Oracle9*i*AS Containers for J2EE

Support for JavaServer Pages Reference

Release 2 (9.0.2)

January 2002

Part No. A95882-01

ORACLE

Oracle9*i*AS Containers for J2EE Support for JavaServer Pages Reference, Release 2 (9.0.2)

Part No. A95882-01

Primary Author: Brian Wright

Contributing Author: Michael Freedman

Contributors: Julie Basu, Alex Yiu, Sunil Kunisetty, Gael Stevens, Ping Guo, YaQing Wang, Song Lin, Hal Hildebrand, Jasen Minton, Ashok Banerjee, Matthieu Devin, Jose Alberto Fernandez, Olga Peschansky, Jerry Schwarz, Clement Lai, Shinji Yoshida, Kenneth Tang, Robert Pang, Kannan Muthukkaruppan, Ralph Gordon, Shiva Prasad, Sharon Malek, Jeremy Lizt, Kuassi Mensah, Susan Kraft, Sheryl Maring, Ellen Barnes, Angie Long, Sanjay Singh, Olaf van der Geest

Contents

2 Overview of the Oracle JSP Implementation

3 Getting Started

4 Basic Programming Issues

5 Key Considerations

6 JSP Translation and Deployment

A Servlet and JSP Technical Background

B The Apache JServ Environment

C Third Party Licenses

Send Us Your Comments

Oracle9*i*AS Containers for J2EE Support for JavaServer Pages Reference, Release 2 (9.0.2)
Part No. A95882-01

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

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

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

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

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

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

Preface

This document introduces and explains the Oracle implementation of JavaServer Pages (JSP) technology, specified by Sun Microsystems. It summarizes standard features, as specified by Sun, but focuses primarily on Oracle implementation details and value-added features.

The OC4J JSP container in Oracle9*i*AS 9.0.2 is a complete implementation of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, with many JSP 1.2-compliant features as well. There will be a fully 1.2-compliant release in the near future.

This preface contains these topics:

Note: The Sample Applications chapter was removed for Oracle9*i*AS 9.0.2. Applications that were listed there are available in the OC4J demos, from either of the following locations:

- the OC4J demo instance, included with the Oracle9*i*AS product
- the JSP download page on the Oracle Technology Network (requiring an OTN membership, which is free):

  `http://otn.oracle.com/tech/java/servlets/content.html`

Intended Audience

This document is intended for developers interested in creating Web applications based on JavaServer Pages technology. It assumes that working Web and servlet environments already exist, and that readers are already familiar with the following:

- general Web technology
- general servlet technology (some technical background is provided in Appendix A)
- how to configure their Web server and servlet environments
- HTML
- Java
- Oracle JDBC (for JSP applications accessing an Oracle database)
- Oracle SQLJ (for JSP database applications using SQLJ)

While some information about standard JSP 1.1 technology and syntax is provided in Chapter 1 and elsewhere, there is no attempt at completeness in this area. For additional information about standard JSP 1.1 features, consult the Sun Microsystems *JavaServer Pages Specification, Version 1.1* or other appropriate reference materials.

Because the JSP 1.1 specification relies on a servlet 2.2 environment, this document is geared largely toward such environments, as well as some JSP 1.2 and servlet 2.3 features. The OC4J JSP container has special features for earlier servlet environments, however, and there is special discussion of these features in Appendix B as they relate to servlet 2.0 environments, particularly Apache JServ, which is included with the Oracle9*i* Application Server.

For documentation of tag libraries and utilities that are provided with the OC4J product, please refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* (although an overview is provided here, in "Overview of Tag Libraries and Utilities Provided with OC4J" on page 2-10).

For a quick primer about getting started with JSP pages in OC4J, see the *Oracle9iAS Containers for J2EE User's Guide*.

Documentation Accessibility

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

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

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

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

Organization

This document contains:

Chapter 1, "General JSP Overview"

This chapter highlights standard JSP 1.1 technology. It is not intended as a complete reference.

Chapter 2, "Overview of the Oracle JSP Implementation"

This chapter provides an overview of the JSP implementation provided with OC4J, including both portable and Oracle-specific value-added features.

Chapter 3, "Getting Started"

This contains information about required files for the OC4J JSP container, OC4J Web server configuration, and JSP configuration.

Chapter 4, "Basic Programming Issues"

This chapter introduces basic JSP programming considerations and provides a starter sample for database access.

Chapter 5, "Key Considerations"

This chapter discusses a variety of general programming and configuration issues and data-access considerations that the developer should be aware of. It also covers considerations specific to the OC4J environment.

Chapter 6, "JSP Translation and Deployment"

This chapter describes features of the Oracle9*i*AS JSP translator and Oracle `ojspc` pre-translation utility, and discusses general and OC4J-specific deployment considerations.

Chapter 7, "JSP Tag Libraries"

This chapter introduces the basic JSP 1.1 framework for custom tag libraries.

Chapter 8, "JSP Globalization Support"

This chapter covers features for globalization support.

Appendix A, "Servlet and JSP Technical Background"

This appendix provides a brief background of servlet technology and introduces the standard JSP interfaces for translated pages.

Appendix B, "The Apache JServ Environment"

This appendix provides details for the JServ servlet 2.0 environment, including deployment, configuration, and special programming considerations.

Appendix C, "Third Party Licenses"

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

Related Documentation

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

- *Oracle9iAS Containers for J2EE User's Guide*

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

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

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

- *Oracle9iAS Containers for J2EE Servlet Developer's Guide*

  This book provides information for servlet developers regarding use of servlets and the servlet container in OC4J. It also documents relevant OC4J configuration files.

- *Oracle9iAS Containers for J2EE Services Guide*

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

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

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

Also available from the Oracle Java Platform group:

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

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

- *Oracle9i Application Server Administrator's Guide*
- *Oracle Enterprise Manager Administrator's Guide*
- *Oracle HTTP Server Administration Guide*

- *Oracle9i Application Server Performance Guide*
- *Oracle9i Application Server Globalization Support Guide*
- *Oracle9iAS Web Cache Administration and Deployment Guide*
- *Oracle9i Application Server: Migrating from Oracle9i Application Server 1.x*

The following are available from the JDeveloper group:

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

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

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

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

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

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

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

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

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

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

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

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

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

The following Oracle Technology Network (OTN) resources are available for further information about JavaServer Pages:

- OTN Web site for Java servlets and JavaServer Pages:

  `http://otn.oracle.com/tech/java/servlets/`

- OTN JSP discussion forums, accessible through the following address:

  `http://www.oracle.com/forums/forum.jsp?id=399160`

The following resources are available from Sun Microsystems:

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

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

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

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

- `jsp-interest` discussion group for JavaServer Pages

  To subscribe, send an e-mail to `listserv@java.sun.com` with the following line in the body of the message:

  `subscribe jsp-interest` *`yourlastname yourfirstname`*

  It is recommended, however, that you request only the daily digest of the posted e-mails. To do this add the following line to the message body as well:

  `set jsp-interest digest`

Conventions

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

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

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

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
UPPERCASE monospace (fixed-width) font	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a NUMBER column. You can back up the database by using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES data dictionary view. Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter sqlplus to open SQL*Plus. The password is specified in the orapwd file. Back up the data files and control files in the /disk1/oracle/dbs directory. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user. The JRepUtil class implements these methods.

Convention	Meaning	Example
lowercase italic monospace (fixed-width) font	Lowercase italic monospace font represents place holders or variables.	You can specify the *parallel_clause*. Run *old_release*.SQL where *old_release* refers to the release you installed prior to upgrading.

Conventions in Code Examples

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

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

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
...	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates place holders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`* `DB_NAME = `*`database_name`*

Convention	Meaning	Example

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

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

1

General JSP Overview

This chapter reviews standard features and functionality of JavaServer Pages technology, then concludes with a discussion of JSP execution models. For further general information, consult the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

For an overview of the JSP implementation in OC4J, see Chapter 2, "Overview of the Oracle JSP Implementation". Also note that Appendix A, "Servlet and JSP Technical Background", provides related background on standard servlet and JSP technology.

The following topics are covered here:

- Introduction to JavaServer Pages
- Overview of JSP Syntax Elements
- JSP Execution

Introduction to JavaServer Pages

JavaServer Pages(TM) is a technology specified by Sun Microsystems as a convenient way of generating dynamic content in pages that are output by a Web application (an application running on a Web server).

This technology, which is closely coupled with Java servlet technology, allows you to include Java code snippets and calls to external Java components within the HTML code (or other markup code, such as XML) of your Web pages. JavaServer Pages (JSP) technology works nicely as a front-end for business logic and dynamic functionality in JavaBeans and Enterprise JavaBeans (EJBs).

JSP code is distinct from other Web scripting code, such as JavaScript, in a Web page. Anything that you can include in a normal HTML page can be included in a JSP page as well.

In a typical scenario for a database application, a JSP page will call a component such as a JavaBean or Enterprise JavaBean, and the bean will directly or indirectly access the database, generally through JDBC or perhaps SQLJ.

A JSP page is translated into a Java servlet before being executed, and processes HTTP requests and generates responses similarly to any other servlet. JSP technology offers a more convenient way to code the servlet. The translation typically occurs on demand, but sometimes in advance.

Furthermore, JSP pages are fully interoperable with servlets—JSP pages can include output from a servlet or forward to a servlet, and servlets can include output from a JSP page or forward to a JSP page.

Note: See the OC4J demos for some basic JSP sample applications.

What a JSP Page Looks Like

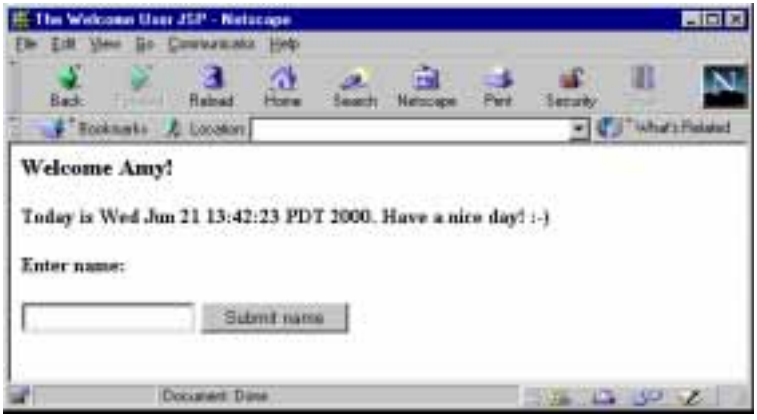
Here is an example of a simple JSP page. For an explanation of JSP syntax elements used here, see "Overview of JSP Syntax Elements" on page 1-7.

```
<HTML>
<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
```

```
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

In a JSP page, Java elements are set off by tags such as <% and %>, as in the preceding example. In this example, Java snippets get the user name from an HTTP request object, print the user name, and get the current date.

This JSP page will produce the following output if the user inputs the name "Amy":



Convenience of JSP Coding Versus Servlet Coding

Combining Java code and Java calls into an HTML page is more convenient than using straight Java code in a servlet. JSP syntax gives you a shortcut for coding dynamic Web pages, typically requiring much less code than Java servlet syntax. Following is an example contrasting servlet code and JSP code.

Servlet Code

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

public class Hello extends HttpServlet
{
   public void doGet(HttpServletRequest rq, HttpServletResponse rsp)
   {
```

```
      rsp.setContentType("text/html");
      try {
         PrintWriter out = rsp.getWriter();
         out.println("<HTML>");
         out.println("<HEAD><TITLE>Welcome</TITLE></HEAD>");
         out.println("<BODY>");
         out.println("<H3>Welcome!</H3>");
         out.println("<P>Today is "+new java.util.Date()+".</P>");
         out.println("</BODY>");
         out.println("</HTML>");
      } catch (IOException ioe)
      {
        // (error processing)
      }
   }
}
```

See "The Servlet Interface" on page A-3 for some background information about the standard `HttpServlet` abstract class, `HttpServletRequest` interface, and `HttpServletResponse` interface.

JSP Code

```
<HTML>
<HEAD><TITLE>Welcome</TITLE></HEAD>
<BODY>
<H3>Welcome!</H3>
<P>Today is <%= new java.util.Date() %>.</P>
</BODY>
</HTML>
```

Note how much simpler JSP syntax is. Among other things, it saves Java overhead such as package imports and `try...catch` blocks.

Additionally, the JSP translator automatically handles a significant amount of servlet coding overhead for you in the `.java` file that it outputs, such as directly or indirectly implementing the standard `javax.servlet.jsp.HttpJspPage` interface (see "Standard JSP Interfaces and Methods" on page A-12) and adding code to acquire an HTTP session.

Also note that because the HTML of a JSP page is not embedded within Java print statements, as it is in servlet code, you can use HTML authoring tools to create JSP pages.

Separation of Business Logic from Page Presentation—Calling JavaBeans

JSP technology allows separating the development efforts between the HTML code that determines static page presentation, and the Java code that processes business logic and presents dynamic content. It therefore becomes much easier to split maintenance responsibilities between presentation and layout specialists who may be proficient in HTML but not Java, and code specialists who may be proficient in Java but not HTML.

In a typical JSP page, most Java code and business logic will *not* be within snippets embedded in the JSP page—instead, it will be in JavaBeans or Enterprise JavaBeans that are invoked from the JSP page.

JSP technology offers the following syntax for defining and creating an instance of a JavaBeans class:

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

This example creates an instance, `pageBean`, of the `mybeans.NameBean` class. The `scope` parameter will be explained later in this chapter.

Later in the page, you can use this bean instance, as in the following example:

```
Hello <%= pageBean.getNewName() %> !
```

This prints "Hello Julie !", for example, if the name "Julie" is in the `newName` attribute of `pageBean`, which might occur through user input.

The separation of business logic from page presentation allows convenient division of responsibilities between the Java expert who is responsible for the business logic and dynamic content—this developer owns and maintains the code for the `NameBean` class—and the HTML expert who is responsible for the static presentation and layout of the Web page that the application user sees—this developer owns and maintains the code in the `.jsp` file for this JSP page.

Tags used with JavaBeans—`useBean` to declare the JavaBean instance and `getProperty` and `setProperty` to access bean properties—are further discussed in "JSP Actions and the <jsp: > Tag Set" on page 1-15.

JSP Pages and Alternative Markup Languages

JavaServer Pages technology is typically used for dynamic HTML output, but the Sun Microsystems *JavaServer Pages Specification, Version 1.1* also supports additional types of structured, text-based document output. A JSP translator does not process

text outside of JSP elements, so any text that is appropriate for Web pages in general is typically appropriate for a JSP page as well.

A JSP page takes information from an HTTP request and accesses information from a data server (such as through a SQL database query). It combines and processes this information and incorporates it, as appropriate, into an HTTP response with dynamic content. The content can be formatted as HTML, DHTML, XHTML, or XML, for example.

For information about XML support, refer to "XML-Alternative Syntax" on page 5-23 and to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Overview of JSP Syntax Elements

You have seen a simple example of JSP syntax in "What a JSP Page Looks Like" on page 1-2. Now here is a top-level list of syntax categories and topics:

- *directives*—These convey information regarding the JSP page as a whole.
- *scripting elements*—These are Java coding elements such as declarations, expressions, scriptlets, and comments.
- *objects* and *scopes*—JSP objects can be created either explicitly or implicitly and are accessible within a given "scope", such as from anywhere in the JSP page or the session.
- *actions*—These create objects or affect the output stream in the JSP response (or both).

This section introduces each category, including basic syntax and a few examples. For more information, see the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

Notes: There are XML-compatible alternatives to the syntax for JSP directives, declarations, expressions, and scriptlets. See "XML-Alternative Syntax" on page 5-23.

Directives

Directives provide instruction to the JSP container regarding the entire JSP page. This information is used in translating or executing the page. The basic syntax is as follows:

```
<%@ directive attribute1="value1" attribute2="value2"... %>
```

The JSP 1.1 specification supports the following directives:

- `page`—Use this directive to specify any of a number of page-dependent attributes, such as the scripting language to use, a class to extend, a package to import, an error page to use, or the JSP page output buffer size. For example:

  ```
  <%@ page language="java" import="packages.mypackage" errorPage="boof.jsp" %>
  ```

  or, to set the JSP page output buffer size to 20 KB (the default is 8 KB):

  ```
  <%@ page buffer="20kb" %>
  ```

or, to unbuffer the page:

```
<%@ page buffer="none" %>
```

Notes:

- A JSP page using an error page must be buffered. Forwarding to an error page (not outputting it to the browser) clears the buffer.
- In Oracle's JSP implementation, `java` is the default language setting. It is good programming practice to set it explicitly, however. You can also use a `sqlj` setting for SQLJ JSP pages.

- `include`—Use this directive to specify a resource that contains text or code to be inserted into the JSP page when it is translated.

  Example:

  ```
  <%@ include file="/jsp/userinfopage.jsp" %>
  ```

  Specify either a page-relative or context-relative path to the resource. (See "Requesting a JSP Page" on page 1-25 for discussion of page-relative and context-relative paths.)

  Notes:

  - The `include` directive, referred to as a "static include", is comparable in nature to the `jsp:include` action discussed later in this chapter, but takes effect at JSP translation-time instead of request-time. See "Static Includes Versus Dynamic Includes" on page 5-3.
  - The `include` directive can be used only between pages in the same servlet context (application).

- `taglib`—Use this directive to specify a library of custom JSP tags that will be used in the JSP page. Vendors can extend JSP functionality with their own sets of tags. This directive indicates the location of a *tag library description* file and a prefix to distinguish use of tags from that library. For example:

  ```
  <%@ taglib uri="/oracustomtags" prefix="oracust" %>
  ```

Later in the page, use the `oracust` prefix whenever you want to use one of the tags in the library (presume this library includes a tag `dbaseAccess`):

```
<oracust:dbaseAccess ... >
...
</oracust:dbaseAccess>
```

JSP tag libraries and tag library description files are introduced later in this chapter, in "Tag Libraries" on page 1-21, and discussed in detail in Chapter 7, "JSP Tag Libraries".

Scripting Elements

JSP scripting elements include the following categories of snippets of Java code that can appear in a JSP page:

- *declarations*—These are statements declaring methods or member variables that will be used in the JSP page.

  A JSP declaration uses standard Java syntax within the `<%!...%>` declaration tags to declare a member variable or method. This will result in a corresponding declaration in the generated servlet code. For example:

  ```
  <%! double f1=0.0; %>
  ```

  This example declares a member variable, `f1`. In the servlet class code generated by the JSP translator, `f1` will be declared at the class top level.

  Note: Method variables, as opposed to member variables, are declared within JSP scriptlets as described below.

- *expressions*—These are Java expressions that are evaluated, converted into string values as appropriate, and displayed where they are encountered on the page.

  A JSP expression does *not* end in a semicolon, and is contained within `<%=...%>` tags. For example:

  ```
  <P><B> Today is <%= new java.util.Date() %>. Have a nice day! </B></P>
  ```

> **Note:** A JSP expression in a request-time attribute, such as in a `jsp:setProperty` statement, need not be converted to a string value.

- *scriptlets*—These are portions of Java code intermixed within the markup language of the page.

  A scriptlet, or code fragment, can consist of anything from a partial line to multiple lines of Java code. You can use them within the HTML code of a JSP page to set up conditional branches or a loop, for example.

  A JSP scriptlet is contained within `<%...%>` scriptlet tags, using normal Java syntax.

  Example 1:

```
<% if (pageBean.getNewName().equals("")) { %>
   I don't know you.
<% } else { %>
   Hello <%= pageBean.getNewName() %>.
<% } %>
```

  Three one-line JSP scriptlets are intermixed with two lines of HTML code (one of which includes a JSP expression, which does *not* require a semicolon). Note that JSP syntax allows HTML code to be the code that is conditionally executed within the `if` and `else` branches (inside the Java brackets set out in the scriptlets).

  The preceding example assumes the use of a JavaBean instance, `pageBean`.

  Example 2:

```
<% if (pageBean.getNewName().equals("")) { %>
   I don't know you.
   <% empmgr.unknownemployee();
} else { %>
   Hello <%= pageBean.getNewName() %>.
   <% empmgr.knownemployee();
} %>
```

  This example adds more Java code to the scriptlets. It assumes the use of a JavaBean instance, `pageBean`, and assumes that some object, `empmgr`, was previously instantiated and has methods to execute appropriate functionality for a known employee or an unknown employee.

Note: Use a JSP scriptlet to declare method variables, as opposed to member variables, as in the following example:

```
<% double f2=0.0; %>
```

This scriptlet declares a method variable, `f2`. In the servlet class code generated by the JSP translator, `f2` will be declared as a variable within the service method of the servlet.

Member variables are declared in JSP declarations as described above.

For a comparative discussion, see "Method Variable Declarations Versus Member Variable Declarations" on page 5-8.

- *comments*—These are developer comments embedded within the JSP code, similar to comments embedded within any Java code.

  Comments are contained within `<%--.....--%>` tags. Unlike HTML comments, these comments are not visible when a user views the page source.

  Example:

  ```
  <%-- Execute the following branch if no user name is entered. --%>
  ```

JSP Objects and Scopes

In this document, the term *JSP object* refers to a Java class instance declared within or accessible to a JSP page. JSP objects can be either:

- *explicit*—Explicit objects are declared and created within the code of your JSP page, accessible to that page and other pages according to the `scope` setting you choose.

or:

- *implicit*—Implicit objects are created by the underlying JSP mechanism and accessible to Java scriptlets or expressions in JSP pages according to the inherent `scope` setting of the particular object type.

This section covers the following topics:

- Using an Implicit Object
- Object Scopes

Explicit Objects

Explicit objects are typically JavaBean instances that are declared and created in `jsp:useBean` action statements. The `jsp:useBean` statement and other action statements are described in "JSP Actions and the <jsp: > Tag Set" on page 1-15, but here is an example:

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

This statement defines an instance, `pageBean`, of the `NameBean` class that is in the `mybeans` package. The `scope` parameter is discussed in "Object Scopes" on page 1-14.

You can also create objects within Java scriptlets or declarations, just as you would create Java class instances in any Java program.

Implicit Objects

JSP technology makes available to any JSP page a set of *implicit objects*. These are Java class instances that are created automatically by the JSP container and that allow interaction with the underlying servlet environment.

The following implicit objects are available. For information about methods available with these objects, refer to the Sun Microsystems Javadoc for the noted classes and interfaces at the following locations (for servlet 2.2 and servlet 2.3 classes, respectively):

```
http://java.sun.com/products/servlet/2.2/javadoc/index.html
```

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

- `page`

  This is an instance of the JSP page implementation class that was created when the page was translated, and that implements the interface `javax.servlet.jsp.HttpJspPage`; `page` is synonymous with `this` within a JSP page.

- `request`

  This represents an HTTP request and is an instance of a class that implements the `javax.servlet.http.HttpServletRequest` interface, which extends the `javax.servlet.ServletRequest` interface.

- `response`

  This represents an HTTP response and is an instance of a class that implements the `javax.servlet.http.HttpServletResponse` interface, which extends the `javax.servlet.ServletResponse` interface.

  The `response` and `request` objects for a particular request are associated with each other.

- `pageContext`

  This represents the *page context* of a JSP page, which is provided for storage and access of all `page` scope objects of a JSP page instance. A `pageContext` object is an instance of the `javax.servlet.jsp.PageContext` class.

  The `pageContext` object has `page` scope, making it accessible only to the JSP page instance with which it is associated.

- `session`

  This represents an HTTP session and is an instance of a class that implements the `javax.servlet.http.HttpSession` class.

- `application`

  This represents the servlet context for the Web application and is an instance of the `javax.servlet.ServletContext` class.

