Extends `MyGenericFilter` and implements `doFilter()` code to alter the HTTP response, inserting a line before the HTML page output and a line after the HTML page output.

Write the Custom Output Stream Code

This class, `FilterServletOutputStream`, extends the standard `ServletOutputStream` class to implement special functionality that the response wrapper will use.

```
package mypkg;

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);
  }
```

```
}
```

Write the Response Wrapper Code

This class, `GenericResponseWrapper`, extends the standard `HttpServletResponseWrapper` class to implement custom functionality to manipulate an HTTP response.

```
package mypkg;

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;
  }
}
```

Write the Base Filter Code

This class, `MyGenericFilter`, is an empty filter, providing a template that is extended by the response filter of this example.

```
package mypkg;

import javax.servlet.*;

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

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

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

  public void destroy() {
  }
}
```

Write the Response Filter Code

This class, `PrePostFilter`, which extends `MyGenericFilter`, is a filter that alters the response of the target HTML page, prepending a line of output before the HTML page output, and appending a line of output after the HTML page output.

```
package mypkg;

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(new String("<HR>PRE<HR>").getBytes());
  GenericResponseWrapper wrapper =
         new GenericResponseWrapper((HttpServletResponse) response);
  chain.doFilter(request,wrapper);
  out.write(wrapper.getData());
  out.write(new String("<HR>POST<HR>").getBytes());
  out.close();
  }
}
```

Write the Target HTML Page

This HTML page, `prepostfilter.html`, is requested by the user in this example. The filter inserts output before and after the output of this page.

```
<HTML>
<HEAD>
<TITLE>PrePostFilter Example</TITLE>
</HEAD>
```

```
<BODY>
This is a testpage. You should see<br>
this text when you invoke prepostfilter.html, <br>
as well as the additional material added<br>
by the PrePostFilter class.
<br>
</BODY>
</HTML>
```

Configure the Response Filter

Here is the configuration in `web.xml` that declares the response filter and maps it to requests for `prepostfilter.html`:

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<web-app>
   <filter>
      <filter-name>prepost</filter-name>
      <filter-class>mypkg.PrePostFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>prepost</filter-name>
      <url-pattern>/prepostfilter.html</url-pattern>
   </filter-mapping>
</web-app>
```

Package the Response Filter Example

The WAR file for this example, which we name `myresponsewrapper.war`, has the following contents and structure:

```
prepostfilter.html
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/mypkg/FilterServletOutputStream.class
WEB-INF/classes/mypkg/FilterServletOutputStream.java
WEB-INF/classes/mypkg/GenericResponseWrapper.class
WEB-INF/classes/mypkg/GenericResponseWrapper.java
WEB-INF/classes/mypkg/MyGenericFilter.class
WEB-INF/classes/mypkg/MyGenericFilter.java
WEB-INF/classes/mypkg/PrePostFilter.class
WEB-INF/classes/mypkg/PrePostFilter.java
```

And the EAR file is as follows:

```
myresponsewrapper.war
META-INF/application.xml
META-INF/Manifest.mf
```

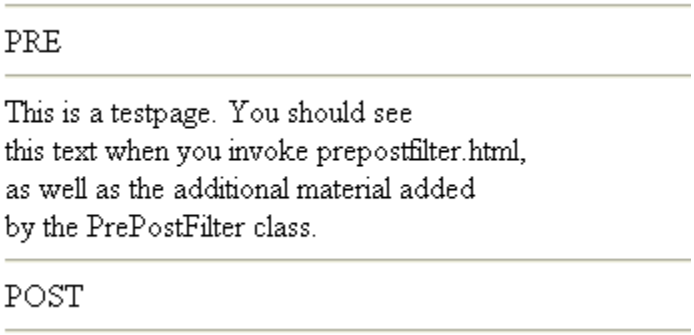
(The `Manifest.mf` files are created automatically by the JAR utility.)

Invoke the Response Filter Example

For this example, assume that `application.xml` maps the context path `/mywrapper` to `myresponsewrapper.war`. In this case, after deployment, you invoke `prepostfilter.html` as follows, and the filter is executed as a result:

```
http://host:port/mywrapper/prepostfilter.html
```

The following is output to the browser, with "PRE", "POST", and the horizontal rules coming from the filter:



Form Authentication Filter

An OC4J proprietary filter dispatcher was introduced in 10.1.3 that enables a filter to access the username/password credentials passed in to OC4J through Form Authentication. For example, you would do this if you want to perform further authentication on external resources.

To enable this feature, specify the `FORMAUTH` value in the `<dispatcher>` element for the filter in the `orion-web.xml` file.

The following examples show the code that declares the filter and the code that specifies the `FORMAUTH` feature:

> **Note:**
>
> The `<filter>` element can be declared in either the `orion-web.xml` file or the `web.xml` file.
>
> The `<filter-mapping>` element must be declared in the `orion-web.xml` file.

In `orion-web.xml` or `web.xml`:

```
<filter>
   <filter-name>MyFilter</filter-name>
   <filter-class>myFilterClass</filter-class>
</filter>
```

In orion-web.xml:

```
<orion-web-app>
...
   <web-app>
   ...
      <filter-mapping>
         <filter-name>MyFilter</filter-name>
         <dispatcher>FORMAUTH</dispatcher>
      </filter-mapping>
   </web-app>
</orion-web-app>
```

Any filter declared this way will be executed after the authentication form is submitted, but before authentication is performed in OC4J.

The filter can call `request.getParameters()` to get the `j_username` and `j_password` parameters from the request object.

Here is an example of the calling code:

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

public class MyFilter implements Filter {

    public MyFilter() {
        super();
    }

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain filterChain)
        throws IOException, ServletExxception {
        HttpServletRequest req = (HttpServletRequest) request;

String username = req.getParameter("j_username");
String password = req.getParameter("j_password");
// use these credentials to access a remote DB
....
filterChain.doFilter(request, response);
    }

    public void init(FilterConfig filterConfig) throws ServletException {
    }

    public void destroy() {
    }
}
```

5

Understanding and Using Event Listeners

The servlet specification includes features to track key events in your Web applications through *event listeners*, which were introduced in "When to Use Event Listeners for Servlet Notification" on page 6-17. You can use listeners for automated processing and more efficient resource management based on event status. You can implement listeners for request events, session events, and servlet context events. This is described in the following sections:

- Overview of How Event Listeners Work
- Event Listener Interfaces
- Implementing and Configuring Event Listeners
- Session Lifecycle Listener Example

Overview of How Event Listeners Work

There are eight event listener categories:

- Servlet context lifecycle (startup or shutdown)
- Servlet context attribute changes (adding, deleting, or replacing)
- Session lifecycle (startup, invalidation, or timeout)
- Session attribute changes (adding, deleting, or replacing)
- Session migration (activation or passivation)
- Session object binding (objects being bound to or unbound from a session)
- Request lifecycle (start of request processing)
- Request attribute changes (adding, deleting, or replacing)

You can create one or more event listener classes for each of these event categories. Or, 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` or `javax.servlet.http` package. Table 5–1 summarizes the categories and their associated interfaces.

Table 5–1 Event Listener Categories and Interfaces

Event Category	Event Descriptions	Java Interface
Servlet context lifecycle	Servlet context creation, at which point the first request can be serviced	javax.servlet. ServletContextListener
	Imminent shutdown of the servlet context	
Servlet context attributes	Addition of servlet context attributes	javax.servlet. ServletContextAttributeListener
	Removal of servlet context attributes	
	Replacement of servlet context attributes	
Session lifecycle	Session creation	javax.servlet.http. HttpSessionListener
	Session invalidation	
	Session timeout	
Session attributes	Addition of session attributes	javax.servlet.http. HttpSessionAttributeListener
	Removal of session attributes	
	Replacement of session attributes	
Session migration	Session activation	javax.servlet. HttpSessionActivationListener
	Session passivation	
Session object binding	Object bound to session	javax.servlet. HttpSessionBindingListener
	Object unbound from session	
Request lifecycle	Start of request processing	javax.servlet. ServletRequestListener
Request attributes	Addition of session attributes	javax.servlet. ServletRequestAttributeListener
	Removal of session attributes	
	Replacement of session attributes	

Configure a listener through a `<listener>` element (subelement of the `<web-app>` element) in the `web.xml` file. See "Configure the Listener" on page 5-7.

After your application starts up and before it services the first request, the servlet container creates and registers an instance of each listener class that is declared in `web.xml`. 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.

Event Listener Interfaces

This section documents methods of the interfaces that are summarized in Table 5–1. Each method described here is called by the servlet container when the appropriate 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. Implement this interface in a class you will use for tracking servlet context lifecycle events.

- `void contextInitialized(ServletContextEvent sce)`

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

- `void contextDestroyed(ServletContextEvent sce)`

  The servlet container calls this method when 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 shut down, from which you can obtain information as desired. See "Servlet Contexts: the Application Container" on page 1-10 for information about the `javax.servlet.ServletContext` interface.

ServletContextAttributeListener Methods, ServletContextAttributeEvent Class

The `ServletContextAttributeListener` interface specifies the following methods. Implement this interface in a class you will use for tracking servlet context attribute events.

- `void attributeAdded(ServletContextAttributeEvent scae)`

  The servlet container calls this method when an attribute is added to the servlet context.

- `void attributeRemoved(ServletContextAttributeEvent scae)`

  The servlet container calls this method when an attribute is removed from the servlet context.

- `void attributeReplaced(ServletContextAttributeEvent scae)`

  The servlet container calls this method when an attribute is replaced (its value changed) in the servlet context.

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

- `String getName()`

  Use this 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.

HttpSessionListener Methods, HttpSessionEvent Class

The `HttpSessionListener` interface specifies the following methods. Implement this interface in a listener class you will use to track session lifecycle events.

- `void sessionCreated(HttpSessionEvent hse)`

  The servlet container calls this method when the session is created.

- `void sessionDestroyed(HttpSessionEvent hse)`

  The servlet container calls this method when the session is about to be terminated.

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 is about to be terminated, from which you can obtain information as desired.

HttpSessionAttributeListener Methods, HttpSessionBindingEvent Class

The `HttpSessionAttributeListener` interface specifies the following methods. Implement this interface in a listener class you will use to track session attribute events.

- `void attributeAdded(HttpSessionBindingEvent hsbe)`

  The servlet container calls this method when an attribute is added to the session.

- `void attributeRemoved(HttpSessionBindingEvent hsbe)`

  The servlet container calls this method when an attribute is removed from the session.

- `void attributeReplaced(HttpSessionBindingEvent hsbe)`

  The servlet container calls this method when an attribute is replaced (its value changed) 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.

HttpSessionActivationListener Methods

The `HttpSessionActivationListener` interface specifies the following methods. Implement this interface in a listener class you will use to track session migration (activation or passivation) events.

- `void sessionDidActivate(HttpSessionEvent hse)`

  The servlet container calls this method when the session is activated.

- `void sessionWillPassivate(HttpSessionEvent hse)`

  The servlet container calls this method when the session is about to be passivated.

The servlet container creates an instance of the `HttpSessionEvent` class to use as input for `HttpSessionActivationListener` method calls. See "HttpSessionListener Methods, HttpSessionEvent Class" on page 5-3 for information about this class.

HttpSessionBindingListener Methods

The `HttpSessionBindingListener` interface specifies the following methods. Implement this interface in a class whose instances will be bound to a session.

- `void valueBound(HttpSessionBindingEvent hsbe)`

  The servlet container calls this method when the object (that implements `HttpSessionBindingListener`) is bound to a session, which is identified.

- `void valueUnbound(HttpSessionBindingEvent hsbe)`

  The servlet container calls this method when the object is unbound from a session, which is identified. The object can be unbound explicitly, or as a result of session invalidation or timeout.

The servlet container creates an instance of the `HttpSessionBindingEvent` class to use as input for `HttpSessionBindingListener` method calls. See "HttpSessionAttributeListener Methods, HttpSessionBindingEvent Class" on page 5-4 for information about this class.

ServletRequestListener Methods, ServletRequestEvent Class

The `ServletRequestListener` interface specifies the following methods. Implement this interface in a listener class you will use to track request lifecycle events.

- `void requestInitialized(ServletRequestEvent sre)`

  The servlet container calls this method when the request is about to come into scope of the Web application.

- `void requestDestroyed(ServletRequestEvent sre)`

  The servlet container calls this method when the request is about to go out of scope of the Web application.

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

- `ServletRequest getServletRequest()`

  Use this method to retrieve the servlet request whose status is changing.

- `ServletContext getServletContext()`

  Use this method to retrieve the servlet context of the Web application.

ServletRequestAttributeListener Methods, ServletRequestAttributeEvent Class

The `ServletRequestAttributeListener` interface specifies the following methods. Implement this interface in a listener class you will use to track request attribute events.

- `void attributeAdded(ServletRequestAttributeEvent srae)`

  The servlet container calls this method when an attribute is added to the request.

- `void attributeRemoved(ServletRequestAttributeEvent srae)`

  The servlet container calls this method when an attribute is removed from the request.

- `void attributeReplaced(ServletRequestAttributeEvent srae)`

  The servlet container calls this method when an attribute is replaced (its value changed) in the request.

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

Implementing and Configuring Event Listeners

This section describes the basic steps of implementing and configuring listeners. See "Session Lifecycle Listener Example" on page 5-8 for a complete example.

Implement the Listener Code

A listener class can be used for any or all categories of events summarized in "Overview of How Event Listeners Work" on page 5-1. A single class can implement multiple listeners. Implement code as follows:

- For a servlet context lifecycle listener, implement the `ServletContextListener` interface and write code for the methods `contextInitialized()` (for actions at application startup) and `contextDestroyed()` (for actions at application shutdown), as appropriate.
- For a servlet context attribute listener, implement the `ServletContextAttributeListener` interface and write code for the methods `attributeAdded()` (for actions when an attribute is added), `attributeRemoved()` (for actions when an attribute is removed), and `attributeReplaced()` (for actions when an attribute value is changed), as appropriate.
- For a session lifecycle listener, implement the `HttpSessionListener` interface and write code for one the methods `sessionCreated()` (for actions at session creation) and `sessionDestroyed()` (for actions at session invalidation), as

appropriate. See "Write the Session Lifecycle Listener Code" on page 5-10 for an elementary example.

- For a session attribute listener, implement the `HttpSessionAttributeListener` interface and write code for the methods `attributeAdded()`, `attributeRemoved()`, and `attributeReplaced()`, as appropriate.
- For a session migration listener, implement the `HttpSessionActivationListener` interface and write code for the methods `sessionDidActivate()` (for actions at session activation) and `sessionWillPassivate()` (for actions at session passivation), as appropriate.
- For a session binding listener, implement the `HttpSessionBindingListener` interface and write code for the methods `valueBound()` (for actions when the object is bound to the session) or `valueUnbound()` (for actions when the object is unbound from the session), as appropriate.
- For a request lifecycle listener, implement the `ServletRequestListener` interface and write code for the methods `requestInitialized()` (for actions when the request comes into scope) and `requestDestroyed()` (for actions when the request goes out of scope), as appropriate.
- For a request attribute listener, implement the `ServletRequestAttributeListener` interface and write code for the methods `attributeAdded()`, `attributeRemoved()`, and `attributeReplaced()`, as appropriate.

Notes:

- 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.
- 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.

Configure the Listener

To configure each listener class, use a `<listener>` element (subelement of `<web-app>`) and its `<listener-class>` subelement in the application `web.xml` file:

```
<web-app>
   <listener>
      <listener-class>SessionLifeCycleEventExample</listener-class>
   </listener>
...
   <servlet>
      <servlet-name>name</servlet-name>
      <servlet-class>class</servlet-class>
   </servlet>
...
   <servlet-mapping>
```

```
      <servlet-name>name</servlet-name>
      <url-pattern>path</url-pattern>
   </servlet-mapping>
...
</web-app>
```

A listener is not associated with any particular servlet. At application startup, for each event category, the servlet container registers listeners in the order in which they are declared in `web.xml`. As the application runs, event listeners for each category are invoked in the order of their registration whenever an applicable event occurs. Listeners remain active until the last request for the application has been serviced.

Upon application shutdown, however, listeners are notified in the reverse order of their declarations, with request and session listeners being notified before servlet context listeners.

Physical File Required for Welcome File

A physical file must be present for a welcome file to dispatch to a servlet. To create a servlet mapped to `/index.html` that maps to the JSP page `/index.jsp` and have it serve as a welcome file, the `web.xml` file should include the following entries:

```
<servlet>
   <servlet-name> index_jsp </servlet-name>
   <jsp-file> /index.jsp </jsp-file>
</servlet>

<servlet-mapping>
   <servlet-name>index_jsp</servlet-name>
   <url-pattern>/index.html</url-pattern>
</servlet-mapping>
```

This works *only* if there is a physical file, `/index.html`, in the Web application. The file can be zero length. As long as the file exists, this servlet will be loaded as the welcome file. Otherwise, a `java.lang.StringIndexOutOfBoundsException` exception will be thrown.

Session Lifecycle Listener Example

This is an elementary example of a session lifecycle event listener that writes messages to the OC4J console whenever a session is created or terminated. There is code for the following components:

- `index.jsp`: This is the application welcome page, which has a link to invoke `SessionCreateServlet` to create an HTTP session.
- `SessionCreateServlet`: This servlet creates an HTTP session and has a link to `SessionDestroyServlet` to terminate the session.
- `SessionDestroyServlet`: This servlet terminates the HTTP session and has a link back to the welcome page.
- `SessionLifeCycleEventExample`: This is the event listener class, which implements the `HttpSessionListener` interface with code for the `sessionCreated()` and `sessionDestroyed()` methods to write a console message when a session is created or terminated.

Write the JSP Welcome Page

Here is the JSP welcome page, `index.jsp`, from which you can invoke the session creation servlet by clicking the **Create New Session** link. In this example, we assume `/mylistener` is the context path of the application, and `mysessioncreate` is configured in `web.xml` to be the servlet path of the session creation servlet.

```
<%@page session="false" %>
<HTML>
<BODY>
<H2>OC4J Session Event Listener</H2>
<P>
This example demonstrates the use of a session event listener.
</P>
<P>
<a href="/mylistener/mysessioncreate">Create New Session</A><br><br>
</P>
<P>
Click the <b>Create</b> link above to start a new session.<br>
A session listener has been configured for this application.<br>
The servlet container will send an event to this listener when a new session
is<br>
created or destroyed. The output from the event listener will be visible in
the<br>
console window from where OC4J was started.
</P>
</BODY>
</HTML>
```

Write the Session Creation Servlet

This servlet, `SessionCreateServlet`, creates an HTTP session object and displays some information for the created session. You can terminate the session by clicking the **Destroy Session** link, which invokes `SessionDestroyServlet`. In this example, we assume `/mylistener` is the context path of the application, and `mysessiondestroy` is configured in `web.xml` to be the servlet path of the session invalidation servlet.

```
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 for the response.
      res.setContentType("text/html");

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

      out.println("<HTML><BODY>");
      out.println("<A HREF=\"/mylistener/mysessiondestroy\">Destroy Session</A>");
```

```
      out.println("<h2>Session Created</h2>");
      out.println("Also check the OC4J console.");
      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("</BODY></HTML>");

   }
}
```

Write the Session Invalidation Servlet

This servlet, `SessionDestroyServlet`, destroys the HTTP session object. You can go back to the JSP welcome page to create a new session by clicking the **Reload Welcome Page** link. In this example, we assume `/mylistener` is the context path of the application.

```
import java.io.*;
import java.util.Enumeration;
import java.util.Date;
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 the session.
      session.invalidate();

      // Set content type for response.
      res.setContentType("text/html");

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

      out.println("<HTML><BODY>");
      out.println("<A HREF=\"/mylistener/index.jsp\">Reload Welcome Page</A>");
      out.println("<h2>Session Destroyed</h2>");
      out.println("Also check the OC4J console.");
      out.println("</BODY></HTML>");
      out.close();
   }
}
```

Write the Session Lifecycle Listener Code

This section shows the session lifecycle listener class, `SessionLifeCycleEventExample`, which implements the `HttpSessionListener` interface. Its `sessionCreated()` method is called by the servlet container whenever an HTTP session is created, which occurs when you click **Create New Session** from the JSP welcome page. When `sessionCreated()` is called, it writes a "CREATED" message to the OC4J console indicating the ID of the new session.

The `sessionDestroyed()` method is called by the servlet container whenever an HTTP session is destroyed, which occurs when you click **Destroy Session** from the session creation servlet. When `sessionDestroyed()` is called, it prints a "DESTROYED" message to the OC4J console indicating the ID of the terminated session.

(This class also implements the `ServletContextListener` interface and has `contextInitialized()` and `contextDestroyed()` methods, but these features are not used in this example.)

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

public class SessionLifeCycleEventExample
       implements ServletContextListener, HttpSessionListener
{
   ServletContext servletContext;

   /* Methods for 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("CREATED",hse);
   }

   public void sessionDestroyed(HttpSessionEvent hse)
   {
      log("DESTROYED",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);
   }
}
```

Configure the Session Lifecycle Listener Example

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 SessionLifeCycleEventExample methods, as appropriate, upon the occurrence of session lifecycle events (or servlet context lifecycle events, but that is not relevant for this example). Here are the `web.xml` entries:

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<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>
   <servlet-mapping>
      <servlet-name>sessioncreate</servlet-name>
      <url-pattern>mysessioncreate</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>sessiondestroy</servlet-name>
      <url-pattern>mysessiondestroy</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
</web-app>
```

Package the Session Lifecycle Listener Example

The WAR file for this example, which we name `sessionlistener.war`, has the following contents and structure:

```
index.jsp
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/SessionCreateServlet.class
WEB-INF/classes/SessionCreateServlet.java
WEB-INF/classes/SessionDestroyServlet.class
WEB-INF/classes/SessionDestroyServlet.java
WEB-INF/classes/SessionLifeCycleEventExample.class
WEB-INF/classes/SessionLifeCycleEventExample.java
```

And the EAR file is as follows:

```
sessionlistener.war
META-INF/application.xml
META-INF/Manifest.mf
```

(The `Manifest.mf` files are created automatically by the JAR utility.)