  The `application` object is accessible from any JSP page instance running as part of any instance of the application within a single JVM. (The programmer should be aware of the server architecture regarding use of JVMs.)

- `out`

  This is an object that is used to write content to the output stream of a JSP page instance. It is an instance of the `javax.servlet.jsp.JspWriter` class, which extends the `java.io.Writer` class.

  The `out` object is associated with the `response` object for a particular request.

- `config`

  This represents the servlet configuration for a JSP page and is an instance of a class that implements the `javax.servlet.ServletConfig` interface. Generally speaking, servlet containers use `ServletConfig` instances to provide information to servlets during initialization. Part of this information is the appropriate `ServletContext` instance.

- `exception` (JSP error pages only)

  This implicit object applies only to JSP error pages—these are pages to which processing is forwarded when an exception is thrown from another JSP page. They must have the `page` directive `isErrorPage` attribute set to `true`.

  The implicit `exception` object is a `java.lang.Exception` instance that represents the uncaught exception that was thrown from another JSP page and that resulted in the current error page being invoked.

  The `exception` object is accessible only from the JSP error page instance to which processing was forwarded when the exception was encountered.

  For an example of JSP error processing and use of the `exception` object, see "JSP Runtime Error Processing" on page 4-13.

Using an Implicit Object

Any of the implicit objects discussed in the preceding section may be useful. The following example uses the `request` object to retrieve and display the value of the `username` parameter from the HTTP request:

```
<H3> Welcome <%= request.getParameter("username") %> ! <H3>
```

The `request` object, like the other implicit objects, is available automatically; it is not explicitly instantiated.

Object Scopes

Objects in a JSP page, whether explicit or implicit, are accessible within a particular *scope*. In the case of explicit objects, such as a JavaBean instance created in a `jsp:useBean` action, you can explicitly set the scope with the following syntax, as in the example in "Explicit Objects" on page 1-12:

```
scope="scopevalue"
```

There are four possible scopes:

- `scope="page"` (default scope)—The object is accessible only from within the JSP page where it was created. A page-scope object is stored in the implicit `pageContext` object. The `page` scope ends when the page stops executing.

  Note that when the user refreshes the page while executing a JSP page, new instances will be created of all page-scope objects.

- `scope="request"`—The object is accessible from any JSP page servicing the same HTTP request that is serviced by the JSP page that created the object. A

request-scope object is stored in the implicit `request` object. The `request` scope ends at the conclusion of the HTTP request.

- `scope="session"`—The object is accessible from any JSP page that is sharing the same HTTP session as the JSP page that created the object. A session-scope object is stored in the implicit `session` object. The `session` scope ends when the HTTP session times out or is invalidated.
- `scope="application"`—The object is accessible from any JSP page that is used in the same Web application as the JSP page that created the object, within any single Java virtual machine. The concept is similar to that of a Java static variable. An application-scope object is stored in the implicit `application` servlet context object. The `application` scope ends when the application itself terminates, or when the JSP container or servlet container shuts down.

You can think of these four scopes as being in the following progression, from narrowest scope to broadest scope:

`page < request < session < application`

If you want to share an object between different pages in an application, such as when forwarding execution from one page to another, or including content from one page in another, you cannot use `page` scope for the shared object; in this case, there would be a separate object instance associated with each page. The narrowest scope you can use to share an object between pages is `request`. (For information about including and forwarding pages, see "JSP Actions and the <jsp: > Tag Set" below.)

Note: The `request`, `session`, and `application` scopes also apply to servlets.

JSP Actions and the <jsp: > Tag Set

JSP action elements result in some sort of action occurring while the JSP page is being executed, such as instantiating a Java object and making it available to the page. Such actions may include the following:

- creating a JavaBean instance and accessing its properties
- forwarding execution to another HTML page, JSP page, or servlet
- including an external resource in the JSP page

Action elements use a set of standard JSP tags that use `<jsp:`*`tag ... `*`>` syntax. Although directives and scripting elements described earlier in this chapter are

sufficient to code a JSP page, the <jsp:> tags described here provide additional functionality and convenience.

Here is the general JSP tag syntax:

```
<jsp:tag attr1="value1" attr2="value2" ... attrN="valueN">
...body...
</jsp:tag>
```

or, where there is no body:

```
<jsp:tag attr1="value1", ..., attrN="valueN" />
```

The JSP specification includes the following standard action tags, which are introduced and briefly discussed here:

- jsp:useBean

  The jsp:useBean tag creates an instance of a specified JavaBean class, gives the instance a specified name, and defines the scope within which it is accessible (such as from anywhere within the current JSP page instance).

  Example:

  ```
  <jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
  ```

  This example creates a page-scoped instance pageBean of the mybeans.NameBean class. This instance is accessible only from the JSP page instance that creates it.

- jsp:setProperty

  The jsp:setProperty tag sets one or more bean properties. The bean must have been previously specified in a jsp:useBean tag. You can directly specify a value for a specified property, or take the value for a specified property from an associated HTTP request parameter, or iterate through a series of properties and values from the HTTP request parameters.

  The following example sets the user property of the pageBean instance (defined in the preceding jsp:useBean example) to a value of "Smith":

  ```
  <jsp:setProperty name="pageBean" property="user" value="Smith" />
  ```

Note: Beginning with Oracle9*i*AS 9.0.2, the OC4J JSP container properly handles settings for bean properties that are object types (`java.lang.Object`). A string specified for the `value` setting in a `jsp:setProperty` tag is converted to a `java.lang.Object` instance and passed in to the corresponding JavaBean setter method. This features complies with the JSP 1.2 specification.

The following example sets the `user` property of the `pageBean` instance according to the value set for a parameter called `username` in the HTTP request:

```
<jsp:setProperty name="pageBean" property="user" param="username" />
```

Or, if the bean property and request parameter have the same name (`user`), you can simply set the property as follows:

```
<jsp:setProperty name="pageBean" property="user" />
```

The following example results in iteration over the HTTP request parameters, matching bean property names with request parameter names and setting bean property values according to the corresponding request parameter values:

```
<jsp:setProperty name="pageBean" property="*" />
```

Important: For `property="*"`, the JSP 1.1 specification does not stipulate the order in which properties are set. If order matters, and if you want to ensure that your JSP page is portable, you should use a separate `jsp:setProperty` statement for each property.

Also, if you use separate `jsp:setProperty` statements, then the JSP translator can generate the corresponding `setXXX()` methods directly. In this case, introspection occurs only during translation. There will be no need to introspect the bean during runtime, which is more costly.

- `jsp:getProperty`

  The `jsp:getProperty` tag reads a bean property value, converts it to a Java string, and places the string value into the implicit `out` object so that it can be displayed as output. The bean must have been previously specified in a `jsp:useBean` tag. For the string conversion, primitive types are converted

directly, and object types are converted using the `toString()` method specified in the `java.lang.Object` class.

The following example puts the value of the `user` property of the `pageBean` bean into the `out` object:

```
<jsp:getProperty name="pageBean" property="user" />
```

- `jsp:param`

  You can use `jsp:param` tags in conjunction with `jsp:include`, `jsp:forward`, and `jsp:plugin` tags (described below).

  Used with `jsp:forward` and `jsp:include` tags, a `jsp:param` tag optionally provides key / value pairs for parameter values in the HTTP `request` object. New parameters and values specified with this action are added to the `request` object, with new values taking precedence over old.

  The following example sets the `request` object parameter `username` to a value of `Smith`:

  ```
  <jsp:param name="username" value="Smith" />
  ```

- `jsp:include`

  The `jsp:include` tag inserts additional static or dynamic resources into the page at request-time as the page is displayed. Specify the resource with a relative URL (either page-relative or application-relative).

  The `flush` attribute is mandatory. The Sun Microsystems *JavaServer Pages Specification, Version 1.1*, supports only a "true" setting, which results in the buffer being flushed to the browser when a `jsp:include` action is executed. The JSP 1.2 specification and OC4J JSP in Oracle9*i*AS 9.0.2, however, support a "false" setting.

  You can also have an action body with `jsp:param` tags, as shown in the second example.

  Examples:

  ```
  <jsp:include page="/templates/userinfopage.jsp" flush="true" />
  ```

  or:

  ```
  <jsp:include page="/templates/userinfopage.jsp" flush="true" >
     <jsp:param name="username" value="Smith" />
     <jsp:param name="userempno" value="9876" />
  </jsp:include>
  ```

Note that the following syntax would work as an alternative to the preceding example:

```
<jsp:include page="/templates/userinfopage.jsp?username=Smith&userempno=9876" flush="true" />
```

Notes:

- The `jsp:include` tag, known as a "dynamic include", is similar in nature to the `include` directive discussed earlier in this chapter, but takes effect at request-time instead of translation-time. See "Static Includes Versus Dynamic Includes" on page 5-3.
- The `jsp:include` tag can be used only between pages in the same servlet context.

- `jsp:forward`

  The `jsp:forward` tag effectively terminates execution of the current page, discards its output, and dispatches a new page—either an HTML page, a JSP page, or a servlet.

  The JSP page must be buffered to use a `jsp:forward` tag; you cannot set `buffer="none"` in a `page` directive. The action will clear the buffer, not outputting contents to the browser.

  As with `jsp:include`, you can also have an action body with `jsp:param` tags, as shown in the second of the following examples.

  Examples:

  ```
  <jsp:forward page="/templates/userinfopage.jsp" />
  ```

  or:

  ```
  <jsp:forward page="/templates/userinfopage.jsp" >
     <jsp:param name="username" value="Smith" />
     <jsp:param name="userempno" value="9876" />
  </jsp:forward>
  ```

Notes:

- The difference between the `jsp:forward` examples here and the `jsp:include` examples earlier is that the `jsp:include` examples insert `userinfopage.jsp` within the output of the current page; the `jsp:forward` examples stop executing the current page and display `userinfopage.jsp` instead.
- The `jsp:forward` tag can be used only between pages in the same servlet context.
- The `jsp:forward` tag results in the original `request` object being forwarded to the target page. As an alternative, if you do not want the `request` object forwarded, you can use the `sendRedirect(String)` method specified in the standard `javax.servlet.http.HttpServletResponse` interface. This sends a temporary redirect response to the client using the specified redirect-location URL. You can specify a relative URL; the servlet container will convert the relative URL to an absolute URL.

- `jsp:plugin`

  The `jsp:plugin` tag results in the execution of a specified applet or JavaBean in the client browser, preceded by a download of Java plugin software if necessary.

  Specify configuration information, such as the applet to run and the code base, using `jsp:plugin` attributes. The JSP container might provide a default URL for the download, but you can also specify attribute `nspluginurl="`*`url`*`"` (for a Netscape browser) or `iepluginurl="`*`url`*`"` (for an Internet Explorer browser).

  Use nested `jsp:param` tags within `<jsp:params>` and `</jsp:params>` start and end tags to specify parameters to the applet or JavaBean. (Note that these `jsp:params` start and end tags are *not* necessary when using `jsp:param` in a `jsp:include` or `jsp:forward` action.)

  Use `<jsp:fallback>` and `</jsp:fallback>` start and end tags to delimit alternative text to execute if the plugin cannot run.

The following example, from the *Sun Microsystems JavaServer Pages Specification, Version 1.1*, shows the use of an applet plugin:

```
<jsp:plugin type=applet code="Molecule.class" codebase="/html" >
   <jsp:params>
      <jsp:param name="molecule" value="molecules/benzene.mol" />
   </jsp:params>
   <jsp:fallback>
      <p> Unable to start the plugin. </p>
   </jsp:fallback>
</jsp:plugin>
```

Many additional parameters—such as `ARCHIVE`, `HEIGHT`, `NAME`, `TITLE`, and `WIDTH`—are allowed in the `jsp:plugin` tag as well. Use of these parameters is according to the general HTML specification.

Tag Libraries

In addition to the standard JSP tags discussed above, the JSP specification lets vendors define their own *tag libraries* and also lets vendors implement a framework allowing customers to define their own tag libraries.

A tag library defines a collection of custom tags and can be thought of as a JSP sub-language. Developers can use tag libraries directly when manually coding a JSP page, but they might also be used automatically by Java development tools. A tag library must be portable between different JSP container implementations.

Import a tag library into a JSP page using the `taglib` directive introduced in "Directives" on page 1-7.

Key concepts of standard JavaServer Pages support for JSP tag libraries include the following topics:

- tag handlers

  A *tag handler* implements the action of a custom tag. A tag handler is an instance of a Java class that implements either the `Tag` or `BodyTag` interface (depending on whether the tag uses a body between a start tag and an end tag) in the standard `javax.servlet.jsp.tagext` package.

- scripting variables

  Custom tag actions can create server-side objects available for use by the tag itself or by other scripting elements such as scriptlets. This is accomplished by creating or updating *scripting variables*.

Details regarding scripting variables that a custom tag defines must be specified in a subclass of the standard `javax.servlet.jsp.tagext.TagExtraInfo` abstract class. This document refers to such a subclass as a *tag-extra-info class*. The JSP container uses instances of these classes during translation.

- tag library description files

  A *tag library description* (TLD) file is an XML document that contains information about a tag library and about individual tags of the library. The file name of a TLD has the `.tld` extension.

  A JSP container uses the TLD file in determining what action to take when it encounters a tag from the library.

- use of `web.xml` for tag libraries

  The Sun Microsystems *Java Servlet Specification, Version 2.2* (and higher) describes a standard deployment descriptor for servlets—the `web.xml` file. JSP applications can use this file in specifying the location of a JSP tag library description file.

  For JSP tag libraries, the `web.xml` file can include a `taglib` element and two subelements: `taglib-uri` and `taglib-location`.

For information about these topics, see "Standard Tag Library Framework" on page 7-2. For further information, see the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

For complete information about the tag libraries provided with OC4J, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

JSP Execution

This section provides a top-level look at how a JSP page is run, including on-demand translation (the first time a JSP page is run), the role of the *JSP container* and the servlet container, and error processing.

Note: The term *JSP container* is used in the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, replacing the term *JSP engine* that was used in earlier specifications. The two terms are synonymous.

JSP Containers in a Nutshell

A JSP container is an entity that translates, executes, and processes JSP pages and delivers requests to them.

The exact make-up of a JSP container varies from implementation to implementation, but it will consist of a servlet or collection of servlets. The JSP container, therefore, is executed by a servlet container. Servlet containers are summarized in "Servlet Containers" on page A-3.

A JSP container can be incorporated into a Web server if the Web server is written in Java, or the container can be otherwise associated with and used by the Web server.

JSP Execution Models

There are two distinct execution models for JSP pages:

- In most implementations and situations, the JSP container translates pages *on demand* before triggering their execution; that is, at the time they are requested by the user.
- In some scenarios, however, the developer may want to translate the pages in advance and deploy them as working servlets. Command-line tools are available to translate the pages, load them, and "publish" them to make them available for execution. You can have the translation occur either on the client or in the server. When the end-user requests the JSP page, it is executed directly, with no translation necessary.

On-Demand Translation Model

It is typical to run JSP pages in an on-demand translation scenario. When a JSP page is requested from a Web server that incorporates the JSP container, a front-end

servlet is instantiated and invoked, assuming proper Web server configuration. This servlet can be thought of as the front-end of the JSP container. In OC4J, it is `oracle.jsp.runtimev2.JspServlet`.

`JspServlet` locates the JSP page, translates and compiles it if necessary (if the translated class does not exist or has an earlier timestamp than the JSP page source), and triggers its execution.

Note that the Web server must be properly configured to map the `*.jsp` file name extension (in a URL) to `JspServlet`. This is handled automatically during OC4J installation, as discussed in "JSP Container Setup" on page 3-3.

Pre-Translation Model

As an alternative to the typical on-demand scenario, developers may want to pre-translate their JSP pages before deploying them. This can offer the following advantages, for example:

- It can save time for the end-users when they first request a JSP page, because translation at execution time is not necessary.
- It is also useful if you want to deploy binary files only, perhaps because the software is proprietary or you have security concerns and you do not want to expose the code.

For more information, see "JSP Pre-Translation" on page 6-31 and "Deployment of Binary Files Only" on page 6-33.

Oracle supplies the `ojspc` command-line utility for pre-translating JSP pages. This utility has options that allow you to set an appropriate base directory for the output files, depending on how you want to deploy the application. The `ojspc` utility is documented in "The ojspc Pre-Translation Utility" on page 6-13.

JSP Pages and On-Demand Translation

Presuming the typical on-demand translation scenario, a JSP page is usually executed through the following steps:

1. The user requests the JSP page through a URL ending with a `.jsp` file name.
2. Upon noting the `.jsp` file name extension in the URL, the servlet container of the Web server invokes the JSP container.
3. The JSP container locates the JSP page and translates it if this is the first time it has been requested. Translation includes producing servlet code in a `.java` file and then compiling the `.java` file to produce a servlet `.class` file.

The servlet class generated by the JSP translator extends a class (provided by the JSP container) that implements the `javax.servlet.jsp.HttpJspPage` interface (described in "Standard JSP Interfaces and Methods" on page A-12). The servlet class is referred to as the *page implementation class*. This document will refer to instances of page implementation classes as *JSP page instances*.

Translating a JSP page into a servlet automatically incorporates standard servlet programming overhead into the generated servlet code, such as implementing the `HttpJspPage` interface and generating code for its service method.

4. The JSP container triggers instantiation and execution of the page implementation class.

The JSP page instance will then process the HTTP request, generate an HTTP response, and pass the response back to the client.

Note: The preceding steps are loosely described for purposes of this discussion. As mentioned earlier, each vendor decides how to implement its JSP container, but it will consist of a servlet or collection of servlets. For example, there may be a front-end servlet that locates the JSP page, a translation servlet that handles translation and compilation, and a wrapper servlet class that is extended by each page implementation class (because a translated page is not a pure servlet and cannot be run directly by the servlet container). A servlet container is required to run each of these components.

Requesting a JSP Page

A JSP page can be requested either directly—through a URL—or indirectly—through another Web page or servlet.

Directly Requesting a JSP Page

As with a servlet or HTML page, the end-user can request a JSP page directly by URL. For example, suppose you have a `HelloWorld` JSP page that is located under a `myapp` directory, as follows, where `myapp` is mapped to the `myapproot` root context in the Web server:

```
myapp/dir1/HelloWorld.jsp
```

You can request it with a URL such as the following:

```
http://host:port/myapproot/dir1/HelloWorld.jsp
```

The first time the end-user requests `HelloWorld.jsp`, the JSP container triggers both translation and execution of the page. With subsequent requests, the JSP container triggers page execution only; the translation step is no longer necessary.

> **Note:** This is just a general example. By default in OC4J in Oracle9*i*AS 9.0.2, the context path must start with "/j2ee" if you want processing to be routed to OC4J through the Oracle HTTP Server and `mod_oc4j`, as in a typical deployment environment. Oracle HTTP Server and mod_oc4j are introduced in "Role of the Oracle HTTP Server and mod_oc4j" on page 2-7. General servlet and JSP invocation are discussed in the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*.

Indirectly Requesting a JSP Page

JSP pages, like servlets, can also be executed indirectly—linked from a regular HTML page or referenced from another JSP page or from a servlet.

When invoking one JSP page from a JSP statement in another JSP page, the path can be either relative to the application root—known as *context-relative* or *application-relative*—or relative to the invoking page—known as *page-relative*. An application-relative path starts with "/"; a page-relative path does not.

Be aware that, typically, neither of these paths is the same path as used in a URL or HTML link. Continuing the example in the preceding section, the path in an HTML link is the same as in the direct URL request, as follows:

```
<a href="/myapp/dir1/HelloWorld.jsp" /a>
```

The application-relative path in a JSP statement is:

```
<jsp:include page="/dir1/HelloWorld.jsp" flush="true" />
```

The page-relative path to invoke `HelloWorld.jsp` from a JSP page in the same directory is:

```
<jsp:forward page="HelloWorld.jsp" />
```

("JSP Actions and the <jsp: > Tag Set" on page 1-15 discusses the `jsp:include` and `jsp:forward` statements.)

2

Overview of the Oracle JSP Implementation

The JSP container provided with Oracle9*i*AS Containers for J2EE (OC4J) in the Oracle9*i* Application Server is a complete implementation of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, with support for some JSP 1.2 features as well. Complete JSP 1.2 support will follow in the near future.

This chapter provides overviews of the Oracle9*i* Application Server, OC4J, the OC4J JSP implementation and features, and portable tag libraries and utilities that are also supplied (documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*).

The following topics are covered here:

Overview of the Oracle9*i* Application Server and JSP Support

This section provides a brief overview of the Oracle9*i* Application Server, its J2EE environment, its JSP implementation, and its Web server:

> **Note:** For past users of Oracle9*i*AS release 1.0.2.2, see *Migrating from Oracle9i Application Server 1.x* for information about issues in migrating to Oracle9*i*AS release 2.

Overview of the Oracle9*i* Application Server

Oracle9*i* Application Server is a scalable, secure, middle-tier application server. It can be used to deliver Web content, host Web applications, connect to back-office applications, and make these services accessible to any client browser. Users can access information, perform business analysis, and run business applications on the Internet or corporate intranets or extranets. Major areas of functionality include Business Intelligence, e-Business Integration, J2EE Web Services, Performance and Caching, Portals, Wireless, and Management and Security.

To deliver this range of content and services, the Oracle9*i* Application Server incorporates many components, including the Oracle HTTP Server, Oracle9*i*AS Web Cache, Oracle9*i*AS Portal, Oracle9*i*AS Wireless, Oracle9*i*AS Forms Services and Reports Services (to support Oracle Forms-based applications and reports generation), Oracle9*i*AS Personalization, and various business logic runtime environments that support Enterprise JavaBeans, stored procedures, and Oracle Business Components for Java.

For its J2EE environment, Oracle9*i*AS provides the Oracle9*i*AS Containers for J2EE (OC4J), which includes the JSP container described in this manual, a servlet container, and an EJB container.

(In addition, Oracle9*i*AS includes an Apache JServ servlet environment, documented in Appendix B, "The Apache JServ Environment".)

Overview of OC4J

OC4J is a high-performance J2EE-compliant environment providing a scalable and reliable server infrastructure by supporting *clusters* and *load balancing*. (See the *Oracle9i Application Server Performance Guide* for information about clustering.)

Each OC4J instance runs in a single Java virtual machine. The JVM running an OC4J instance is referred to as a *node*. One or more nodes—typically about two to four—form an *island*. Multiple islands together form a cluster. For each cluster, there is a JVM for each OC4J instance, plus a JVM for the load balancer.

In addition to load balancing, which improves performance by distributing requests among multiple servers, the clustering mechanism provides fault tolerance, which allows any particular server to redirect a client to another server in the event of failure.

OC4J also supports standard EAR/WAR deployment, provides a convenient auto-deployment feature, and supports the following services:

- J2EE Connector Architecture (JCA)—JCA defines a standard architecture for connecting J2EE platforms to heterogeneous enterprise information systems such as ERP systems, mainframe transaction processing, database systems, and legacy applications.
- Java Transaction API (JTA) and two-phase commits—JTA allows simultaneous updates to multiple resources in a single, coordinated transaction.
- Java Message Service (JMS) integration—This integration allows compatibility between Oracle's JMS implementation and those of other JMS providers.
- Java Naming and Directory Interface (JNDI)—JNDI associates names with resources for lookup purposes.

OC4J supplies the following containers:

- a JSP container complying with the Sun JSP 1.1 specification, with selected JSP 1.2 features added, and full JSP 1.2 compliance to follow shortly

  The JSP bundle also supplies tag libraries to implement SQL access, caching capabilities, file access, and other features. For further overview of the JSP container provided with OC4J, see "Overview of the JSP Implementation in OC4J" on page 2-4.
- a servlet container complying with the Sun Microsystems servlet 2.3 specification (see below for key features)

- an EJB container complying with the Sun EJB 1.1 specification, with EJB 2.0 compliance to follow shortly (see below for key features)

Key Servlet Container Features The OC4J servlet container supports the following features:

- servlet filtering—This allows transformation of the content of an HTTP request or response, and modification of header information.
- application-level events—This feature allows greater control over interaction with servlet context and HTTP session objects and, therefore, greater efficiency in managing resources that the application uses.
- integration with SSO and OID—This occurs through the Oracle implementation of the Java Authentication and Authorization Service (JAAS) standard, which provides authentication and fine-grained authorization.

Key EJB Container Features The OC4J EJB container supports the following features:

- session beans—A session bean is used for task-oriented requests. You can define a session bean as stateless or stateful. Stateless session beans cannot maintain state information across calls; stateful session beans can maintain state across calls and so are used for "conversations".
- entity beans—An entity bean represents data. It can use the container to maintain the data persistently, which is referred to as container-managed persistence (CMP), or it can use the bean implementation to manage the data, which is referred to as bean-managed persistence (BMP).
- message-driven beans (MDB)—A message-driven bean is used to receive JMS messages from a queue or topic. It can then invoke other EJBs to process the JMS message.

Overview of the JSP Implementation in OC4J

The JSP container in OC4J is compliant with the Sun Microsystems JSP 1.1 specification. It also offers selected JSP 1.2 features, with full 1.2 compliance to follow in the near future.

In addition, because the JSP 1.1 specification requires a servlet 2.1 or higher environment, the JSP container offers servlet 2.2 emulation for earlier environments, such as the Apache JServ servlet 2.0 environment. JServ is included with Oracle9*i*AS; however, OC4J is the recommended environment.

This section offers additional information on the following topics:

History and Integration of JSP Containers

In Oracle9*i*AS release 1.0.2.2, the first release to include OC4J, there were two JSP containers: 1) a container developed by Oracle and formerly known as OracleJSP; 2) a container licensed from Ironflare AB and formerly known as the "Orion JSP container".

The OracleJSP container offered several advantages, including useful value-added features and enhancements such as for globalization and SQLJ support. The Orion container also offered advantages, including superior speed, but had disadvantages as well. It did not always exhibit standard behavior when compared to the JSP 1.1 reference implementation (Tomcat), and its support for internationalization and globalization was not as complete.

Oracle9*i*AS release 2 integrates the OracleJSP and Orion containers into a single JSP container known as the "OC4J JSP container". This container offers the best features of both previous versions, runs efficiently as a servlet in the OC4J servlet container, and is integrated with other OC4J containers as well. The integrated container primarily consists of the OracleJSP translator and the Orion container runtime, running with a newly simplified dispatcher and the OC4J 1.0.2.2 core runtime classes. The result is one of the fastest JSP engines on the market.

Key Features of the JSP Container in OC4J

The OC4J JSP container adds new features in Oracle9*i*AS 9.0.2, including the following:

- mode switch for automatic page recompilation and class reloading (in OC4J environment only)

  You have a choice of: 1) running JSP pages without any automatic reloading of classes or recompilation of JSP pages; 2) automatically reloading any classes that are used by the JSP page and have changed; or 3) automatically recompiling any JSP pages that have changed, as well as reloading any classes that have changed.

In "JSP Configuration Parameters" on page 3-4, see the description of the `main_mode` parameter.

- single-threaded-model JSP instance pooling (OC4J only)

  For single-threaded (non-thread-safe) JSP pages, page instances are pooled. There is no switch for this feature—it is always enabled.

- tag handler instance pooling (OC4J or JServ)

  To save time in tag handler creation and garbage collection, you can optionally enable pooling of tag handler instances. They are pooled in `application` scope. You can use different settings in different pages, or even in different sections of the same page. See "Disabling or Enabling Tag Handler Instance Pooling" on page 7-19.

- `flush="false"` setting for included pages (OC4J or JServ)

  The current OC4J release does not fully support the JSP 1.2 specification, but does support the setting `flush="false"` for the `jsp:include` tag, as in the following example:

  ```
  <jsp:include flush="false" page="foo.jsp" />
  ```

  The JSP 1.1 specification supports only a `flush="true"` setting, which results in the buffer being flushed to the browser whenever a `jsp:include` tag is executed.

Note: Also see "Overview of Oracle Value-Added Features" on page 2-10.

Portability Across Servlet Environments

The JSP container is provided as a component of OC4J, but is portable to other environments. Because of its servlet 2.2 emulation features, this includes pre-2.1 servlet environments, in particular the Apache JServ servlet 2.0 environment.

The servlet 2.0 specification was limited in that it provided only a single servlet context per Java virtual machine, instead of a servlet context for each application. The OC4J JSP servlet 2.2 emulation allows a full application framework in a servlet 2.0 environment, including providing applications with distinct `ServletContext` and `HttpSession` objects.

Because of this extended functionality, the OC4J JSP container is not limited by the underlying servlet environment.

In addition to JServ 1.1, the OC4J JSP container has been tested with Tomcat 3.1 (servlet 2.2) from the Apache Software Foundation, and JSWDK 1.0 (JavaServer Web Developer's Kit, servlet 2.1) from Sun Microsystems.

JSP Front-End Servlet and Configuration

The JSP container in OC4J uses the front-end servlet `oracle.jsp.runtimev2.JspServlet`. See "JSP Configuration in OC4J" on page 3-3.

For non-OC4J environments, including Apache JServ, use the old front-end servlet, `oracle.jsp.JspServlet`. See "Getting Started in a JServ Environment" on page B-2.

(The front-end servlet for the old Orion JSP container is `com.evermind.server.http.JSPServlet`, but its use is not recommended.)

Role of the Oracle HTTP Server and mod_oc4j

Oracle HTTP Server, powered by the Apache Web server, is included with Oracle9*i* Application Server as the HTTP entry point for Web applications. By default, it is the front-end for all OC4J processes—client requests go through Oracle HTTP Server first.

When the Oracle HTTP Server is used, dynamic content is delivered through various Apache *mod* components provided either by the Apache Software Foundation or by Oracle. Static content is typically delivered from the file system, which is more efficient in this case. An Apache mod is typically a module of C code, running in the Apache address space, that passes requests to a particular mod-specific processor. The mod software will have been written specifically for use with the particular processor.

Oracle9*i*AS supplies the `mod_oc4j` Apache mod, which is used for communication between the Oracle HTTP Server and OC4J. It routes requests from the Oracle HTTP Server to OC4J processes, and forwards responses from OC4J processes to Web clients.

Communication is through the Apache JServ protocol (AJP). AJP was chosen over HTTP because of a variety of AJP features allowing faster communication, including use of binary format and more efficient processing of message headers.

The following features are provided with `mod_oc4j`:

- load balancing capabilities across many back-end OC4J clusters

- stateless session routing of stateful servlets

  This is accomplished through enhanced use of cookies. Routing information is maintained in the cookie itself to ensure that stateful servlets are always routed to the same OC4J JVM.

- high availability

  A `mod_oc4j` module can restart an OC4J instance automatically, if necessary.

Notes:

- Oracle9*i*AS also includes `mod_jserv`, from Apache, for the JServ servlet environment. This feature is documented in Appendix B, "The Apache JServ Environment". Additional Apache mod components provided with Oracle9*i*AS are not relevant for JSP applications.
- It is possible to bypass the Oracle HTTP Server and access OC4J directly through its own Web listener, which may be convenient in getting a quick start for development or basic testing. For information about port configuration and default settings, see the *Oracle9iAS Containers for J2EE User's Guide*.

Oracle JDeveloper JSP Support

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

- integration of the OC4J JSP container to support the full application development cycle—editing, debugging, and running JSP pages
- debugging of deployed JSP pages
- an extensive set of data-enabled and Web-enabled JavaBeans, known as JDeveloper Web beans
- the JSP Element Wizard, which offers a convenient way to add predefined Web beans to a page
- support for incorporating custom JavaBeans
- a deployment option for JSP applications that rely on the JDeveloper Business Components for Java (BC4J)

See "Application Deployment with Oracle9i JDeveloper" on page 6-30 for more information about JSP deployment support.

For debugging, JDeveloper can set breakpoints within JSP page source and can follow calls from JSP pages into JavaBeans. This is much more convenient than manual debugging techniques, such as adding print statements within the JSP page to output state into the response stream (for viewing in your browser) or to the server log (through the `log()` method of the implicit `application` object).

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

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

For an overview of JSP tag libraries provided with JDeveloper, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Overview of Oracle Value-Added Features

OC4J value-added features for JSP pages can be grouped into three major categories:

- features implemented through custom tag libraries, custom JavaBeans, or custom classes that are generally portable to other JSP environments
- features that are Oracle-specific
- features supporting caching technologies

The rest of this section provides feature overviews in each of these areas.

Overview of Tag Libraries and Utilities Provided with OC4J

This section provides an overview of extended OC4J JSP features that are implemented through custom tag libraries and custom JavaBeans and classes that are generally portable to other JSP environments. These features are fully documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*. Here is a summary list:

- extended types implemented as JavaBeans that can have a specified scope
- `JspScopeListener` for event-handling
- integration with XML and XSL
- data-access JavaBeans
- data-access tag library
- the JSP Markup Language (JML) custom tag library, which reduces the level of Java proficiency required for JSP development
- Oracle Personalization tag library
- additional utility tags and JavaBeans for uploading files, downloading files, sending e-mail from within an application, using EJBs, and using miscellaneous utilities

Extended Type JavaBeans

JSP pages generally rely on core Java types in representing scalar values. However, neither of the following type categories is fully suitable for use in JSP pages:

- primitive types such as `int`, `float`, and `double`

  Values of these types cannot have a specified scope—they cannot be stored in a JSP scope object (for `page`, `request`, `session`, or `application` scope), because only objects can be stored in a scope object.

- wrapper classes in the standard `java.lang` package, such as `Integer`, `Float`, and `Double`

  Values of these types are objects, so they can theoretically be stored in a JSP scope object. However, you cannot declare them in a `jsp:useBean` action, because the wrapper classes do not follow the JavaBean model and do not provide a zero-argument constructor.

  Additionally, instances of the wrapper classes are immutable. To change a value, you must create a new instance and assign it appropriately.

To work around these limitations, OC4J provides the `JmlBoolean`, `JmlNumber`, `JmlFPNumber`, and `JmlString` JavaBean classes in package `oracle.jsp.jml` to wrap the most common Java types.

JspScopeListener for Event-Handling

OC4J provides the `JspScopeListener` interface for lifecycle management of Java objects of various scopes within a JSP application.

Standard servlet and JSP event-handling is provided through the `javax.servlet.http.HttpSessionBindingListener` interface, but this handles session-based events only. You can integrate the Oracle `JspScopeListener` with `HttpSessionBindingListener` to handle session-based events, as well as page-based, request-based, and application-based events.

Integration with XML and XSL

You can use JSP syntax to generate any text-based MIME type, not just HTML code. In particular, you can dynamically create XML output. When you use JSP pages to generate an XML document, however, you often want a stylesheet applied to the XML data before it is sent to the client. This is difficult in JavaServer Pages technology, because the standard output stream used for a JSP page is written directly back through the server.

OC4J provides special tags to specify that all or part of a JSP page should be transformed through an XSL stylesheet before it is output. Input can be from the tag body or from an XML DOM object, and output can be to an XML DOM object to the browser.

You can use these tags multiple times in a single JSP page if you want to specify different style sheets for different portions of the page. Note that these tags are portable to other JSP environments.

There is additional XML support as well:

- There is a utility tag to convert data from an input stream to an XML DOM object.
- There are several tags, for such things as caching and SQL operations, that now can take XML objects as input or send them as output.
- The JSP translator supports XML-alternative syntax as specified in the Sun Microsystems *JavaServer Pages Specification, Version 1.1*. For information, see "XML-Alternative Syntax" on page 5-23.

For information about XML-related tags, refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Custom Data-Access JavaBeans

OC4J supplies a set of custom JavaBeans for database access. The following beans are provided in the `oracle.jsp.dbutil` package:

- `ConnBean` opens a database connection. This bean also supports data sources and connection pooling.
- `ConnCacheBean` uses the Oracle connection caching implementation for database connections. (This requires JDBC 2.0.)
- `DBBean` executes a database query.
- `CursorBean` provides general DML support for queries; `UPDATE`, `INSERT`, and `DELETE` statements; and stored procedure calls.

Custom Data-Access Tag Library

OC4J provides a custom SQL tag library for database access. The following tags are provided:

- `dbOpen`—Open a database connection. This tag also supports data sources and connection pooling.

- `dbClose`—Close a database connection.
- `dbQuery`—Execute a query.
- `dbCloseQuery`—Close the cursor for a query.
- `dbNextRow`—Process the rows of a result set.
- `dbExecute`—Execute any SQL statement (DML or DDL).
- `dbSetParam`—Set a parameter to bind into a `dbQuery` or `dbExecute` tag.
- `dbSetCookie`—Set a cookie.

JSP Markup Language (JML) Custom Tag Library

Although the Sun Microsystems *JavaServer Pages Specification, Version 1.1* supports scripting languages other than Java, Java is the primary language used. Even though JavaServer Pages technology is designed to separate the dynamic/Java development effort from the static/HTML development effort, it is no doubt still a hindrance if the Web developer does not know any Java, especially in small development groups where no Java experts are available.

OC4J provides custom tags as an alternative—the JSP Markup Language (JML). The Oracle JML tag library provides an additional set of JSP tags so that you can script your JSP pages without using Java statements. JML provides tags for variable declarations, flow control, conditional branches, iterative loops, parameter settings, and calls to objects. The JML tag library also supports XML functionality, as noted previously.

The following example shows use of the `jml:for` tag, repeatedly printing "Hello World" in progressively smaller headings (H1, H2, H3, H4, H5):

```
<jml:for id="i" from="<%= 1 %>" to="<%= 5 %>" >
     <H<%=i%>>
            Hello World!
     </H<<%=i%>>
</jml:for>
```

Note: Oracle JSP implementations preceding the JSP 1.1 specification used an Oracle-specific compile-time implementation of the JML tag library. This implementation is still supported as an alternative to the standard runtime implementation.

Oracle9*i*AS Personalization Tag Library

Web site personalization is a mechanism to personalize recommendations to users of a site, based on behavioral and demographic data. Recommendations are made in real-time, during a user's Web session. User behavior is saved to a database repository for use in building models for predictions of future user behavior.

Oracle9*i*AS Personalization uses data mining algorithms in the Oracle database to choose the most relevant content available for a user. Recommendations are calculated by an Oracle9*i*AS Personalization recommendation engine, using typically large amounts of data regarding past and current user behavior. This is superior to other approaches that rely on "common sense" heuristics and require manual definition of rules in the system.

The Oracle9*i*AS Personalization tag library brings this functionality to a wide audience of JSP developers for use in HTML, XML, or JavaScript pages. The tag interface is layered on top of the lower level Java API of the recommendation engine.

JSP Utility Tags

OC4J provides utility tags to accomplish the following from within Web applications:

- sending e-mail messages
- uploading and downloading files
- using EJBs
- using miscellaneous utilities

For sending e-mail messages, you can use the `sendMail` tag or the `oracle.jsp.webutil.email.SendMailBean` JavaBean.

For uploading files, you can use the `httpUpload` tag or the `oracle.jsp.webutil.fileaccess.HttpUploadBean` JavaBean. For downloading, there is the `httpDownload` tag or the `HttpDownloadBean` JavaBean.

For using EJBs, there are tags to create a home instance, create an EJB instance, and iterate through a collection of EJBs.

There are also utility tags for displaying a date, displaying an amount of money in the appropriate currency, displaying a number, iterating through a collection, evaluating and including the tag body depending on whether the user belongs to a specified role, and displaying the last modification date of the current file.

Overview of Oracle-Specific Features

This section provides an overview of Oracle-specific programming extensions supported by the OC4J JSP container:

- support for SQLJ, a standard syntax for embedding SQL statements directly into Java code
- "global includes", a mechanism to automatically statically include a file or files in multiple pages
- Dynamic Monitoring Service support for performance measurements
- enhanced application framework and globalization support for servlet 2.0 environments

SQLJ Support

Dynamic server pages commonly include data extracted from databases. JSP developers typically rely on the standard Java Database Connectivity (JDBC) API or a custom set of database JavaBeans.

SQLJ is a standard syntax for embedding static SQL instructions directly in Java code, greatly simplifying database-access programming. The OC4J JSP container supports SQLJ programming in JSP scriptlets.

SQLJ statements are indicated by the `#sql` token. You can trigger the JSP translator to invoke the Oracle SQLJ translator by using the file name extension `.sqljsp` for the JSP source code file, or by specifying `language="sqlj"` in a `page` directive.

For more information, see "JSP Support for Oracle SQLJ" on page 5-19.

Global Includes

In Oracle9*i*AS 9.0.2, the OC4J JSP container introduces a feature called *global includes*. You can use this feature to specify one or more files to statically include into JSP pages in (or under) a specified directory, through virtual JSP `include` directives. During translation, the JSP container looks for a configuration file, `/WEB-INF/ojsp-global-include.xml`, that specifies the included files and the directories for the pages.

This enhancement is particularly useful in migrating applications that had used `globals.jsa` or `translate_params` functionality in previous Oracle JSP releases. For more information, see "Oracle JSP Global Includes" on page 6-9.

Support for Dynamic Monitoring System

The Dynamic Monitoring System (DMS) adds performance-monitoring features to a number of Oracle9*i*AS components, including OC4J. The goal of DMS is to provide information about runtime behavior, using 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.

The OC4J JSP container supports DMS features, calculating relevant statistics and providing information to DMS servlets such as the spy servlet and monitoring agent. Statistics include the following (using averages, maximums, and minimums, as applicable):

- processing time of HTTP request
- processing time of JSP service method
- number of JSP instances created or available
- number of active JSP instances

(Counts of JSP instances are applicable only for single-threaded situations, where `isThreadSafe` is set to `false`.)

Standard configuration for these servlets is in the OC4J `global-web-application.xml` configuration file. Use the Oracle Enterprise Manager to access DMS, for displaying DMS information and, as appropriate, altering DMS configuration.

Enhanced Servlet 2.0 Support

OC4J supports special features for the servlet 2.0 JServ environment. It is highly advisable to migrate to the servlet 2.3 OC4J environment as soon as practical, but in the meantime, be aware of the following:

- an enhanced application framework for servlet 2.0 environments

  See "JSP Application and Session Support for JServ" on page B-28.

- extended globalization support for servlet 2.0 environments

  See "Multibyte Parameter Encoding in JServ" on page B-21.

The referenced sections include migration information.

Overview of Tags and API for Caching Support

Faced with Web performance challenges, e-businesses must invest in more cost-effective technologies and services to improve the performance of their Internet sites. *Web caching*, the caching of both static and dynamic Web content, is a key technology in this area. Benefits of Web caching include performance, scalability, high availability, cost savings, and network traffic reduction.

OC4J provides the following support for Web caching technologies:

- the JESI tag set for Edge Side Includes (ESI), an XML-style markup language that allows dynamic content assembly away from the Web server

  The Oracle9*i*AS Web Cache provides an ESI engine.

- a tag set and servlet API for the Web Object Cache, an application-level cache that is embedded and maintained within a Java Web application

  The Web Object Cache uses the Oracle9*i* Application Server Java Object Cache as its default repository.

These features are documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

3

Getting Started

This chapter covers basic issues in your JSP environment, including key support files, key OC4J configuration files, configuration of the JSP container, and other initial considerations such as application root and classpath functionality.

Before getting started, it is assumed that you can do the following on your system:

- run Java
- run a Java compiler (typically the standard `javac`)
- run an HTTP servlet

The following topics are covered here:

Notes: JSP pages will run with any standard browser supporting HTTP 1.0 or higher. The JDK or other Java environment in the end-user's Web browser is irrelevant, because all the Java code in a JSP page is executed in the Web server.

Key Support Files Provided with OC4J

This section summarizes JAR and ZIP files that are used by the JSP container or JSP applications. These files are installed on your system and into your classpath with OC4J:

- `ojsp.jar`—classes for the JSP container
- `ojsputil.jar`—classes for tag libraries and utilities provided with OC4J
- `xmlparserv2.jar`—for XML parsing; required for the `web.xml` deployment descriptor and any tag library descriptor files and XML-related tag functionality
- `xsu12.jar` / `xsu111.jar`—for XML functionality on the client (for JDK 1.2.x or higher, or 1.1.x, respectively)
- `classes12.zip` / `classes111.zip`—for the Oracle JDBC drivers (for JDK 1.2.x or higher, or 1.1.x, respectively)
- `translator.zip`—for the Oracle SQLJ translator
- `runtime12.zip` / `runtime12ee.zip` / `runtime11.zip` / `runtime.zip` / `runtime-nonoracle.zip`—for the Oracle SQLJ runtime (respectively: for JDK 1.2.x or higher with Oracle9*i* JDBC, JDK 1.2.x or higher enterprise edition with Oracle9*i* JDBC, JDK 1.1.x with Oracle9*i* JDBC, any 1.1.x or higher JDK with any Oracle JDBC version, or any JDK environment with non-Oracle JDBC drivers)
- `jndi.jar`—for JNDI service for lookup of resources such as JDBC data sources and Enterprise JavaBeans
- `jta.jar`—for the Java Transaction API

There are also files relating to particular areas, such as particular tag libraries. These include the following:

- `mail.jar`—for e-mail functionality within applications (standard `javax.mail` package)
- `activation.jar`—Java activation files for e-mail functionality
- `cache.jar`—for the Oracle9*i* Application Server Java Object Cache (which can be used by the OC4J Web Object Cache as the back-end repository)

Note: The `.zip` files are also provided as `.jar` files.

JSP Configuration in OC4J

This section covers the following topics regarding configuration of the JSP environment:

- JSP Container Setup
- JSP Configuration Parameters

> **Note:** For non-OC4J environments, including JServ, use the old `oracle.jsp.JspServlet` front-end servlet instead of the `oracle.jsp.runtimev2.JspServlet` version. See "Getting Started in a JServ Environment" on page B-2.

JSP Container Setup

The JSP container is appropriately pre-configured in OC4J. The following settings appear in the OC4J `global-web-application.xml` file to map the name of the front-end JSP servlet, and to map the appropriate file name extensions for JSP pages:

```
<orion-web-app ... >
   ...
   <web-app>
       ...
      <servlet>
         <servlet-name>jsp</servlet-name>
         <servlet-class>oracle.jsp.runtimev2.JspServlet</servlet-class>
         ...
            init params
         ...
      </servlet>
      ...
      <servlet-mapping>
         <servlet-name>jsp</servlet-name>
         <url-pattern>/*.jsp</url-pattern>
      </servlet-mapping>
      <servlet-mapping>
          <servlet-name>jsp</servlet-name>
          <url-pattern>/*.JSP</url-pattern>
       </servlet-mapping>
       <servlet-mapping>
           <servlet-name>jsp</servlet-name>
           <url-pattern>/*.sqljsp</url-pattern>
```

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

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

See the *Oracle9iAS Containers for J2EE Servlet Developer's Guide* for more information about the `global-web-application.xml` file.

JSP Configuration Parameters

The JSP front-end servlet in OC4J, `oracle.jsp.runtimev2.JspServlet`, supports a number of configuration parameters to control JSP operation. This section describes those parameters, starting with a summary table and then providing more complete descriptions.