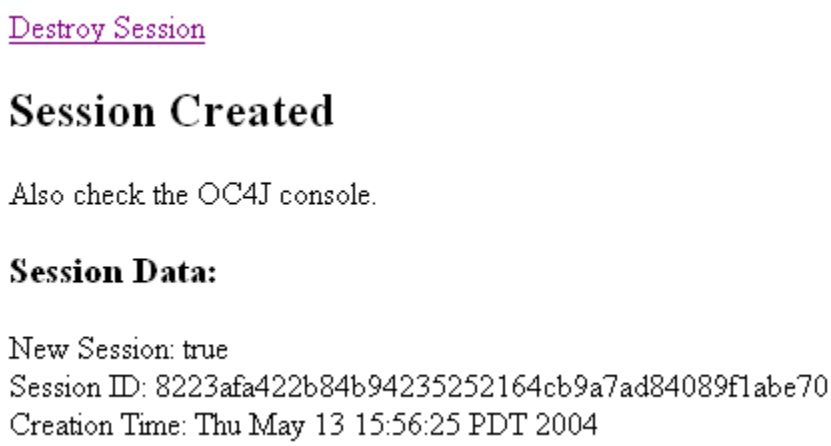
Invoke the Session Lifecycle Listener Example

For this example, assume that `application.xml` maps the context path `/mylistener` to `sessionlistener.war`. In this case, after deployment, you invoke the JSP welcome page as follows:

```
http://host:port/mylistener/index.jsp
```

The welcome page outputs the following:

Clicking **Create New Session** invokes the session creation servlet. In a test run, this results in the following output:



And the OC4J console reports the following:

```
04/05/13 15:56:25 SessionLifeCycleEventExample
Session ID: 8223afa422b84b94235252164cb9a7ad84089f1abe70     CREATED
```

Clicking **Destroy Session** invokes the session termination servlet. This results in the following output:

And in a test run, the OC4J console reports the following:

```
04/05/13 15:58:08 SessionLifeCycleEventExample
Session ID: 8223afa422b84b94235252164cb9a7ad84089f1abe70     DESTROYED
```

Clicking **Reload Welcome Page** takes you back to the JSP welcome page, where you can create another session.

6

Developing Servlets

This chapter, consisting of the following sections, provides basic information for developing servlets for OC4J and the Oracle Application Server:

- Writing a Basic Servlet
- Simple Servlet Example
- Using HTML Forms and Request Parameters
- Dispatching to Other Servlets Through Includes and Forwards
- When to Use Filters for Pre-Processing and Post-Processing
- When to Use Event Listeners for Servlet Notification
- Migrating an Application from Apache Tomcat to OC4J

For more general OC4J development information, refer to the *Oracle Containers for J2EE Developer's Guide*.

Note: For use during development, there is a convenience flag to direct automatic recompilation of servlet source files in a specified directory. 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 the `development` flag under "<orion-web-app>" on page B-16.

Writing a Basic Servlet

HTTP servlets follow a standard form. They are written as public classes that extend the `javax.servlet.http.HttpServlet` class. Most servlets override either the `doGet()` method or the `doPost()` method of `HttpServlet`, to handle HTTP `GET` or `POST` requests, respectively. It may also be appropriate to override the `init()` and `destroy()` methods if special processing 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.

The following subsections cover basic scenarios for implementing these methods, show how to set up the response, and go step-by-step through the code of a Hello World servlet:

- When to Implement Methods of the Servlet Interface
- Setting Up the Response

- Step-by-Step Through a Simple Servlet

When to Implement Methods of the Servlet Interface

Here is a basic code template for servlet development:

```
package ...;
import ...;

public class MyServlet extends HttpServlet {

  public void init(ServletConfig config) {
  }

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

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

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

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

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

  public void destroy() {
  }

}
```

The subsections that follow discuss the scenarios for overriding any of these methods.

When to Override the init() Method

You can override the `init()` method to perform special actions that are required only once in the servlet lifetime, such as the following:

- Establish database connections.
- Get initialization parameters from the servlet configuration object and store the values.
- Recover persistent data that the servlet requires.
- Create expensive session objects, such as hashtables.

For example, to establish a database connection through a data source:

```
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
```

```
    }
    ...
}
```

When to Override the doGet() or doPost() Method

Almost any servlet will override the `doGet()` method, to handle an HTTP `GET` request, or the `doPost()` method, to handle an HTTP `POST` request, for the bulk of its processing. `GET` and `POST` are the two HTTP methods for passing form data to the server. A detailed discussion of when to use one versus the other is beyond the scope of this manual, but the `doPost()` method may be more appropriate if security is a particular concern, given that the `GET` method places form parameters directly in the URL string, or for large sequences of data, allowing the client to send data of unlimited length to the server.

In implementing `doGet()` or `doPost()`, in addition to writing the code that generates the data to pass to the client, you will typically write code to read data from the HTTP request, set up the HTTP response, and write the response. For additional information, see "Setting Up the Response", which follows shortly, and "Using HTML Forms and Request Parameters" on page 6-7.

"Step-by-Step Through a Simple Servlet" on page 6-4 shows the steps for an elementary `doGet()` implementation.

When to Override the doPut() Method

Use this method to execute an HTTP `PUT` request, which allows a file to be written from the client to the server. The `doPut()` method must be able to handle a content header (or issue an error message if it cannot), and must leave any content headers it encounters intact.

When to Override the doDelete() Method

Use this method to execute an HTTP `DELETE` request, which allows a file or Web page to be removed from the server.

When to Override the getServletInfo() Method

Use this method to retrieve information from the servlet, such as author and version. By default, this method returns an empty string, so you must override it to provide any meaningful information.

When to Override the destroy() Method

This method is called by the servlet container when the servlet is about to be shut down. You can override it for any cleanup prior to shutdown that is appropriate for your servlet, such as the following:

- Update any persistent data to make sure it is current.
- Clean up any resources, such as database connections or file handles.

For example, to close the database connection that was opened in "When to Override the init() Method" on page 6-2:

```
public void destroy() {
   try {
      conn.close();
   }
   ...
}
```

Setting Up the Response

To send a response from your servlet, use the `HttpServletResponse` instance that is passed in to the servlet method you are using, typically `doGet()` or `doPost()`. The key steps are as follows:

1. Set a content type, and optionally a character encoding (MIME character set), for the response.

   Note: The OC4J default content type, if any, is reflected in the `<default-mime-type>` element of the OC4J `global-web-application.xml` (global) or `orion-web.xml` (application-level) Web application configuration file. You can set it through the deployment plan editor in the Application Server Control Console, introduced in "A Brief Overview of OC4J Administration" on page 2-1.

2. Get a writer object (`java.io.PrintWriter`), for character data, or an output stream (`javax.servet.ServletOutputStream`), for binary data, from the response object.
3. Write the response data to the writer object or output stream.

Here is some code that shows these steps.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><body><h1>Hello World</h1></body></html>");
    ...
}
```

See "Key Methods of the HttpServletResponse Interface" on page 1-7 for a summary of response methods.

Development Tip: The servlet container implicitly closes the writer object or output stream after committing the response, but it is still good programming practice to close it explicitly.

Step-by-Step Through a Simple Servlet

This chapter shows a Hello World example that overrides the `doGet()` method. This servlet is shown in its entirety in "Simple Servlet Example" on page 6-5, but we also go through it step-by-step here.

Initial steps in the servlet example:

1. Declare a package, as appropriate. The servlet example declares `mytest`:

   ```
   package mytest;
   ```

2. Import required Java packages, particularly the servlet packages. The following are typically required:

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

3. Declare the servlet class, which always extends `HttpServlet` for HTTP operations:

```
public class HelloWorld extends HttpServlet {
   ...
}
```

4. Declare any servlet method that you want to override. The servlet methods for HTTP operations all take the same parameters (an HTTP request object and an HTTP response object) and throw the same exceptions. The servlet example overrides `doGet()`:

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

Steps in the servlet example `doGet()` method:

1. Set the content type for the response object. This may not always be required, but is generally advisable:

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

   Optionally, you can also specify a character encoding, such as UTF-8 in the following example:

```
response.setContentType("text/html; charset=UTF-8");
```

2. Get a writer object from the response object:

```
PrintWriter out = response.getWriter();
```

3. Write the data to the response object:

```
out.println("<html>");
out.println("<head>");
out.println("<title>Hello World!</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Hi Amy!</h1>");
out.println("</body>");
out.println("</html>");
```

4. Close the output stream, which also commits the response.

```
out.close();
```

(In this simple example, manipulating request data is not required.)

Simple Servlet Example

This section shows the complete simple servlet example that is discussed, step-by-step, in the preceding section. This example is deployed and invoked in "Deploying and Invoking the Simple Servlet Example" on page 2-12.

Write the Sample Code

The following code writes "Hi Amy!" to the browser. Enter the code into a file called `HelloWorld.java`. According to the `package` statement, the `HelloWorld` class will be in package `mytest`.

```
package mytest;

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

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hi Amy!</h1>");
        out.println("</body>");
        out.println("</html>");

        out.close();
    }
}
```

Compile the Sample Code

Compile the sample code. If you are using a JDK from Sun Microsystems, with their default compiler, accomplish this as follows, from the directory where the `.java` file is located (assuming `%` is the system prompt):

```
% javac HelloWorld.java
```

Development Tips:

- Add the location of the Java executables—such as the JVM, Java compiler, and JAR utility—to your system file path so you can run them from any location. For example, for the Sun Microsystems JDK 1.4.2, version 4, add *jdkroot*`/j2sdk1.4.2_04/bin` to the file path, where *jdkroot* is the full path to the directory where the JDK is installed, so you can run `java`, `javac`, and `jar` from any location. How to accomplish this varies, depending on your operating system.
- The standard servlet classes and interfaces are provided with OC4J in a file called `servlet.jar` in the *oc4jroot*`/j2ee/home/lib` directory, where *oc4jroot* is the full path to the directory where OC4J is installed. You must make `servlet.jar` available to the Java compiler. One way to accomplish this is to add *oc4jroot*`/j2ee/home/lib/servlet.jar` to a system or user classpath environment variable. If you are using a Sun JDK, an alternative way to accomplish this is to copy `servlet.jar` to the JDK `jre/lib/ext` extensions directory. For example, for JDK 1.4.2, version 4, copy it to the *jdkroot*`/j2sdk1.4.2_04/jre/lib/ext` directory.

Using HTML Forms and Request Parameters

A typical servlet might ask the user to enter some information for the servlet to display or manipulate. The servlet can use HTML forms to take the information, store it in parameters of the HTTP request object, and send it to the server. You can also retrieve other information from the request object, such as the protocol, HTTP method, and request URI being used.

The following sections show some examples:

- Using an HTML Form for User Input
- Displaying Request Parameter Data Specified in User Input
- Complete Example Using a Form and Request Parameters
- Using the POST Method for URL Security
- Calling Information Methods of the Request Object
- Complete Example Retrieving Request Information

HTTP request parameters will not be available to servlet filters that are meant to be executed before dispatch of the request to a static resource (an `.html` file, for example). Filters that execute before dynamic resources, such as a servlet or JSP page, will have access to the parameters.

See "Key Methods of the HttpServletRequest Interface" on page 1-6 for a summary of request methods.

Using an HTML Form for User Input

A servlet can use an HTML form to take input from the user, then submit these data to the server as parameters of the HTTP request object. Here is an example:

```
PrintWriter out = response.getWriter();
```

```
...
out.print("<form action=\"");
out.print("RequestParamExample\" ");
out.println("method=GET>");
out.println("Enter a new first name: ");
out.println("<input type=text size=20 name=firstname>");
out.println("<br>");
out.println("Enter a new last name: ");
out.println("<input type=text size=20 name=lastname>");
out.println("<br>" + "<br>");
out.println("<input type=submit>");
out.println("</form>");
```

This example prompts the user to enter his or her first name, stores it in a request parameter called `firstname`, prompts for the last name, and stores it in a request parameter called `lastname`. The request object is sent to the server, where the information is processed as desired (as shown in the next section).

A significant disadvantage to using the `GET` method for this operation, however, is that the parameter names and values are appended to the servlet URL string. To prevent this, you can use the `POST` method instead, as shown in "Using the POST Method for URL Security" on page 6-10.

Displaying Request Parameter Data Specified in User Input

This section shows sample code that displays request parameter data that were specified by the user through an HTML form (shown in the preceding section).

```
PrintWriter out = response.getWriter();
...
String firstName = request.getParameter("firstname");
String lastName = request.getParameter("lastname");
out.println("First and last name from request:" + "<br>" + "<br>");
if (firstName != null || lastName != null) {
    out.println("First name ");
    out.println(" = " + firstName + "<br>");
    out.println("Last name ");
    out.println(" = " + lastName + "<br>");
} else {
    out.println("(No names entered. Please enter first and last name.)");
}
```

The values of the request parameters `firstname` and `lastname` are stored in the strings `firstName` and `lastName`, then output to the user.

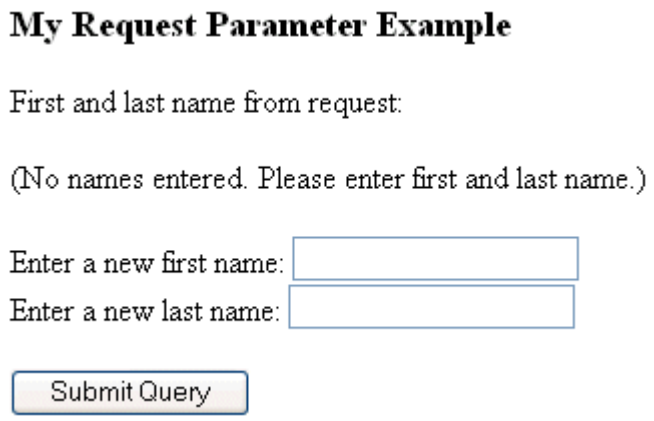
Complete Example Using a Form and Request Parameters

Here is the complete servlet with the code from the preceding sections. It prompts the user for first name and last name, with the information being written to the request object, then it retrieves the names from the request object and outputs them to the user. (The output code comes first, indicating "No names entered" until the user first enters some names.)

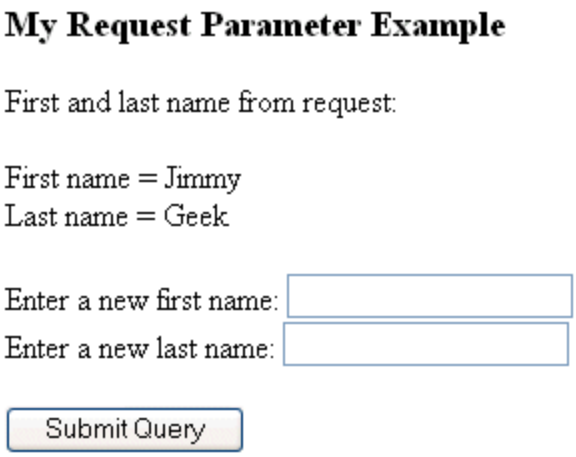
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestParamExample extends HttpServlet {

    public void doGet(HttpServletRequest request,
```

```
                    HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");

        out.println("<h3>" + "My Request Parameter Example" + "</h3>");
        String firstName = request.getParameter("firstname");
        String lastName = request.getParameter("lastname");
        out.println("First and last name from request:" + "<br>" + "<br>");
        if (firstName != null || lastName != null) {
            out.println("First name ");
            out.println(" = " + firstName + "<br>");
            out.println("Last name ");
            out.println(" = " + lastName + "<br>");
        } else {
            out.println("(No names entered. Please enter first and last name.)");
        }
        out.println("<P>");
        out.print("<form action=\"");
        out.print("RequestParamExample\" ");
        out.println("method=GET>");
        out.println("Enter a new first name: ");
        out.println("<input type=text size=20 name=firstname>");
        out.println("<br>");
        out.println("Enter a new last name: ");
        out.println("<input type=text size=20 name=lastname>");
        out.println("<br>" + "<br>");
        out.println("<input type=submit>");
        out.println("</form>");

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

When the servlet first starts, it shows the following:



If you enter "Jimmy" and "Geek" and then click **Submit Query**, it shows the following:



Development Tip: This servlet uses the HTTP `GET` method, resulting in the request parameter names and values being appended to the servlet URL. In this example, the string "`?firstname=Jimmy&lastname=Geek`" is appended. See the next section, "Using the POST Method for URL Security" on page 6-10, for how to avoid this.

Using the POST Method for URL Security

The preceding example used the HTTP `GET` method, which results in request parameter names and values being appended to the servlet URL. To avoid this (typically for security considerations), you can use the `POST` method instead. In the following code, the preceding example has been modified to use the `POST` method in the form, and to use a `doPost()` method to call the `doGet()` method. Changes are highlighted in **bold**.

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

public class RequestParamExample extends HttpServlet {


    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");

        out.println("<h3>" + "My Request Parameter Example" + "</h3>");
        String firstName = request.getParameter("firstname");
        String lastName = request.getParameter("lastname");
        out.println("First and last name from request:" + "<br>" + "<br>");
        if (firstName != null || lastName != null) {
            out.println("First name ");
            out.println(" = " + firstName + "<br>");
```

```
            out.println("Last name ");
            out.println(" = " + lastName + "<br>");
        } else {
            out.println("(No names entered. Please enter first and last name.)");
        }
        out.println("<P>");
        out.print("<form action=\"");
        out.print("RequestParamExample\" ");
        out.println("method=POST>");
        out.println("Enter a new first name: ");
        out.println("<input type=text size=20 name=firstname>");
        out.println("<br>");
        out.println("Enter a new last name: ");
        out.println("<input type=text size=20 name=lastname>");
        out.println("<br>" + "<br>");
        out.println("<input type=submit>");
        out.println("</form>");

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

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException
    {
        doGet(request, response);
    }
}
```

Development Tip: There is still a `doGet()` method in this example, rather than using `doPost()` directly, because browsers use `GET` requests.

Calling Information Methods of the Request Object

"Key Methods of the HttpServletRequest Interface" on page 1-6 lists some methods of the request object that you can use to retrieve information about the HTTP request. Here is a code sample that calls some of the information methods of a request object and outputs the information:

```
PrintWriter out = response.getWriter();
...
out.println("Method:");
out.println(request.getMethod());
out.println("Request URI:");
out.println(request.getRequestURI());
out.println("Protocol:");
out.println(request.getProtocol());
```

This example retrieves and displays the HTTP method (such as `GET` or `POST`), the request URI (consisting of the context path and servlet path in this example), and the protocol (such as HTTP). The next section shows a complete example.

Complete Example Retrieving Request Information

Here is a complete servlet that retrieves the HTTP method, request URI, and protocol, and outputs them in an HTML table.

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

public class RequestInfoExample extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html;charset=UTF-8");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");

        out.println("<h3>" + "My Request Info Example" + "</h3>");
        out.println("<table border=0><tr><td>");
        out.println("Method:");
        out.println("</td><td>");
        out.println(request.getMethod());
        out.println("</td></tr><tr><td>");
        out.println("Request URI:");
        out.println("</td><td>");
        out.println(request.getRequestURI());
        out.println("</td></tr><tr><td>");
        out.println("Protocol:");
        out.println("</td><td>");
        out.println(request.getProtocol());
        out.println("</td></tr>");
        out.println("</table>");

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

This results in output such as the following:

My Request Info Example

Method:	GET
Request URI:	/servlet/RequestInfoExample
Protocol:	HTTP/1.1

Dispatching to Other Servlets Through Includes and Forwards

Many servlets use other servlets in the course of their processing, either by *including* the response of another servlet or by *forwarding* the request to another servlet. The following subsections discuss these features and show examples:

- Basics of Includes and Forwards
- Why Use Includes and Forwards?
- Step-by-Step Through the Include or Forward Process

- Complete Example of a Servlet Include

> **Note:** The target of an include or forward can be a JSP page as well as a servlet. Wherever target servlets are discussed in the following text, you can assume the same applies to target JSP pages.

Basics of Includes and Forwards

In servlet terminology, a servlet *include* is the process by which a servlet includes the 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—this is the mechanism for dispatching an HTTP request to an alternative servlet. Use either of the following servlet context methods:

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

For `getRequestDispatcher()`, input the URI 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 execute any includes or forwards:

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

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

Why Use Includes and Forwards?

A servlet include is a convenient way to do any of the following:

- Reuse existing code without having to rewrite it.
- Include the same processing or output in multiple servlets, without having to implement the code in each individual servlet.
- Include content from a static file.

You are including the output of the target servlet *in addition to* the output of the originating servlet.

These points are similarly true for servlet forwards, but remember that with a forward, the output of the target servlet is *instead of* the output of the originating servlet, not in addition to it.

Step-by-Step Through the Include or Forward Process

Here are basic steps to implement an include or forward:

1. Use the `getServletConfig()` method of the servlet (specified in the `javax.servlet.Servlet` interface) to retrieve a servlet configuration object.

   ```
   ServletConfig config = getServletConfig();
   ```

2. Use the `getServletContext()` method of the servlet configuration object to retrieve the servlet context object for the servlet.

   ```
   ServletContext context = config.getServletContext();
   ```

3. Use the `getRequestDispatcher()` or `getNamedDispatcher()` method of the servlet context object to retrieve a `RequestDispatcher` object. For `getRequestDispatcher()`, specify the URI path of the target servlet; for `getNamedDispatcher()`, specify the name of the target servlet, according to the relevant `<servlet-name>` element in the `web.xml` file.

   ```
   RequestDispatcher rd = context.getRequestDispatcher("path");

   RequestDispatcher rd = context.getNamedDispatcher("name");
   ```

4. Use the `include()` or `forward()` method of the request dispatcher, as appropriate, to execute the include or forward, respectively. Pass the servlet request and response objects.

   ```
   rd.include(request, response);

   rd.forward(request, response);
   ```

You can combine all four steps into a single statement, as in the following example:

```
getServletConfig().getServletContext().getRequestDispatcher
                          ("path").include(request, response);
```

> **Note:** Alternatively, you can retrieve a request dispatcher through the `getRequestDispatcher()` method of the request object (`HttpServletRequest` instance).

The next section shows a complete example for a servlet include.

Complete Example of a Servlet Include

This section provides a complete example of a servlet including the output of another servlet. The `RequestInfoExample` class, shown in "Complete Example Retrieving Request Information" on page 6-11, is updated to include output from a slightly modified version of the `HelloWorld` class shown in "Simple Servlet Example" on page 6-5.