Configuration Parameter Summary Table

Table 3–1 summarizes the configuration parameters supported by `JspServlet`. For each parameter, the table notes any equivalent `ojspc` translation options for pages you are pre-translating, and whether the parameter is for runtime or compile-time use.

Notes: See "The ojspc Pre-Translation Utility" on page 6-13 for a description of the `ojspc` options.

Table 3–1 JSP Configuration Parameters, OC4J Environment

Parameter	Related ojspc Options	Description	Default	Runtime / Compile-Time
debug_mode	(n / a)	Set this boolean to `true` to print the stack trace when a runtime exception occurs.	true	runtime
emit_debuginfo	-debug	Set this boolean to `true` to generate a line map to the original `.jsp` file for debugging (for development).	false	compile-time
external_resource	-extres	Set this boolean to `true` to place all static content of the page into a separate Java resource file during translation.	false	compile-time
javaccmd	-noCompile	Use this if you want to specify a `javac` command line, or if you want to specify an alternative Java compiler, optionally with command-line settings (for development). If you specify an alternative compiler, it will be spawned in a separate JVM (`javac` runs in the same JVM). A `null` setting means use the JDK `javac` with default settings.	null	compile-time
main_mode	(n / a)	This determines whether classes are automatically reloaded or JSP pages are automatically recompiled, in case of changes. Possible settings are `justrun`, `reload`, and `recompile`.	recompile	runtime
old_include_from_top	-oldIncludeFromTop	Set this boolean to `true` for page locations in nested `include` directives to be relative to the top-level page, for backward compatibility with behavior prior to Oracle9*i*AS release 2.	false	compile-time
precompile_check	(n / a)	Set this boolean to `true` to check the HTTP request for a standard `jsp_precompile` setting.	false	runtime

Table 3–1 JSP Configuration Parameters, OC4J Environment (Cont.)

Parameter	Related ojspc Options	Description	Default	Runtime / Compile-Time
reduce_tag_code	-reduceTagCode	Set this boolean to `true` for further reduction in the size of generated code for custom tag usage.	false	compile-time
req_time_introspection	-reqTimeIntrospection	Set this boolean to `true` to enable request-time JavaBean introspection whenever compile-time introspection is not possible.	false	compile-time
sqljcmd	-S	Use this if you want to specify a SQLJ command line, or if you want to specify an alternative SQLJ translator, optionally with command-line settings (for development). If you specify an alternative translator, it will be spawned in a separate JVM. A `null` setting means use the Oracle SQLJ version provided with Oracle9*i*AS, with its default option settings.	null	compile-time
static_text_in_chars	-staticTextInChars	Set this boolean to `true` to instruct the JSP translator to generate static text in JSP pages as characters instead of bytes.	false	compile-time
tags_reuse_default	(n/a)	This specifies a default setting for JSP tag handler pooling (`true` to enable by default; `false` to disable by default). You can override this default setting for any particular JSP page.	true	runtime
xml_validate	-xmlValidate	This specifies whether XML validation is performed on the `web.xml` file and TLD files.	false	compile-time

Configuration Parameter Descriptions

This section describes the JSP configuration parameters for OC4J in more detail.

debug_mode (boolean; default: `true`)

Use the default `true` setting to print a stack trace whenever a runtime exception occurs. Set it to `false` to disable this feature.

Note: When `debug_mode` is `false` and a file is not found, the full path of the missing file is *not* displayed.

---

emit_debuginfo (boolean; default: `false`)

During development, set this flag to `true` to instruct the JSP translator to generate a line map to the original `.jsp` file for debugging. Otherwise, lines will be mapped to the generated page implementation class `.java` file.

Notes:

- Oracle9*i* JDeveloper enables `emit_debuginfo`.
- For pre-translating pages, the `ojspc -debug` option is equivalent.

---

external_resource (boolean; default: `false`)

Set this flag to true to instruct the JSP translator to place generated static content (the Java print commands that output static HTML code) into a Java resource file, instead of into the service method of the generated page implementation class.

The resource file name is based on the JSP page name, with the `.res` suffix. With Oracle9*i*AS 9.0.2, translation of `MyPage.jsp`, for example, would create `_MyPage.res` in addition to normal output. The exact implementation may change in future releases, however.

The translator places the resource file into the same directory as generated class files.

If there is a lot of static content in a page, this technique will speed translation and may speed execution of the page. For more information, see "Workarounds for Large Static Content in JSP Pages" on page 5-7.

Note: For pre-translating pages, the `ojspc -extres` option is equivalent.

---

javaccmd (compiler executable; default: `null`)

This parameter is useful in either of the following circumstances:

- if you want to set `javac` command-line options (although default settings are typically sufficient)
- if you want to use a compiler other than `javac` (optionally including command-line options)
- if you want to run the Java compiler in a separate process from the JSP container

Specifying an alternative compiler results in that executable being spawned as a separate process in a separate JVM, instead of within the same JVM as the JSP container. You can fully specify the path for the executable, or specify only the executable and let the JSP container look for it in the system path.

For example, set `javaccmd` to the value `javac -verbose` to run the compiler in verbose mode.

Notes:

- The specified Java compiler must be installed in the classpath and any front-end utility (if applicable) must be installed in the system path.
- For pre-translating pages, the `ojspc -noCompile` option allows similar functionality. It results in no compilation by `javac`, so you can compile the translated classes manually, using any desired compiler.

---

main_mode (mode switch for reloading or recompilation; default: `recompile`)

This is a flag to direct the mode of operation of the JSP container, particularly for automatic recompilation of JSP pages and reloading of Java classes that have changed.

Here are the supported settings:

- `justrun`—The runtime dispatcher will not perform any timestamp checking, so there is no recompilation of JSP pages or reloading of Java classes. This mode is the most efficient mode for a deployment environment, where code will not change.
- `reload`—The dispatcher will check if any classes have been modified since loading, including translated JSP pages, JavaBeans invoked from pages, and any other dependency classes.
- `recompile` (default)—The dispatcher will check the timestamp of the JSP page, retranslate it and reload it if has been modified since loading, and execute all `reload` functionality as well.

old_include_from_top (backward compatibility for `include`; default: `false`)

This is for backward compatibility with Oracle JSP versions prior to Oracle9*i*AS release 2, for functionality of `include` directives. If set to `true`, page locations in nested `include` directives are relative to the top-level page. If set to `false`, page locations are relative to the immediate parent page, as per the JSP 1.2 specification.

Note: For pre-translating pages, the `ojspc -oldIncludeFromTop` option is equivalent.

---

precompile_check (`jsp_precompile` checking; default: `false`)

Set this to `true` to check the HTTP request for a standard `jsp_precompile` setting. If `precompile_check` is `true` and the request enables `jsp_precompile`, then the JSP page will be pre-translated only, without execution. Setting `precompile_check` to `false` improves performance and ignores any `jsp_precompile` setting in the request.

For more information about `jsp_precompile`, see "Standard JSP Pre-Translation Without Execution" on page 6-32, and the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

reduce_tag_code (flag for size reduction of custom tag code; default: `false`)

The Oracle9*i*AS release 2 implementation reduces the size of generated code for custom tag usage, but setting `reduce_tag_code` to `true` results in even further size reduction. There may be performance consequences regarding tag handler reuse, however. See "Tag Handler Code Generation" on page 7-19.

> **Note:** For pre-translating pages, the `ojspc -reduceTagCode` option is equivalent.

req_time_introspection (flag for request-time introspection; default: `false`)

A `true` setting enables request-time JavaBean introspection whenever compile-time introspection is not possible. When compile-time introspection is possible and succeeds, this parameter is ignored and there is no request-time introspection.

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 request-time (runtime). In this case, if `req_time_introspection` is enabled, the JSP container will delay introspection until request-time. (See "Scripting Variables and Tag-Extra-Info Classes" on page 7-7 for information about use of `VariableInfo`.)

> **Note:** For pre-translating pages, the `ojspc -reqTimeIntrospection` option is equivalent.

sqljcmd (SQLJ translator executable and options; default: `null`)

This parameter is useful in any of the following circumstances:

- if you want to set one or more SQLJ command-line options
- if you want to use a different SQLJ translator, or at least a different version, than the one provided with OC4J
- if you want to run SQLJ in a separate process from the JSP container

Specifying a SQLJ translator executable results in its being spawned as a separate process in a separate JVM, instead of within the same JVM as the JSP container.

You can fully specify the path for the executable, or specify only the executable and let the JSP container look for it in the system path.

For example, to run SQLJ with online semantics checking as user `scott/tiger` and with the `-ser2class` option enabled, set `sqljcmd` to the following value:

```
sqlj -user=scott/tiger -ser2class=true
```

Appropriate SQLJ libraries must be in the classpath, and any front-end utility (such as `sqlj` in the example) must be in the system path. For Oracle SQLJ, the

`translator` ZIP or JAR file and the appropriate SQLJ runtime ZIP or JAR file must be in the classpath. See "Key Support Files Provided with OC4J" on page 3-2.

Notes: For pre-translating pages, the `ojspc -S` option provides related functionality.

---

static_text_in_chars (flag to generate static text as characters; default: `false`)

A `true` setting directs the JSP translator to generate static text in JSP pages as characters, instead of bytes. 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"); %>
```

Place this statement as close to the beginning of the page as possible.

The `false` default setting improves performance in outputting static text blocks.

Note: For pre-translating pages, the `ojspc -staticTextInChars` option is equivalent.

---

tags_reuse_default (default setting for tag handler pooling; default: `true`)

You can specify whether JSP tag handler instances are pooled by setting an attribute in the JSP page context. See "Disabling or Enabling Tag Handler Instance Pooling" on page 7-19 for information about this.

If there is no setting in the page context, then the behavior is determined by the `tags_reuse_default` setting, or is according to a setting of `true` (the "default default") if `tags_reuse_default` is not specified.

xml_validate (XML validation of `web.xml` and TLD files; default: `false`)

This flag specifies whether XML validation is performed on the application `web.xml` file and any tag library description (TLD) files. Because the Tomcat JSP reference implementation does not perform XML validation, `xml_validate` is `false` by default.

Note: For pre-translating pages, the `ojspc -xmlValidate` option is equivalent.

Setting JSP Configuration Parameters in OC4J

In the OC4J environment, you can set JSP configuration parameters in the `global-web-application.xml` file, inside the `servlet` tag for the JSP front-end servlet. In the portion of this file shown in "JSP Container Setup" on page 3-3, the settings would go where the *`init params`* place holder appears.

The following example lists `servlet` tag and subtag settings for the JSP front-end servlet. This sample enables the `precompile_check` flag, sets the `main_mode` flag to run without checking timestamps, and runs the Java compiler in verbose mode.

```
<servlet>
   <servlet-name>jsp</servlet-name>
   <servlet-class>oracle.jsp.runtimev2.JspServlet</servlet-class>
   <init-param>
      <param-name>precompile_check</param-name>
      <param-value>true</param-value>
   </init-param>
   <init-param>
      <param-name>main_mode</param-name>
      <param-value>justrun</param-value>
   </init-param>
   <init-param>
      <param-name>javaccmd</param-name>
      <param-value>javac -verbose</param-value>
   </init-param>
</servlet>
```

You can override any settings in the `global-web-application.xml` file with settings in the `web.xml` file for a particular application.

Key OC4J Configuration Files

Be aware of the following key configuration files in the OC4J environment.

Global files for all OC4J applications, in the OC4J `/j2ee/home/config` directory:

- `server.xml`—This has an overall `<application-server>` element, with an `<application>` subelement for each J2EE application. Each `<application>` subelement specifies the name of the application and the name and location of its EAR deployment file. The `<application-server>` element specifies the name of the general application source directory, where EAR files are placed for deployment and extracted, and the application deployment directory, where OC4J-specific configuration files are generated. Additionally, there is a `<web-site>` element for the default Web site, and you can add a `<web-site>` element for each additional Web site you want to have on the server.
- `default-web-site.xml`—This includes a `<web-app>` element for each Web application for the default Web site, mapping the application name to the "Web application name". The Web application name corresponds to the WAR deployment file name. Additional Web site XML files, as specified for additional Web sites in the `server.xml` file, have the same functionality.
- `global-web-application.xml`—This is a global configuration file for OC4J Web applications. It establishes default configurations and includes setup and configuration of the JSP front-end servlet, `JspServlet`.
- `application.xml`—This is another parent configuration file for OC4J applications.
- `data-sources.xml`—This specifies data sources for database connections.

In addition to the global `application.xml` file, there is a standard `application.xml` file, and optionally an `orion-application.xml` file, for each application. These files are in the application EAR file.

Also, in an application WAR file, which is inside the application EAR file, there is a standard `web.xml` file and optionally an `orion-web.xml` file. These are for application-specific and deployment-specific configuration settings, overriding `global-web-application.xml` settings or providing additional settings as appropriate. The `global-web-application.xml` and `orion-web.xml` files support the same elements, which is a superset of those supported by the `web.xml` file.

If the `orion-application.xml` and `orion-web.xml` files are not present in the archive files, they will be generated during initial deployment, according to settings in the `global-web-application.xml` file.

For information about the use of these files, see the *Oracle9iAS Containers for J2EE User's Guide* and the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*. For additional information, see "Overview of EAR/WAR Deployment" on page 6-28.

Some Initial Considerations

This section discusses some initial considerations you should be aware of before running or deploying JSP pages:

Application Root Functionality

The servlet 2.2 and 2.3 specifications provide for each Web application to have its own servlet context. Each servlet context is associated with a directory path in the server file system, which is the base path for modules of the Web application. This is the *application root*. Each Web application has its own application root. For a Web application in a servlet 2.2 or 2.3 environment, servlets, JSP pages, and static files such as HTML files are all based out of this application root. (By contrast, in servlet 2.0 environments the application root for servlets and JSP pages is distinct from the doc root for static files.)

Note that a servlet URL has the following general form:

`http://host[:port]/contextpath/servletpath`

When a servlet context is created, a mapping is specified between the application root and the *context path* portion of a URL. The *servlet path* is defined in the application `web.xml` file. The `<servlet>` element within `web.xml` associates a servlet class with a servlet name. The `<servlet-mapping>` element within `web.xml` associates a URL pattern with a named servlet. When a servlet is executed, the servlet container will compare a specified URL pattern with known servlet paths, and pick the servlet path that matches. See the *Oracle9iAS Containers for J2EE Servlet Developer's Guide* for more information.

For example, consider an application with the application root `/home/dir/mybankapp/mybankwebapp`, which is mapped to the context path `/mybank`. Further assume the application includes a servlet whose servlet path is `loginservlet`. You can invoke this servlet as follows:

`http://host[:port]/mybank/loginservlet`

The application root directory name itself is not visible to the end-user.

To continue this example for an HTML page in this application, the following URL points to the file `/home/dir/mybankapp/mybankwebapp/dir1/abc.html`:

`http://host[:port]/mybank/dir1/abc.html`

For each servlet environment there is also a default servlet context. For this context, the context path is simply "/", which is mapped to the default servlet context application root. For example, assume the application root for the default context is `/home/dir/defaultapp/defaultwebapp`, and a servlet with the servlet path `myservlet` uses the default context. Its URL would be as follows:

```
http://host[:port]/myservlet
```

The default context is also used if there is no match for the context path specified in a URL.

Continuing this example for an HTML file, the following URL points to the file `/home/dir/defaultapp/defaultwebapp/dir2/def.html`:

```
http://host[:port]/dir2/def.html
```

Classpath Functionality

The JSP container uses standard locations on the Web server to look for translated JSP pages, as well as `.class` files and `.jar` files for any required classes (such as JavaBeans). The container will find files in these locations without any Web server classpath configuration, and has the ability to automatically reload classes in these locations, depending on configuration settings.

The locations for dependency classes are as follows and are relative to the application root:

```
/WEB-INF/classes/...
/WEB-INF/lib
```

The location for JSP page implementation classes (translated pages) is as follows:

```
.../_pages/...
```

The `/WEB-INF/classes` directory is for individual Java `.class` files. You should store these classes in subdirectories under the `classes` directory, according to Java package naming conventions. For example, consider a JavaBean called `LottoBean` whose code defines it to be in the `oracle.jsp.sample.lottery` package. The JSP container will look for `LottoBean.class` in the following location relative to the application root:

```
/WEB-INF/classes/oracle/jsp/sample/lottery/LottoBean.class
```

The `lib` directory is for `.jar` files. Because Java package structure is specified in the `.jar` file structure, the `.jar` files are all directly in the `lib` directory (not in subdirectories). As an example, `LottoBean.class` might be stored in `lottery.jar`, located as follows relative to the application root:

```
/WEB-INF/lib/lottery.jar
```

The `_pages` directory is under the following directory in OC4J:

```
/j2ee/home/application-deployments/app-name/web-app-name/temp
```

The *`app-name`* is determined through an `<application>` element in the OC4J `server.xml` file; the *`web-app-name`*, which corresponds to the WAR file name, is mapped to the *`app-name`* through a `<web-app>` element in the OC4J `default-web-site.xml` file. See the *Oracle9iAS Containers for J2EE User's Guide* and the *Oracle9iAS Containers for J2EE Servlet Developer's Guide* for more information.

Generated page implementation classes for translated JSP pages are placed in subdirectories under the `_pages` directory according to the locations of the original `.jsp` files. See "Generated Files and Locations" on page 6-6 for information.

Important: Implementation details, such as the location of the `_pages` directory, are subject to change in future releases.

4

Basic Programming Issues

This chapter discusses basic programming issues for JSP pages, followed by a JSP "starter sample" for data access.

The following topics are included:

- JSP-Servlet Interaction
- JSP Resource Management
- JSP Runtime Error Processing
- JSP Starter Sample for Data Access

JSP-Servlet Interaction

Although coding JSP pages is convenient in many ways, some situations call for servlets. One example is when you are outputting binary data, as discussed in "Reasons to Avoid Binary Data in JSP Pages" on page 5-13.

Therefore, it is sometimes necessary to go back and forth between servlets and JSP pages in an application. This section discusses how to accomplish this, covering the following topics:

- Invoking a Servlet from a JSP Page
- Passing Data to a Servlet Invoked from a JSP Page
- Invoking a JSP Page from a Servlet
- Passing Data Between a JSP Page and a Servlet
- JSP-Servlet Interaction Samples

Important: This discussion assumes a servlet 2.2 or higher environment. Appropriate reference is made to other sections of this document for related considerations for Apache JServ and other servlet 2.0 environments.

Invoking a Servlet from a JSP Page

As when invoking one JSP page from another, you can invoke a servlet from a JSP page through the `jsp:include` and `jsp:forward` action tags. (See "JSP Actions and the <jsp: > Tag Set" on page 1-15.) Following is an example:

```
<jsp:include page="/servlet/MyServlet" flush="true" />
```

When this statement is encountered during page execution, the page buffer is output to the browser and the servlet is executed. When the servlet has finished executing, control is transferred back to the JSP page and the page continues executing. This is the same functionality as for `jsp:include` actions from one JSP page to another.

And as with `jsp:forward` actions from one JSP page to another, the following statement would clear the page buffer, terminate the execution of the JSP page, and execute the servlet:

```
<jsp:forward page="/servlet/MyServlet" />
```

Important: You cannot include or forward to a servlet in JServ or other servlet 2.0 environments; you would have to write a JSP wrapper page instead. For information, see "Dynamic Includes and Forwards in JServ" on page B-17.

Passing Data to a Servlet Invoked from a JSP Page

When dynamically including or forwarding to a servlet from a JSP page, you can use a `jsp:param` tag to pass data to the servlet (the same as when including or forwarding to another JSP page).

You can use a `jsp:param` tag within a `jsp:include` or `jsp:forward` tag. Consider the following example:

```
<jsp:include page="/servlet/MyServlet" flush="true" >
   <jsp:param name="username" value="Smith" />
   <jsp:param name="userempno" value="9876" />
</jsp:include>
```

For more information about the `jsp:param` tag, see "JSP Actions and the <jsp: > Tag Set" on page 1-15.

Alternatively, you can pass data between a JSP page and a servlet through an appropriately scoped JavaBean or through attributes of the HTTP request object. Using attributes of the request object is discussed later, in "Passing Data Between a JSP Page and a Servlet" on page 4-4.

Invoking a JSP Page from a Servlet

You can invoke a JSP page from a servlet through functionality of the standard `javax.servlet.RequestDispatcher` interface. Complete the following steps in your code to use this mechanism:

1. Get a servlet context instance from the servlet instance:

   ```
   ServletContext sc = this.getServletContext();
   ```

2. Get a request dispatcher from the servlet context instance, specifying the page-relative or application-relative path of the target JSP page as input to the `getRequestDispatcher()` method:

   ```
   RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
   ```

Prior to or during this step, you can optionally make data available to the JSP page through attributes of the HTTP request object. See "Passing Data Between a JSP Page and a Servlet" below for information.

3. Invoke the `include()` or `forward()` method of the request dispatcher, specifying the HTTP request and response objects as arguments. For example:

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

or:

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

The functionality of these methods is similar to that of `jsp:include` and `jsp:forward` tags. The `include()` method only temporarily transfers control; execution returns to the invoking servlet afterward.

Note that the `forward()` method clears the output buffer.

Notes:

- The request and response objects would have been obtained earlier, using standard servlet functionality such as the `doGet()` method specified in the `javax.servlet.http.HttpServlet` class.
- This functionality was introduced in the servlet 2.1 specification.

Passing Data Between a JSP Page and a Servlet

The preceding section, "Invoking a JSP Page from a Servlet", notes that when you invoke a JSP page from a servlet through the request dispatcher, you can optionally pass data through the HTTP request object.

You can accomplish this using either of the following approaches:

- You can append a query string to the URL when you obtain the request dispatcher, using "?" syntax with *name=value* pairs. For example:

```
RequestDispatcher rd =
       sc.getRequestDispatcher("/jsp/mypage.jsp?username=Smith");
```

In the target JSP page (or servlet), you can use the `getParameter()` method of the implicit `request` object to obtain the value of a parameter set in this way.

- You can use the `setAttribute()` method of the HTTP request object. For example:

  ```
  request.setAttribute("username", "Smith");
  RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
  ```

  In the target JSP page (or servlet), you can use the `getAttribute()` method of the implicit `request` object to obtain the value of a parameter set in this way.

  Notes: You can use the mechanisms discussed in this section instead of the `jsp:param` tag to pass data from a JSP page to a servlet.

JSP-Servlet Interaction Samples

This section provides a JSP page and a servlet that use functionality described in the preceding sections. The JSP page `Jsp2Servlet.jsp` includes the servlet `MyServlet`, which includes another JSP page, `welcome.jsp`.

Code for Jsp2Servlet.jsp

```
<HTML>
<HEAD> <TITLE> JSP Calling Servlet Demo </TITLE> </HEAD>
<BODY>
<!-- Forward processing to a servlet -->
<% request.setAttribute("empid", "1234"); %>
<jsp:include page="/servlet/MyServlet?user=Smith" flush="true"/>

</BODY>
</HTML>
```

Code for MyServlet.java

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

public class MyServlet extends HttpServlet {
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
      throws IOException, ServletException {
```

```
      PrintWriter out= response.getWriter();
      out.println("<B><BR>User:" + request.getParameter("user"));
      out.println
          (", Employee number:" + request.getAttribute("empid") + "</B>");
      this.getServletContext().getRequestDispatcher("/jsp/welcome.jsp").
        include(request, response);
    }
}
```

Code for welcome.jsp

```
<HTML>
<HEAD> <TITLE> The Welcome JSP  </TITLE> </HEAD>
<BODY>

<H3> Welcome! </H3>
<P><B> Today is <%= new java.util.Date() %>.  Have a nice day! </B></P>
</BODY>
</HTML>
```

JSP Resource Management

This section discusses standard features and Oracle value-added features for resource management:

- Standard Session Resource Management—HttpSessionBindingListener (servlet 2.2 or higher environments)
- Overview of Oracle Value-Added Features for Resource Management (`JspScopeListener` for servlet 2.3 environments, `globals.jsa` for servlet 2.0 environments)

Standard Session Resource Management—HttpSessionBindingListener

A JSP page must appropriately manage resources acquired during its execution, such as JDBC connection, statement, and result set objects. The standard `javax.servlet.http` package provides the `HttpSessionBindingListener` interface and `HttpSessionBindingEvent` class to manage session-scope resources. Through this mechanism, a session-scope query bean could, for example, acquire a database cursor when the bean is instantiated and close it when the HTTP session is terminated. (The example in "JSP Starter Sample for Data Access" on page 4-16 opens and closes the connection for each query, which adds overhead.)

This section describes use of the `HttpSessionBindingListener` `valueBound()` and `valueUnbound()` methods.

Note: The bean instance must register itself in the event notification list of the HTTP session object, but the `jsp:useBean` statement takes care of this automatically.

The valueBound() and valueUnbound() Methods

An object that implements the `HttpSessionBindingListener` interface can implement a `valueBound()` method and a `valueUnbound()` method, each of which takes an `HttpSessionBindingEvent` instance as input. These methods are called by the servlet container—the `valueBound()` method when the object is stored in the session; the `valueUnbound()` method when the object is removed from the session or when the session times-out or becomes invalid. Usually, a developer will use `valueUnbound()` to release resources held by the object (in the example below, to release the database connection).

"JDBCQueryBean JavaBean Code" below provides a sample JavaBean that implements `HttpSessionBindingListener` and a sample JSP page that calls the bean.

JDBCQueryBean JavaBean Code

Following is the sample code for `JDBCQueryBean`, a JavaBean that implements the `HttpSessionBindingListener` interface. (It uses the JDBC OCI driver for its database connection; use an appropriate JDBC driver and connection string if you want to run this example yourself.)

`JDBCQueryBean` gets a search condition through the HTML request (as described in "The UseJDBCQueryBean JSP Page" on page 4-10), executes a dynamic query based on the search condition, and outputs the result.

This class also implements a `valueUnbound()` method (as specified in the `HttpSessionBindingListener` interface) that results in the database connection being closed at the end of the session.

```
package mybeans;

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

public class JDBCQueryBean implements HttpSessionBindingListener
{
  String searchCond = "";
  String result = null;

  public void JDBCQueryBean() {
  }

  public synchronized String getResult() {
    if (result != null) return result;
    else return runQuery();
  }

  public synchronized void setSearchCond(String cond) {
    result = null;
    this.searchCond = cond;
  }

  private Connection conn = null;
```

```
private String runQuery() {
  StringBuffer sb = new StringBuffer();
  Statement stmt = null;
  ResultSet rset = null;
  try {
    if (conn == null) {
      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
      conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                         "scott", "tiger");

    }

    stmt = conn.createStatement();
    rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
          (searchCond.equals("") ? "" : "WHERE " + searchCond ));
    result = formatResult(rset);
    return result;

  } catch (SQLException e) {
    return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
  }
  finally {
    try {
      if (rset != null) rset.close();
      if (stmt != null) stmt.close();
    }
    catch (SQLException ignored) {}
  }
}

private String formatResult(ResultSet rset) throws SQLException  {
  StringBuffer sb = new StringBuffer();
  if (!rset.next())
    sb.append("<P> No matching rows.<P>\n");
  else {
    sb.append("<UL><B>");
    do {  sb.append("<LI>" + rset.getString(1) +
          " earns $ " + rset.getInt(2) + "</LI>\n");
    } while (rset.next());
    sb.append("</B></UL>");
  }
  return sb.toString();
}
```

```
  public void valueBound(HttpSessionBindingEvent event) {
    // do nothing -- the session-scope bean is already bound
  }

  public synchronized void valueUnbound(HttpSessionBindingEvent event) {
    try {
      if (conn != null) conn.close();
    }
    catch (SQLException ignored) {}
  }
}
```

Note: The preceding code serves as a sample only. This is not necessarily an advisable way to handle database connection pooling in a large-scale Web application.

The UseJDBCQueryBean JSP Page

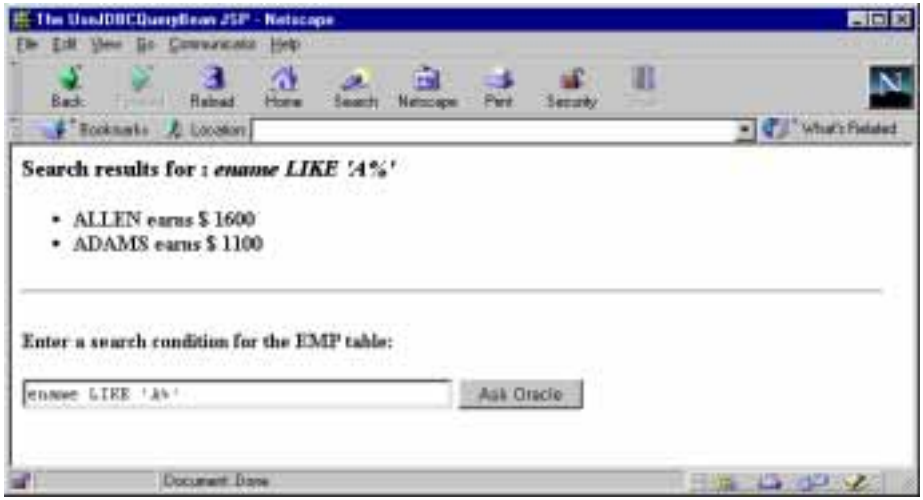
The following JSP page uses the `JDBCQueryBean` JavaBean defined in "JDBCQueryBean JavaBean Code" above, invoking the bean with `session` scope. It uses `JDBCQueryBean` to display employee names that match a search condition entered by the user.

`JDBCQueryBean` gets the search condition through the `jsp:setProperty` tag in this JSP page, which sets the `searchCond` property of the bean according to the value of the `searchCond` request parameter input by the user through the HTML form. (The HTML `INPUT` tag specifies that the search condition entered in the form be named `searchCond`.)

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session" />
<jsp:setProperty name="queryBean" property="searchCond" />

<HTML>
<HEAD> <TITLE> The UseJDBCQueryBean JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String searchCondition = request.getParameter("searchCond");
   if (searchCondition != null) { %>
      <H3> Search results for : <I> <%= searchCondition %> </I> </H3>
      <%= queryBean.getResult() %>
      <HR><BR>
<% }  %>
```

```
<B>Enter a search condition for the EMP table:</B>

<FORM METHOD="get">
<INPUT TYPE="text" NAME="searchCond" VALUE="ename LIKE 'A%' " SIZE="40">
<INPUT TYPE="submit" VALUE="Ask Oracle">
</FORM>

</BODY>
</HTML>
```

Following is sample input and output for this page:



Advantages of HttpSessionBindingListener

In the preceding example, an alternative to the `HttpSessionBindingListener` mechanism would be to close the connection in a `finalize()` method in the JavaBean. The `finalize()` method would be called when the bean is garbage-collected after the session is closed. The `HttpSessionBindingListener` interface, however, has more predictable behavior than a `finalize()` method. Garbage collection frequency depends on the memory consumption pattern of the application. By contrast, the `valueUnbound()` method of the `HttpSessionBindingListener` interface is called reliably at session shutdown.

Overview of Oracle Value-Added Features for Resource Management

OC4J JSP provides the following features for managing application and session resources as well as page and request resources:

- `JspScopeListener` interface—for managing application-scope, session-scope, page-scope, or request-scope resources in a servlet 2.3 environment such as OC4J

  This mechanism adheres to servlet and JSP standards in supporting objects of `page`, `request`, `session`, or `application` scope. To create a class that supports session scope as well as other scopes, you can integrate `JspScopeListener` with `HttpSessionBindingListener` by having the class implement both interfaces. For `page` scope in OC4J or JServ environments, you also have the option of using an Oracle-specific runtime implementation.

  For information about configuration and how to integrate with `HttpSessionBindingListener`, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

- `globals.jsa` file—for start and end events for application-scope and session-scope objects in a servlet 2.0 environment such as JServ

  See "The globals.jsa Event-Handlers" on page B-32 for information.

JSP Runtime Error Processing

While a JSP page is executing and processing client requests, runtime errors can occur either inside the page or outside the page (such as in a called JavaBean). This section describes the JSP error processing mechanism and provides an elementary example.

Using JSP Error Pages

Any runtime error encountered during execution of a JSP page is handled using the standard Java exception mechanism in one of two ways:

- You can catch and handle exceptions in a Java scriptlet within the JSP page itself, using standard Java exception-handling code.
- Exceptions you do not catch in the JSP page will result in forwarding of the request and uncaught exception to an error page. This is the preferred way to handle JSP errors.

You can specify the URL of an error page by setting the `errorPage` parameter in a `page` directive in the originating JSP page. (For an overview of JSP directives, including the `page` directive, see "Directives" on page 1-7.)

In a servlet 2.2 or higher environment, you can also specify a default error page in the `web.xml` deployment descriptor with instructions such as the following:

```
<error-page>
   <error-code>404</error-code>
   <location>/error404.html</location>
</error-page>
```

See the Sun Microsystems *Java Servlet Specification, Version 2.2* (or higher) for more information about default error pages.

An error page must have a `page` directive setting the `isErrorPage` parameter to `true`.

The exception object describing the error is a `java.lang.Exception` instance that is accessible in the error page through the implicit `exception` object.

Only an error page can access the implicit `exception` object. (For information about JSP implicit objects, including the `exception` object, see "Implicit Objects" on page 1-12.)

See "JSP Error Page Example" below for an example of error page usage.

Note: The JSP 1.1 specification is ambiguous regarding exception types that can be handled through the JSP mechanism.

In the current Oracle JSP implementation, a page implementation class generated by the translator can handle an instance of the `java.lang.Exception` class or a subclass, but cannot handle an instance of the `java.lang.Throwable` class or any subclass other than `Exception`. A `Throwable` instance will be thrown by the JSP container to the servlet container.

The JSP 1.2 specification will address this ambiguity.

JSP Error Page Example

The following example, `nullpointer.jsp`, generates an error and uses an error page, `myerror.jsp`, to output contents of the implicit `exception` object.

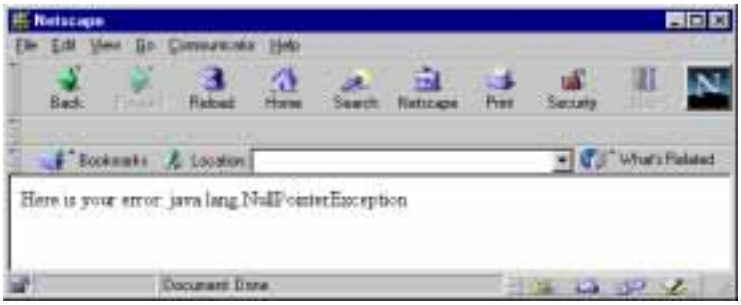
Code for nullpointer.jsp

```
<HTML>
<BODY>
<%@ page errorPage="myerror.jsp" %>
Null pointer is generated below:
<%
   String s=null;
   s.length();
%>
</BODY>
</HTML>
```

Code for myerror.jsp

```
<HTML>
<BODY>
<%@ page isErrorPage="true" %>
Here is your error:
<%= exception %>
</BODY>
</HTML>
```

This example results in the following output:



Note: The line "Null pointer is generated below:" in `nullpointer.jsp` is not output when processing is forwarded to the error page. This shows the difference between JSP "include" and "forward" functionality—with a "forward", the output from the "forward-to" page *replaces* the output from the "forward-from" page.

JSP Starter Sample for Data Access

Chapter 1, "General JSP Overview", provides a couple of elementary JSP examples; however, if you are using OC4J, you presumably want to access a database. This section offers a more interesting sample that uses standard JDBC code in a JSP page to perform a query.

Introduction to JSP Support for Data Access

Because the JDBC API is simply a set of Java interfaces, JavaServer Pages technology directly supports its use within JSP scriptlets.

Oracle JDBC provides several driver alternatives: 1) the JDBC OCI driver for use with an Oracle client installation; 2) a 100%-Java JDBC Thin driver that can be used in essentially any client situation (including applets); 3) a JDBC server-side Thin driver to access one Oracle database from within another Oracle database; and 4) a JDBC server-side internal driver to access the database within which the Java code is running (such as from a Java stored procedure or Enterprise JavaBean). For more information about Oracle JDBC, see the *Oracle9i JDBC Developer's Guide and Reference*.

The OC4J JSP container also supports EJB calls, as well as SQLJ (embedded SQL in Java) for static SQL operations. These and other data-access considerations are covered in "JSP Data-Access Considerations and Features" on page 5-14.

There are also custom JavaBeans and custom SQL tags for data access, documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Data Access Sample

The following example creates a query dynamically from search conditions the user enters through an HTML form (typed into a box, and entered with an `Ask Oracle` button). To perform the specified query, it uses JDBC code in a method called `runQuery()` that is defined in a JSP declaration. It also defines a method, `formatResult()`, within the JSP declaration to produce the output. The `runQuery()` method uses the `scott` schema with password `tiger`. (JDBC is used because SQLJ is primarily for static SQL, although Oracle SQLJ adds extensions for dynamic SQL.)

The HTML `INPUT` tag specifies that the string entered in the form be named `cond`. Therefore, `cond` is also the input parameter to the `getParameter()` method of the implicit `request` object for this HTTP request, and the input parameter to the `runQuery()` method (which puts the `cond` string into the `WHERE` clause of the query).

Notes:

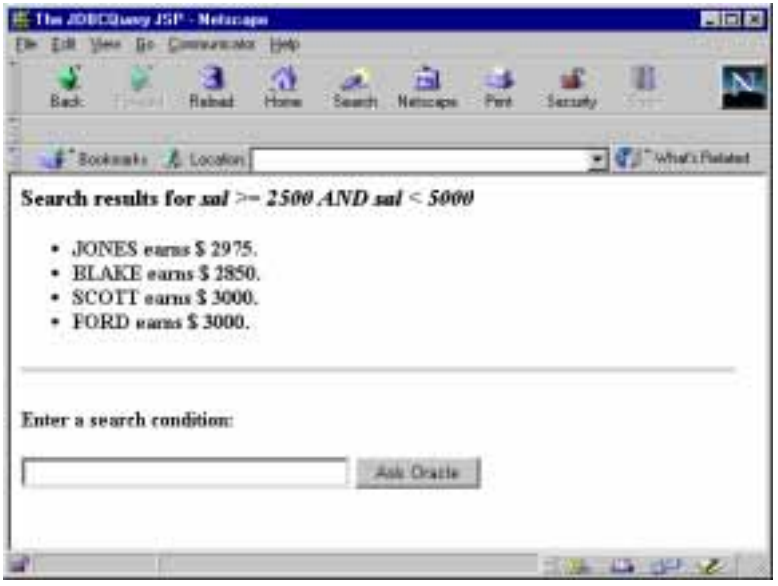
- Another approach to this example would be to define the `runQuery()` method in `<%...%>` scriptlet syntax instead of `<%!...%>` declaration syntax.
- This example uses the JDBC OCI driver, which requires an Oracle client installation. If you want to run this sample, use an appropriate JDBC driver and connection string.

```
<%@ page language="java" import="java.sql.*" %>
<HTML>
<HEAD> <TITLE> The JDBCQuery JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">
<% String searchCondition = request.getParameter("cond");
   if (searchCondition != null) { %>
      <H3> Search results for  <I> <%= searchCondition %> </I> </H3>
      <B> <%= runQuery(searchCondition) %> </B> <HR><BR>
<% }  %>
<B>Enter a search condition:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="cond" SIZE=30>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%-- Declare and define the runQuery() method. --%>
<%! private String runQuery(String cond) throws SQLException {
     Connection conn = null;
     Statement stmt = null;
     ResultSet rset = null;
     try {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                           "scott", "tiger");
        stmt = conn.createStatement();
        // dynamic query
        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                          (cond.equals("") ? "" : "WHERE " + cond ));
       return (formatResult(rset));
     } catch (SQLException e) {
         return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
     } finally {
```

```
        if (rset!= null) rset.close();
        if (stmt!= null) stmt.close();
        if (conn!= null) conn.close();
     }
  }
  private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
      sb.append("<P> No matching rows.<P>\n");
    else {  sb.append("<UL>");
            do {  sb.append("<LI>" + rset.getString(1) +
                            " earns $ " + rset.getInt(2) + ".</LI>\n");
            } while (rset.next());
           sb.append("</UL>");
    }
    return sb.toString();
  }
%>
```

The graphic below illustrates sample output for the following input:

```
sal >= 2500 AND sal < 5000
```



5

Key Considerations

This chapter discusses important programming and runtime considerations in developing and executing JSP applications. The following topics are covered:

- General JSP Programming Strategies, Tips, and Traps
- JSP Data-Access Considerations and Features
- JSP Runtime Considerations and Optimization

General JSP Programming Strategies, Tips, and Traps

This section discusses issues you should consider when programming JSP pages, regardless of the particular target environment. The following assortment of topics are covered:

- JavaBeans Versus Scriptlets
- Static Includes Versus Dynamic Includes
- When to Consider Creating and Using JSP Tag Libraries
- Use of a Central Checker Page
- Workarounds for Large Static Content in JSP Pages
- Method Variable Declarations Versus Member Variable Declarations
- Page Directive Characteristics
- JSP Preservation of White Space and Use with Binary Data

> **Note:** In addition to being aware of what is discussed in this section, you should be aware of JSP translation and deployment issues and behavior. See Chapter 6, "JSP Translation and Deployment".

JavaBeans Versus Scriptlets

The section "Separation of Business Logic from Page Presentation—Calling JavaBeans" on page 1-5 describes a key advantage of JavaServer Pages technology: Java code containing the business logic and determining the dynamic content can be separated from the HTML code containing the request processing, presentation logic, and static content. This separation allows HTML experts to focus on presentation, while Java experts focus on business logic in JavaBeans that are called from the JSP page.

A typical JSP page will have only brief snippets of Java code, usually for Java functionality for request processing or presentation. The sample page in "JSP Starter Sample for Data Access" on page 4-16, although illustrative, is probably not an ideal design. Data access, such as in the `runQuery()` method in the sample, is usually more appropriate in a JavaBean. However, the `formatResult()` method in the sample, which formats the output, is more appropriate for the JSP page itself.

Static Includes Versus Dynamic Includes

The `include` directive, described in "Directives" on page 1-7, makes a copy of the included page and copies it into a JSP page (the "including page") during translation. This is known as a *static include* (or *translate-time include*) and uses the following syntax:

```
<%@ include file="/jsp/userinfopage.jsp" %>
```

The `jsp:include` tag, described in "JSP Actions and the <jsp: > Tag Set" on page 1-15, dynamically includes output from the included page within the output of the including page, during runtime. This is known as a *dynamic include* (or *runtime include*) and uses the following syntax:

```
<jsp:include page="/jsp/userinfopage.jsp" flush="true" />
```

For those familiar with C syntax, a static include is comparable to a `#include` statement. A dynamic include is similar to a function call. They are both useful, but serve different purposes.

Note: You can use static includes and dynamic includes only between pages in the same servlet context.

Logistics of Static Includes

A static include increases the size of the generated code for the including JSP page, as though the text of the included page is physically copied into the including page during translation (at the point of the `include` directive). If a page is included multiple times within an including page, multiple copies are made.

A JSP page that is statically included does not need to stand as an independent, translatable entity. It simply consists of text that will be copied into the including page. The including page, with the included text copied in, must then be translatable. And, in fact, the including page does not have to be translatable prior to having the included page copied into it. A sequence of statically included pages can be fragments unable to stand on their own.

Logistics of Dynamic Includes

A dynamic include does *not* significantly increase the size of the generated code for the including page, although method calls, such as to the request dispatcher, will be added. The dynamic include results in runtime processing being switched from the

including page to the included page, as opposed to the text of the included page being physically copied into the including page.

A dynamic include *does* increase processing overhead, with the necessity of the additional call to the request dispatcher.

A page that is dynamically included must be an independent entity, able to be translated and executed on its own. Likewise, the including page must be independent as well, able to be translated and executed without the dynamic include.

Advantages, Disadvantages, and Typical Uses

Static includes affect page size; dynamic includes affect processing overhead. Static includes avoid the overhead of the request dispatcher that a dynamic include necessitates, but may be problematic where large files are involved. (The service method of the generated page implementation class has a 64 KB size limit—see "Workarounds for Large Static Content in JSP Pages" on page 5-7.)

Overuse of static includes can also make debugging your JSP pages difficult, making it harder to trace program execution. Avoid subtle interdependencies between your statically included pages.

Static includes are typically used to include small files whose content is used repeatedly in multiple JSP pages. For example:

- Statically include a logo or copyright message at the top or bottom of each page in your application.
- Statically include a page with declarations or directives (such as imports of Java classes) that are required in multiple pages.
- Statically include a central "status checker" page from each page of your application. (See "Use of a Central Checker Page" on page 5-6.)

Dynamic includes are useful for modular programming. You may have a page that sometimes executes on its own but sometimes is used to generate some of the output of other pages. Dynamically included pages can be reused in multiple including pages without increasing the size of the including pages.

Note: OC4J offers "global includes" as a convenient way to statically include a file into multiple pages. See "Oracle JSP Global Includes" on page 6-9.

When to Consider Creating and Using JSP Tag Libraries

Some situations dictate that the development team consider creating and using custom tags. In particular, consider the following situations:

- JSP pages would otherwise have to include a significant amount of Java logic regarding presentation and format of output.
- You want to provide convenient JSP programming access to functionality that would otherwise require the use of a Java API.
- Special manipulation or redirection of JSP output is required.

Replacing Java Syntax

Because one cannot count on JSP developers being experienced in Java programming, they may not be ideal candidates for coding Java logic in the page—logic that dictates presentation and format of the JSP output, for example.

This is a situation where JSP tag libraries might be helpful. If many of your JSP pages will require such logic in generating their output, a tag library to replace Java logic would be a great convenience for JSP developers.

An example of this is the JML tag library provided with OC4J. This library, documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*, includes tags that support logic equivalent to Java loops and conditionals.

Providing Convenient JSP Programming Access to API Features

Instead of having Web application programmers rely on Java APIs from servlets or JSP scriptlets to use product functionality or extensions, you can provide a tag library. A tag library can make the programmer's task much more convenient.

For example, tags as well as JavaBeans are provided with OC4J for sendmail and file access functionality. There is also a tag library as well as a Java API provided with the OC4J Web Object Cache. Similarly, while Oracle9*i*AS Personalization provides a Java API, OC4J also provides a tag library that you can use instead if you want to program a personalization application.

Manipulating or Redirecting JSP Output

Another common situation for custom tags is if special runtime processing of the response output is required. Perhaps the desired functionality requires an extra processing step, or redirection of the output to somewhere other than the browser.

An example is to create a custom tag that you can place around a body of text whose output will be redirected into a log file instead of to a browser, such as in the following example (where `cust` is the prefix for the tag library, and `log` is one of the tags of the library):

```
<cust:log>
   Today is <%= new java.util.Date() %>
   Text to log.
   More text to log.
   Still more text to log.
</cust:log>
```

See "Tag Handlers" on page 7-4 for information about processing of tag bodies.

Use of a Central Checker Page

For general management or monitoring of your JSP application, it may be useful to use a central "checker" page that you include from each page in your application. A central checker page could accomplish tasks such as the following during execution of each page:

- Check session status.
- Check login status (such as checking the cookie to see if a valid login has been accomplished).
- Check usage profile (if a logging mechanism has been implemented to tally events of interest, such as mouse clicks or page visits).

There could be many more uses as well.

As an example, consider a session checker class, `MySessionChecker`, that implements the `HttpSessionBindingListener` interface. (See "Standard Session Resource Management—HttpSessionBindingListener" on page 4-7.)