Here is the slightly modified Hello World example whose output will be included. The class is now called `HelloIncluded` and is not in a package:

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

public class HelloIncluded extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hi Amy!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Here is the updated request information example class, now called `RequestInfoWithInclude`, that includes the output from `HelloIncluded`. Key code is highlighted in **bold**:

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

public class RequestInfoWithInclude extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html;charset=UTF-8");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");

        out.println("<h3>" + "My Request Info Example" + "</h3>");
        out.println("<table border=0><tr><td>");
        out.println("Method:");
        out.println("</td><td>");
        out.println(request.getMethod());
        out.println("</td></tr><tr><td>");
        out.println("Request URI:");
        out.println("</td><td>");
        out.println(request.getRequestURI());
        out.println("</td></tr><tr><td>");
        out.println("Protocol:");
        out.println("</td><td>");
        out.println(request.getProtocol());
        out.println("</td></tr>");
        out.println("</table>");
```

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

        getServletConfig().getServletContext().getRequestDispatcher
                          ("/mypath/helloincluded").include(request, response);
    }
}
```

The path `/mypath/helloincluded` is a URI consisting of the context path and servlet path. The assumption is that the application has been configured so that `HelloIncluded` can also be requested directly, as follows:

```
http://host:port/mypath/helloincluded
```

See Chapter 2, "Deploying and Invoking Servlets" for related information.

You could similarly include a JSP page instead of a servlet, such as in the following example:

```
        getServletConfig().getServletContext().getRequestDispatcher
                          ("/mypath/hello.jsp").include(request, response);
```

Invoking `RequestInfoWithInclude` results in output such as the following:

When to Use Filters for Pre-Processing and Post-Processing

Request objects and response objects 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 servlets and the servlet container to wrap and preprocess requests or to wrap and postprocess responses. A filter is invoked when there is a request for a resource that the filter has been mapped to in the servlet configuration.

Filters can effectively transform requests and responses. Use filters if you want to apply preprocessing or postprocessing for a group of servlets. (If you want to modify the request or response for just one servlet, there is no need to create a filter. You can just do what is required directly in the servlet itself.)

You can use filters in scenarios such as accessing a resource, or processing a request to that resource, prior to the request being invoked; or wrapping a request or response in a customized request object or response object, respectively. You can act on a servlet with a chain of filters in a specified order.

One example is an encryption filter. 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.) Other examples are filters for authentication, logging, auditing, data compression, and caching.

See Chapter 4, "Understanding and Using Servlet Filters" for details.

When to Use Event Listeners for Servlet Notification

The servlet specification adds the capability to track key events in your Web applications through *event listeners*. You can implement listeners to notify your application of application events, session events, or request events. This functionality allows more efficient resource management and automated processing based on event status.

Use event listeners if there is reason for your application to be notified of any of the following.

- For the servlet context:
  - The servlet context is newly created or is about to be shut down.
  - Servlet context attributes are added, removed, or replaced.
- For a session:
  - A session is newly created or is newly invalidated or timed out.
  - Session attributes are added, removed, or replaced.
  - A session is newly active or passive.
  - An object is newly bound to or unbound from a session.
- For a request:
  - A request is being newly processed.
  - Request attributes are added, removed, or replaced.

As an example, consider a Web application comprising servlets that access a database. You can create a servlet context lifecycle event listener to manage the database connection. This event listener may function as follows:

1. The event 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 event 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 event listener closes the database connection.

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 event listener method.

See Chapter 5, "Understanding and Using Event Listeners" for details.

How to Display the Stack Trace

The following changes have been made to the error messages that are presented when an exception occurs and there is no error page to handle it.

The following error message appears for security-sensitive exceptions:

```
Servlet error: An exception occurred. For security reasons, it
may not be included in this response. Please consult the
application log for details.
```

For other exceptions, the following error message appears:

```
Servlet error: An exception occurred. The current application
deployment descriptors does not allow for including it in this
response. Please consult the application log for details.
```

If you have tests that rely on the display of an exception or stack trace, you can cause the stack trace to be displayed by running the application in developer mode.

To run an application in development mode, set the `development` attribute of the `<orion-web-app>` element to `"true"` in the `orion-web.xml` file.

Here is an example:

```
<?xml version="1.0"?>
<orion-web-app
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-web-1
0_0.xsd"
      deployment-version="null"
      deployment-time="1152223108413"
      jsp-cache-directory="./persistence"
       jsp-cache-tlds="standard"
      temporary-directory="./temp"
      context-root="/DevelopmentThrowException"
      schema-major-version="10"
      schema-minor-version="0"
      development="true">
<!-- Uncomment this element to control web application class loader behavior.
 <web-app-class-loader search-local-classes-first="true"
include-war-manifest-class-path="true" />
 -->
   <web-app>
   </web-app>
</orion-web-app>}}
```

Migrating an Application from Apache Tomcat to OC4J

This section provides information about migrating a Web application that includes servlets, JSP modules, or both from Apache Tomcat to OC4J. The following topics are discussed:

- Pointers for Migrating from Tomcat to OC4J
- JNDI Lookups in Tomcat and OC4J

- Tomcat-to-OC4J JSP Compilation Issues
- Tomcat-to-OC4J Clustering Issues

Pointers for Migrating from Tomcat to OC4J

This section provides pointers about migrating Java servlets from Tomcat to Oracle Application Server. This section contains the following topics:

- Introduction
- Migration Approach for Servlets
- Migrating a Simple Servlet
- Migrating a WAR File
- Migrating an Exploded Web Application
- Tips From the Field

Introduction

Migrating Java servlets from Tomcat to OC4J is straightforward, requiring little or no code changes to the servlets migrated, depending on some of the choices made in the Tomcat environment.

Oracle Application Server 10g Release 3 (10.1.3.x) is fully compliant with Sun Microsystem's J2EE Servlet specification, version 2.4. Tomcat 5.5 is also compatible with version 2.4.

In addition, Oracle Application Server 10g Release 3 (10.1.3.x) is backward compatible to Servlet 2.3. Hence, servlets written to the standard 2.3 specification should work correctly in OC4J and require minimal migration effort.

The primary tasks involved in migrating servlets to a new environment are configuration and deployment.

The tasks involved in migrating servlets also depend on how the servlets have been packaged and deployed. Servlets can be deployed as a simple servlet, as a Web application packaged with other resources in a standard directory structure, or as a Web archive (WAR) file.

Migration Approach for Servlets

The typical steps for migrating servlets to OC4J are as follows:

1. Configuration: Create or modify the Oracle Application Server deployment descriptors for the servlets.
2. Packaging:
   - Simple servlets can be deployed individually.
   - Servlets can be packaged as part of a Web application in a WAR file.
3. Deployment: Application Server Control Console can be used to deploy servlets in a WAR file. Individual servlets and servlets in exploded Web applications can be deployed automatically by copying them to the appropriate directories.

Oracle JDeveloper provides tools and wizards to automate these steps.

Migrating a Simple Servlet

Simple servlets are easily configured and deployed in OC4J. The manual process used to deploy a servlet is the same in both Tomcat and OC4J.

> **Note:** The recommended and preferred way to deploy a servlet is by packaging it in a WAR or EAR file and using Oracle Enterprise Manager 10g Application Server Control Console.
>
> Alternatively, you can deploy manually using the `admin_client.jar` command line utility. The manual processes described in this chapter of editing XML files and starting OC4J at the command line using the java command should preferably be used in a development environment. For information on `admin_client.jar`, see the *Oracle Containers for J2EE Configuration and Administration Guide* and the *Oracle Containers for J2EE Deployment Guide*.

A servlet must be registered and configured as part of a Web application. To register and configure a servlet, several entries must be added to the Web application deployment descriptor.

The typical steps to deploy a simple servlet are as follows:

1. Update the Web application deployment descriptor (`web.xml`) with the name of the servlet class and the URL pattern used to resolve requests for the servlet.
2. Copy the servlet class file to the `WEB-INF/classes/` directory. If the servlet class file contains a package statement, create additional subdirectories for each level of the package statement. The servlet class file must then be placed in the lowest subdirectory created for that package.
3. Extract and copy the supporting utility class file and any other supporting files required by the servlet to the appropriate directory in the Oracle Application Server installation.
4. Start or restart the home OC4J "home" instance.
5. Invoke the servlet from your browser by entering its URL.

Migrating a WAR File

A Web application can be configured and deployed as a WAR file. This is most easily accomplished in OC4J by using the Application Server Control Console administration GUI. Alternatively, you can manually copy the WAR file to the appropriate directory. This is also true for Tomcat.

> **Note:** Manually copying a WAR file to the appropriate directory to deploy it should only be done in a development environment where OC4J is in standalone mode (not a component of an Oracle Application Server instance).

Production Web applications are typically deployed using WAR or EAR files through Application Server Control Console or the utility. During the development of a Web application, it may be faster to deploy and test edited code using an exploded directory format.

See the *Oracle Containers for J2EE Deployment Guide* for more information on deployment.

The typical steps for migrating a WAR file from Tomcat to OC4J are as follows:

1. Create the WAR file for the sample application.
2. Deploy the sample application to OC4J.
3. Test the deployed application.

Migrating an Exploded Web Application

Web applications can also be configured and deployed as a collection of files stored in a standard directory structure or exploded directory format. This can be accomplished in OC4J by manually copying the contents of the standard directory structure to the appropriate directory in the OC4J installation. The same method can also be used for Tomcat.

Deploying a Web application in exploded directory format is used primarily during the development of a Web application. It provides a fast and easy way to deploy and test changes. When deploying a production Web application, package the Web application in a WAR file and deploy the WAR file using Application Server Control Console.

The typical steps for manually deploying an exploded Web application in OC4J are as follows:

1. Copy the top-level directory containing the exploded Web application into the following directory of your OC4J installation:
   `<ORACLE_HOME>/j2ee/home/applications`

   Then, modify the application deployment descriptor `<ORACLE_HOME>/config/application.xml` to include the Web application, as follows:

   ```
   <web-module id="migratedHR" path="../applications/hrapp" />
   ```

2. Bind the Web application to your Web site by adding an entry in the descriptor file `<ORACLE_HOME>/config/default-web-site.xml`, as follows:

   ```
   <web-app application="default" name="migratedHR " root="/hr" />
   ```

3. Finally, register the new application by adding a new `<application>` tag entry in the following file: `<ORACLE_HOME>/config/server.xml`

When you modify `server.xml` and save it, OC4J detects the timestamp change of this file and deploys the application automatically. OC4J need not be restarted.

Tips From the Field

Here are some issues to watch out for when porting from Tomcat to OC4J.

- Make Sure to Use the Initial Slash "/" in Path Names
- JNDI Context Factory Summary
- Xerces and Xalan Require Additional Steps

Make Sure to Use the Initial Slash "/" in Path Names

Tomcat is not entirely compliant with the Servlet specification with respect to the following methods of the `ServletContext` class:

- `ServletContext.getResource()`
- `ServletContext.getResourceAsStream()`

The methods `getResource()` and `getResourceAsStream()` take a path parameter. This path allows you to access files in your web application directory structure such as `WEB-INF/config.xml`.

The J2EE API documentation and specifications state the following:

"The path must begin with a "/" and is interpreted as relative to the current context root."

Tomcat, contrary to the Servlet specification, allows path names without the initial slash (/).

OC4J, in full compliance with the Servlet specification, requires the initial slash (/).

Don't forget the initial slash (/) on path names to access files with the `ServletContext` class.

JNDI Context Factory Summary

In OC4J, you can use several different JNDI context factories, each of which creates an `InitialContext` and comes with different functionalities.

The most commonly used context factories are the following:

- The internal context factory - You get the internal context factory when you are within the OC4J container and you call the default constructor of `InitialContext` without `jndi.properties` in the class path.
- `RMIInitialContextFactory` - This context factory is commonly used to connect to the OC4J container. It ignores every configuration and does not require the `java:comp/env/` prefix.
- `ApplicationClientContextFactory` - This context factory takes care of your JNDI environment (especially in `META-INF/application-client.xml`) and honors the `java:comp/env/` environment prefix.

Xerces and Xalan Require Additional Steps

If the code relies on Xerces/Xalan, then additional steps must be taken.

Metalink has additional information about this topic.

JNDI Lookups in Tomcat and OC4J

Tomcat enables you to look up resources using `java:`*`comp/env/ResourceName`* without defining a `<resource-ref>` element in `web.xml`.

OC4J requires the `<resource-ref>` element in `web.xml` or the lookup is just as *`ResourceName`*.

OC4J can do this by default for a Data Source defined on the server:

```
initialContext.lookup("jdbc/ScottDS")
```

Whereas Tomcat can do it like this:

```
initialContext.lookup("java:comp/env/jdbc/ScottDS")
```

If you do not want make code changes and you need to use the Tomcat option, then you must modify `web.xml` to include the `<resource-ref>` entry for the resource, as follows:

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

Tomcat-to-OC4J JSP Compilation Issues

In general, OC4J interprets the J2EE specifications more strictly than Tomcat.

You can avoid compilation issues by replacing custom Tomcat JSP tags with standard JSP tags before you deploy a JSP page on OC4J.

Be careful when the words "should" and "may" appears in the specification. Treat these cases with caution because they allow choice, which results in differing behavior.

Here is an example. The JSP specification says:

> A scripting language variable of the specified type (if given) or class (if type is not given) is defined with the given id in the current lexical scope of the scripting language. The type attribute should be used to specify a Java type that cannot be instantiated as a JavaBean (i.e. a Java type that is an abstract class, an interface, or a class with no public no-args constructor). If the class attribute is used for a Java type that cannot be instantiated as a JavaBean, the container may consider the page invalid, and is recommended to (but not required to) produce a fatal translation error at translation time, or a java.lang.Instantiation-Exception at request time.

In this case, OC4J implements the recommendation that the `type` attribute should be used instead of the `class` attribute when the JavaBean does not have a `zero-args` constructor. Tomcat does not insist on the `type` attribute but will accept the `class` attribute as well. This difference results in the following problem when migrating the application from Tomcat to OC4J:

If you use the `type` attribute in your `index.jsp` and the JavaBean (`my.MyClass`) does not have a public `no-args` constructor, then your page works properly both in Tomcat and in OC4J, which is the desirable behavior.

The following example shows this **preferred** usage:

```
<jsp:useBean id="codeDesc" scope="session" type="my.MyClass"/>
```

If, on the other hand, you use the `class="my.MyClass"` attribute in this situation, Tomcat may accept this usage and behave correctly. But OC4J will throw a `JSPCompilationException` because OC4J uses the more strict interpretation of the specification.

The following example shows the less-strict usage:

```
<jsp:useBean id="codeDesc" scope="session" class="my.MyClass"/>
```

Tomcat-to-OC4J Clustering Issues

This section provides an overview of how clustering in OC4J and Tomcat relate to one another. The following topics are discussed:

- Basic Configuration in Tomcat and OC4J
- Network Considerations in Tomcat and OC4J
- State Persistence Mechanisms in Tomcat and OC4J
- Replication Algorithms in Tomcat and OC4J
- Application Design in Tomcat and OC4
- Load Balancing in Tomcat and OC4J

> **Note:** The commentary in this section is based on the behavior of Tomcat 5.

In OC4J, clustering is defined on an application-by-application basis when it is deployed.

For information about clustering in OC4J, including how to configure, see Chapter 9 "Application Clustering in OC4J" of the Oracle Containers for J2EE Configuration and Administration Guide.

For information about load balancing, see Appendix D of the Oracle® HTTP Server Administrator's Guide.

Basic Configuration in Tomcat and OC4J

In Tomcat, clustering is configured using the <cluster> element and its subelements in the server.xml file. Clustering is enabled on a container basis.

In OC4J, clustering is defined on an application-by-application basis using the <cluster> element in the orion-application.xml file of a deployed application.

This enables applications with and without clustering enabled to co-exist within the same OC4J instance. Clustering in OC4J is configured using the orion-application.xml file of the application in question. To enable container-level clustering, enable clustering for the default application and all deployed applications will inherit this setting. Also, since the clustering is configured on a per-application basis in OC4J, different applications can be configured to enable clustering, using different protocols and different options.

Network Considerations in Tomcat and OC4J

Tomcat uses both IP multicast and IP sockets to facilitate clustering. In terms of terminology, Tomcat has a single in-memory replication option, where it combines the two network models.

- IP multicast is used to perform group membership operations - such as when Tomcat instances are discovering and checking the availability of one another.
- IP sockets are used to perform the actual replication of the session state from one Tomcat instance to another Tomcat instance. Presumably this is for the in-memory state persistence mechanism.

OC4J differs in its use and terminology for in-memory session-state replication. OC4J has two different ways to replicate session state in-memory - it can use either IP

multicast OR IP sockets. In either case, the Group membership activities occur using the same network model.

- IP Multicast

  If you configure OC4J to use IP Multicast as the replication protocol, then OC4J uses IP Multicast as the medium to participate in the Group membership protocol as well as for distributing session state to the other members of the Group. The multicast network address is defaulted to a known value. It can be configured as desired using the <multicast> subelement of the <cluster> element. In OC4J, the multicast replication protocol has the added value of guaranteed delivery - so, even though the actual network model used has no reliability guarantees, the way in which is it used by OC4J guarantees that all the packets are delivered and no loss of session state will occur.

- IP Sockets

  If you configure OC4J to use Peer-Peer as the replication protocol, then OC4J uses IP sockets as the medium to participate in the Group membership protocol as well as for distributing session state to its selected peers. In the case of OC4J used on its own (standalone), then the list of peers must be statically defined in a configuration file. The TCP socket address defaults to a well-known value. It can be configured using the peer subelement of the cluster protocol, tag.

  In the case of OC4J used in the Oracle Application Server environment (clustered), the initial list of peers is provided by the OPMN server. OPMN also allocates the port numbers used by the Peer replication protocol so that there are no port conflicts on a server where multiple OC4J instances are started.

State Persistence Mechanisms in Tomcat and OC4J

Tomcat supports the following mechanisms to handle the distribution of session state to other Tomcat instances:

- In-memory
- Database
- File-based

OC4J supports the following session persistence mechanisms:

- In-memory
- Database

There is potential confusion here because OC4J supports two methods of in-memory-based replication (multicast and peer) whereas Tomcat uses a combination of both to support in-memory replication.

OC4J does not support a file-based state replication protocol.

OC4J uses the term "replication protocol" instead of "state persistence".

- In Tomcat, the "state persistence" mechanism is defined by specifying the name of the class used to implement the persistence mechanism.
- In OC4J, the analogous concept, "replication protocol", is configured as a value in the `orion-application.xml` file using the `<protocol>` tag and specifying one of the `<multicast>`, `<peer>` or `<database>` subtags to indicate which protocol should be used.

  The database replication protocol requires the JNDI location of a Data Source, which points to the database instance in which the state replicas will be stored.

Replication Algorithms in Tomcat and OC4J

In Tomcat, the type of state replicated can be configured to be either the full session or just the change set. This is configured by specifying the class that is used to perform the replication task.

OC4J supports the same concept of sending either the full session or just the deltas. This is configurable using the `<replication-policy>` element and the `scope` attribute. The options are `modifiedAttributes` or `allAttributes`.

State Replication Transmission

In Tomcat, the transmission of the state replications can be configured to be synchronous, asynchronous, or pooled.

OC4J by default uses an asynchronous replication model for replication transmissions. This can be changed to a synchronous model using the synchronous-replication option, whereupon the OC4J instance sending the replica will wait for an acknowledgment from the receiving OC4J before proceeding.

Application Design in Tomcat and OC4

OC4J and Tomcat share the same application design requirements for applications that will run in a clustered environment.

Load Balancing in Tomcat and OC4J

Tomcat provides a built-in HTTP server. However, it is possible to scale up using a generic Apache HTTP Server with `mod_proxy` or `mod_rewrite` or the AJP-based `mod_jk` connector to route between multiple instances of Tomcat.

OC4J also provides a built-in HTTP server. However, to scale up and benefit from the automatic load balancing capabilities it provides, Oracle recommends using the Oracle HTTP server. Using OC4J HTTP requires a third-party load balancing component.

In an Oracle Application Server environment, the Oracle HTTP Server is used to route requests to applications running on OC4J instances. The Oracle HTTP Server is configured to discover the OC4J instances running in the same Cluster Topology. As applications are deployed to the OC4J instances, Oracle HTTP Server automatically receives updates about the new deployment and automatically adds the application context root to its list of routable URLs.

Oracle HTTP Server provides automatic failover for clustered applications with sticky session support so that session-based requests are always routed to the same OC4J instance until a failover occurs.

Oracle HTTP Server uses a round-robin load balancing algorithm by default. The load balancing model can be changed to several different options.

For further discussion, see Appendix D, "Load Balancing Using mod_oc4j" in the *Oracle HTTP Server Administrator's Guide*.

7

Using Annotations for Services and Resource References

OC4J supports annotations that the servlet 2.5 specification describes, by injecting resource references before a component instance is made available to an application. The following sections describe how to use annotations in OC4J:

- Overview of How Annotations Work
- Annotations and Injection
- Annotations in OC4J
- Annotation Rules and Guidelines
- How Annotations Affect Performance with Servlet Version 2.5

Overview of How Annotations Work

In J2SE 5.0 or greater, you can specify configuration data and dependency on external resources in Java code as metadata, also referred to as annotations. You can define such data in configuration files or in annotations for services, such as EJBs or Web services, and for resource references, such as data sources and JMS destinations.

For example, in J2EE 1.4, before a servlet can refer to an EJB, the developer must define an `ejb-local-ref` element, like this one:

```
<ejb-local-ref>
 <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
 <local>oracle.ejb.HelloWorld</local>
</ejb-local-ref>
```

Then, to refer to the EJB in code, the developer has to use JNDI, as in this code fragment:

```
Context ic = new InitialContext();

HelloWorld helloWorld = (HelloWorld)ic.lookup("java:comp/env/ejb/HelloWorld");
helloWorld.greet("Hello!");
```

In servlet 2.5, the client code is simplified and OC4J injects the correct resource or service. The client only needs to specify the resource or service in an annotation, such as this one:

```
@EJB
private HelloWorld helloWorld;
helloWorld.greet("Hello!");
```

Annotations and Injection

The `metadata-complete` attribute of the `web-app` element in a Web application's deployment descriptor specifies whether the Web descriptor and other related deployment descriptors for this module (such as Web service descriptors) are complete. If the `web.xml` file uses servlet 2.5 by setting `version="2.5"` or points to the servlet 2.5 schema namespace, the OC4J servlet container will check the `metadata-complete` flag to determine whether or not to process annotations. If `version` is set to `2.4` or an earlier version, the servlet container does not process any annotations.