```
public class MySessionChecker implements HttpSessionBindingListener
{
   ...
   valueBound(HttpSessionBindingEvent event)
   {...}

   valueUnbound(HttpSessionBindingEvent event)
   {...}

   ...
}
```

You can create a checker JSP page, suppose `centralcheck.jsp`, that includes something like the following:

```
<jsp:useBean id="sessioncheck" class="MySessionChecker" scope="session" />
```

In any page that includes `centralcheck.jsp`, the servlet container will call the `valueUnbound()` method implemented in the `MySessionChecker` class as soon as `sessioncheck` goes out of scope (at the end of the session). Presumably this is to manage session resources. You could include `centralcheck.jsp` at the end of each JSP page in your application.

Note: OC4J offers "global includes" as a convenient way to statically include a file into multiple pages. See "Oracle JSP Global Includes" on page 6-9.

Workarounds for Large Static Content in JSP Pages

JSP pages with large amounts of static content (essentially, large amounts of HTML code without content that changes at runtime) may result in slow translation and execution.

There are two primary workarounds for this (either workaround will speed translation):

- Put the static HTML into a separate file and use a dynamic include (`jsp:include`) to include its output in the JSP page output at runtime. See "JSP Actions and the <jsp: > Tag Set" on page 1-15 for information about the `jsp:include` tag.

  Important: A static `include` directive would not work. It would result in the included file being included at translation-time, with its code being effectively copied back into the including page. This would not solve the problem.

- Put the static HTML into a Java resource file.

  The JSP translator will do this for you if you enable the `external_resource` configuration parameter. This parameter is documented in "Configuration Parameter Descriptions" on page 3-7.

For pre-translation, the `-extres` option of the `ojspc` tool also offer this functionality.

Note: Putting static HTML into a resource file may result in a larger memory footprint than the `jsp:include` workaround mentioned above, because the page implementation class must load the resource file whenever the class is loaded.

Another possible, though unlikely, problem with JSP pages that have large static content is that most (if not all) JVMs impose a 64 KB size limit on the code within any single method. Although `javac` would be able to compile it, the JVM would be unable to execute it. Depending on the implementation of the JSP translator, this may become an issue for a JSP page, because generated Java code from essentially the entire JSP page source file goes into the service method of the page implementation class. (Java code is generated to output the static HTML to the browser, and Java code from any scriptlets is copied directly.)

Another possible, though rare, scenario is for the Java scriptlets in a JSP page to be large enough to create a size limit problem in the service method. If there is enough Java code in a page to create a problem, however, then the code should be moved into JavaBeans.

Method Variable Declarations Versus Member Variable Declarations

In "Scripting Elements" on page 1-9, it is noted that JSP <%! ... %> declarations are used to declare member variables, while method variables must be declared in <% ... %> scriptlets.

Be careful to use the appropriate mechanism for each of your declarations, depending on how you want to use the variables:

- A variable that is declared in <%! ... %> JSP declaration syntax is declared at the class level in the page implementation class that is generated by the JSP translator. In this case, if declaring an object instance, the object can be accessed simultaneously from multiple requests. Therefore, the object must be thread-safe, unless `isThreadSafe="false"` is declared in a `page` directive.
- A variable that is declared in <% ... %> JSP scriptlet syntax is local to the service method of the page implementation class. Each time the method is called, a separate instance of the variable or object is created, so there is no need for thread safety.

Consider the following example, `decltest.jsp`:

```
<HTML>
<BODY>
<% double f2=0.0; %>
<%! double f1=0.0; %>
Variable declaration test.
</BODY>
</HTML>
```

This results in something like the following code in the page implementation class:

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

public class decltest extends oracle.jsp.runtime.HttpJsp {
   ...
   // ** Begin Declarations
   double f1=0.0;                  // *** f1 declaration is generated here ***
   // ** End Declarations
   public void _jspService
               (HttpServletRequest request, HttpServletResponse response)
               throws IOException, ServletException {
      ...
      try {
         out.println( "<HTML>");
         out.println( "<BODY>");
         double f2=0.0;      // *** f2 declaration is generated here ***
         out.println( "");
         out.println( "");
         out.println( "Variable declaration test.");
         out.println( "</BODY>");
         out.println( "</HTML>");
         out.flush();
      }
      catch( Exception e) {
         try {
            if (out != null) out.clear();
         }
         catch( Exception clearException) {
         }
      finally {
         if (out != null) out.close();
      }
   }
}
```

Note: This code is provided for conceptual purposes only. Most of the class is deleted for simplicity, and the actual code of a page implementation class generated by the JSP translator would differ somewhat.

Page Directive Characteristics

This section discusses the following `page` directive characteristics:

- A `page` directive is static and takes effect during translation—you cannot specify parameter settings to be evaluated at runtime.
- Java `import` settings in `page` directives are cumulative within a JSP page.

Page Directives Are Static

A `page` directive is static; it is interpreted during translation. You cannot specify dynamic settings to be interpreted at runtime. Consider the following examples:

Example 1 The following `page` directive is *valid*.

```
<%@ page contentType="text/html; charset=EUCJIS" %>
```

Example 2 The following `page` directive is *not valid* and will result in an error. (`EUCJIS` is hard-coded here, but the example also holds true for any character set determined dynamically at runtime.)

```
<% String s="EUCJIS"; %>
<%@ page contentType="text/html; charset=<%=s%>" %>
```

For some `page` directive settings there are workarounds. Reconsidering the second example, there is a `setContentType()` method that allows dynamic setting of the content type, as described in "Dynamic Content Type Settings" on page 8-4.

Page Directive Import Settings Are Cumulative

Java `import` settings in `page` directives within a JSP page are cumulative.

Within any single JSP page, the following two examples are equivalent:

```
<%@ page language="java" %>
<%@ page import="sqlj.runtime.ref.DefaultContext, java.sql.*" %>
```

or:

```
<%@ page language="java" %>
<%@ page import="sqlj.runtime.ref.DefaultContext" %>
<%@ page import="java.sql.*" %>
```

After the first `page` directive `import` setting, the `import` setting in the second `page` directive adds to the set of classes or packages to be imported, as opposed to replacing the classes or packages to be imported.

JSP Preservation of White Space and Use with Binary Data

JSP containers generally preserve source code white space, including carriage returns and linefeeds, in what is output to the browser. Insertion of such white space may not be what the developer intended, and typically makes JSP technology a poor choice for generating binary data.

White Space Examples

The following two JSP pages produce different HTML output, due to the use of carriage returns in the source code.

Example 1—No Carriage Returns

The following JSP page does *not* have carriage returns after the `Date()` and `getParameter()` calls. (The third and fourth lines, starting with the `Date()` call, actually form a single wraparound line of code.)

`nowhitsp.jsp`:

```
<HTML>
<BODY>
<%= new java.util.Date() %> <% String user=request.getParameter("user"); %> <%=
(user==null) ? "" : user %>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

This code results in the following HTML output to the browser. (Note that there are no blank lines after the date.)

```
<HTML>
<BODY>
Tue May 30 20:07:04 PDT 2000
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

Example 2—Carriage Returns

The following JSP page *does* include carriage returns after the `Date()` and `getParameter()` calls.

`whitesp.jsp`:

```
<HTML>
<BODY>
<%= new java.util.Date() %>
<% String user=request.getParameter("user"); %>
<%= (user==null) ? "" : user %>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

This code results in the following HTML output to the browser.

```
<HTML>
<BODY>
Tue May 30 20:19:20 PDT 2000


<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

Note the two blank lines between the date and the "Enter name:" line. In this particular case the difference is not significant, because both examples produce the same appearance in the browser, as shown below. However, this discussion nevertheless demonstrates the general point about preservation of white space.

Reasons to Avoid Binary Data in JSP Pages

For the following reasons, JSP pages are a poor choice for generating binary data. Generally, you should use servlets instead.

- JSP implementations are not designed to handle binary data—there are no methods for writing raw bytes in the `JspWriter` object.
- During execution, the JSP container preserves whitespace. Whitespace is sometimes unwanted, making JSP pages a poor choice for generating binary output to the browser (a `.gif` file, for example) or other uses where whitespace is significant.

  Consider the following example:

  ```
  <% out.getOutputStream().write(...binary data...) %>
  <% out.getOutputStream().write(...more binary data...) %>
  ```

  In this case, the browser will receive an unwanted newline characters in the middle of the binary data or at the end, depending on the buffering of your output buffer. You can avoid this problem by not using a carriage return between the lines of code, but this is an undesirable programming style.

Trying to generate binary data in JSP pages largely misses the point of JSP technology anyway, which is intended to simplify the programming of dynamic textual content.

JSP Data-Access Considerations and Features

This section discusses OC4J JSP features to consider when accessing data, covering the following topics:

- Use of JDBC Performance Enhancement Features
- EJB Calls from JSP Pages
- JSP Support for Oracle SQLJ
- Oracle XML Support

Note: For information about additional OC4J JSP data-access features—portable JavaBeans and tags for SQL functionality—see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Use of JDBC Performance Enhancement Features

You can use the following performance enhancement features, supported through Oracle JDBC extensions, in JSP applications in OC4J:

- caching database connections
- caching JDBC statements
- batching update statements
- prefetching rows during a query
- caching rowsets

Most of these performance features are supported by the `ConnBean` and `ConnCacheBean` data-access JavaBeans (but not by `DBBean`). These beans are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Database Connection Caching

Creating a new database connection is an expensive operation that you should avoid whenever possible. Instead, use a cache of database connections. A JSP application can get a logical connection from a pre-existing pool of physical connections, and return the connection to the pool when done.

You can create a connection pool at any one of the four JSP scopes—`application`, `session`, `page`, or `request`. It is most efficient to use the maximum possible

scope—`application` scope if that is permitted by the Web server, or `session` scope if not.

The Oracle JDBC connection caching scheme, built upon standard connection pooling as specified in the JDBC 2.0 standard extensions, is implemented in the `ConnCacheBean` data-access JavaBean provided with OC4J. Alternatively, you can use standard data source connection pooling functionality, which is supported by the `ConnBean` data-access JavaBean. These beans are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

It is also possible to use the Oracle JDBC `OracleConnectionCacheImpl` class directly, as though it were a JavaBean, as in the following example (although all `OracleConnectionCacheImpl` functionality is available through `ConnCacheBean`):

```
<jsp:useBean id="occi" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
             scope="session" />
```

The same properties are available in `OracleConnectionCacheImpl` as in `ConnCacheBean`. They can be set either through `jsp:setProperty` tags or directly through the class setter methods.

Refer to the OC4J demos for examples of using `OracleConnectionCacheImpl` directly. For information about the Oracle JDBC connection caching scheme and the `OracleConnectionCacheImpl` class, see the *Oracle9i JDBC Developer's Guide and Reference*.

JDBC Statement Caching

Statement caching, an Oracle JDBC extension, improves performance by caching executable statements that are used repeatedly within a single physical connection, such as in a loop or in a method that is called repeatedly. When a statement is cached, the statement does not have to be re-parsed, the statement object does not have to be recreated, and parameter size definitions do not have to be recalculated each time the statement is executed.

The Oracle JDBC statement caching scheme is implemented in the `ConnBean` and `ConnCacheBean` data-access JavaBeans that are provided with OC4J. Each of these beans has a `stmtCacheSize` property that can be set through a `jsp:setProperty` tag or the bean `setStmtCacheSize()` method. The beans are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Statement caching is also available directly through the Oracle JDBC `OracleConnection` and `OracleConnectionCacheImpl` classes. For

information about the Oracle JDBC statement caching scheme and the `OracleConnection` and `OracleConnectionCacheImpl` classes, see the *Oracle9i JDBC Developer's Guide and Reference*.

Important: Statements can be cached only within a single physical connection. When you enable statement caching for a connection cache, statements can be cached across multiple logical connection objects from a single pooled connection object, but not across multiple pooled connection objects.

Update Batching

The Oracle JDBC update batching feature associates a batch value (limit) with each prepared statement object. With update batching, instead of the JDBC driver executing a prepared statement each time its "execute" method is called, the driver adds the statement to a batch of accumulated execution requests. The driver will pass all the operations to the database for execution once the batch value is reached. For example, if the batch value is 10, then each batch of ten operations will be sent to the database and processed in one trip.

OC4J supports Oracle JDBC update batching directly, through the `executeBatch` property of the `ConnBean` data-access JavaBean. You can set this property through a `jsp:setProperty` tag or through the setter method of the bean. If you use `ConnCacheBean` instead, you can enable update batching through Oracle JDBC functionality in the connection and statement objects you create. These beans are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

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

Row Prefetching

The Oracle JDBC row prefetching feature allows you to set the number of rows to prefetch into the client during each trip to the database while a result set is being populated during a query, reducing the number of round-trips to the server.

OC4J supports Oracle JDBC row prefetching directly, through the `preFetch` property of the `ConnBean` data-access JavaBean. You can set this property through a `jsp:setProperty` tag or through the setter method of the bean. If you use `ConnCacheBean` instead, you can enable row prefetching through Oracle JDBC functionality in the connection and statement objects you create. These beans are

described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

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

Rowset Caching

A cached rowset provides a disconnected, serializable, and scrollable container for retrieved data. This feature is useful for small sets of data that do not change often, particularly when the client requires frequent or continued access to the information. By contrast, using a normal result set requires the underlying connection and other resources to be held. Be aware, however, that large cached rowsets consume a lot of memory on the client.

In Oracle9*i*, Oracle JDBC provides a cached rowset implementation. If you are using an Oracle JDBC driver, use code inside a JSP page to create and populate a cached rowset, as follows:

```
CachedRowSet crs = new CachedRowSet();
crs.populate(rset); // rset is a previously created JDBC ResultSet object.
```

Once the rowset is populated, the connection and statement objects used in obtaining the original result set can be closed.

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

EJB Calls from JSP Pages

JSP pages can call EJBs to perform additional processing or data access. A typical application design uses JavaServer Pages as a front-end for the initial processing of client requests, with Enterprise JavaBeans being called to perform the work that involves reading from and writing to data sources. This section provides an overview of EJB usage, covering the following topics:

- Overview of Configuration and Deployment for EJBs
- Code Steps and Approaches for EJB Calls
- Use of the OC4J EJB Tag Library

See the OC4J demos for a complete example incorporating JSP pages and EJBs.

Overview of Configuration and Deployment for EJBs

The configuration and deployment steps for calling EJBs from JSP pages are similar to the steps for calling EJBs from servlets, which are described in the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*. These steps include the following:

- Define an `<ejb-ref>` element in the application `web.xml` file for each EJB called from a JSP page.
- Create an `ejb-jar.xml` deployment descriptor that contains an `<enterprise-beans>` element with appropriate subelements, such as `<session>` or `<entity>`, that specify the types of EJBs. Within these subelements, specify the name, class name, and other details for each called EJB.
- Package the `ejb-jar.xml` file in the EJB archive. Deployment requirements are very similar to the requirements for servlets.

Code Steps and Approaches for EJB Calls

As with a servlet, the key steps required for a JSP page to invoke an EJB are the following:

1. Import the EJB package for the bean home and remote interfaces into each JSP page that makes EJB calls. (In a JSP page, use a `page` directive for this.)
2. Use JNDI to look up the EJB home interface.
3. Create the EJB remote object from the home.
4. Invoke business methods on the remote object.

Because you can use almost any servlet code in a JSP page in the form of a scriptlet, one straightforward way to call EJBs from a JSP page is to use the same code in a scriptlet that you would use in a servlet. This is one way to accomplish steps 2, 3, and 4.

Alternatively, you can use tags from the EJB tag library provided with OC4J. These tags simplify the coding. Essentially, they allow you to treat Enterprise JavaBeans similarly to regular JavaBeans, which are commonly used in JSP pages.

Use of the OC4J EJB Tag Library

Refer to "Code Steps and Approaches for EJB Calls" above. As in that section, import the appropriate package in a `page` directive. Then use the OC4J EJB tags as follows. (See Chapter 7, "JSP Tag Libraries", for general information about JSP tag libraries.)

- Use a `taglib` directive to specify the tag prefix you will use and the location of the tag library description (TLD) file.
- For step 2 above, use an EJB `useHome` tag.
- For step 3 above, you can use an EJB `createBean` tag inside an EJB `useBean` tag.
- For step 4 above, the EJB `iterate` convenience tag allows you to apply business methods to each member of a collection of EJB objects, usually returned by a `find` method.

For more information about the EJB tag library, including detailed tag syntax, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Deployment requirements are the same for the tag library approach as for the scriptlet code approach. As with any tag library, the TLD file and the library support files (tag handler classes and tag-extra-info classes) must be made accessible to your application.

JSP Support for Oracle SQLJ

SQLJ is a standard syntax for embedding static SQL instructions directly in Java code, greatly simplifying database-access programming. The OC4J JSP container supports Oracle SQLJ, allowing you to use SQLJ syntax in JSP statements. SQLJ statements are indicated by the `#sql` token.

For general information about Oracle SQLJ programming features, syntax, and command-line options, see the *Oracle9i SQLJ Developer's Guide and Reference*.

SQLJ JSP Code Example

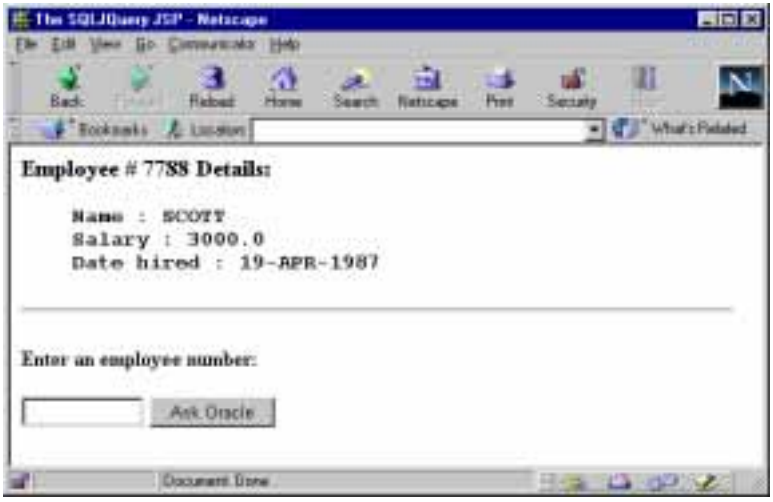
Following is a sample SQLJ JSP page. (The `page` directive imports classes that are typically required by SQLJ.)

```
<%@ page language="sqlj"
    import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>
<HTML>
<HEAD> <TITLE> The SQLJQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<% String empno = request.getParameter("empno");
if (empno != null) { %>
<H3> Employee # <%=empno %> Details: </H3>
<%= runQuery(empno) %>
<HR><BR>
<% } %>
```

```
<B>Enter an employee number:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="empno" SIZE=10>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%!

private String runQuery(String empno) throws java.sql.SQLException {
   DefaultContext dctx = null;
   String ename = null; double sal = 0.0; String hireDate = null;
   StringBuffer sb = new StringBuffer();
   try {
     dctx = Oracle.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");
     #sql [dctx] {
       SELECT ename, sal, TO_CHAR(hiredate,'DD-MON-YYYY')
       INTO :ename, :sal, :hireDate
       FROM scott.emp WHERE UPPER(empno) = UPPER(:empno) };
     sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
     sb.append("Name : " + ename + "\n");
     sb.append("Salary : " + sal + "\n");
     sb.append("Date hired : " + hireDate);
     sb.append("</PRE></B></BIG></BLOCKQUOTE>");
     } catch (java.sql.SQLException e) {
       sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
     } finally {
     if (dctx!= null) dctx.close();
     }
  return sb.toString();
}

%>
```

This example uses the JDBC OCI driver, which requires an Oracle client installation. The `Oracle` class used in getting the connection is provided with Oracle SQLJ.

Entering employee number 7788 for the schema used in the example results in the following output:



Notes:

- In case a JSP page is invoked multiple times in the same JVM, it is recommended that you always use an explicit connection context, such as `dctx` in the example, instead of the default connection context. (Note that `dctx` is a local method variable.)
- If you use Oracle SQLJ, the OC4J JSP container requires SQLJ release 8.1.6.1 or higher.

For further examples of using SQLJ in JSP pages, refer to the OC4J demos.

Triggering the SQLJ Translator

You can trigger the OC4J JSP translator to invoke the Oracle SQLJ translator in one of two ways:

- by using the file name extension `.sqljsp` for the JSP source file

or:

- by specifying `language="sqlj"` in a `page` directive

Either of these results in the JSP translator generating a `.sqlj` file instead of a `.java` file. The Oracle SQLJ translator is then invoked to translate the `.sqlj` file into a `.java` file.

Using SQLJ results in additional output files; see "Generated Files and Locations" on page 6-6.

Important:

- To use Oracle SQLJ, you must install appropriate SQLJ ZIP files (depending on your environment) and add them to your classpath. See "Key Support Files Provided with OC4J" on page 3-2.
- Do not use the same base file name for a `.jsp` file and a `.sqljsp` file in the same application, because this would result in duplicate generated class names and `.java` file names.

Setting Oracle SQLJ Options

When you execute or pre-translate a SQLJ JSP page, you can specify desired Oracle SQLJ option settings. This is true both in on-demand translation scenarios and pre-translation scenarios, as follows:

- In an on-demand translation scenario, use the JSP `sqljcmd` configuration parameter. This parameter, in addition to allowing you to specify a particular SQLJ translator executable, allows you to set SQLJ command-line options.

  For information about `sqljcmd`, see "JSP Configuration Parameters" on page 3-4.

- In a pre-translation scenario with the `ojspc` pre-translation tool, use the `ojspc -S` option. This option allows you to set SQLJ command-line options.

  For information, see "Command-Line Syntax for ojspc" on page 6-17 and "Option Descriptions for ojspc" on page 6-18.

Oracle XML Support

This section describes the following Oracle support features for XML:

- XML-Alternative Syntax
- OracleXMLQuery Class

For information about additional JSP support for XML and XSL that is provided through custom tags, refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

XML-Alternative Syntax

JSP tags, such as <%...%> for scriptlets, <%!...%> for declarations, and <%=...%> for expressions, are not syntactically valid within an XML document. Sun Microsystems addressed this in the *JavaServer Pages Specification, Version 1.1* by defining equivalent JSP tags using syntax that is XML-compatible. This is implemented through a standard DTD that you can specify within a `jsp:root` start tag at the beginning of an XML document.

This functionality allows you, for example, to write XML-based JSP pages in an XML authoring tool.

The OC4J JSP container does not use this DTD directly or require you to use a `jsp:root` tag, but the JSP translator includes logic to recognize the alternative syntax specified in the standard DTD. Table 5–1 documents this syntax.

Table 5–1 XML-Alternative Syntax

Standard JSP Syntax	XML-Alternative JSP Syntax
<%@ *directive* ...%>	<jsp:directive.*directive* ... />
Such as: <%@ page ... %> <%@ include ... %>	Such as: <jsp:directive.page ... /> <jsp:directive.include ... />
<%! ... %> (declaration)	<jsp:declaration> *...declarations go here...* </jsp:declaration>
<%= ... %> (expression)	<jsp:expression> *...expression goes here...* </jsp:expression>
<% ... %> (scriptlet)	<jsp:scriptlet> *...code fragment goes here...* </jsp:scriptlet>

JSP action tags, such as `jsp:useBean`, generally use syntax that already complies with XML. Changes due to quoting conventions or for request-time attribute expressions may be necessary, however.

OracleXMLQuery Class

The `oracle.xml.sql.query.OracleXMLQuery` class is provided with Oracle9*i* and Oracle9*i*AS as part of the XML-SQL utility for XML functionality in database queries. This class requires file `xsu12.jar` (or `xsu111.jar` for JDK 1.1.x), which is also required for XML functionality in some of the custom tags and JavaBeans provided with OC4J. This file is provided with Oracle9*i* and Oracle9*i*AS as well.

For a JSP sample using `OracleXMLQuery`, refer to the OC4J demos.

For information about the `OracleXMLQuery` class and other XML-SQL utility features, refer to the *Oracle9i XML Developer's Kits Guide - XDK*.

JSP Runtime Considerations and Optimization

This section describes some of the JSP runtime functionality, particularly regarding dynamic page retranslation and class reloading, and points out some considerations for optimizing execution. The following topics are covered:

- Dynamic Page Retranslation and Class Reloading
- Additional Optimization Considerations

Dynamic Page Retranslation and Class Reloading

By default, particularly for use in development environments where code is in flux, the JSP container has the following behavior during page execution:

- For each page being executed, the container checks whether a page implementation class already exists, compare the `.class` file timestamp against the `.jsp` source file timestamp, and retranslate the page if the `.class` file is older (indicating that the page has been modified since the page implementation class was loaded).
- For any request that will execute a Java class that was loaded by the JSP class loader, the container checks to see if the class file has been modified since it was last loaded. If the class has been modified, then the JSP class loader reloads it. This applies to class files in the following locations:
  - under the `/WEB-INF/classes` directory
  - in JAR files in the `/WEB-INF/lib` directory
  - under the `_pages` output directory (generated page implementation classes)

  See "Classpath Functionality" on page 3-16 for related information.
- The container reloads a JSP page (in other words, reload the generated page implementation class) in the following circumstances:
  - the page is retranslated
  - a Java class that is called by the page and was loaded by the JSP class loader (and not the system class loader) is modified
  - any page in the same application is reloaded

In a typical deployment environment, where source code will not change, comparing timestamps is unnecessary. In this case, you can avoid all timestamp

comparisons and any possible retranslation and reloading by setting the JSP `main_mode` flag to `justrun`. This will optimize program execution.

If you want to reload modified class files but not retranslate modified JSP pages, you can set `main_mode` to `reload`.

For more information about the `main_mode` flag, see "JSP Configuration Parameters" on page 3-4.

Notes for Dynamic Retranslation and Reloading

- This discussion is not relevant for pre-translation scenarios.
- Because of the usage of in-memory values for class file last-modified times, removing a page implementation class file from the file system will *not* by itself cause retranslation of the associated JSP page source.
- The page implementation class file will be regenerated when the memory cache is lost. This happens whenever a request is directed to this page after the server is restarted or after another page in this application has been retranslated.
- In OC4J, if a statically included page is updated (that is, a page included through an `include` directive), the page that includes it will be automatically retranslated the next time it is invoked. (This is not true in JServ.)

Additional Optimization Considerations

This section describes additional settings you can consider to optimize JSP performance.

Unbuffering a JSP Page

By default, a JSP page uses an area of memory known as a *page buffer*. This buffer (8 KB by default) is required if the page uses dynamic globalization support content type settings, forwards, or error pages. If it does not use any of these features, you can disable the buffer in a `page` directive:

```
<%@ page buffer="none" %>
```

This will improve the performance of the page by reducing memory usage and saving the output step of copying the buffer.

Not Using an HTTP Session

If a JSP page does not need an HTTP session (essentially, does not need to store or retrieve session attributes), then you can direct that no session be used. Specify this with a `page` directive such as the following:

```
<%@ page session="false" %>
```

This will improve the performance of the page by eliminating the overhead of session creation or retrieval.

Note that although servlets by default do *not* use a session, JSP pages by default *do* use a session. For background information, see "Servlet Sessions" on page A-4.

6

JSP Translation and Deployment

This chapter discusses operation of the OC4J JSP translator, then discusses the `ojspc` utility and situations where pre-translation is useful, followed by general discussion of a number of additional JSP deployment considerations.

The chapter is organized as follows:

Functionality of the JSP Translator

JSP translators generate standard Java code for a JSP page implementation class. This class is essentially a servlet class wrapped with features for JSP functionality.

This section discusses general functionality of the JSP translator, focusing on its behavior in on-demand translation scenarios such as in OC4J in the Oracle9*i* Application Server. The following topics are covered:

- Generated Code Features
- General Conventions for Output Names
- Generated Package and Class Names
- Generated Files and Locations
- Oracle JSP Global Includes

> **Important:** Implementation details in this section regarding package and class naming, file and directory naming, output file locations, and generated code are for illustrative purposes. The exact details are subject to change from release to release.

Generated Code Features

This section discusses general features of the page implementation class code that is produced by the JSP translator in translating JSP source (`.jsp` and `.sqljsp` files).

Features of Page Implementation Class Code

When the JSP translator generates servlet code in the page implementation class, it automatically handles some of the standard programming overhead. For both the on-demand translation model and the pre-translation model, generated code automatically includes the following features:

- It extends a wrapper class provided by the JSP container that implements the standard `javax.servlet.jsp.HttpJspPage` interface, which extends the more generic `javax.servlet.jsp.JspPage` interface, which in turn extends the standard `javax.servlet.Servlet` interface.
- It implements the `_jspService()` method specified by the `HttpJspPage` interface. This method, often referred to generically as the "service" method, is the central method of the page implementation class. Code from any Java

scriptlets, expressions, and JSP tags in the JSP page is incorporated into this method implementation.

- It includes code to request an HTTP session, unless your JSP source code specifically sets `session="false"` in a `page` directive.

For introductory information about key JSP and servlet classes and interfaces, see Appendix A, "Servlet and JSP Technical Background".

Inner Class for Static Text

The service method, `_jspService()`, of the page implementation class includes print statements—`out.print()` or equivalent calls on the implicit `out` object—to print any static text in the JSP page. The JSP translator, however, places the static text itself in an inner class within the page implementation class. The service method `out.print()` statements reference attributes of the inner class to print the text.

This inner class implementation results in an additional `.class` file when the page is translated and compiled. In a client-side pre-translation scenario, be aware this means there is an extra `.class` file to deploy.

The name of the inner class will always be based on the base name of the `.jsp` file or `.sqljsp` file. For `mypage.jsp`, for example, the inner class (and its `.class` file) will always include "mypage" in its name.

Note: The OC4J JSP translator can optionally place the static text in a Java resource file, which is advantageous for pages with large amounts of static text. (See "Workarounds for Large Static Content in JSP Pages" on page 5-7.) You can request this feature through the JSP `external_resource` configuration parameter for on-demand translation, or the `ojspc -extres` option for pre-translation.

Even when static text is placed in a resource file, the inner class is still produced, and its `.class` file must be deployed. (This is noteworthy only if you are in a client-side pre-translation scenario.)

General Conventions for Output Names

The JSP translator follows a consistent set of conventions in naming output classes, packages, files, and directories. *However, this set of conventions and other implementation details may change from release to release.*

One fact that is *not* subject to change, however, is that the base name of a JSP page will be included intact in output class and file names as long as it does not include special characters. For example, translating `MyPage123.jsp` will always result in the string "MyPage123" being part of the page implementation class name, Java source file name, and class file name.

In the current release, the base name is preceded by an underscore ("_"). Translating `MyPage123.jsp` results in the page implementation class `_MyPage123` in the source file `_MyPage123.java`, which is compiled into `_MyPage123.class`.

Similarly, where path names are used in creating Java package names, each component of the path is preceded by an underscore. Translating `/jspdir/myapp/MyPage123.jsp`, for example, results in class `_MyPage123` being in the following package:

```
_jspdir._myapp
```

The package name is used in creating directories for output `.java` and `.class` files, so the underscores are also evident in output directory names. For example, in translating a JSP page in a directory such as `webapp/test`, the JSP translator by default will create a directory such as `webappdeployment/_pages/_test` for the page implementation class source. All output directories are created under the standard `_pages` directory, as described in "Generated Files and Locations" on page 6-6.

If you include special characters in a JSP page name or path name, the JSP translator takes steps to ensure that no illegal Java characters appear in the output class, package, and file names. For example, translating `My-name_foo12.jsp` results in `_My_2d_name__foo12` being the class name, in source file `_My_2d_name__foo12.java`. The hyphen is converted to a string of alpha-numeric characters. (An extra underscore is also inserted before "foo12".) In this case, you can only be assured that alphanumeric components of the JSP page name will be included intact in the output class and file names. For example, you could search for "My", "name", or "foo12".

These conventions are demonstrated in examples provided later in this chapter.

Generated Package and Class Names

Although the Sun Microsystems *JavaServer Pages Specification, Version 1.1* defines a uniform process for parsing and translating JSP text, it does not describe how the generated classes should be named—that is up to each JSP implementation.

This section describes how the OC4J JSP translator creates package and class names when it generates code during translation.

Note: For information about general conventions that the OC4J JSP translator uses in naming output classes, packages, and files, see "General Conventions for Output Names" on page 6-4

Package Naming

In an on-demand translation scenario, the URL path that is specified when the user requests a JSP page—specifically, the path relative to the doc root or application root—determines the package name for the generated page implementation class. Each directory in the URL path represents a level of the package hierarchy.

It is important to note, however, that generated package names are *always* lowercase, regardless of the case in the URL.

Consider the following URL as an example:

```
http://host[:port]/HR/expenses/login.jsp
```

In the current OC4J JSP implementation, this results in the following package specification in the generated code (implementation details are subject to change in future releases):

```
package _hr._expenses;
```

No package name is generated if the JSP page is at the application root directory, where the URL is as follows:

```
http://host[:port]/login.jsp
```

Class Naming

The base name of the `.jsp` file (or `.sqljsp` file) determines the class name in the generated code.

Consider the following URL example:

```
http://host[:port]/HR/expenses/UserLogin.jsp
```

In the current OC4J JSP implementation, this yields the following class name in the generated code (implementation details are subject to change in future releases):

```
public class _UserLogin extends ...
```

Be aware that the case (lowercase / uppercase) that end users type in the URL must match the case of the actual `.jsp` or `.sqljsp` file name. For example, they can specify `UserLogin.jsp` if that is the actual file name, or `userlogin.jsp` if that is the actual file name, but not `userlogin.jsp` if `UserLogin.jsp` is the actual file name.

Currently, the translator determines the case of the class name according to the case of the file name. For example:

- `UserLogin.jsp` results in the class `_UserLogin`.
- `Userlogin.jsp` results in the class `_Userlogin`.
- `userlogin.jsp` results in the class `_userlogin`.

If you care about the case of the class name, then you must name the `.jsp` file or `.sqljsp` file accordingly. However, because the page implementation class is invisible to the end user, this is usually not a concern.

Generated Files and Locations

This section describes files that are generated by the JSP translator and where they are placed. For pre-translation scenarios, `ojspc` places files differently and has its own set of relevant options—see "Summary of ojspc Output Files, Locations, and Related Options" on page 6-26.

Wherever JSP configuration parameters are mentioned, see "JSP Configuration Parameters" on page 3-4 for more information.

Note: For information about general conventions used in naming output classes, packages, and files, see "General Conventions for Output Names" on page 6-4

Files Generated by the JSP Translator

This section considers both regular JSP pages (`.jsp` files) and SQLJ JSP pages (`.sqljsp` files or files with `language="sqlj"` in a `page` directive) in listing files that are generated by the JSP translator. For the file name examples, presume a file `Foo.jsp` or `Foo.sqljsp` is being translated.

Source files:

- A `.sqlj` file (for example, `_Foo.sqlj`) is produced by the OC4J JSP translator if the page is a SQLJ JSP page.
- A `.java` file (for example, `_Foo.java`) is produced for the page implementation class and inner class. It is produced either directly by the JSP translator from the `.jsp` file, or by the SQLJ translator from the `.sqlj` file if the page is a SQLJ JSP page. The currently installed Oracle SQLJ translator is used by default, but you can specify an alternative translator by using the `sqljcmd` JSP configuration parameter.

Binary files:

- In the case of a SQLJ JSP page, by default one or more binary files are produced during SQLJ translation for SQLJ profiles. By default these are `.ser` Java resource files, but they will be `.class` files if you enable the SQLJ `-ser2class` option through the `sqljcmd` configuration parameter. The resource file or `.class` file has "Foo" as part of its name.

  Note: Discussion of SQLJ profiles assumes standard SQLJ code generation. Oracle9*i* SQLJ provides an option, `-codegen=oracle`, for Oracle-specific code generation, in which case no profiles are produced.

- A `.class` file is produced by the Java compiler for the page implementation class. The Java compiler is the JDK `javac` by default, but you can specify an alternative compiler using the JSP `javaccmd` configuration parameter.
- An additional `.class` file is produced for the inner class of the page implementation class. This file will have "Foo" as part of its name; in the current implementation it would be `_Foo$__jsp_StaticText.class`.
- A `.res` Java resource file (for example, `_Foo.res`) is optionally produced for the static page content if the `external_resource` JSP configuration parameter is enabled.

Note: The exact names of generated files for the page implementation class may change in future releases, but will still have the same general form. The names would always include the base name, such as "Foo" in these examples, but may include slight variations.

JSP Translator Output File Locations

The JSP translator places generated output files under a base `temp/_pages` directory as follows:

`/j2ee/home/`*`app-deployment/app-name/web-app-name`*`/temp/_pages/...`

Note the following, and refer to "Key OC4J Configuration Files" on page 3-13 for related information about the noted configuration files:

- *`app-deployment`* is the OC4J deployment directory, specified in the OC4J `server.xml` file. It is typically the `application-deployments` directory.
- *`app-name`* is the application name, according to an `<application>` element in `server.xml`.
- *`web-app-name`* is the corresponding "Web application name", mapped to the application name in a `<web-app>` element in the OC4J `default-web-site.xml` file.

The path under the `_pages` directory depends on the path of the `.jsp` file under the application root directory.

As an example, consider the page `welcome.jsp` in the `examples/jsp` subdirectory under the OC4J default Web application directory. The path would be as follows:

`/j2ee/home/default-web-app/examples/jsp/welcome.jsp`

Assuming the default application deployment directory, the JSP translator would place the output files (`_welcome.java`, `_welcome.class`, and `_welcome$__jsp_StaticText.class` for the page implementation class inner class) in the following directory:

`/j2ee/home/application-deployments/default/defaultWebApp/temp/_pages/_examples/_jsp`

Note the following:

- `application-deployments` is the OC4J default deployment directory.
- `default` is the OC4J default application name and `defaultWebApp` is the default Web application name, both used for JSP pages placed in the `default-web-app` directory.
- Because the `.jsp` source file is in an `examples/jsp` subdirectory under the application root directory, the JSP translator generates `_examples._jsp` as the package name, and places the output files into an `_examples/_jsp` subdirectory under the `_pages` directory.

> **Important:** Implementation details, such as the location of generated output files and use of "_" in output file names, are subject to change in future releases.

Oracle JSP Global Includes

In Oracle9*i*AS 9.0.2, the OC4J JSP container introduces a feature called *global includes*. You can use this feature to specify one or more files to statically include into JSP pages in (or under) a specified directory, through virtual JSP `include` directives. During translation, the JSP container looks for a configuration file, `/WEB-INF/ojsp-global-include.xml`, that specifies the included files and the directories for the pages.

This enhancement is particularly convenient for migrating applications that used `globals.jsa` or `translate_params` functionality in previous Oracle JSP releases.

Globally included files can be used for the following, for example:

- global bean declarations (formerly supported through `globals.jsa`)
- common page headers or footers
- `translate_params` equivalent code (typically for a JServ environment)

The ojsp-global-include.xml File

The `ojsp-global-include.xml` file specifies the names of files to include, whether they should be included at the tops or bottoms of JSP pages, and the locations of JSP pages to which the global includes should apply. This section describes the elements of `ojsp-global-include.xml`.

<ojsp-global-include>

This is the root element of the `ojsp-global-include.xml` file. It has no attributes.

Subelements:

<include>

<include ... >

Use this subelement of `<ojsp-global-include>` to specify a file to be included, and whether it should be included at the top or bottom of JSP pages.

Subelements:

<into>

Attributes:

- `file`: Specify the file to be included, such as `"/header.html"` or `"/WEB-INF/globalbeandeclarations.jsph"`. The file name setting must start with a slash ("/"). In other words, it must be context-relative, not page-relative.
- `position`: Specify whether the file is to be included at the top or bottom of JSP pages. Supported values are "top" (default) and "bottom".

<into ... >

Use this subelement of `<include>` to specify a location (a directory, and possibly subdirectories) of JSP pages into which the specified file is to be included. This element has no subelements.

Attributes:

- `directory`: Specify a directory. Any JSP pages in this directory, and optionally its subdirectories, will statically include the file specified in the `file` attribute of the `<include>` element. The `directory` setting must start with a slash ("/"), such as `"/dir1"`. The setting can also include a slash after the directory name, such as `"/dir1/"`, or a slash will be appended internally during translation.
- `subdir`: Use this to specify whether JSP pages in all subdirectories of the `directory` should also have the file statically include. Supported values are "true" (default) and "false".

Global Include Examples

This section provides examples of global includes.

Example: Header/Footer Assume the following `ojsp-global-include.xml` file:

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE ojsp-global-include SYSTEM 'ojsp-global-include.dtd'>

<ojsp-global-include>
  <include file="/header.html">
     <into directory="/dir1" />
  </include>
  <include file="/footer1.html" position="bottom">
     <into directory="/dir1" subdir="false" />
     <into directory="/dir1/part1/" subdir="false" />
  </include>
  <include file="/footer2.html" position="bottom">
     <into directory="/dir1/part2/" subdir="false" />
  </include>
</ojsp-global-include>
```

This example accomplishes three objectives:

- The `header.html` file is included at the top of any JSP page in or under the `dir1` directory. The result would be the same as if each `.jsp` file in or under this directory had the following `include` directive at the top of the page:

  ```
  <%@ include file="/header.html" %>
  ```

- The `footer1.html` file is included at the bottom of any JSP page in the `dir1` directory or its `part1` subdirectory. The result would be the same as if each `.jsp` file in those directories had the following `include` directive at the bottom of the page:

  ```
  <%@ include file="/footer1.html" %>
  ```

- The `footer2.html` file is included at the bottom of any JSP page in the `part2` subdirectory of `dir1`. The result would be the same as if each `.jsp` file in that directory had the following `include` directive at the bottom of the page:

  ```
  <%@ include file="/footer2.html" %>
  ```

Note: If multiple header or multiple footer files are included into a single JSP page, the order of inclusion is according to the order of `<include>` elements in the `ojsp-global-include.xml` file.

Example: translate_params Equivalent Code Assume the following `ojsp-global-include.xml` file:

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE ojsp-global-include SYSTEM 'ojsp-global-include.dtd'>

<ojsp-global-include>
  <include file="/WEB-INF/nls/params.jsf">
     <into directory="/" />
  </include>
</ojsp-global-include>
```

And assume `params.jsf` contains the following:

```
<% request.setCharacterEncoding(response.getCharacterEncoding(); %>
```

The `params.jsf` file (essentially, the `setCharacterEncoding()` method call) is included at the top of any JSP page in or under the application root directory. In other words, it is included in any JSP page in the application. The result would be the same as if each `.jsp` file in or under this directory had the following `include` directive at the top of the page:

```
<%@ include file="/WEB-INF/nls/parms.jsf" %>
```

Also see "Migration Away from translate_params" on page B-27.

The ojspc Pre-Translation Utility

This section describes the `ojspc` utility, provided with OC4J for pre-translation of JSP pages. For consideration of pre-translation scenarios, see "JSP Pre-Translation" on page 6-31.

The following topics are covered here:

- Overview of ojspc Functionality
- Option Summary Table for ojspc
- Command-Line Syntax for ojspc
- Option Descriptions for ojspc
- Summary of ojspc Output Files, Locations, and Related Options

---

Important: To use `ojspc`, you must be using a Sun Microsystems JDK (version 1.1.8 or higher) and you must have `tools.jar` (for JDK 2.0 or higher) or `classes.zip` (for JDK 1.1.8) in your classpath.

Overview of ojspc Functionality

For a simple JSP (not SQLJ JSP) page, default functionality for `ojspc` is as follows:

- It takes a `.jsp` file as an argument.
- It invokes the JSP translator to translate the `.jsp` file into Java page implementation class code, producing a `.java` file. The page implementation class includes an inner class for static page content.
- It invokes the Java compiler to compile the `.java` file, producing two `.class` files—one for the page implementation class itself and one for the inner class.

Following is the default `ojspc` functionality for a SQLJ JSP page:

- It takes a `.sqljsp` file as an argument instead of a `.jsp` file, or it takes a `.jsp` file with `language="sqlj"` in a `page` directive.
- It invokes the JSP translator to translate the SQLJ JSP page into a `.sqlj` file for the page implementation class (and inner class).
- It invokes the Oracle SQLJ translator to translate the `.sqlj` file. This produces a `.java` file for the page implementation class (and inner class) and a SQLJ "profile" file that is, by default, a `.ser` Java resource file.

> **Note:** Discussion of SQLJ profiles assumes standard SQLJ code generation. Oracle9*i* SQLJ provides an option, `-codegen=oracle`, for Oracle-specific code generation, in which case no profiles are produced.

For information about SQLJ profiles and Oracle-specific code generation, see the *Oracle9i SQLJ Developer's Guide and Reference*.

- It invokes the Java compiler to compile the `.java` file, producing two `.class` files—one for the page implementation class itself and one for the inner class.

Under some circumstances (see the `-extres` option descriptions below), `ojspc` options direct the JSP translator to produce a `.res` Java resource file for static page content, instead of putting this content into the inner class of the page implementation class. However, the inner class is still created and must still be deployed with the page implementation class.

Because `ojspc` invokes the JSP translator, `ojspc` output conventions are the same as for the translator in general, as applicable. For general information about JSP translator output, including generated code features, general conventions for output names, generated package and class names, and generated files and locations, see "Functionality of the JSP Translator" on page 6-2.

> **Note:** The `ojspc` command-line tool is a front-end utility that invokes the `oracle.jsp.tool.Jspc` class.

Option Summary Table for ojspc

Table 6–1 describes the options supported by the `ojspc` pre-translation utility. These options are further discussed in "Option Descriptions for ojspc" on page 6-18.

The second column notes comparable or related JSP configuration parameters for on-demand translation environments, such as OC4J.

Notes:

- Enable a boolean `ojspc` option by typing only the option name, not by setting it to `true`. Setting it to `true` will cause an error. All defaults are `false`.
- For a JServ environment, use the `ojspc_jserv` command instead of the `ojspc` command. See "Using ojspc for JServ" on page B-14. Be aware that the `-staticTextInChars` option is not relevant for JServ, so is not supported by `ojspc_jserv`.