If `metadata-complete` is set to `true`, the default value for Servlet 2.4, the servlet container ignores any servlet annotations that are in the class files of the application. If the `metadata-complete` attribute is missing or is set to `false`, the default value for Servlet 2.5, and `version` is set to `2.5`, the servlet container examines the class files of the application for servlet annotations and supports the annotations as follows:

1. The OC4J servlet container inspects resource or service references for annotation references for classes of a Web application that are located in the `WEB-INF/` directory or in a JAR file under the `WEB-INF/lib/` directory.

   The servlet container also provides annotation support for jar files listed in `MANIFEST.MF`.

2. The OC4J servlet container provides annotation support for managed component classes that are declared in the Web application deployment descriptor and that implement the following interfaces:

   - `javax.servlet.Servlet`
   - `javax.servlet.Filter`
   - `javax.servlet.ServletContextListener`
   - `javax.servlet.ServletContextAttributeListencer`
   - `javax.servlet.ServletRequestListener`
   - `javax.servlet.ServletRequestAttributeListener`
   - `javax.servlet.http.HttpSessionListener`
   - `javax.servlet.http.HttpSessionAttributeListener`

3. The OC4J servlet container injects resource or service references before any lifecycle methods are called on the instance.

   - `(init()` for `javax.servlet.Servlet` and `javax.servlet.Filter`
   - `contextInitialized()` for `javax.servlet.ServletContextListener`
   - `requestInitialized()` for `javax.servlet.ServletRequestListener`

4. If both an annotation and a deployment descriptor entry declare an environment entry, information in the deployment descriptor entry can override some of the information in a `Resource`, `EJB`, or `WebServiceRef` annotation.

   The following specifications describe the rules for using a deployment descriptor entry to override annotation information:

   - For rules to override `Resource` annotations, see the Java EE 5 specification.
   - For rules to override `EJB` annotations, see the EJB specification.

- For rules to override `WebServiceRef` annotations, see the Web Services specification.

5. If the OC4J servlet container cannot find a resource or service that it needs to inject for a class, the initialization of the class fails and OC4J issues this warning message:

   ```
   Some resource(s) and/or service(s) to be injected cannot be found for class
   name. class name will not be put to service.
   ```

6. After all the resource or service references have been injected, and before any lifecycle methods are called on the instance, the method marked with the `PostConstruct` annotation, if any, must be invoked to give the servlet a chance to initialize the injected resources. See "PostConstruct Annotation" on page 7-5.
7. Before the servlet is taken out of service, the method marked with a `PreDestroy` annotation, if any, must be invoked to give the servlet a chance to release the injected resources. See "PreDestroy Annotation" on page 7-5.

Annotations in OC4J

This section describes the annotations that OC4J supports:

EJB Annotation

Resource Annotation

Resources Annotation

PostConstruct Annotation

PreDestroy Annotation

PersistenceUnit(s) Annotation

PersistenceContext(s) Annotation

WebServiceRef Annotation

DeclaresRoles Annotation

RunAs Annotation

EJB Annotation

An `EJB` annotation on a field or method of an application component is equivalent to an `ejb-ref` or `ejb-local-ref` element in the deployment descriptor. If a field has an `EJB` annotation, OC4J injects the field with a reference to the corresponding EJB component.

In an `EJB` annotation, you can refer to the local or remote home interface of the bean or to the business interface of an EJB 3 bean. If the reference is to the EJB 3 business interface, OC4J injects a reference to an instance of the enterprise bean.

The following example shows a short `EJB` annotation:

```
@EJB private ShoppingCart myCart;
```

The next example shows a longer `EJB` annotation, which uses all of the annotation fields:

```
@EJB(
   name = "ejb/shopping-cart",
   beanName = "Cart1",
```

```
    beanInterface = ShoppingCart.class,
    description = "Items for purchase"
)
private ShoppingCart myCart;
```

For more details about the `EJB` annotation, see the EJB specification.

The example set in the "Annotation Example" section on page 7-8 includes EJB annotation.

Resource Annotation

Use the `Resource` annotation to declare a reference to a resource such as a data source, JMS destination, or environment entry. If you use this annotation, you do not need to declare the reference in a `<resource-ref>`, `<message-destination-ref>`, `<env-ref>`, or `<resource-env-ref>` element in the deployment descriptor.

When a `Resource` annotation is applied on a field or a setter method, OC4J injects a reference to the resource declared by the annotation and maps the references to the JNDI name for the resource. When the annotation is applied to a class, the annotation declares a resource that the application will look up at runtime.

Each injection corresponds to a JNDI lookup. If the annotation does not explicitly specify the JNDI name, the name of the field combined with the fully qualified name of the class is used as the JNDI name. For example, the default JNDI name of a field named `myDb` in a class `MyApp` in the package `com.example` would be `java:comp/env/com.example.MyApp/myDb`. All JNDI names are relative to `java:comp/env/`. If the annotation is applied on a setter method, the default is the JavaBeans property name corresponding to the method qualified by the class name. When the annotation is applied to a class, there is no default and the name must be specified.

The following example does not specify the JNDI name, so OC4J would use the default JNDI name:

```
@Resource
private DataSource myDB;
```

The next example explicitly specifies the JNDI name:

```
@Resource(name="customerDB")
private DataSource myDB;
```

For general information about annotations, see the Java EE 5 specification.

The example set in the "Annotation Example" section on page 7-8 includes resource annotation.

Resources Annotation

Repeated annotations are not allowed, so the `Resources` annotation acts as a container for multiple `Resource` annotations, in this format:

```
public @interface Resources {
    Resource[] value;
}
// value - Array of multiple resources.
```

The following example shows a `Resources` annotation that contains two `Resource` annotations on a class, for a data source and a connection factory:

```
@Resources ({
   @Resource (name = "myDB", type=javax.sql.DataSource),
   @Resource (name = "myCF", type=javax.jms.ConnectionFactory)
)
public class MulResClass {
   //...
}
```

PostConstruct Annotation

OC4J invokes the method with the `PostConstruct` annotation after all other resource injections have been completed and before any lifecycle methods on a component is called. This allows the component to do post-create processing. OC4J invokes this method even if the class has no other annotations.

The following example shows a `PostConstruct` annotation:

```
@PostConstruct
void doPostInjectionProcessing {
   //...
```

Note: The `PostConstruct` annotation lets an arbitrary method of a servlet act as the `init()` method.

PreDestroy Annotation

OC4J invokes the method with the `PreDestroy` annotation before taking the servlet out of service. This allows the servlet to release the injected resources. OC4J invokes this method even if the class has no other annotations.

The following example shows a `PreDestroy` annotation:

```
@PreDestroy
void doPreDestroyProcessing {
   //...
   }
```

Note: The `PreDestroy` annotation lets an arbitrary method of a servlet act as the `destroy()` method.

PersistenceUnit(s) Annotation

A `PersistenceUnit` annotation is required for using EJB 3.0 persistence. A persistence unit has configuration details about entity managers (persistence contexts) that manage a set of related entity beans. "PersistenceContext(s) Annotation" on page 7-6 describes the annotation for persistence contexts.

You can annotate a field or a method of a servlet with a PersistenceUnit annotation. A logical persistent unit reference refers to an entity manager factory for a persistence unit.

The following example declares a single persistence unit:

```
@PersistenceUnit
EntityManagerFactory emf;
```

If you declare multiple persistence units in a servlet, you must specify an explicit unitName for each unit, as follows:

```
@PersistenceUnit(unitName="InventoryManagement")
   EntityManagerFactory emf;
```

A `PersistenceUnit` annotation is equivalent to a `persistence-unit-ref` element in `persistence.xml`.

For more information about persistence unit references, see the Java EE 5 specification.

PersistenceContext(s) Annotation

A `PersistenceContext` annotation is required for using EJB 3.0 persistence. A persistence context is an entity manager that manages a set of related entity beans. A persistence unit has configuration details about a persistence context. "PersistenceUnit(s) Annotation" on page 7-5 describes the annotation for persistence units. An example of a `PersistenceContext` annotation follows:

```
@PersistenceContext
EntityManager em;
```

If multiple persistence units are declared in the servlet, an explicit unit name must be specified in the `unitName` attribute, as follows:

```
@PersistenceContext(unitName="InventoryManagement")
EntityManager em;
```

This annotation is equivalent to a `persistence-context-ref` element in `persistence.xml`.

For more information about persistence context references, see the Java EE 5 specification.

WebServiceRef Annotation

Annotating with the `WebServiceRef` annotation is equivalent to declaring a `<resource-ref>` element in the deployment descriptor that would provide a reference to a Web service.

This annotation has two main uses:

- Define a reference whose type is a generated service interface
- Define a reference whose type is a service endpoint interface (`SEI`)

An example of a `WebServiceRef` annotation follows:

```
// Generated Service Interface
@WebServiceRef
private StockQuoteService stockQuoteService;

// SEI
@WebServiceRef(StockQuoteService.class)
private StockQuoteProvider stockQuoteProvider;
```

For more details about the `WebServiceRef` annotation, see *Java API for XML-Based Web Services, 2.0* (JSR 224).

DeclaresRoles Annotation

The `DeclaresRoles` annotation defines all the security roles that compose the security model of an application. You can specify this annotation on a class, defining roles that you can test from within the methods of the annotated class, by calling `isCallerInRole`.

The following example shows a `DeclaresRoles` annotation:

```
@DeclaresRoles("Manager")
public class CorporationServlet {
   //...
}
```

This annotation is equivalent to the following code in a `web.xml` file:

```
<web-app>
   <security-role>
      <role-name>Manager</role-name>
   </security-role>
</web-app>
```

For more details about the `DeclaresRoles` annotation, see *Common Annotations for the JavaTM Platform* (JSR 250)

RunAs Annotation

The `RunAs` annotation is equivalent to the `<run-as>` element in the deployment descriptor. This annotation can only be used in classes that implement the `javax.servlet.Servlet` interface or a subclass of it.

The following example shows a `RunAs` annotation:

```
@RunAs("Admin")
public class CorporationServlet {
   //...
}
```

This annotation is equivalent to the following code in a `web.xml` file:

```
<servlet>
   <servlet-name>CorporationServlet</servlet-name>
   <run-as>Admin</run-as>
</servlet>
```

For more details about the `RunAs` annotation, see *Common Annotations for the JavaTM Platform* (JSR 250)

Annotation Rules and Guidelines

Here are some rules for using annotations. For more information on general annotation guidelines, see the Java EE 5 specification.

- A field or a method of the following container-managed component classes can be annotated to request that an entry from the component's environment be injected into the class:

```
javax.servlet.Servlet
javax.servlet.Filter
javax.servlet.ServletContextListener
javax.servlet.ServletContextAttributeListencer
```

```
javax.servlet.ServletRequestListener
javax.servlet.ServletRequestAttributeListener
javax.servlet.http.HttpSessionListener
javax.servlet.http.HttpSessionAttributeListener
```

- A field or method can have any access qualifier (such as `public` or `private`).
- A field or method cannot be static, except for the fields or methods of the application client main class that have been annotated for injection. These fields or methods must be static.
- Any violation of the preceding rules is an error that will result in a message or messages being logged. The violating class will not be put in service.
- Annotations applied on a class do not cause resource injections. Instead, they declare an entry in an application component's environment that the application component can look up through JNDI or the component context lookup method.
- Resource annotations can be specified in any of the classes in the first item of this list or on any of their superclasses, as the following example shows. Injection of resources follows the Java language overriding rules for visibility of fields and methods.

```
Class A {

   @Resource
   private DataSource myDS; // Injected resource not visible in class B
   //...
}

Class B extends Class A {
   public DataSource myDS; // Not a resource, needs to do JNDI lookup
   //...
}
```

How Annotations Affect Performance with Servlet Version 2.5

Whether or not annotations are used for Web applications with servlet version 2.5, if the `metadata-complete` attribute of `<web-app>` is missing from the deployment descriptor or is set to `false` (the servlet 2.5 default value), OC4J needs to load all the classes under `WEB-INF/` and `WEB-INF/lib` to look for annotations. This loading can have some impact on Oracle Application Server performance at startup. When you are not using annotations, you can avoid this performance impact by specifying `metadata-complete="true"`.

This startup performance impact is applicable only for Web applications with servlet version 2.5. For Web applications with servlet version 2.4 or lower, there will be no performance impact.

Annotation Example

This section contains an example of a servlet using annotations and the corresponding `web.xml` file.

The servlet illustrates the Servlet 2.5 way of doing things and then commented out is the Servlet 2.4 way of doing things as a comparison. The servlet demonstrates one `@EJB` annotation and two `@Resource` annotations.

The `web.xml` file illustrates the `version=` and `metadata-complete` settings required to enable annotations to be used:

```
<web-app version="2.5" metadata-complete="false">
 ...
</web-app>
```

For additional information, see the document "*How-To: Using Dependency Injection In Web Module*" on the following site:

http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/index.html

Here is the web.xml file example.

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version="2.5"
        metadata-complete="false"
>

    <display-name>Annotation Example</display-name>
<description>A few examples of Servlet 2.5 Annotation and Resource
Injection</description>

    <servlet>
        <display-name>hello</display-name>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>HelloServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

  <env-entry>
    <env-entry-name>EmpNo</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>15</env-entry-value>
  </env-entry>
</web-app>
```

Here is the HelloServlet.java servlet example.

```
import java.io.*;

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

```
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import org.acme.*;
import javax.rmi.*;
import javax.ejb.*;
public class HelloServlet extends HttpServlet {

    @EJB HelloObject bean;
    @Resource(name="EmpNo") int empNo;
    @Resource(name="jdbc/OracleDS") private DataSource db;

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/html");

        PrintWriter writer = res.getWriter();

        // ejb invocation
        writer.println(bean.sayHello()+"<br>");

        // fetch value set by deployer
        writer.println("EmpNo="+empNo+"<br>");

        // make a db connection
        writer.println("db="+getConnection()+"<br>");
    }

    public Connection getConnection() {
      Connection conn = null;

      try {
          if (db != null) {
              conn = db.getConnection();
          }
      } catch (Exception e) {
          e.printStackTrace();
      }
      return conn;
    }

    // The following is the pre-2.5 way to do it.
/*

   public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/html");

        PrintWriter writer = res.getWriter();

        InitialContext ctx = new InitialContext();
        Object obj = ctx.lookup("java:comp/env/ejb/Hello");

        HelloHome ejbHome = (HelloHome)
                    PortableRemoteObject.narrow(obj,HelloHome.class);
        HelloObject bean = ejbHome.create();

        // ejb invocation
        writer.println(bean.sayHello()+"<br>");
```

```
        // fetch value set by deployer
        Context myEnv = (Context) ctx.lookup("java:comp/env");
        Integer empNoInteger = (Integer) myEnv.lookup("EmpNo");
        int empNo = empNoInteger.intValue();

        writer.println("EmpNo="+empNo+"<br>");

        // make a db connection
        writer.println("db="+getConnection()+"<br>");
    }

    public Connection getConnection() {

        Context initCtx = new InitialContext();
        javax.sql.DataSource db =
           (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/OracleDS");

        Connection conn = null;

        try {
            if (db != null) {
                conn = db.getConnection();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return conn;
    }
*/
}
```

8

Using JDBC or Enterprise JavaBeans

Dynamic Web applications typically access a database to provide content. This chapter shows how servlets can use JDBC, the Java standard API for database connectivity. It also provides an overview of Enterprise JavaBeans, which you can call from servlets to perform server-side business logic or manage data persistence for an application. The following sections are included:

- Using JDBC in Servlets
- Overview of Enterprise JavaBeans

Using 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 for the database connection, and to use JNDI, the Java Naming and Directory Interface, to look up the data source. The following subsections describe the basic steps involved and supply an example of this functionality:

- Why Use JDBC?
- Configuring a Data Source and Resource Reference
- Implementing JDBC Calls
- Database Query Servlet Example

For information about JDBC, see the *Oracle Database JDBC Developer's Guide and Reference*.

Why Use JDBC?

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

JDBC is the standard Java mechanism for accessing a database.

Notes:

- The general assumption is that you will use an Oracle database and Oracle JDBC driver. For connection to a non-Oracle database, you can use a DataDirect JDBC driver, provided with Oracle Application Server.
- Instead of using JDBC directly from a servlet, you can use EJBs to access data instead. Also see "Overview of Enterprise JavaBeans" on page 8-8.

Configuring a Data Source and Resource Reference

Your database connection will presumably use a standard data source. This section describes steps to configure a data source that you can use through JNDI:

1. Configure the Data Source
2. Configure the Resource Reference

See the *Oracle Containers for J2EE Services Guide* for more information about data sources and their configuration in OC4J.

Configure the Data Source

To use a data source, you must add it to the central OC4J data source configuration. Typically perform this step through Oracle Enterprise Manager 10*g* Application Server Control.

In the Application Server Control Console:

1. From the applicable Application Home page, or from the OC4J Home page, select the Administration tab.
2. Go to the task "JDBC Resources".
3. From the JDBC Resources page, you can create a data source. You can also edit a data source created previously. You can also create or edit connection pools from this page.

Configuring a data source results in new or updated entries in the `j2ee/home/config/data-sources.xml` file, following the form shown below (in this example, to use the Oracle JDBC Thin driver). Note the following:

- The `<connection-pool>` element has settings for a JDBC connection pool and specifies the name of the pool. (Connection pooling improves performance by taking a connection from an existing pool of connection objects, rather than going through the overhead of creating a new connection object.)
- The `<connection-factory>` subelement of `<connection-pool>` specifies the class to use as a factory for connections (in this case presumably a class representing a data source) and the database user name, password, and connection string.
- The `<managed-data-source>` element specifies the name (`name`) and JNDI location (`jndi-name`) of the data source, and references the connection pool specified in the `<connection-pool>` element.

See "Configure the Data Source for the Query Servlet" on page 8-5 for an example.

```
<data-sources ... >
```

```
    <connection-pool name="poolname">
        <connection-factory factory-class="package.Classname"
                            user="user"
                            password="password"
                            url="jdbc:oracle:thin:@host:port/service"/>
    </connection-pool>
    <managed-data-source connection-pool-name="poolname"
                         jndi-name="jndiname"
                         name="name"/>
</data-sources>
```

Note: For the `url` entry, the *`host`*:*`port`*:*`sid`* form is also still supported, but deprecated.

Configure the Resource Reference

To use a data source and JNDI lookup, there must also be an appropriate resource reference entry in the `web.xml` file. Here is an example, which corresponds to the data source configuration example shown in the preceding section:

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

This establishes that the `jdbc/OracleDS` resource is of type `DataSource`, for use as a data source.

Note: Always use the `Container` setting for the `<res-auth>` element, indicating that the container, as opposed to application component code, performs the sign-on to the resource.

Implementing JDBC Calls

This section shows typical steps to access a database through JDBC in servlet code. See "Write the Query Servlet" on page 8-5 for a complete example.

1. Import required packages. In addition to the servlet and `java.io` packages, there are packages that include classes for JDBC, data sources, and JNDI:

```
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.*;
```

2. Implement the `init()` method to perform the JNDI lookup of the data source and to establish the database connection, inside a `try...catch` block. The lookup here corresponds to the examples shown in "Configuring a Data Source and Resource Reference" on page 8-2.

```
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);
    }
  }
```

3. Implement the appropriate servlet `doXXX()` method, such as `doGet()`, and use JDBC to perform the desired SQL operations. In this example, assume a SQL query string has been constructed in a string `query`. The code creates a JDBC statement object, performs the query, loops through the result set to print the data records (where `out` is a `PrintWriter` object), then closes the statement and result set objects. SQL operations are also performed inside a `try...catch` block.

```
try {
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  while (rs.next()) {
     out.println(rs.getString(1) + rs.getInt(2));
  }
  rs.close();
  stmt.close();
}
catch (SQLException se) {
  se.printStackTrace(out);
}
```

4. Implement the `destroy()` method to close the database connection (also inside a `try...catch` block).

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

Database Query Servlet Example

This example has an HTML welcome page that prompts the user for the `LIKE` specification that completes the following query:

```
SELECT ename, empno FROM emp WHERE ename LIKE xxx
```

The welcome page then invokes a servlet to perform the query and output the results.

The following sections show how to implement and configure the example:

- Configure the Data Source for the Query Servlet
- Write the HTML Welcome Page
- Write the Query Servlet
- Configure the Servlet and JNDI Resource Reference
- Package the Query Example
- Invoke the Query Example

Configure the Data Source for the Query Servlet

Here is the data source configuration for this example, as reflected in the OC4J `data-sources.xml` file, configurable through the Application Server Control Console, as described in "Configure the Data Source" on page 8-2. This example uses the Oracle JDBC Thin driver to access a database on host `myhost` through port `5521` using service name `myservice`, connecting as user `scott`. (This is a simplified example—there are ways to avoid exposing the password in `data-sources.xml`.) The example also uses connection pooling, and the class `OracleDataSource` to represent the data source from which connections are obtained. The `jndi-name` entry, `jdbc/OracleDS`, is used by the servlet for the JNDI lookup of the data source.

```
<data-sources>
       <connection-pool name="ConnectionPool1">
             <connection-factory factory-class="oracle.jdbc.pool.OracleDataSource"
                                 url="jdbc:oracle:thin:@myhost:5521/myservice"
                                 user="scott" password="tiger"/>
       </connection-pool>
       <managed-data-source connection-pool-name="ConnectionPool1"
                            jndi-name="jdbc/OracleDS" name="OracleDS"/>
</data-sources>
```

Write the HTML Welcome Page

Here is the welcome page, `empinfo.html`, prompting the user to complete the query, then invoking the query servlet. For this example, the servlet is deployed to be invoked with the context path and servlet path of `/myquery/getempinfo`.

```
<html>
<head>
<title>Query the Employees Table</title>
</head>
<body>
<form method=GET ACTION="/myquery/getempinfo">
The query is<br>
SELECT ename, empno FROM emp WHERE ename LIKE xxx

<p>
Specify the WHERE clause xxx parameter.<br>
Enclose entry in single-quotes; use % for wildcard. Search is case-sensitive.<br>
Example: 'S%' (for all names starting with 'S').<br>
<input type=text name="queryVal">
<p>
<input type=submit>
</form>
</body>
</html>
```

Write the Query Servlet

Here is the query servlet, `GetEmpInfo`, implementing the steps described in "Implementing JDBC Calls" on page 8-3. There is also formatting for an HTML table for the output, and a counter for the number of rows retrieved.

```
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 LIKE specification for the WHERE clause from the user, through the */
/* HTTP request, then construct the SQL query.                                */
    String queryVal = req.getParameter("queryVal");
    String query =
      "select ename, empno from emp " +
      "where ename like " + queryVal;

    resp.setContentType("text/html");

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

/* Create a JDBC statement object and execute the query.     */
    try {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(query);

/* HTML table formatting for the output.                     */
      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. Using 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();
    }
  }
}
```

Configure the Servlet and JNDI Resource Reference

The `web.xml` file, in addition to configuration for the servlet, must include a resource reference entry for the data source. There is also configuration to declare `empinfo.html` as a welcome file. Here is the file for this example:

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<web-app>
   <servlet>
      <servlet-name>empinfoquery</servlet-name>
      <servlet-class>GetEmpInfo</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>empinfoquery</servlet-name>
      <url-pattern>getempinfo</url-pattern>
   </servlet-mapping>
   <resource-ref>
      <res-auth>Container</res-auth>
      <res-ref-name>jdbc/OracleDS</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
   </resource-ref>
   <welcome-file-list>
      <welcome-file>empinfo.html</welcome-file>
   </welcome-file-list>
</web-app>
```

Package the Query Example

The WAR file for this example, which we name `empinfo.war`, has the following contents and structure:

```
empinfo.html
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/GetEmpInfo.class
WEB-INF/classes/GetEmpInfo.java
```

And the EAR file is as follows:

```
empinfo.war
META-INF/Manifest.mf
META-INF/application.xml
```

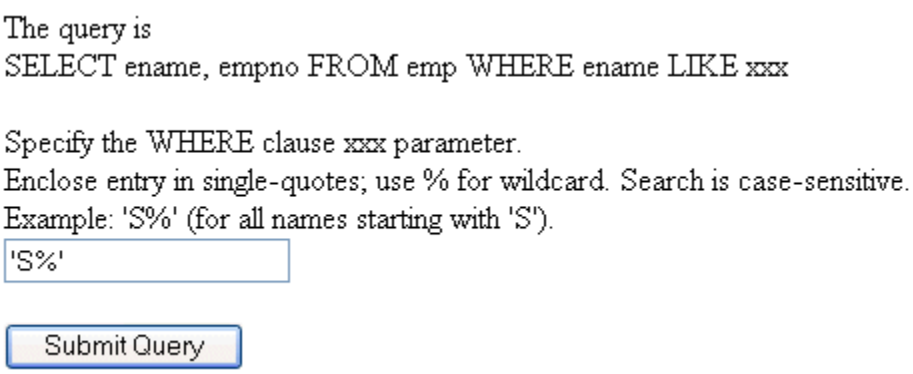
(The `Manifest.mf` files are created automatically by the JAR utility.)

Invoke the Query Example

For this example, assume that `application.xml` maps the context path `/myquery` to `empinfo.war`. In this case, after deployment, you can invoke the welcome page `empinfo.html` as follows (given the declaration of `empinfo.html` as a welcome page in `web.xml`):

```
http://host:port/myquery
```

In a test run, we specify `'S%'` to look for any names starting with "S":



For a database used for the test run, this returned two entries:

Last Name	Employee ID
SMITH	7369
SCOTT	7788

2 rows retrieved

TopLink Servlet Examples

A servlet can use Oracle TopLink to provide J2EE persistence for an application. You can find TopLink servlet examples on the Oracle Technology Network, at this Web site:

`http://www.oracle.com/technology/products/ias/toplink/examples/index.html`

Overview of Enterprise JavaBeans

A servlet can call Enterprise JavaBeans to access a database or perform additional business logic. The following sections offer an overview of EJBs and their use from servlets:

- Why Use Enterprise JavaBeans?
- EJB Support in OC4J and Oracle Application Server
- Servlet-EJB Lookup Scenarios

- EJB Local Interfaces Versus Remote Interfaces
- Using the Remote Flag for Remote Lookup within the Same Application

For detailed information about EJB features, and for servlet-EJB examples in an Oracle Application Server environment, refer to the *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide*.

> **Note:** OC4J provides an EJB tag library to make accessing EJBs from JSP pages more convenient. See the *Oracle Containers for J2EE JSP Tag Libraries and Utilities Reference* for information.

Why Use Enterprise JavaBeans?

EJBs have many uses in business applications, including the use of session beans for server-side business logic and entity beans to manage data persistence. EJB technology provides a more robust infrastructure than JSP or servlet technology, for use in secure, transactional, server-side processing.

A typical application design often uses a servlet as a front-end controller to process HTTP requests, with EJBs being called to access or update a database, and finally another servlet or JSP page being used to display data for the requester.

There are three categories of EJBs: session beans, entity beans, and message-driven beans. Container Managed Persistence entity beans, in particular, are well-suited to manage persistent data, because they make it unnecessary to use the JDBC API directly when accessing a database. Instead, you can let the EJB container handle database operations for you transparently.

Session beans are useful to model business logic and may be either stateless or stateful, with stateful beans typically being used where transaction state must be maintained across method calls or servlet requests. Stateless beans contain individual business logic methods that are independent of application state.

EJB Support in OC4J and Oracle Application Server

OC4J provides full support for session beans, entity beans, and message driven beans. The entity bean implementation provides Bean Managed Persistence (BMP), Container Managed Persistence (CMP), local interfaces, container-managed relationships, and the ability to perform queries using the EJB query language.

Within the entity bean implementation, a basic persistence manager supports both simple mapping and complex mapping, supporting one-to-one, one-to-many, many-to-one, and many-to-many object-relational mappings. It also automatically maps fields of an entity bean to a corresponding database table.

To facilitate application maintenance and deployment, Oracle Application Server provides a number of enhancements, including dynamic EJB stub generation. CORBA interoperability provides the capability to build EJBs and access them as CORBA services from CORBA clients.

Servlet-EJB Lookup Scenarios

There are three scenarios in calling an EJB from a servlet:

- 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. For this, use EJB local interfaces.

- Remote lookup within the same application: The servlet calls an EJB that is in the same application, but on a different host, where the 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.
- Remote lookup outside the application: The servlet calls an EJB that is not in the same application. This requires EJB remote interfaces. The EJB may be on a different host or on the same host, but is not running in the same JVM.

Servlet-EJB communications use JNDI for local and remote EJB calls. When a remote lookup is performed, JNDI uses either ORMI (the Oracle implementation of RMI) or IIOP (the standard and interoperable Internet Inter-Orb Protocol). In versions of EJB before 3.0, only home interfaces require JNDI lookup. They are then used to create EJBs for use by the application. J2EE components can use the default `no-args` constructor to look up objects within the same application. The `RMIInitialContextFactory` or `IIOPInitialContextFactory` class can be used for remote lookups. See the *Oracle Containers for J2EE Services Guide* for more information about JNDI in OC4J.

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, 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 configuration of the OC4J EJB `remote` flag for your application, on each host. See "Using the Remote Flag for Remote Lookup within the Same Application" on page 8-11.

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

EJB Local Interfaces Versus Remote Interfaces

In initial versions 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` parameter, which maps to an 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 Containers for J2EE Enterprise JavaBeans Developer's Guide* for information. Note that this parameter is configurable as `copyByValue` in the Application Server Control deployment plan editor, as discussed in the *Oracle Containers for J2EE Deployment Guide*.

In more recent versions, the EJB specification supports *local interfaces* for co-located EJB calls. 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 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.

Using the Remote Flag for Remote Lookup within the Same Application

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 OC4J EJB `remote` flag appropriately on each tier. When this flag is set to "`true`" on a server, beans will be looked up on a remote server instead of the EJB service being used on the local server.

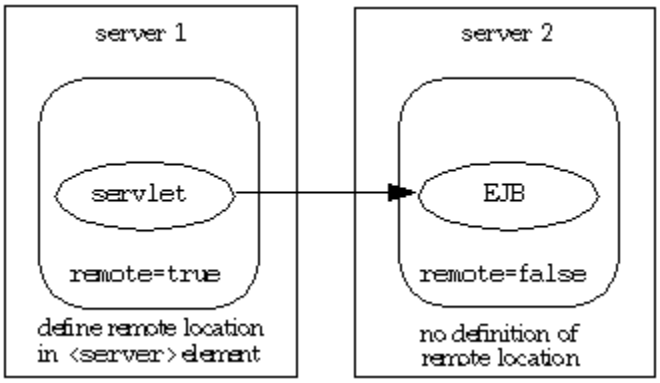
The `remote` flag maps to an attribute in the `<ejb-module>` subelement of an `<orion-application>` element in the `orion-application.xml` file. The default setting is `remote="false"`. Update the file to set this flag to "`true`", as follows:

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

(You cannot set this flag through Oracle Enterprise Manager 10*g* Application Server Control.)

You can deploy the application EAR file to both servers with a remote flag value of "`false`", then set it to "`true`" on server 1, the servlet tier. This is illustrated in Figure 8–1.

Figure 8–1 Setup for Remote Lookup within Application



You must properly configure server 2 as a remote host to instruct OC4J to look for EJBs there. Specify `host`, `port`, `username`, and `password` settings in the `<server>` subelement of the applicable `<rmi-server>` element in the `rmi.xml` file on server 1, as follows:

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

See the *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide* for details about using remote hosts and the `remote` flag.

Note: Use the default administrative user name for the remote host, and the administrative password set up on the remote host. This avoids possible JAZN configuration issues. See the *Oracle Containers for J2EE Security Guide* for information about JAZN.

9

Best Practices and Performance

This chapter provides programming tips for how to maximize the efficiency and performance of your servlets—general suggestions as well as suggestions regarding sessions, security, and thread models. There is also an introduction to the Oracle Dynamic Monitoring Service (DMS) to monitor performance. The following topics are covered:

- Best Practices for Sessions
- Best Practices for Security
- Considerations for Thread Models
- Best Practices for Performance
- Monitoring Performance

Best Practices for Sessions

This section discusses considerations when using sessions:

- 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.
- For performance reasons, OC4J does not wait to confirm successful replication of session state.

See the *Oracle Containers for J2EE Developer's Guide* for information about clustering in OC4J.

Best Practices for Security

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, verify appropriate settings as reflected in 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. You can edit them through the `jaznWebApp` property in the Application Server Control deployment plan editor, as described in the *Oracle Containers for J2EE Deployment Guide*.

- 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.
- 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. (See the *Oracle Containers for J2EE Configuration and Administration Guide* for general information about OC4J system properties.)

  – Set a `servlet-webdir` value of `""` (empty quotes), either through `global-web-application.xml` or `orion-web.xml`. This is editable through the `servletWebdir` property in the Application Server Control deployment plan editor.

  (Invocation by class name is described in "Invoking a Servlet by Class Name During OC4J Development" on page 2-11, 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.

For additional information, also see the reference documentation under "Elements and Attributes of orion-web.xml, global-web-application.xml" on page B-4.

Considerations for Thread Models

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 central `service()` method of the servlet.

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/docs/books/tutorial/essential/threads/multithreaded.html`

Custom Thread Pool

You can create one or more custom thread pools for selected applications to use instead of the default thread pool.

You create a custom thread pool in the `server.xml` file and then you make it available for applications to use by referring to it in one or more `*-web-site.xml` files.

Creating a Custom Thread Pool

To create a custom thread pool, add a `<custom-thread-pool>` element to the `server.xml` file. The `<custom-thread-pool>` element is a sub-element of the `<application-server>` element.

The name attribute is required, all other attributes are optional. The `<custom-thread-pool>` element has the same attributes as the other thread pool elements discussed in Chapter 10, "Task Manager and Thread Pool Configuration", of the *Oracle Containers for J2EE Configuration and Administration Guide*. The `server.xml` file is discussed in the "Overview of the OC4J Server Configuration File (server.xml)" section of Appendix B, "Configuration Files Used in OC4J" of the *Oracle Containers for J2EE Configuration and Administration Guide*.

The following example creates a custom thread pool named `mypool` in the `server.xml` file. The example specifies the following for `mypool`:

- The minimum number of threads to create in the pool is 10.
- The maximum number of threads that can be created in the pool is 100.
- The number of requests outstanding in each queue can be 50 requests.
- Idle threads are kept alive for 700 seconds.

```
<application-server ...>
...
```

```
   <custom-thread-pool name="mypool" min="10" max="100"
    queue="50" keepAlive="700000" debug="true"/>
...
</application-server>
```

Assigning a Custom Thread Pool to Applications

To instruct applications to use a custom thread pool instead of the default thread pool, add a reference to that custom thread pool in the `custom-thread-pool` attribute in the `<web-site>` element in one or more `*-web-site.xml` files.

Each web application assigned to the web site and port specified in the `*-web-site.xml` file by being named in a `<web-app>` element will use the custom thread pool named in the `custom-thread-pool` attribute.

The `*-web-site.xml` file is discussed in the "Overview of the Web Site Configuration Files (*-web-site.xml)" section of Appendix B, "Configuration Files Used in OC4J" of the *Oracle Containers for J2EE Configuration and Administration Guide*.

The following example shows the `custom-thread-pool` attribute used to name the `mypool` custom thread pool to be used by all applications named in the `<web-app>` elements of the `srdemo-web-site.xml` file:

```
  <?xml version="1.0" ?>
- <web-site xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/web-site-10_0.xsd"
port="12501" protocol="ajp13" display-name="SRDEMO Web Site" custom-thread-pool="mypool"
schema-major-version="10" schema-minor-version="0">
  <default-web-app application="default" name="defaultWebApp" root="/j2ee" />
  <web-app application="system" name="dms" root="/dmsoc4j" access-log="false" />
  <web-app application="system" name="JMXSoapAdapter-web" root="/JMXSoapAdapter" />
  <web-app application="default" name="jmsrouter_web" load-on-startup="true" root="/jmsrouter" />
  <web-app application="ascontrol" name="ascontrol" load-on-startup="true" root="/em" />
  <web-app application="bc4j" name="webapp" load-on-startup="true" root="/webapp" />
  <web-app application="SRDEMO" name="SRDEMO-WEB" load-on-startup="true" root="/SRDEMO" />
  <access-log path="../log/default-web-access.log" split="day" />
  </web-site>
```

Best Practices for Performance

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, as reflected in the `<expiration-setting>` subelement of `<orion-web-app>` in `global-web-application.xml` or `orion-web.xml`. A more appropriate setting decreases load on the application and improves performance. This is editable through the `expirationSettings` property in the Application Server Control deployment plan editor, as described in the *Oracle Containers for J2EE Deployment Guide*.
- 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 "Considerations for Thread Models" on page 9-2.

- Servlet configuration parameters can significantly affect performance. In particular be careful in using File Modification Check Interval, as reflected in the `file-modification-check-interval` attribute of the `<orion-web-app>` element in `global-web-application.xml` or `orion-web.xml`. This is configurable through the Application Server Control Console Configuration Properties Page (documented in "Configuration Properties Page" on page A-3) or through the `fileModificationCheckInterval` property in the Application Server Control deployment plan editor.

  Also be wary of the `use-keep-alives` attribute of the `<web-site>` element in `default-web-site.xml`. This attribute is discussed in the *Oracle Containers for J2EE Configuration and Administration Guide*.

- Additional JSP-related configuration parameters can significantly affect performance, particularly the `simple-jsp-mapping` and `enable-jsp-dispatcher-shortcut` attributes of the `<orion-web-app>` element in `global-web-application.xml` or `orion-web.xml`. These are configurable through the `EnableJspDispatcherShortcut` and `simpleJspMapping` properties in the Application Server Control deployment plan editor.

- OC4J in a standalone environment 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. You can enable this feature through the `shared` attribute of the `<web-app>` element in `default-web-site.xml`.

- If you ever use standalone OC4J as a production environment (although this is not typical), ensure that the `check-for-updates` flag in the `<application-server>` element in `server.xml` is turned off. This parameter is discussed in both the *Oracle Containers for J2EE Configuration and Administration Guide* and the *Oracle Containers for J2EE Deployment Guide*.

For further information, also see the reference documentation under "Elements and Attributes of orion-web.xml, global-web-application.xml" on page B-4.

Monitoring Performance

This section includes information about monitoring the performance of servlets.

Oracle Application Server Dynamic Monitoring Service

In an Oracle Application Server environment, the Dynamic Monitoring Service (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 DMS modules is reflected in the OC4J `system-application.xml` file and the `default-web-site.xml` file. In `system-application.xml`, the Web module `dms` and the path to its WAR file is specified. The `default-web-site.xml` file specifies that this Web module is

deployed to the OC4J default application and binds it to its context path. Do not directly alter any of these DMS configurations.

Use Application Server Control to access DMS, display DMS information, and, if appropriate, alter DMS configuration.

Refer to the *Oracle Application Server Performance Guide* for information about DMS, the *Oracle Containers for J2EE Developer's Guide* for information about the global `application.xml` file (which uses the same specification as the `orion-application.xml` file), and the *Oracle Containers for J2EE Configuration and Administration Guide* for information about Web site XML files.

A

Web Module Administration

This appendix provides reference documentation of OC4J features for administering Web modules through Oracle Enterprise Manager 10*g* Application Server Control, covering the following topics:

- Application Server Control Console Top-Level Web Module Pages
- Application Server Control Web Module Configuration Pages
- Summary of Web Module MBeans and Administration

Application Server Control Console Top-Level Web Module Pages

The Application Server Control Console provides a Web-based user interface for deploying, configuring, and monitoring applications, as well as managing the OC4J instance and the Web services used by your applications. It is installed, preconfigured, and started automatically when you install the OC4J software (either in a standalone or Oracle Application Server environment), and is bound to whichever port the OC4J instance is using. In a standalone environment, the port is typically 8888. In an Oracle Application Server environment, the port is usually 7777.

For example, in a standalone OC4J environment, you can use port 8888 of the appropriate host to access the console:

`http://host:1888`

See the Application Server Control Console online help for detailed instructions on using this interface.

The console is organized into functional areas for applications, administration, performance, and Web services. You can manage Web modules through the applications area.

Web module configuration starts from the applicable Web Module Home page in the Application Server Control Console. This page has a General tab, a Performance tab, and an Administration tab.

The rest of this section covers the following topics:

- How to Get to a Web Module Home Page
- Summary of Top-Level Web Module Pages

How to Get to a Web Module Home Page

To get to a Web Module Home page from the OC4J Home page, select the Applications tab. From there, you can get to a Web Module Home page in one of two ways:

1. View applications, by selecting "Applications" in the View dropdown menu.
2. Select the application of interest.
3. In the resulting Application Home page, select the module of interest.

or:

1. View modules from all applications, by selecting "Modules" in the View dropdown menu.
2. Select the module of interest.

Summary of Top-Level Web Module Pages

From a Web Module Home page, you can access the following:

- General tab (the Web Module Home page itself): This lists the host, port, and context path for the Web module, as well as the module's active servlets and JSP pages. For each servlet and JSP page, the General tab page lists the following metrics:
  - Active Requests
  - Current Client Processing Time (average processing time for each request over the preceding five minutes)
  - Requests per Second
  - Requests Processed
  - Total Client Processing Time (since startup of the servlet or JSP page)

  (These metrics are not persistent if OC4J is restarted.)
- Performance tab: This page graphically displays the following metrics for servlets and JSP pages that have been executed.
  - Active Sessions: The number of sessions active at any given point, over the indicated period of time.
  - Active Requests: The number of requests active at any given point, over the indicated period of time.
  - Response and Load: The average request processing time over the preceding five minutes, together with the requests per second over the preceding five minutes, over the indicated period of time.
- Administration tab: From this page, you can view or edit certain Web module configuration properties, view the `web.xml` or `orion-web.xml` file, and view or edit various kinds of mappings that relate to the Web module. Specifically, you can reach the following pages:
  - Configuration Properties page
  - View Deployment Descriptor page
  - View Proprietary Deployment Descriptor page
  - Servlet Mappings page
  - Filter Mappings page
  - Resource Reference Mappings page
  - EJB Reference Mappings page
  - Environment Entry Mappings page

See the next section, "Application Server Control Web Module Configuration Pages", for information about these pages.

For more information about the pages summarized in the preceding paragraphs, see the context-sensitive topics "Web Module Home Page", "Web Module Performance Page", and "Web Module Administration Page" in the Application Server Control online help. For more information about OC4J performance metrics, see the topic "Summary of the OC4J Performance Metrics" in the online help.

Application Server Control Web Module Configuration Pages

From the Web Module Administration page, you can select "Go to Task" for several viewing and editing functions related to Web module configuration. These are described in the following sections:

- Configuration Properties Page
- Deployment Descriptor Viewing Pages
- Servlet Mappings Page
- Filter Mappings Page
- Resource Reference Mappings Page
- EJB Reference Mappings Page
- Environment Entry Mappings Page
- Resource Reference Lookup Context Page

Notes:

- For `orion-web.xml` properties you can view or edit in these pages, corresponding elements and attributes are described under "Elements and Attributes of orion-web.xml, global-web-application.xml" on page B-4.
- Where properties are documented as being under the `<web-app>` element, existing values may be coming from either the `web.xml` file or the `orion-web.xml` file. The `<web-app>` element is the root element of `web.xml`, with the schema being defined in the servlet specification. This definition is also imported into the `orion-web.xml` schema definition. Settings under the `<web-app>` element in `orion-web.xml` effectively override any of the same settings under the `<web-app>` element in `web.xml`. Changes made to any of these properties through the Application Server Control Console are persisted to the `orion-web.xml` file.

Configuration Properties Page

Table A–1 discusses the Web module properties you can configure through the Configuration Properties page of the Application Server Control Console. This page is accessible from the Web Module Administration page. For additional information about the properties, see the appropriate elements under "Elements and Attributes of orion-web.xml, global-web-application.xml" on page B-4.

The Configuration Properties page also displays the Web module name and application name.

Also see the context-sensitive topic "Web Module Configuration Properties Page" in the Application Server Control online help.

Table A–1 Properties of the Configuration Properties Page

Application Server Control Property	Corresponding XML Entity	Description
Display Name	`<display-name>` element under `<web-app>`	A short name for the Web module, for display by tools. **Note**: This is read-only through Application Server Control
Description	`<description>` element under `<web-app>`	An optional description of the Web module. **Note**: This is read-only through Application Server Control
Distributable	`<distributable>` element under `<web-app>`	Indicates whether the application is distributable, as described in the servlet specification. This property is not editable through this page. **Note**: This is read-only through Application Server Control
Classpath	`<classpath>` subelement of `<orion-web-app>`	Informs OC4J of additional code locations for Web application class loading—either library files or locations for individual class files.
Persistence Path	`persistence-path` attribute of `<orion-web-app>`	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 `application-deployments` directory. If no value is specified, then there is no persistence of session objects across restarts or redeployments. **Note**: This attribute is ignored if OC4J clustering is enabled.
Temporary Directory	`temporary-directory` attribute of `<orion-web-app>`	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.

Table A–1 (Cont.) Properties of the Configuration Properties Page

Application Server Control Property	Corresponding XML Entity	Description
File Modification Check Interval	`file-modification-check-interval` attribute of `<orion-web-app>`	This attribute, in milliseconds, determines when to check a static file, such as an HTML file, to see whether its timestamp has changed and it should therefore be reloaded from the file system. The default is `"1000"`. For performance reasons, a very large value (`"1000000"`, for example) is recommended in a production environment.
Session Timeout	`<session-timeout>` subelement of `<session-config>` element under `<web-app>`	Defines the default session timeout for all sessions created in the Web application, in minutes. For a value of 0 or less, there is no timeout.
Default Buffer Size	`default-buffer-size` attribute of `<orion-web-app>`	Specifies the default size of the output buffer for servlet responses, in bytes. Without specifying, the default is `"2048"`.
Allow Directory Browsing	`directory-browsing` attribute of `<orion-web-app>`	Specifies whether to allow directory browsing for a URL that ends in `"/"`. Supported values are `"allow"` and `"deny"` (default). See "<orion-web-app>" on page B-16 for additional information about directory browsing.

Deployment Descriptor Viewing Pages

You can view, but not edit, the application `web.xml` file and `orion-web.xml` file through the View Deployment Descriptor page and View Proprietary Deployment Descriptor page, respectively, of the Application Server Control Console.

Servlet Mappings Page

In the Servlet Mappings page of the Application Server Control Console, you can view, but not edit, mappings between servlet names and URL patterns. These mappings are described in Table A–2.

Also see the context-sensitive topic "Web Module Servlet Mappings Page" in the Application Server Control online help.

Table A–2 Properties of the Servlet Mappings Page

Application Server Control Property	Corresponding XML Entity	Description
Servlet Name	`<servlet-name>` subelement of `<servlet-mapping>` element under `<web-app>`	A reference to the desired name of the servlet, as defined in the `<servlet-name>` subelement of a corresponding `<servlet>` element under `<web-app>`.

Table A–2 (Cont.) Properties of the Servlet Mappings Page

Application Server Control Property	Corresponding XML Entity	Description
URL Pattern	<url-pattern> subelement of <servlet-mapping> element under <web-app>	The desired URL pattern (servlet path) for the servlet, to map to the corresponding servlet name. The URL to invoke the servlet includes this pattern.

Filter Mappings Page

Table A–3 discusses the properties you can configure for servlet filters, through the Filter Mappings page.

Also see the context-sensitive topic "Web Module Filter Mappings Page" in the Application Server Control online help.

Table A–3 Configurable Properties of the Filter Mappings Page

Application Server Control Property	Corresponding XML Entity	Description
Filter Name	<filter-name> subelement of <filter-mapping> element under <web-app>	A reference to the desired name of the filter, as defined in the <filter-name> subelement of a corresponding <filter> element under <web-app>.
URL Pattern or Servlet Name	<url-pattern> or <servlet-name> subelement of <filter-mapping> element under <web-app>	This is to map either a URL pattern or a servlet name (not both) to the corresponding filter name.
Apply to Forwards	<dispatcher> subelement of <filter-mapping> element under <web-app>, with value of FORWARD	Use this for the filter to be applied to any "forward" targets matching the servlet name or URL pattern.
Apply to Requests	<dispatcher> subelement of <filter-mapping> element under <web-app>, with value of REQUEST	Use this in addition to an "Apply to Forwards" or "Apply to Includes" setting for the filter to also be applied to direct request targets matching the servlet name or URL pattern.
Apply to Includes	<dispatcher> subelement of <filter-mapping> element under <web-app>, with value of INCLUDE	Use this for the filter to be applied to any "include" targets matching the servlet name or URL pattern.
Apply to Errors	<dispatcher> subelement of <filter-mapping> element under <web-app>, with value of ERROR	Use this for the filter to be applied under the error page mechanism.

Table A–4 discusses addition servlet filter properties that are displayed for reference in the Filter Mappings page.

Table A–4 Reference Properties of the Filter Mappings Page

Application Server Control Property	Corresponding XML Entity	Description
Name	`<filter-name>` subelement of `<filter>` element under `<web-app>`	The desired name of the filter.
Class	`<filter-class>` subelement of `<filter>` element under `<web-app>`	The fully qualified name of the class containing the filter code.
Description	`<description>` subelement of `<filter>` element under `<web-app>`	An optional description of the filter.

Note: See Chapter 4, "Understanding and Using Servlet Filters", for related information.

Resource Reference Mappings Page

Use the Resource Reference Mappings page to specify a JNDI location for a resource such as a data source, JMS queue, or mail session. Table A–5 shows properties you can configure in this page.

Also see the context-sensitive topic "Web Module Resource Reference Mappings" in the Application Server Control online help.

From the Resource Reference Mappings page, you can also edit a JNDI lookup context, taking you to the Resource Reference Lookup Context page, described in "Resource Reference Lookup Context Page" on page A-9.

Table A–5 Properties of the Resource Reference Mappings Page

Application Server Control Property	Corresponding XML Entity	Description
Name	`name` attribute of `<resource-ref-mapping>` subelement of `<orion-web-app>`	The name of the resource, which refers to the value of a `<res-ref-name>` subelement of `<resource-ref>` under `<web-app>`. More specifically, this is the name of a resource manager connection factory reference.
Type	`<res-type>` subelement of `<resource-ref>` element under `<web-app>`	The type of the data source or other resource. This is the fully qualified Java type that is implemented by the resource.
Authorization	`<res-auth>` subelement of `<resource-ref>` element under `<web-app>`	Indicates whether sign-on to the resource manager is programmatic in the application component (Authorization value is `Application`) or is managed by the OC4J container (Authorization value is `Container`).
JNDI Location	`location` attribute of `<resource-ref-mapping>` element under `<orion-web-app>`	The desired JNDI location from which to look up the resource.

Table A–5 (Cont.) Properties of the Resource Reference Mappings Page

Application Server Control Property	Corresponding XML Entity	Description
Lookup Context	`location` attribute of `<lookup-context>` subelement of `<resource-ref-mapping>` element under `<orion-web-app>`	Specifies an optional JNDI context that will be used instead of the default context in looking up the resource.

EJB Reference Mappings Page

Use the EJB Reference Mappings page to specify a JNDI location for an EJB. Table A–6 discusses the properties in this page.

Also see the context-sensitive topic "Web Module EJB Reference Mappings Page" in the Application Server Control online help.

Table A–6 Properties of the EJB Reference Mappings Page

Application Server Control Property	Corresponding XML Entity	Description
Name	`name` attribute of `<ejb-ref-mapping>` subelement of `<orion-web-app>`	The name of the EJB, which refers to the value of an `<ejb-ref-name>` subelement of `<ejb-ref>` under `<web-app>`.
Type	`<ejb-ref-type>` subelement of `<ejb-ref>` element under `<web-app>`	The type of EJB, either `Entity` or `Session`.
Home Interface	`<home>` subelement of `<ejb-ref>` element under `<web-app>`	The fully qualified name of the home interface of the EJB.
Remote Interface	`<remote>` subelement of `<ejb-ref>` element under `<web-app>`	The fully qualified name of the remote interface of the EJB.
JNDI Location	`location` attribute of `<ejb-ref-mapping>` subelement of `<orion-web-app>`	The desired JNDI location from which to look up the EJB.

Note:

- See "Overview of Enterprise JavaBeans" on page 8-8 for related information.

Environment Entry Mappings Page

Use the Environment Entry Mappings Page to set a new value for an environment entry. Table A–7 discusses the properties in this page.

Also see the context-sensitive topic "Web Module Environment Entry Mappings Page" in the Application Server Control online help.

Table A–7 Properties of the Environment Entry Mappings Page

Application Server Control Property	Corresponding XML Entity	Description
Name	`name` attribute of `<env-entry-mapping>` subelement of `<orion-web-app>`	The name of the environment entry, and refers to the value of the `<env-entry-name>` subelement of an `<env-entry>` element under `<web-app>`.
Type	`<env-entry-type>` subelement of `<env-entry>` element under `<web-app>`	The Java type of the environment entry.
Description	`<description>` subelement of `<env-entry>` element under `<web-app>`	An optional description of the environment entry.
Value	`<env-entry-value>` subelement of `<env-entry>` element under `<web-app>`	The assembled value of the environment entry (typically from the `web.xml` file).
Deployed Value	`<env-entry-mapping>` subelement of `<orion-web-app>`	The desired deployment value of the environment entry, to override the assembled value. The value of the `<env-entry-mapping>` element overrides the value of the `<env-entry-value>` element.

Resource Reference Lookup Context Page

This page is linked from the Resource Reference Mappings page, described in "Resource Reference Mappings Page" on page A-7, allowing you to specify a new context from which to look up a resource. You can also edit context attributes.

Also see the context-sensitive topic "Resource Reference Lookup Context Page" in the Application Server Control online help.

Table A–8 Properties of the Resource Reference Lookup Context Page

Application Server Control Property	Corresponding XML Entity	Description
Resource Reference Name	`name` attribute of `<resource-ref-mapping>` subelement of `<orion-web-app>`	The name of the resource, which refers to the value of a `<res-ref-name>` subelement of `<resource-ref>` under `<web-app>`. More specifically, this is the name of a resource manager connection factory reference.
Lookup Context Location	`location` attribute of `<lookup-context>` subelement of `<resource-ref-mapping>` element under `<orion-web-app>`	Specifies an optional JNDI context that will be used instead of the default context in looking up the resource.
Name (of context attribute)	`name` attribute of `<context-attribute>` subelement of `<lookup-context>`	The name of an attribute to send to a nondefault, such as third-party, JNDI context specified as the corresponding Lookup Context.

Table A–8 (Cont.) Properties of the Resource Reference Lookup Context Page

Application Server Control Property	Corresponding XML Entity	Description
Value (of context attribute)	`value` attribute of `<context-attribute>` subelement of `<lookup-context>`	The desired value of the attribute.

Summary of Web Module MBeans and Administration

Standards-compliant MBeans play a role in OC4J runtime configuration. The following sections provide an overview:

- General Overview of OC4J MBean Administration
- Summary of OC4J Web Module MBeans

General Overview of OC4J MBean Administration

OC4J support for the JMX specification allows standard interfaces to be created for managing resources dynamically, including resources relating to resource adapters, in a J2EE environment. The OC4J implementation of JMX provides a JMX client, the System MBean Browser, that you can use to manage an OC4J instance through MBeans that are provided with OC4J.

An MBean is a Java object that represents a JMX manageable resource. Each manageable resource within OC4J, such as an application or a resource adapter, is managed through an instance of the appropriate MBean. Each MBean provided with OC4J exposes a management interface that is accessible through the System MBean Browser in the Application Server Control Console. You can set MBean attributes, execute operations to call methods on an MBean, subscribe to notifications of errors or specific events, and display execution statistics.

Note: This information is provided for reference, but key Web module configuration settings are exposed in a more user-friendly manner through other features of the Application Server Control Console, as discussed under "Application Server Control Web Module Configuration Pages" on page A-3.

To access the browser from the OC4J home page, select the Administration tab and then, under the list of tasks, go to the JMX task "System MBean Browser". From the browser, you can do the following:

- Select the MBean of interest in the left-hand frame.
- Use the Attributes tab in the right-hand frame to view or change attributes. An attribute that can be set has a field where you can type in a new value. Then apply the change.
- Use the Operations tab in the right-hand frame to invoke methods on the MBean. Select the operation of interest. In the Operation window, you can invoke it with specified parameter settings.
- Use the Notifications tab (where applicable) in the right-hand frame to subscribe to notifications. You can select each item for which you want notification, and then apply the changes.

- Use the Statistics tab (where applicable) in the right-hand frame to display execution statistics.

Be aware that MBeans and their attributes vary regarding when changes take effect. In the *runtime model*, changes take effect immediately. In the *configuration model*, some changes take effect when the resource is restarted, others when the application is restarted, and still others when OC4J is restarted. There is also variation in whether changes are persisted.

See the *Oracle Containers for J2EE Configuration and Administration Guide* for details. The System MBean Browser itself also provides information about the MBeans.

Summary of OC4J Web Module MBeans

OC4J exports a set of MBeans for each Web module, including the OC4J default Web application, to support administration during application runtime. Some OC4J MBeans are required in order to support the J2EE management specification, but may offer extended features. Other OC4J MBeans are Oracle extensions to the model.

Table A–9 summarizes the OC4J implementation of MBeans that relate to Web modules and are required of an application server according to JSR-77. These implementations are in the `oracle.oc4j.admin.management.mbeans` package.

Notes:

- MBeans are self-documenting in the System MBean Browser, providing some documentation of MBean attributes, operations, and notifications (as applicable).
- Most (if not all) MBean statistical properties are derived from DMS statistics. See "Oracle Application Server Dynamic Monitoring Service" on page 9-5 for an introduction to DMS.
- Regarding the default Web application: OC4J is installed with a default configuration that includes a default J2EE application (also known as the global application). The default application is, by default, the parent of all other J2EE applications in OC4J, except Application Server Control Console. 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 OC4J global `application.xml` file. In standalone OC4J, the default Web application is bound to a Web site through the `default-web-site.xml` file, and the default context path is "/".

Table A–9 Mandatory System MBeans for Web Modules

MBean	Description
Servlet	Manages an instance of a servlet, with properties corresponding to a `<servlet>` element in the `web.xml` file.

Table A–9 (Cont.) Mandatory System MBeans for Web Modules

MBean	Description
WebModule	Manages standard features of a Web module, with properties corresponding to the `<web-app>` element (outside of `<servlet>` subelements) in the `web.xml` file. The loadAllServletMBeans() method of the WebModule MBean creates the servlet MBeans without loading the underlying servlets. OC4J loads a servlet when it is invoked or when preloading is requested, as described under "Preloading Servlets" on page 2-16.

Table A–10 summarizes the OC4J MBean that relates to Web modules and is an Oracle extension. This MBean implementation is also in the `oracle.oc4j.admin.management.mbeans` package.

Table A–10 Additional System MBeans for Web Modules

MBean	Description
OC4JWebModule	Manages OC4J-specific features of a Web module, with properties corresponding to the `<orion-web-app>` element (outside of the `<web-app>` subelement) in the `orion-web.xml` file.

Notes Regarding OC4J Web Module MBeans

- `WebModule` attributes corresponding to the following `<web-app>` subelements are accessible through the Application Server Control Console: `<context-param>`, `<servlet-mapping>`, `<filter-mapping>`, `<session-timeout>`.
- `OC4JWebModule` attributes corresponding to the following `<orion-web-app>` attributes and subelements are accessible through the Application Server Control Console: `file-modification-check-interval`, `directory-browsing`, `<classpath>`, `<resource-ref-mapping>`, `<env-entry-mapping>`, `<ejb-ref-mapping>`.
- The following JSP-related `<orion-web-app>` attributes, being global in nature, do not apply to `OC4JWebModule`: `jsp-cache-directory`, `jsp-cache-tlds`, `jsp-taglib-locations`, `jsp-print-null`, `jsp-timeout`.

B

Web Module Configuration Files

This appendix contains reference information for the OC4J-specific Web module configuration files `global-web-application.xml` (for global and default configuration) and `orion-web.xml` (for application-level configuration). There is also an overview of these files and their relationship to the standard `web.xml` file.

- Overview of Web Application Configuration Files
- Hierarchy of orion-web.xml and global-web-application.xml
- Elements and Attributes of orion-web.xml, global-web-application.xml

Overview of Web Application Configuration Files

A *Web descriptor* specifies and configures a set of J2EE Web components: static pages, servlets, and JSP pages. The Web components can together form an independent Web application and be deployed in an independent 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 web.xml Configuration File
- Oracle global-web-application.xml Configuration File
- Oracle orion-web.xml Configuration File
- Summary of Relationship Between Web Application Configuration Files

Standard web.xml Configuration File

The servlet specification defines the concept and XSD 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 sample `web.xml` configuration specifying, among other things, a servlet, the servlet mapping, and a local EJB lookup:

```
<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>
```

Oracle global-web-application.xml Configuration File

The OC4J `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 XSD `orion-web.xsd`. This is the same XSD as for the application-level, OC4J-specific Web descriptor, `orion-web.xml`, described in the next section, "Oracle orion-web.xml Configuration File".

The `orion-web` XSD is a superset of the standard XSD for `web.xml`. A `<web-app>` subelement of the `<orion-web-app>` top-level element in the `orion-web` XSD 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` if possible. 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, see "Elements and Attributes of orion-web.xml, global-web-application.xml" on page B-4.

Oracle orion-web.xml Configuration File

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 a corresponding XSD. This is the same XSD as for the global Web application descriptor that was described in the previous section, "Oracle global-web-application.xml Configuration File".

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`..

> **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, see "Elements and Attributes of orion-web.xml, global-web-application.xml" on page B-4.

Summary of Relationship Between Web Application Configuration Files

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`.

Hierarchy of orion-web.xml and global-web-application.xml

Here is an overview of the element hierarchy for the `global-web-application.xml` and `orion-web.xml` files.

```
<orion-web-app>
   <classpath>
   <context-param-mapping>
   <mime-mappings>
   <virtual-directory>
   <access-mask>
      <host-access>
      <ip-access>
   <servlet-chaining>
   <request-tracker>
   <session-tracking>
      <session-tracker>
   <resource-ref-mapping>
      <lookup-context>
         <context-attribute>
   <resource-env-ref-mapping>
   <security-role-mapping>
      <group>
      <user>
   <env-entry-mapping>
   <ejb-ref-mapping>
   <service-ref-mapping>
   <expiration-setting>
   <web-app>
   <jazn-web-app>
   <web-app-class-loader>
   <ojsp-init>
```

Note: You cannot use an <ojsp-init> element in `global-web-application.xml`.

Elements and Attributes of orion-web.xml, global-web-application.xml

This section is an alphabetical dictionary of elements of the `orion-web.xml` and `global-web-application.xml` files. See the preceding section, "Hierarchy of orion-web.xml and global-web-application.xml", if you are interested in the hierarchy.

The element descriptions in this section generally apply to either `global-web-application.xml` or to an application-specific `orion-web.xml` configuration file. The `global-web-application.xml` file configures the global application and sets defaults; the `orion-web.xml` file can override these defaults for a particular application deployment, as appropriate. See "Summary of Relationship Between Web Application Configuration Files" on page B-3 for a summary.

Notes:

- Where attributes are discussed, note that attribute values are always set inside quotes: `attribute="value"`.
- Most attributes of interest can be set through the Application Server Control deployment plan editor. See the *Oracle Containers for J2EE Deployment Guide* for information.

<access-mask>

Parent element: <orion-web-app>

Child elements: <host-access>, <ip-access>

Required? Optional; zero or one

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.

Table B–1 <access-mask> Attributes

Name	Description
default	Values: allow | deny Default: allow Specifies whether to allow requests from clients not identified through a `<host-access>` or `<ip-access>` subelement. 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.

<classpath>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

Use this element to inform OC4J of additional code locations for Web application class loading—either library files or locations for individual class files.

Table B–2 <classpath> Attributes

Name	Description

path	Values: String
	Default: n/a (required)
	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-attribute>

Parent element: <lookup-context>

Child elements: None

Required? Required if you use `<lookup-context>`; one or more

Each occurrence of this element specifies an attribute to send to a nondefault, such as third-party, JNDI context named in the parent `<lookup-context>` element.

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

Table B–3 <context-attribute> Attributes

Name	Description
name	Values: String Default: n/a (required) Specifies the name of the attribute.
value	Values: String Default: n/a (required) Specifies the desired value of the attribute.

<context-param-mapping>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

This element carries information in the content of the element itself, as follows:

In `orion-web.xml`, a `<context-param-mapping>` element overrides the value specified through a corresponding `<context-param>` element in `web.xml`, for a servlet context parameter. Use the `name` attribute to match a `<param-name>` setting in `web.xml`, and use the element value to specify the new value:

```
<context-param-mapping name="..." >deploymentValue</context-param-mapping>
```

Table B–4 <context-param-mapping> Attributes

Name	Description
name	Values: String Default: n/a (required) The name of the parameter for which you are specifying a new value.

<ejb-ref-mapping>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

Use this element to declare a JNDI location for an EJB. This is in conjunction with a corresponding `<ejb-ref>` or `<ejb-local-ref>` element to declare the EJB in the `web.xml` file. The `<ejb-ref-mapping>` element `name` attribute corresponds to an `<ejb-ref-name>` element in `web.xml`, and the `location` attribute specifies a JNDI location.

Table B–5 <ejb-ref-mapping> Attributes

Name	Description
name	Values: String Default: n/a (required) Specifies the EJB reference name, from an `<ejb-ref-name>` element in `web.xml`.
location	Values: String Default: n/a (required) Specifies a JNDI location from which to look up the EJB home.

<env-entry-mapping>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

In `orion-web.xml`, an `<env-entry-mapping>` element overrides the value specified through a corresponding `<env-entry>` element in `web.xml`, for an environment entry. Use the `name` attribute to match an `<env-entry-name>` setting in `web.xml`, and use the element value to specify the new value:

```
<env-entry-mapping name="..." >deploymentValue</env-entry-mapping>
```

Table B–6 <env-entry-mapping> Attributes

Name	Description
name	Values: String Default: n/a (required) The name of the environment entry for which you are specifying a new value.

<expiration-setting>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

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.

Table B–7 <expiration-setting> Attributes

Name	Description
expires	Values: String (integer, seconds) Default: 0 Specifies the number of seconds before expiration, or `"never"` for no expiration. The default setting calls for immediate expiration.
url-pattern	Values: String Default: n/a (required) Specifies the URL pattern that the expiration applies to. This could be as in the following example: `url-pattern="*.gif"`

<group>

Parent element: <security-role-mapping>

Child elements: None

Required? Optional; zero or more

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

Table B–8 <group> Attributes

Name	Description
name	Values: String Default: n/a (required) Specifies the name of the group.

<host-access>

Parent element: <access-mask>

Child elements: None

Required? Optional; zero or more

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

Table B–9 <host-access> Attributes

Name	Description
domain	Values: String Default: n/a (required) Specifies the host or domain.
mode	Values: allow | deny Default: n/a (required) Specifies whether to allow or deny access from the specified host or domain.

<ip-access>

Parent element: <access-mask>

Child elements: None

Required? Optional; zero or more

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

Table B–10 <ip-access> Attributes

Name	Description
ip	Values: String Default: n/a (required) Specifies the IP address, as a 32-bit value (for example, `"123.124.125.126"`).
netmask	Values: String Default: No default Specifies the relevant subnet mask, if any (example: `"255.255.255.0"`).

Table B–10 (Cont.) <ip-access> Attributes

Name	Description
mode	Values: allow | deny Default: n/a (required) Specifies whether to allow or deny access from the specified IP address and subnet mask.

<jazn-web-app>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or one

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.

When Oracle Identity Management is being used as the security provider for a Web application, with SSO for authentication, you can synchronize a servlet session with the OracleAS JAAS Provider user context through `<jazn-web-app>`. To synchronize the session with the user context, set the `sso.session.synchronize` property to "`true`", the default, in a `<property>` subelement under `<jazn-web-app>`:

```
<jazn-web-app ...>
<property name="sso.session.synchronize" value="true"/>
</jazn-web-app>
```

Or you can set the property to "false".

For additional information about JAAS and the features described for this element, see the *Oracle Containers for J2EE Security Guide*. You can also refer to related Sun Microsystems documentation at the following location:

`http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html`

Table B–11 <jazn-web-app> Attributes

Name	Description
auth-method	Values: BASIC | SSO | COREIDSSO Default: BASIC This is the method of HTTP client authentication. "`BASIC`" is for basic J2EE authentication; "`SSO`" is for Oracle Single Sign-On. "`COREIDSSO`" is for using Oracle COREid Access and Identity as the security provider, with single sign-on Another value for `auth-method`, "`DIGEST`", is deprecated in the OC4J 10.1.3 implementation, but still supported for backward compatibility. Instead, you can use the standard `DIGEST` setting for `auth-method` in the `web.xml` file. **Note**: Use "`BASIC`" if your application uses a custom `LoginModule` instance.

Table B–11 (Cont.) <jazn-web-app> Attributes

Name	Description
runas-mode	Values: Boolean Default: false This mode is deprecated in the OC4J 10.1.3 implementation, but still supported for backward compatibility. A new, consolidated JAAS mode replaces the `runas-mode` and `doasprivileged-mode` for servlets. 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	Values: Boolean Default: true This mode is deprecated in the OC4J 10.1.3 implementation, but still supported for backward compatibility. A new, consolidated JAAS mode replaces the `runas-mode` and `doasprivileged-mode` for servlets. `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).

Warning Issued for servlet.init() Not Working with run-as

For a Web application, when `run-as` *`user`* is specified in the `web.xml` file, all method invocations except the `Servlet.init()` method will be invoked as the specified user. With the JMS Router being the default application of OC4J, calls need to be authorized to the router's EJBs. This is done by defining the application role `"jmsRouter"`, which is mapped to the JAAS `"oc4j-administrators"` role, and specifying `<method-permission>` for all methods of the router's EJBs.

The `init()` method of the servlet within the router's Web model creates a router EJB object. Regardless of whether `run-as` is specified in `web.xml` for the servlet, a security exception is thrown:

```
@ oracle.oc4j.rmi.OracleRemoteException: anonymous is not allowed to call this EJB
method, check your security settings (method-permission in ejb-jar.xml and
security-role-mapping in orion-application.xml).
```

Workaround for run-as Warning

You can remove the security warning by commenting out `'*'` in the `<method-name>` element of `<method-permission>` in `ejb-jar.xml` and explicitly enumerating all methods in `AdminMgrBean` that the `jmsRouter` role can access, as follows.

```
<!--
    <method-permission>
      <role-name>jmsRouter</role-name>
      <method>
         <ejb-name>AdminMgrBean</ejb-name>
         <method-name>*</method-name>
      </method>
    </method-permission>
-->
    <method-permission>
      <role-name>jmsRouter</role-name>
      <method>
         <ejb-name>AdminMgrBean</ejb-name>
         <method-name>getConfig</method-name>
      </method>
    </method-permission>
    ...
```

`runAsRoleName` is correctly parsed in `ServletDescriptor.java`, stored in `info` and `thread` in `HttpApplication.loadServlet()`.

<lookup-context>

Parent element: <resource-ref-mapping>

Child elements: <context-attribute>

Required? Optional; zero or one

This element, through its `location` attribute, specifies an optional JNDI context that will be used instead of the default context in looking up the resource mapped in the parent `<resource-ref-mapping>` element. This is useful when you are 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.)

Table B–12 <lookup-context> Attributes

Name	Description
location	Values: String Default: n/a (required) Specifies the name of a nondefault (such as third-party) JNDI context.

<mime-mappings>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

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

Table B–13 <mime-mappings> Attributes

Name	Description
path	Values: String Default: n/a (required) Specifies the path or URL for the file, either absolute or relative to the location of the `orion-web.xml` file.

<ojsp-init>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or one

This element sets the JSP configuration parameters (attributes) that Table B–14 lists. If you specify this element in the `orion-web.xml` file for a Web application, the values of these attributes, including default values, override the values of any corresponding JSP configuration parameters specified in `<init-param>` elements in the `web.xml` deployment descriptor installed with the Web module. Any corresponding command-line options of the `ojspc` pretranslation utility override the `<ojsp-init>` attributes as well as any corresponding settings in `web.xml`.

> **Note:** You cannot use an <ojsp-init> element in `global-web-application.xml`.

Table B–14 <ojsp-init> Attributes

Name	Description
debug-mode	Values: Boolean Default: false Specifies whether or not to print the stack trace. Set to `true` to print the stack trace when certain runtime exceptions occur. When this parameter is `false` and a file is not found, the full path of the missing file is *not* displayed. This is an important security consideration if you want to suppress the display of the physical file path when nonexistent JSP files are requested.
iso-8859-1-convert	Values: Boolean Default: true Specifies whether or not to convert strings to the iso-8859-1 character set. If the value of `iso-8859-1-convert` is "`true`" a full conversion of the characters will take place, using an encoder. If the value of `iso-8859-1-convert` is "`false`", byte truncation will happen, providing a performance gain. Oracle recommends this byte truncation for most of the common use cases where the string does not contain characters.

Table B–14 (Cont.) <ojsp-init> Attributes

Name	Description
jsr45-debug	Values: none, class, file Default: none Specifies the method for JSR-45 support, which *JSR-045: Debugging Support for Other Languages* describes. Setting `jsr45-debug` to "`file`" will generate a separate file containing an SMAP of the JSP lines mapped to the Java lines. A "`class`" setting will put this information into the compiled class file. Setting `jsr45-debug` to "`none`" will turn off JSR-45 debugging support.
main-mode	Values: recompile, reload, justrun Default: recompile Specifies whether JSP-generated classes are automatically reloaded or JSP pages are automatically retranslated when changes are made. If enabled, this feature allows new or modified JSP pages to be loaded into the OC4J runtime, without requiring the Web application to be redeployed or restarted. See *Oracle Containers for J2EE Support for JavaServer Pages Developer's Guide*for additional information. If the value of `main-mode` is `recompile`, the container will check the timestamp of the JSP page, retranslate it, and reload it if has been modified since it was last loaded. The functionality described for `reload` will also be executed. If the value of `main-mode` is `reload`, the container will check the timestamp of classes generated by the JSP translator, such as page implementation classes, and reload any that have changed or been redeployed since they were last loaded. This might be useful, for example, when you deploy or redeploy compiled classes, but not JSP pages, from a development environment to a production environment. If the value of `main-mode` is `justrun`, the container will not perform any timestamp checking, so there is no retranslation of JSP pages or reloading of JSP-generated Java classes. This is the most efficient mode for a production environment, where JSP pages are not expected to change frequently.
precompile-check	Values: Boolean Default: false Specifies whether to check the HTTP request for a standard `jsp_precompile` setting. The default is `false`. If `precompile_check` is `true` and the request enables `jsp_precompile`, then the JSP page will be pretranslated only, without execution. Setting `precompile_check` to `false` improves performance and ignores any `jsp_precompile` setting in the request.
reduce-tag-code	Values: Boolean Default: false If set to `true`, specifies further reduction in the size of generated code for custom tag usage.

Table B–14 (Cont.) <ojsp-init> Attributes

Name	Description

req-time-introspection	Values: Boolean
	Default: false
	If set to `true`, enables request-time JavaBean introspection whenever compile-time introspection is not possible. When compile-time introspection is possible and succeeds, however, there is no request-time introspection regardless of the setting of this flag.
	As an example of a scenario for use of request-time introspection, assume a tag handler returns a generic `java.lang.Object` instance in `VariableInfo` of the tag-extra-info class during translation and compilation, but actually generates more specific objects during runtime. In this case, if `req_time_introspection` is enabled, the Web container will delay introspection until request time.
	An additional effect of this flag is to allow a bean to be declared twice, such as in different branches of an `if..then..else` loop. Consider the example that follows. With the default `false` value of `req_time_introspection`, this code would cause a parse exception. With a `true` value, the code will work without error:
	`<% if (cond) { %>`
	`<jsp:useBean id="foo" class="pkgA.Foo1" />`
	`<% } else { %>`
	`<jsp:useBean id="foo" class="pkgA.Foo2" />`
	`<% } %>`

static-text-in-chars	Values: Boolean
	Default: false
	If set to `true`, instructs the JSP translator to generate static text in JSP pages as characters instead of bytes. The default is `false`.
	Enable this flag if your application requires the ability to change the character encoding dynamically during runtime, such as in the following example:
	`<% response.setContentType("text/html; charset=UTF-8");`
	`%>`
	The `false` default setting improves performance in outputting static text blocks.

Table B–14 (Cont.) <ojsp-init> Attributes

Name	Description
tags-reuse	Values: compiletime, compiletime-with-release, none Default: compiletime Specifies the mode for tag handler reuse, also known as *tag pooling*. ▪ Set to `compiletime` to enable the compile-time model of tag handler reuse in its basic mode. This is the default value. ▪ Set to `compiletime_with_release` to enable the compile-time model of tag handler reuse in its "with release" mode, where the tag handler `release()` method is called between usages of a given tag handler within a given page. ▪ Set to `none` or `false` to disable tag handler reuse. You can override this value in any particular JSP page by setting the JSP page context attribute `oracle.jsp.tags.reuse` to a value of `true`. ▪ Set to `runtime` to enable the runtime model of tag handler reuse. You can override this in any particular JSP page by setting the JSP page context attribute `oracle.jsp.tags.reuse` to a value of `false`. Note that the `runtime` option and its equivalent, `true`, are deprecated in this release of OC4J.

Table B–15 lists the `<ojsp-init>` attributes with corresponding JSP servlet `<init-param>` elements and `ojspc` command-line options. For more information about JSP servlet `<init-param>` elements, see the *Oracle Containers for J2EE Support for JavaServer Pages Developer's Guide*. For more information about JSP servlet `ojspc` command-line options, see the *Oracle Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Table B–15 JSP Servlet Configuration Parameters

<ojsp-init> Attribute	Equivalent JSP Servlet <init-param>	ojspc Command-Line Option
debug-mode	debug_mode	n/a
iso-8859-1-convert	iso-8859-1-convert	n/a
jsr45-debug	debug	-debug
main-mode	main_mode	n/a
precompile-check	precompile_check	n/a
reduce-tag-code	reduce_tag_code	-reduceTagCode
req-time-introspection	req_time_instrospection	-reqTimeIntrospection
static-text-in-chars	static-text-in-chars	-staticTextInChars
tags-reuse	tags_reuse_default	-tagReuse

<orion-web-app>

Parent element: n/a (root)

Child elements: <access-mask>, <classpath>, <context-param-mapping>, <ejb-ref-mapping>, <env-entry-mapping>, <expiration-setting>, <jazn-web-app>, <mime-mappings>, <ojsp-init>, <request-tracker>, <resource-env-ref-mapping>,

<resource-ref-mapping>, <security-role-mapping>, <service-ref-mapping>, <servlet-chaining>, <session-tracking>, <virtual-directory>, <web-app>, <web-app-class-loader>

Required? Required; one only

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

> **Note:** The `always-redeploy`, `deployment-time`, and `deployment-version` attributes are not supported for direct use, so are not documented, although `deployment-time` and `deployment-version` may receive container-generated values for the time of deployment and the OC4J version. Modifying these parameters has no effect.

Table B–16 <orion-web-app> Attributes

Name	Description
autojoin-session	Values: Boolean Default: true Specifies whether users should be assigned a session as soon as they log in to the application.
default-buffer-size	Values: Non-negative integer (bytes) Default: 2048 Specifies the default size of the output buffer for servlet responses, in bytes.
default-charset	Values: String Default: iso-8859-1 In 10.1.3.1 for JSP pages and for the servlet container, this attribute specifies the ISO character set to use by default. In general, for JSP 2.0 users, Oracle instead recommends standard `<page-encoding>` functionality (under the `web.xml` `<jsp-config>` element, according to the JSP 2.0 specification), to specify character sets according to URL patterns. However, `default-charset` may be useful if you have large numbers of JSP pages, particularly across multiple applications, to avoid the necessity of making numerous changes in your EAR files. Also, you can use `default-charset` to set a base default, then use `<page-encoding>` functionality to override the default for particular URL patterns. See the JSP and servlet specifications for more information about the `<jsp-config>` and `<page-encoding>` elements.
default-mime-type	Values: String Default: No default This specifies a default content type for servlet responses, for situations where the `setContentType()` method is not called from the servlet implementation. If `default-mime-type` is not specified, then there is no default content type.

Table B–16 (Cont.) <orion-web-app> Attributes

Name	Description
development	Values: Boolean Default: false This attribute is a convenience flag to use 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. Note that the OC4J JSP container does not take any special action for the `development` flag. Any related functionality is handled by the OC4J servlet container.
directory-browsing	Values: allow | deny Default: deny Specifies whether to allow directory browsing for a URL that ends in "`/`". 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.
enable-jsp-dispatcher-shortcut	Values: Boolean Default: true A "`true`" setting 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.

Table B–16 (Cont.) <orion-web-app> Attributes

Name	Description
file-modification-check-interval	Values: String (integer, milliseconds) Default: 1000 This attribute determines when to check a static file, such as an HTML file, to see whether its timestamp has changed and it should therefore be reloaded from the file system. When a static file is first accessed, it is loaded from the file system and also cached. For each subsequent access, there is the following logic: ■ If the elapsed time since the last check of the file timestamp is less than the specified file-check interval, then the timestamp is not checked and the file is loaded from cache. ■ If the elapsed time since the last check of the file timestamp is greater than the specified file-check interval, then the timestamp is checked. If the timestamp has changed since the last check, the file is loaded from the file system, otherwise it is loaded from cache. Specify this value in milliseconds. Zero or a negative number specifies that the file timestamp is always checked. For performance reasons, a very large value ("`1000000`", for example) is recommended in a production environment.
jsp-cache-directory	Values: String Default: ./persistence (relative to the deployment directory of the application) 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.
jsp-cache-tlds	Values: on | off | standard Default: standard This flag indicates whether persistent TLD caching is enabled for JSP pages. The "`standard`" or "`on`" setting enables caching, and in either case `.tld` files are inherited from the global TLD locations according to the `jsp-taglib-locations` attribute of `<orion-web-app>`. The "`standard`" setting also results in a search for `.tld` files in the application `/WEB-INF` directory, and these are added to the files inherited from the global level. Note that the `/WEB-INF/lib` and `/WEB-INF/classes` subdirectories are *not* searched. The "`on`" setting also results in a search for `.tld` files among all files within the application, and these are added to the files inherited from the global level. The "`off`" setting disables persistent TLD caching.
jsp-print-null	Values: Boolean Default: true Set this flag to "`false`" to print an empty string instead of the default "null" string for null output from a JSP page.

Table B–16 (Cont.) <orion-web-app> Attributes

Name	Description
jsp-taglib-locations	Values: String Default: See below 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 ■ `./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`.
jsp-timeout	Values: Non-negative integer (seconds) Default: 0 (no timeout) Specifies a period of time 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.
persistence-path	Values: String Default: No default (by default, no persistence) 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. If no value is specified, 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. **Note**: This attribute is ignored if OC4J clustering is enabled.
schema-major-version	Values: String Default: No default The major version number of the `orion-web.xml` XSD. If you create `orion-web.xml` manually, set this attribute to 10 for use with the OC4J 10.1.3 implementation. **Note**: This attribute does not appear directly in the XSD for `orion-web.xml`. It is according to the `attributeGroup` specification in the top-level OC4J XSD.

Table B–16 (Cont.) <orion-web-app> Attributes

Name	Description
schema-minor-version	Values: String Default: No default The minor version number of the `orion-web.xml` XSD. If you create `orion-web.xml` manually, set this attribute to 0 for use with the OC4J 10.1.3 implementation. **Note**: This attribute does not appear directly in the XSD for `orion-web.xml`. It is according to the `attributeGroup` specification in the top-level OC4J XSD.
servlet-webdir	Values: String Default: `/servlet/` (see note below) Use this attribute, in conjunction with a `true` setting for the OC4J system property `http.webdir.enable`, to enable servlet invocation by class name in standalone OC4J. Once the system property is set, any `servlet-webdir` setting that starts with a slash ("`/`") enables this feature and specifies a special URL portion to insert after the context path to instruct OC4J to invoke a servlet by class name. Anything appearing after this path in a URL is assumed to be a class name, including the package. See "Invoking a Servlet by Class Name During OC4J Development" on page 2-11 for additional information. This feature is typically for use in an OC4J standalone environment during development and testing; it presents a significant security risk and should not be used in a production environment. (For production deployment, use the standard `web.xml` mechanisms to define the context path and servlet path.) Here is an example of servlet invocation by class name, assuming a context path of "`/`" and a setting of `servlet-webdir="/servlet/"`: `http://www.example.com:8888/servlet/foo.SessionServlet` Invocation by class name is disabled by a setting of `servlet-webdir=""` (empty quotes) or by the OC4J system property setting `http.webdir.enable=false`. **Note**: Any `servlet-webdir` setting, including the default, is overridden by the default `false` setting of `http.webdir.enable`. See the *Oracle Containers for J2EE Configuration and Administration Guide* for general information about OC4J system properties.
simple-jsp-mapping	Values: Boolean Default: false Set this to "`true`" if "`*.jsp`" is mapped to *only* the `oracle.jsp.runtimev2.JspServlet` front-end JSP servlet. This would be specified in the `<servlet>` elements of any Web descriptors affecting your application (`global-web-application.xml`, `web.xml`, and `orion-web.xml`). A "`true`" setting allows performance improvements for JSP pages.

Table B–16 (Cont.) <orion-web-app> Attributes

Name	Description
source-directory	Values: String Default: /WEB-INF/src (if it exists), or /WEB-INF/classes 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. If you use the default location, OC4J keeps track of the location of the `/WEB-INF` directory of your application after deployment. Note that modified source files will be found anywhere under the `source-directory` directory, according to package name.
temporary-directory	Values: String Default: ./temp 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. 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.

Note:

Processing related to the `enable-jsp-dispatcher-shortcut`, `jsp-cache-directory`, `jsp-cache-tlds`, `jsp-print-null`, `jsp-taglib-locations`, `jsp-timeout`, and `simple-jsp-mapping` attributes are handled by the OC4J JSP container. For more information about these attributes and related features, see the *Oracle Containers for J2EE Support for JavaServer Pages Developer's Guide*.

<request-tracker>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

This element specifies a servlet to use as a *request tracker*. 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). 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. There can be multiple request trackers, each one defined in a separate `<request-tracker>` element.

Table B–17 <request-tracker> Attributes

Name	Description
servlet-name	Values: String Default: n/a (required) Specifies the servlet to invoke. You can specify either the designated 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-env-ref-mapping>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

Use this element to declare a JNDI location for an environment resource. This is in conjunction with a corresponding `<resource-env-ref>` element in the `web.xml` file, which declares the resource. The `<resource-env-ref-mapping>` element `name` attribute corresponds to a `<resource-env-ref-name>` element in `web.xml`, and the `location` attribute specifies a JNDI location.

Table B–18 <resource-env-ref-mapping> Attributes

Name	Description
name	Values: String Default: n/a (required) Specifies the resource name, from `web.xml`.
location	Values: String Default: n/a (required) Specifies the JNDI location from which to look up the resource.

<resource-ref-mapping>

Parent element: <orion-web-app>

Child elements: <lookup-context>

Required? Optional; zero or more

Use this element to declare a JNDI location for an external resource, such as a data source, JMS queue, or mail session. This is in conjunction with a corresponding `<resource-ref>` element in the `web.xml` file, which declares the resource. The `<resource-ref-mapping>` element `name` attribute corresponds to a `<res-ref-name>` element in `web.xml`, and the `location` attribute specifies the JNDI location.

Following is an example using this element and its subelements:

```
<resource-ref-mapping location="jdbc/HyperSonicDS" name="jdbc/myDS">
  <lookup-context location="foreign/resource/location">
    <context-attribute name="java.naming.factory.initial" value="classname" />
    <context-attribute name="name" value="value" />
  </lookup-context>
</resource-ref-mapping>
```

Table B–19 <resource-ref-mapping> Attributes

Name	Description
name	Values: String Default: n/a (required) Specifies the resource name, from `web.xml`. For example: `name="jdbc/TheDSVar"`
location	Values: String Default: n/a (required) Specifies a JNDI location from which to look up the resource. For example: `location="jdbc/TheDS"`

<security-role-mapping>

Parent element: <orion-web-app>

Child elements: <group>, <user>

Required? Optional; zero or more

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 specified through a `<security-role>` element in the `web.xml` file. Use either the `impliesAll` attribute or an appropriate combination of subelements—`<group>`, `<user>`, or both.

See the *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide* for additional information about the `<security-role-mapping>` element in OC4J configuration files.

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 `system-jazn-data.xml` (or other valid user manager), 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.

Table B–20 <security-role-mapping> Attributes

Name	Description
impliesAll	Values: Boolean Default: false Specifies whether this mapping applies to all users.
name	Values: String Default: n/a (required) Specifies the name of the security role, matching a name specified in a `<role-name>` subelement of a `<security-role>` element in `web.xml`.

<service-ref-mapping>

Parent element: <orion-web-app>

Child elements: According to its own schema definition

Required? Optional; zero or more

This element is for use in conjunction with a `<service-ref>` element that appears in the `web.xml` file to declare a Web service. You can use it to specify OC4J-specific quality of service features for the corresponding Web service, such as security, logging, and auditing. See the *Oracle Application Server Web Services Developer's Guide* for complete information.

Note that as a `<service-ref>` element can appear in a `web.xml`, `ejb-jar.xml`, or `application-client.xml` file, a corresponding `<service-ref-mapping>` element can appear in an `orion-web.xml`, `orion-ejb-jar.xml`, or `orion-application-client.xml` file. Supported features of the `<service-ref-mapping>` element are according to its own XSD, which is imported into the `orion-web`, `orion-ejb-jar`, and `orion-application-client` XSDs.

<servlet-chaining>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

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.

Important: Servlet chaining is an older and proprietary mechanism with functionality similar to that of standard servlet filtering, which was introduced in version 2.3 of the servlet specification. Use servlet filtering instead. The OC4J `<servlet-chaining>` element is deprecated in the current release and will be desupported in the next release.

Table B–21 <servlet-chaining> Attributes

Name	Description
mime-type	Values: String Default: n/a (required) Specifies the MIME type to trigger the chaining (for example, `"text/html"`).
servlet-name	Values: String Default: n/a (required) 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`.

<session-tracker>

Parent element: <session-tracking>

Child elements: None

Required? Optional; zero or more

This subelement of `<session-tracking>` specifies a servlet to use as a *session tracker*. 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). 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. There can be multiple session trackers, each one defined in a separate `<session-tracker>` element.

Table B–22 <session-tracker> Attributes

Name	Description
servlet-name	Values: String Default: n/a (required) Specifies the servlet to invoke. You can specify either the designated servlet name or the class name, according to the corresponding `<servlet-name>` or `<servlet-class>` element under the relevant `<servlet>` element in the `web.xml` file.
set-secure	Values: Boolean Default: false Specifies whether all session cookies generated by OC4J for an application will be returned by the client only when the HTTPS protocol is being used. If set-secure="`true`", all session cookies will include the `secure` attribute, which instructs the browser to return the cookies only over the secure HTTPS protocol. If set-secure="`false`", the browser will return cookies over any protocol.

<session-tracking>

Parent element: <orion-web-app>

Child elements: <session-tracker>

Required? Optional; zero or one

This element specifies the session-tracking settings for this application. Session tracking is accomplished through cookies, assuming a cookie-enabled browser. The servlet to use as the session tracker is specified through the `<session-tracker>` subelement.

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.

Table B–23 <session-tracking> Attributes

Name	Description
cookies	Values: enabled | disabled Default: enabled Specifies whether to send session cookies. The name of a session cookie is `JSESSIONID`. (See "How OC4J Can Use Cookies for Session Tracking" on page 3-3 for information about the `JSESSIONID` cookie.)

Table B–23 (Cont.) <session-tracking> Attributes

Name	Description
cookie-domain	Values: String Default: No default Specifies the desired domain for `JSESSIONID` session cookies. This overrides any `domain` setting in applicable `Set-Cookie` HTTP response headers. 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 session 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 an OC4J standalone environment, you can use it for a shared application, where `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-path	Values: String Default: No default You can use this to optionally specify the URL path value that applies to `JSESSIONID` session cookies. This specifies the subset of URLs for which session cookies are valid within any applicable domain (which depends on the cookie domain setting). If specified, it overrides any `path` setting in applicable `Set-Cookie` HTTP response headers. If not specified, and if there is no `path` setting in the `Set-Cookie` headers, the default cookie path is the Web application context path.
cookie-max-age	Values: Non-negative integer (seconds) Default: No default This number is sent with `JSESSIONID` session cookies 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.

<user>

Parent element: <security-role-mapping>

Child elements: None

Required? Optional; zero or more

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

Table B–24 <user> Attributes

Name	Description
name	Values: String Default: n/a (required) Specifies the name of the user.

<virtual-directory>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or more

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.

Table B–25 <virtual-directory> Attributes

Name	Description
real-path	Values: String Default: n/a (required) This is a real path, such as `/usr/local/realpath` in UNIX or `C:\testdir` in Windows.
virtual-path	Values: String Default: n/a (required) This is a virtual path to map to the specified real path.

<web-app>

Parent element: <orion-web-app>

Child elements: According to its own schema definition in the servlet specification

Required? Optional; zero or one

This element is used as in the standard `web.xml` file. See "Standard web.xml Configuration File" on page B-1 for an overview, or the servlet specification for complete information. You can establish defaults for `<web-app>` settings in `global-web-application.xml`. 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`.

Table B–26 <web-app> Attributes

Name	Description
id	Values: ID Default:
metadata-complete	Values: Boolean Default: `true` for servlet 2.4, `false` for servlet 2.5 Specifies whether this deployment descriptor and other related deployment descriptors for this module (such as Web service descriptors) are complete or whether the class files available to this module and packaged with this application should be examined for annotations that specify deployment information. If `metadata-complete` is set to `true`, the OC4J servlet container will ignore any annotations that might be present in the class files of the application. If `metadata-complete` is not specified or is set to `false` and if version is set to 2.5 or `web-xml` points to the servlet 2.5 schema namespace, the servlet container will examine the class files of the application for annotations.
version	Values: web-app-versionType Default: n/a (required)

<web-app-class-loader>

Parent element: <orion-web-app>

Child elements: None

Required? Optional; zero or one

Use this element for class-loading instructions.

Table B–27 <web-app-class-loader> Attributes

Name	Description
search-local-classes-first	Values: Boolean Default: false Set this to "`true`" to search and load WAR file classes before system classes. By default, system classes are searched and loaded first.
include-war-manifest-class-path	Values: Boolean Default: true 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). Otherwise, the classplath from the WAR file manifest is included.

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.
- If you want to use an XML parser or JDBC driver packaged with your application in place of the Oracle XML parser or JDBC driver, set the `search-local-classes-first` attribute to "`true`". You also need to specify the default inherited Oracle library in the `<remove-inherited>` tag in `orion-application.xml`. For complete instructions, see *Oracle Containers for J2EE Developer's Guide*.

C

Third Party Licenses

This appendix includes the Third Party License for all the third party products included with Oracle Application Server.

ANTLR

This program contains third-party code from ANTLR. Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the ANTLR software, and the terms contained in the following notices do not change those rights.

The ANTLR License

```
Software License

We reserve no legal rights to the ANTLR--it is fully in the public domain. An
individual or company may do whatever they wish with source code distributed with
ANTLR or the code generated by ANTLR, including the incorporation of ANTLR, or its
output, into commerical software.
We encourage users to develop software with ANTLR. However, we do ask that credit
is given to us for developing ANTLR. By "credit", we mean that if you use ANTLR or
incorporate any source code into one of your programs (commercial product,
research project, or otherwise) that you acknowledge this fact somewhere in the
documentation, research report, etc... If you like ANTLR and have developed a nice
tool with the output, please mention that you developed it using ANTLR. In
addition, we ask that the headers remain intact in our source code. As long as
these guidelines are kept, we expect to continue enhancing this system and expect
to make other tools available as they are completed.
```

Apache

This program contains third-party code from the Apache Software Foundation ("Apache"). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that 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.

The Apache license agreements apply to the following included Apache components:

- Apache HTTP Server
- Apache JServ
- mod_jserv

- Regular Expression package version 1.3
- Apache Expression Language packaged in commons-el.jar
- mod_mm 1.1.3
- Apache XML Signature and Apache XML Encryption v. 1.4 for Java and 1.0 for C++
- log4j 1.1.1
- BCEL v. 5
- XML-RPC v. 1.1
- Batik v. 1.5.1
- ANT 1.6.2 and 1.6.5
- Crimson v. 1.1.3
- ant.jar
- wsif.jar
- bcel.jar
- soap.jar
- Jakarta CLI 1.0
- jakarta-regexp-1.3.jar
- JSP Standard Tag Library 1.0.6 and 1.1
- Struts 1.1
- Velocity 1.3
- svnClientAdapter
- commons-logging.jar
- wsif.jar
- commons-el.jar
- standard.jar
- jstl.jar

The Apache Software License

License for Apache Web Server 1.3.29

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000-2002 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
```

```
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
 *    permission of the Apache Software Foundation.
 *
 * THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 * ====================================================================
 *
 * This software consists of voluntary contributions made by many
 * individuals on behalf of the Apache Software Foundation.  For more
 * information on the Apache Software Foundation, please see
 * <http://www.apache.org/>.
 *
 * Portions of this software are based upon public domain software
 * originally written at the National Center for Supercomputing
Applications,
 * University of Illinois, Urbana-Champaign.
```

License for Apache Web Server 2.0

```
Copyright (c) 1999-2004, The Apache Software Foundation
Licensed under the Apache License, Version 2.0 (the "License"); you may not use
this file except in compliance with the License.  You may obtain a copy of the
License at   ://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software distributed
under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied.  See the License for the
specific language governing permissions and limitations under the License.
Copyright (c) 1999-2004, The Apache Software Foundation
                                 Apache License
                            Version 2.0, January 2004
                         http://www.apache.org/licenses/
```

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions

for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Apache SOAP

This program contains third-party code from the Apache Software Foundation ("Apache"). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache

software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

Apache SOAP License

Apache SOAP license 2.3.1

Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution

notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

Index

Symbols

<access-mask> element (orion-web.xml), B-4
<classpath> element (orion-web.xml), B-5
<context-attribute> element (orion-web.xml), B-6
<context-param-mapping> element (orion-web.xml), B-7
<ejb-ref-mapping> element (orion-web.xml), B-7
<env-entry-mapping> element (orion-web.xml), B-7
<expiration-setting> element (orion-web.xml), B-8
<group> attribute (orion-web.xml), B-8
<host-access> element (orion-web.xml), B-9
<ip-access> element (orion-web.xml), B-9
<jazn-web-app> element (orion-web.xml), B-10
<lookup-context> element (orion-web.xml), B-12
<mime-mappings> element (orion-web.xml), B-12
<ojsp-init> element (orion-web.xml), B-13
<orion-web-app> element (orion-web.xml), B-16
<request-tracker> element (orion-web.xml), B-22
<resource-env-ref-mapping> element (orion-web.xml), B-23
<resource-ref-mapping> element (orion-web.xml), B-23
<security-role-mapping> element (orion-web.xml), B-24
<service-ref-mapping> element (orion-web.xml), B-25
<servlet-chaining> element (orion-web.xml), B-25
<session-tracking> element (orion-web.xml), B-27
<user> element (orion-web.xml), B-28
<virtual-directory> element (orion-web.xml), B-29
<web-app> element (orion-web.xml), B-29
<web-app-class-loader> element (orion-web.xml), B-30

A

access-mask element (orion-web.xml), B-4
administration
  Application Server Control Console pages, A-3
  Application Server Control Console Web Module Administration, A-2
  JSR-77 support, 2-1
  MBeans administration in OC4J, A-10
  overview for OC4J, 2-1
  Web module configuration files, B-1
  Web module MBeans, summary, A-11
annotations
  EJB, 7-3
  overview, 7-1
  PersistenceContext(s), 7-6
  PersistenceUnit(s), 7-5
  PostConstruct, 7-5
  PreDestroy, 7-5
  Resource, 7-4
  Resources, 7-4
  RunAs, 7-7
  WebServiceRef, 7-6, 7-7
Application Server Control Console
  Configuration Properties Page, A-3
  configuring Web modules, A-3
  EJB Reference Mappings page, A-8
  Environment Entry Mappings page, A-8
  Filter Mappings page, A-6
  introduced, 2-2
  overview, A-1
  Resource Reference Lookup Context page, A-9
  Resource Reference Mappings page, A-7
  Servlet Mappings page, A-5
  viewing web.xml and orion-web.xml, A-5
  Web Module Administration, A-2
  Web Module Home page, summary, A-2
  Web Module Performance, A-2
  Web module top-level pages, A-2
Application Server Control Console--Web Module Home page, getting to, A-1
auto-encoding, session (not supported), 3-4, B-27

C

classpath element (orion-web.xml), B-5
classpath setup for servlet.jar, 6-7
co-location of servlet and EJB, 8-9
configuration
  Application Server Control Console pages, A-3
  cookies, 3-11
  session tracking, 3-3
  Web module configuration files, B-1
  Web module MBeans, summary, A-11
container, servlet, 1-8
context path (URL component), 2-5
context root, 2-5

context-attribute element (orion-web.xml), B-6
context-param-mapping element (orion-web.xml), B-7
cookies
configuration, 3-11
Cookie methods, summary, 3-11
enabling or disabling, 3-3
OC4J use for session tracking, 3-3
retrieving, displaying, adding, 3-12
sample servlet, 3-13

D

data sources
configuration, 8-2
JNDI lookup, database connection, 8-3
default application in OC4J, A-11
default Web application in OC4J, A-11
demo location, OTN, 1-1
deployment
deployment plan, 2-2
deployment plan editor, 2-2
JSR-88 support, 2-2
destroy() servlet method, 1-4
dispatching to another servlet (include or forward), 6-12
DMS (Dynamic Monitoring Service), 9-5
doDelete() servlet method, 1-4
doGet() servlet method, 1-4
doPost() servlet method, 1-4
doPut() servlet method, 1-4
Dynamic Monitoring Service (DMS), 9-5

E

EJB annotation, 7-3
ejb-ref-mapping element (orion-web.xml), B-7
EJBs from servlets
Application Server Control Console EJB Reference Mappings page, A-8
co-location, 8-9
local interfaces vs. remote interfaces, 8-10
lookup categories, 8-9
OC4J and OracleAS support, 8-9
ORMI, IIOP usage, 8-10
remote flag for remote lookup, 8-11
scenarios, 8-9
why use, 8-9
Enterprise JavaBeans--see EJBs
env-entry-mapping element (orion-web.xml), B-7
event listeners--see listeners
expiration-setting element (orion-web.xml), B-8

F

Feiner, Amy, Hi, 2-16
Filter interface, 4-3
FilterChain interface, 4-4
FilterConfig interface, 4-4
filters
actions, typical, 4-3
Application Server Control Console Filter Mappings page, A-6
configuration steps, 4-6
example, response filter, 4-11
example, simple filter, 4-7
filter chain construction, 4-7
implementation steps, 4-5
include/forward filtering, 4-9
interfaces, standard, 4-3
introduction, 6-16
invocation by container, 4-1
request/response wrapping, 4-10
forms in servlets, 6-7
forwards
basics, forwarding to another servlet, 6-13
example, 6-14
filtering forwarded servlet, 4-9
implementation steps, 6-14
why use, 6-13

G

GET, HTTP request, 6-1
getServletConfig() servlet method, 1-4
getServletInfo() servlet method, 1-4
global-web-application.xml file
element descriptions, B-4
element hierarchy, B-3
overview, B-2
relationship to web.xml, orion-web.xml, B-3
group attribute (orion-web.xml), B-8

H

host (URL component), 2-4
host-access element (orion-web.xml), B-9
HTML forms in servlets, 6-7
HttpServlet class
overview of methods, 1-4
when to override methods, 6-2
HttpServletRequest interface, 1-5
HttpServletResponse interface, 1-5
HttpSession methods, 3-5
HttpSessionActivationListener interface, 5-4
HttpSessionAttributeListener interface, 5-4
HttpSessionBindingEvent class, 5-4
HttpSessionBindingListener interface, 5-5
HttpSessionEvent class, 5-3
HttpSessionListener interface, 5-3

I

includes
basics, including another servlet, 6-13
example, 6-14
filtering included servlet, 4-9
implementation steps, 6-14
why use, 6-13
init() servlet method, 1-4
input from user, HTML forms, 6-7
invoking a servlet

by class name (OC4J-specific), 2-11
in an Oracle Application Server environment, 2-12
in standalone OC4J, 2-10
summary of URL components, 2-3
ip-access element (orion-web.xml), B-9

J

jazn-web-app element (orion-web.xml), B-10
JDBC driver, using packaged, B-31
JDBC from servlets
code implementation, 8-3
data source configuration, 8-2
example servlet, 8-4
why use, 8-1
JNDI
initial context factory, 8-10
lookup of datasource, 8-3
name and location for data source, 8-2
JSP parameters
enable-jsp-dispatcher-shortcut, B-18
jsp-cache-directory, B-19
jsp-cache-tlds, B-19
jsp-print-null, B-19
jsp-taglib-locations, B-20
jsp-timeout, B-20
simple-jsp-mapping, B-21
JSR-77 support, 2-1
JSR-88 support, 2-2

L

lifecycle, servlet, 1-3
listeners
code to implement, 5-6
configuration, 5-7
example, 5-8
introduction, when to use, 6-17
overview, categories, 5-1
request attribute interface, 5-6
request lifecycle interface, 5-5
servlet context attribute interface, 5-3
servlet context lifecycle interface, 5-2
session attribute interface, 5-4
session binding interface, 5-5
session lifecycle interface, 5-3
session migration interface, 5-4
load-on-startup, OC4J, 2-16
lookup-context element (orion-web.xml), B-12

M

managed OC4J, 2-2
MBeans
administration in OC4J, A-10
definition, 2-1
MBean browser, 2-2
Web module MBeans, summary, A-11
metrics, servlets, A-2
mime-mappings element (orion-web.xml), B-12

O

ojsp-init element (orion-web.xml), B-13
Oracle Application Server environment, 2-2
OracleAS JAAS Provider user context, B-10
orion-web-app element (orion-web.xml), B-16
orion-web.xml file
element descriptions, B-4
element hierarchy, B-3
overview, B-3
relationship to web.xml, global-web-application.xml, B-3
viewing through Application Server Control Console, A-5

P

path setup, 6-7
performance, servlets, 9-5, A-2
PersistenceContext(s) annotation, 7-6
PersistenceUnit(s) annotation, 7-5
persistent session data, 3-2
port (URL component), 2-4
POST, HTTP request, 6-1, 6-10
PostConstruct annotation, 7-5
PreDestroy annotation, 7-5
preloading, servlets in OC4J, 2-16
protocol (URL component), 2-4

R

redirect (to an alternative URL), 1-7
remote flag, for remote EJB lookup, 8-11
request dispatcher, include/forward to another servlet, 6-13
requests
example, form and request parameters, 6-8
example, retrieving request information, 6-11
filtering/wrapping request, 4-10
information retrieval, request, 6-11
listener interface, attribute changes, 5-6
listener interface, lifecycle changes, 5-5
request objects, 1-5
request parameters for user input from forms, 6-7
request-tracker element (orion-web.xml), B-22
Resource annotation, 7-4
resource-env-ref-mapping element (orion-web.xml), B-23
resource-ref-mapping element (orion-web.xml), B-23
Resources annotation, 7-4
responses
example, response filter, 4-11
filtering/wrapping response, 4-10
response objects, 1-5
RunAs annotation, 7-7

S

sample servlets
cookie servlet, 3-13
demo location, OTN, 1-1

filter response, 4-11
filter, simple, 4-7
HTML form and request parameters, 6-8
JDBC query, 8-4
retrieving request info, 6-11
security, using POST, 6-10
servlet include, 6-14
session lifecycle event listener, 5-8
session servlet, 3-7
simple example, 6-5
security
POST method for URL security, 6-10
session tracking through secured connections, 3-4
security-role-mapping element (orion-web.xml), B-24
service() servlet method, 1-3, 1-4
service-ref-mapping element (orion-web.xml), B-25
servlet configuration object, 1-10
servlet container, 1-8
servlet contexts
basics, 1-10
listener interface, attribute changes, 5-3
listener interface, lifecycle changes, 5-2
methods, 1-11
obtaining, 1-11
servlet filters--see filters
Servlet interface, 1-4
servlet path (URL component), 2-5
servlet-chaining element (orion-web.xml), B-25
ServletContextAttributeEvent class, 5-3
ServletContextAttributeListener interface, 5-3
ServletContextEvent class, 5-2
ServletContextListener interface, 5-2
ServletRequestAttributeEvent class, 5-6
ServletRequestAttributeListener interface, 5-6
ServletRequestEvent class, 5-5
ServletRequestListener interface, 5-5
sessions
attributes, adding and retrieving, 3-7
canceling explicitly, 3-16
configuring session tracking, 3-3
cookies versus session attributes, 3-2
cookies, enabling or disabling, 3-3
cookies, general use in servlets, 3-10
cookies, use by OC4J for session tracking, 3-3
HttpSession methods, summary, 3-5
introduction, when to use, 1-12
listener interface, attribute changes, 5-4
listener interface, lifecycle changes, 5-3
listener interface, migration changes, 5-4
listener interface, object binding changes, 5-5
persistent data, 3-2
sample servlet, 3-7
session IDs, overview, 3-2
session objects, overview, 3-1
session tracking in OC4J, 3-2
session tracking through secured connections, 3-4
timeout, specifying, 3-15
URL rewriting, OC4J use for session tracking, 3-4
session-tracking element (orion-web.xml), B-27
shared applications between Web site (standalone), 3-5
single-thread model, servlets, 9-3
standalone environment, 2-2

T

thread models, introduction, 1-12
timeout of session, specifying, 3-15
tips
Date constructor for milliseconds since 1/1/1970, 3-6
no need to close response writer/stream, 6-4
setting up path, classpath, 6-7
using POST for URL security, 6-10

U

unmanaged OC4J, 2-2
URL components, summary, 2-3
URL rewriting, 3-4
user element (orion-web.xml), B-28

V

virtual-directory element (orion-web.xml), B-29

W

Web module vs. Web application, 1-2
web-app element (orion-web.xml), B-29
web-app-class-loader element (orion-web.xml), B-30
WebServiceRef annotation, 7-6, 7-7
web.xml file
overview, B-1
relationship to orion-web.xml, global-web-application.xml, B-3
viewing through Application Server Control Console, A-5

X

XML configuration files for Web modules, B-1
XML parser. using packaged, B-31