Table 6–1 Options for ojspc Pre-Translation Utility

Option	Related JSP Configuration Parameters	Description	Default
-addclasspath	(none)	additional classpath entries for `javac`	empty (no additional path entries)
-appRoot	(none)	application root directory for application-relative static `include` directives from the page	current directory
-debug	emit_debuginfo	boolean to direct `ojspc` to generate a line map to the original `.jsp` file for debugging	`false`
-d	(none)	location where `ojspc` should place generated binary files (`.class` and resource)	current directory
-extend	(none)	class for the generated page implementation class to extend	empty
-extres	external_resource	boolean to direct `ojspc` to generate an external resource file for static text from the `.jsp` file	`false`
-help (or -h)	(none)	boolean to direct `ojspc` to display usage information	`false`

Table 6–1 Options for ojspc Pre-Translation Utility (Cont.)

Option	Related JSP Configuration Parameters	Description	Default
-implement	(none)	interface for the generated page implementation class to implement	empty
-noCompile	javaccmd	boolean to direct `ojspc` *not* to compile the generated page implementation class	`false`
-oldIncludeFromTop	old_include_from_top	boolean to specify that page locations in nested `include` directives are relative to the top-level page, for backward compatibility with Oracle JSP behavior prior to Oracle9*i*AS release 2	`false`
-packageName	(none)	package name for the generated page implementation class	empty (generate package names per `.jsp` file location)
-reduceTagCode	reduce_tag_code	boolean to specify further reduction in the size of generated code for custom tag usage	`false`
-reqTimeIntrospection	req_time_introspection	boolean to allow request-time JavaBean introspection whenever compile-time introspection is not possible	`false`
-S-<*sqlj_option*>	sqljcmd	`-S` prefix followed by an Oracle SQLJ option (for SQLJ JSP pages)	empty
-srcdir	(none)	location where `ojspc` should place generated source files (`.java` and `.sqlj`)	current directory

Table 6–1 Options for ojspc Pre-Translation Utility (Cont.)

Option	Related JSP Configuration Parameters	Description	Default
-staticTextInChars	static_text_in_chars	boolean to instruct the JSP translator to generate static text in JSP pages as characters instead of bytes	`false`
-verbose	(none)	boolean to direct `ojspc` to print status information as it executes	`false`
-version	(none)	boolean to direct `ojspc` to display the JSP version number	`false`
-xmlValidate	xml_validate	boolean to specify whether XML validation is performed on the `web.xml` file and TLD files	`false`

Command-Line Syntax for ojspc

Following is the general `ojspc` command-line syntax (assume % is a UNIX prompt):

```
% ojspc [option_settings] file_list
```

The file list can include `.jsp` files or `.sqljsp` files.

Be aware of the following syntax notes:

- If multiple `.jsp` files are translated, they all must use the same character set (either by default or through `page` directive `contentType` settings).
- Use spaces between file names in the file list.
- Use spaces as separators between option names and option values in the option list.
- Option names are not case sensitive, but option values usually are (such as package names, directory paths, class names, and interface names).
- Enable boolean options, which are disabled by default, by typing only the option name. For example, type `-extres`, *not* `-extres true`.

Following is an example:

```
% ojspc -d /myapp/mybindir -srcdir /myapp/mysrcdir -extres MyPage.sqljsp MyPage2.jsp
```

Option Descriptions for ojspc

This section describes the `ojspc` options in more detail.

-addclasspath (fully qualified path; `ojspc` default: empty)

Use this option to specify additional classpath entries for `javac` to use when compiling generated page implementation class source. Otherwise, `javac` uses only the system classpath.

Notes: The `-addclasspath` setting is also used by the SQLJ translator for SQLJ JSP pages.

-appRoot (fully qualified path; `ojspc` default: current directory)

Use this option to specify an application root directory. The default is the current directory, from which `ojspc` was run.

The specified application root directory path is used as follows:

- for static `include` directives in the page being translated

  The specified directory path is prepended to any application-relative (context-relative) paths in the `include` directives of the translated page.

- in determining the package of the page implementation class

  The package will be based on the location of the file being translated relative to the application root directory. The package, in turn, determines the placement of output files. (See "Summary of ojspc Output Files, Locations, and Related Options" on page 6-26.)

This option is necessary, for example, so included files can still be found if you run `ojspc` from some other directory.

Consider the following example:

- You want to translate the following file:

  ```
  /abc/def/ghi/test.jsp
  ```

- You run `ojspc` from the current directory, `/abc`, as follows (assume `%` is a UNIX prompt):

  ```
  % cd /abc
  % ojspc def/ghi/test.jsp
  ```

- The `test.jsp` page has the following `include` directive:

  ```
  <%@ include file="/test2.jsp" %>
  ```

- The `test2.jsp` page is in the `/abc` directory, as follows:

  ```
  /abc/test2.jsp
  ```

This example requires no `-appRoot` setting, because the default application root setting is the current directory, which is the `/abc` directory. The `include` directive uses the application-relative `/test2.jsp` syntax (note the beginning "/"), so the included page will be found as `/abc/test2.jsp`.

The package in this case is `_def._ghi`, based simply on the location of `test.jsp` relative to the current directory, from which `ojspc` was run (the current directory is the default application root). Output files are placed accordingly.

If, however, you run `ojspc` from some other directory, suppose `/home/mydir`, then you would need an `-appRoot` setting as in the following example:

```
% cd /home/mydir
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

The package is still `_def._ghi`, based on the location of `test.jsp` relative to the specified application root directory.

Note: It is typical for the specified application root directory to be some level of parent directory of the directory where the translated JSP page is located.

-d (fully qualified path; `ojspc` default: current directory)

Use this option to specify a base directory for `ojspc` placement of generated binary files—`.class` files and Java resource files. (The `.res` files produced for static content by the `-extres` option are Java resource files, as are `.ser` profile files produced by the SQLJ translator for SQLJ JSP pages.)

The specified path is taken as a file system path (not an application-relative or page-relative path).

Subdirectories under the specified directory are created automatically, as appropriate, depending on the package. See "Summary of ojspc Output Files, Locations, and Related Options" on page 6-26 for more information.

The default is to use the current directory (your current directory when you executed `ojspc`).

It is recommended that you use this option to place generated binary files into a clean directory so that you easily know what files have been produced.

Notes: In environments such as Windows NT that allow spaces in directory names, enclose the directory name in quotes.

-debug (boolean; `ojspc` default: `false`)

Enable this flag to instruct `ojspc` to generate a line map to the original `.jsp` file for debugging. Otherwise, line-mapping will be to the generated page implementation class.

This flag is useful for source-level JSP debugging, such as when using Oracle9*i* JDeveloper.

Note: In an on-demand translation scenario, the JSP `emit_debuginfo` configuration parameter provides the same functionality.

-extend (fully qualified Java class name; `ojspc` default: empty)

Use this option to specify a Java class that the generated page implementation class will extend.

-extres (boolean; `ojspc` default: `false`)

Enable this flag to instruct `ojspc` to place generated static content (the Java print commands that output static HTML code) into a Java resource file instead of into an inner class of the generated page implementation class.

The resource file name is based on the JSP page name. In the current OC4J JSP implementation, it will be the same core name as the JSP name (unless special characters are included in the JSP name), but with an underscore ("_") prefix and `.res` suffix. Translation of `MyPage.jsp`, for example, would create `_MyPage.res` in addition to normal output. The exact implementation for name generation may change in future releases, however.

The resource file is placed in the same directory as `.class` files.

If there is a lot of static content in a page, this technique will speed translation and may speed execution of the page. For more information, see "Workarounds for Large Static Content in JSP Pages" on page 5-7.

Notes:

- The inner class is still created and must still be deployed.
- In an on-demand translation scenario, the JSP `external_resource` configuration parameter provides the same functionality.

-help (boolean; `ojspc` default: `false`)

Enable this option for `ojspc` to display usage information and then exit. As a shortcut, `-h` is also accepted.

-implement (fully qualified Java interface name; `ojspc` default: empty)

Use this option to specify a Java interface that the generated page implementation class will implement.

-noCompile (boolean; `ojspc` default: `false`)

Enable this flag to direct `ojspc` to *not* compile the generated page implementation class Java source. This allows you to compile it later with an alternative Java compiler.

Notes:

- In an on-demand translation scenario, the JSP `javaccmd` configuration parameter provides related functionality, allowing you to specify an alternative Java compiler directly.
- For a SQLJ JSP page, enabling `-noCompile` does not prevent SQLJ translation, just Java compilation.

-oldIncludeFromTop (backward compatibility for `include`; `ojspc` default: `false`)

This is for backward compatibility with Oracle JSP versions prior to Oracle9*i*AS release 2, for functionality of `include` directives. If you enable this option, page locations in nested `include` directives are relative to the top-level page. If the

option is disabled, page locations are relative to the immediate parent page, as per the JSP 1.2 specification.

Note: In an on-demand translation scenario, the JSP `old_include_from_top` configuration parameter provides the same functionality.

-packageName (fully qualified package name; `ojspc` default: per `.jsp` file location)

Use this option to specify a package name for the generated page implementation class, using Java "dot" syntax.

Without setting this option, the package name is determined according to the location of the `.jsp` file relative to the current directory (from which you ran `ojspc`).

Consider an example where you run `ojspc` from the `/myapproot` directory, while the `.jsp` file is in the `/myapproot/src/jspsrc` directory (assume % is a UNIX prompt):

```
% cd /myapproot
% ojspc -packageName myroot.mypackage src/jspsrc/Foo.jsp
```

This results in `myroot.mypackage` being used as the package name.

If this example did *not* use the `-packageName` option, the JSP translator (in its current implementation) would use `_src._jspsrc` as the package name, by default. (Be aware that such implementation details are subject to change in future releases.)

-reduceTagCode (flag for size reduction of custom tag code; `ojspc` default: `false`)

The Oracle9*i*AS release 2 implementation reduces the size of generated code for custom tag usage, but enabling this option results in even further size reduction. There may be performance consequences regarding tag handler reuse, however. See "Tag Handler Code Generation" on page 7-19.

Note: In an on-demand translation scenario, the JSP `reduce_tag_code` configuration parameter provides the same functionality.

-reqTimeIntrospection (flag for request-time introspection; `ojspc` default: `false`)

Enabling this allows request-time JavaBean introspection whenever compile-time introspection is not possible. When compile-time introspection is possible and succeeds, this parameter is ignored and there is no request-time introspection.

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 request-time (runtime). In this case, if `req_time_introspection` is enabled, the JSP container will delay introspection until request-time. (See "Scripting Variables and Tag-Extra-Info Classes" on page 7-7 for information about use of `VariableInfo`.)

Note: In an on-demand translation scenario, the JSP `req_time_introspection` configuration parameter provides the same functionality.

-S-<*sqlj_option*> <value> (`-S` followed by SQLJ option setting; `ojspc` default: empty)

For SQLJ JSP pages, use the `ojspc -S` option to pass an Oracle SQLJ option to the SQLJ translator. You can use multiple occurrences of `-S`, with one SQLJ option per occurrence.

Unlike when you run the SQLJ translator directly, use a space between a SQLJ option and its value (this is for consistency with other `ojspc` options).

For example (from a UNIX prompt):

```
% ojspc -S-default-customizer -d /myapproot/mybindir MyPage.jsp
```

This command invokes the Oracle SQLJ `-default-customizer` option to choose an alternative profile customizer, as well as setting the `ojspc -d` option.

Here is another example:

```
% ojspc -S-ser2class true -S-status true -d /myapproot/mybindir MyPage.jsp
```

This command enables the SQLJ `-ser2class` option (to convert the profile to a `.class` file) and the SQLJ `-status` option (to display status information as the `.sqlj` file is translated).

> **Note:** As the preceding example shows, you can use an explicit `true` setting in enabling a SQLJ boolean option through the `-S` option setting. This is in contrast to `ojspc` boolean options, which do *not* take an explicit `true` setting.

Note the following for particular Oracle SQLJ options:

- Do not use the SQLJ `-encoding` option; instead, use the `contentType` parameter in a `page` directive in the JSP page.
- Do not use the SQLJ `-classpath` option if you use the `ojspc -addclasspath` option.
- Do not use the SQLJ `-compile` option if you use the `ojspc -noCompile` option.
- Do not use the SQLJ `-d` option if you use the `ojspc -d` option.
- Do not use the SQLJ `-dir` option if you use the `ojspc -srcdir` option.

For information about Oracle SQLJ translator options, see the *Oracle9i SQLJ Developer's Guide and Reference*.

> **Note:** In an on-demand translation scenario, the JSP `sqljcmd` configuration parameter provides related functionality, allowing you to specify an alternative SQLJ translator or specify SQLJ option settings.

-srcdir (fully qualified path; `ojspc` default: current directory)

Use this option to specify a base directory location for `ojspc` placement of generated source files—`.sqlj` files (for SQLJ JSP pages) and `.java` files.

The specified path is taken simply as a file system path (not an application-relative or page-relative path).

Subdirectories under the specified directory are created automatically, as appropriate, depending on the package. See "Summary of ojspc Output Files, Locations, and Related Options" on page 6-26 for more information.

The default is to use the current directory (your current directory when you executed `ojspc`).

It is recommended that you use this option to place generated source files into a clean directory so that you conveniently know what files have been produced.

> **Notes:** In environments such as Windows NT that allow spaces in directory names, enclose the directory name in quotes.

-staticTextInChars (flag to generate static text as characters; `ojspc` default: `false`)

Enabling this option directs the JSP translator to generate static text in JSP pages as characters instead of bytes. The default setting is `false`, which improves performance in outputting static text blocks.

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"); %>
```

> **Note:** In an on-demand translation scenario, the JSP `static_text_in_chars` configuration parameter provides the same functionality.

-verbose (boolean; `ojspc` default: `false`)

Enable this option to direct `ojspc` to report its translation steps as it executes.

The following example shows `-verbose` output for the translation of `myerror.jsp` (in this example, `ojspc` is run from the directory where `myerror.jsp` is located; assume `%` is a UNIX prompt):

```
% ojspc -verbose myerror.jsp
Translating file: myerror.jsp
1 JSP files translated successfully.
Compiling Java file: ./_myerror.java
```

-version (boolean; `ojspc` default: `false`)

Enable this option for `ojspc` to display the JSP version number and then exit.

-xmlValidate (XML validation of `web.xml` and TLD files; `ojspc` default: `false`)

This specifies whether XML validation is performed on the application `web.xml` file and any tag library description (TLD) files. Because the Tomcat JSP reference

implementation does not perform XML validation, this option is disabled by default.

Note: In an on-demand translation scenario, the JSP `xml_validate` configuration parameter provides the same functionality.

Summary of ojspc Output Files, Locations, and Related Options

By default, `ojspc` generates the same set of files that are generated by the JSP translator in an on-demand translation scenario, and places them in or under the current directory (from which `ojspc` was executed).

Here are the files:

- a `.sqlj` source file (SQLJ JSP pages only)
- a `.java` source file
- a `.class` file for the page implementation class
- a `.class` file for the inner class for static text
- a Java resource file (`.ser`) or, optionally, a `.class` file for the SQLJ profile (SQLJ JSP pages only)

  This assumes standard SQLJ code generation. Oracle-specific SQLJ code generation produces no profiles.
- optionally, a Java resource file (`.res`) for the static text of the page

For more information about files that are generated by the JSP translator, see "Generated Files and Locations" on page 6-6.

To summarize some of the commonly used options described under "Option Descriptions for ojspc" on page 6-18, you can use the following `ojspc` options to affect file generation and placement:

- `-appRoot` to specify an application root directory
- `-srcdir` to place source files in a specified location
- `-d` to place binary files (`.class` files and Java resource files) in a specified location

- `-noCompile` to *not* compile the generated page implementation class source (as a result of this, no `.class` files are produced)

  In the case of SQLJ JSP pages, translated `.java` files are still produced, but not compiled.

- `-extres` to put static text into a Java resource file
- `-S-ser2class` (SQLJ `-ser2class` option, for SQLJ JSP pages only) to generate the SQLJ profile in a `.class` file instead of a `.ser` Java resource file

For output file placement, the directory structure underneath the current directory (or directories specified by the `-d` and `-srcdir` options, as applicable) is based on the package. The package is based on the location of the file being translated relative to the application root, which is either the current directory or the directory specified in the `-appRoot` option.

For example, suppose you run `ojspc` as follows (presume % is a UNIX prompt):

```
% cd /abc
% ojspc def/ghi/test.jsp
```

Then the package is `_def._ghi`, and output files will be placed in the directory `/abc/_def/_ghi`, where the `_def/_ghi` subdirectory structure is created as part of the process.

If you specify alternate output locations through the `-d` and `-srcdir` options, a `_def/_ghi` subdirectory structure is created under the specified directories.

Now presume that you run `ojspc` from some other directory, as follows:

```
% cd /home/mydir
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

The package is still `_def._ghi`, according to the location of `test.jsp` relative to the specified application root. Output files will be placed in the directory `/home/mydir/_def/_ghi` or in a `_def/_ghi` subdirectory under locations specified through the `-d` and `-srcdir` options. In either case, the `_def/_ghi` subdirectory structure is created as part of the process.

Notes: It is advisable that you run `ojspc` once for each directory of your JSP application, so files in different directories can be given different package names, as appropriate.

JSP Deployment Considerations

This section covers general deployment considerations and scenarios, mostly independent of your target environment.

It discusses the following topics:

- Overview of EAR/WAR Deployment
- Application Deployment with Oracle9i JDeveloper
- JSP Pre-Translation
- Deployment of Binary Files Only

Overview of EAR/WAR Deployment

This section provides an overview of OC4J deployment features and standard WAR deployment features.

See *Oracle9iAS Containers for J2EE User's Guide* for detailed information about deployment for the OC4J environment.

OC4J Deployment Features

In OC4J, deploy each application through a standard EAR (Enterprise archive) file. Specify the name of the application and the name and location of the EAR file through an `<application>` element in the OC4J `j2ee/home/config/server.xml` file.

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

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

```
dcmctl updateConfig -ct oc4j
```

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

The EAR file includes the following:

- a standard `application.xml` configuration file, in `/META-INF`
- optionally, an `orion-application.xml` configuration file, in `/META-INF`
- a standard WAR (Web archive) file

The WAR file includes the following:

- a standard `web.xml` configuration file, in `/WEB-INF`

  In the `web.xml` file for any particular application, you can override global settings for individual configuration parameters or for the definition of the JSP servlet (`oracle.jsp.runtimev2.JspServlet` by default). Each application uses its own instance of the JSP servlet.
- optionally, an `orion-web.xml` configuration file, in `/WEB-INF`
- classes necessary to run the application (servlets, JavaBeans, and so on), under `WEB-INF/classes` and in JAR files in `WEB-INF/lib`
- JSP pages and static HTML files

The EAR file goes in the OC4J applications directory, which is specified in the `application-directory` setting in the `<application-server>` element of the `server.xml` file, and is typically the `j2ee/home/applications` directory. This would be the same directory as is specified for the EAR file location in the `<application>` element in `server.xml`.

Through the OC4J auto-deployment feature, a new EAR file in the applications directory (as specified in `server.xml`) is detected automatically and hierarchically extracted.

See the *Oracle9iAS Containers for J2EE User's Guide* for more information about deployment and about the `admin.jar` tool (which has additional uses as well). Also see "Key OC4J Configuration Files" on page 3-13 for a summary of important configuration files in OC4J.

Standard WAR Deployment

The Sun Microsystems *JavaServer Pages Specification, Version 1.1* supports the packaging and deployment of Web applications, including JavaServer Pages, according to the Sun Microsystems *Java Servlet Specification, Version 2.2* (and higher).

In typical JSP 1.1 implementations, you can deploy JSP pages through the WAR mechanism, creating WAR files through the JAR utility. The JSP pages can be delivered in source form and are deployed along with any required support classes and static HTML files.

According to the servlet 2.2 (and higher) specification, a Web application includes a deployment descriptor file—`web.xml`—that contains information about the JSP pages and other components of the application. The `web.xml` file must be included in the WAR file.

The servlet 2.2 specification also defines an XML DTD for `web.xml` deployment descriptors and specifies exactly how a servlet container must deploy a Web application to conform to the deployment descriptor.

Through these logistics, a WAR file is the best way to ensure that a Web application is deployed into any standard servlet environment exactly as the developer intends.

Deployment configurations in the `web.xml` deployment descriptor include mappings between servlet paths and the JSP pages and servlets that will be invoked. You can specify many additional features in `web.xml` as well, such as timeout values for sessions, mappings of file name extensions to MIME types, and mappings of error codes to JSP error pages.

For more information about standard WAR deployment, see the Sun Microsystems *Java Servlet Specification, Version 2.2* (and higher).

Application Deployment with Oracle9*i* JDeveloper

Oracle9*i* JDeveloper supports many types of deployment profiles, including simple archive, J2EE application (EAR file), J2EE EJB module (EJB JAR file), J2EE Web module (WAR file), J2EE client module (client JAR file), tag library for JSP 1.1 (tag library JAR file), business components EJB session bean profile, business components CORBA server for VisiBroker, and business components archive profile.

When creating a Business Components for Java (BC4J) Web application using Oracle9*i* JDeveloper, a J2EE Web module deployment archive is generated, containing both the BC4J and the Web application files.

The JDeveloper deployment wizards create all the necessary code to deploy business components as a J2EE Web module. Typically, a JSP client accesses the BC4J application in a J2EE Web Module configuration. The JSP client can also use data tags, data Web beans, or UIX tags to access the business components. (See the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for an overview of the BC4J and UIX tag libraries.)

A J2EE Web module is packaged as a WAR file that contains one or more Web components (servlets and JSP pages) and `web.xml`, the deployment descriptor file.

JDeveloper lets you create the deployment profile containing the Web components and the `web.xml` file, and packages them into a standard J2EE EAR file for deployment. JDeveloper takes the resulting EAR file and deploys it to one or more Oracle9*i*AS instances.

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

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

JSP Pre-Translation

JSP pages are typically used in an on-demand scenario, where pages are translated as they are invoked, in a sequence that is invisible to the user. Another option, however, is to pre-translate JSP pages, which may be useful in saving end users the translation overhead the first time a page is executed.

You also might want to pre-translate pages so that you can deploy binary files only, as discussed in "Deployment of Binary Files Only" on page 6-33.

You can use the Oracle `ojspc` utility for pre-translation, or you can use the standard `jsp_precompile` mechanism.

Pre-Translation with ojspc

When you pre-translate with `ojspc`, use the `-d` option to set an appropriate output base directory for placement of generated binary files.

Consider the example in "JSP Translator Output File Locations" on page 6-8, where the JSP page is located in the `examples/jsp` subdirectory under the OC4J default Web application directory:

`/j2ee/home/default-web-app/examples/jsp/welcome.jsp`

A user would invoke this with a URL such as the following:

`http://`*`host`*`[:`*`port`*`]/examples/jsp/welcome.jsp`

(This is just a general example and does not consider OC4J default configuration for the context path.)

In an on-demand translation scenario for this page, as explained in the example, the JSP translator would by default use the following base directory for placement of generated binary files:

```
/j2ee/home/application-deployments/default/defaultWebApp/temp/_pages
```

When you pre-translate, set your current directory to the application root directory, then in `ojspc` set the `_pages` directory as the output base directory. This results in the appropriate package name and file hierarchy. Continuing the example (assume % is a UNIX prompt):

```
% cd /j2ee/home/default-web-app
% ojspc -d /j2ee/home/application-deployments/default/defaultWebApp/temp/_pages examples/jsp/welcome.jsp
```

The URL noted above specifies an application-relative path of `examples/jsp/welcome.jsp`, so at execution time the JSP container looks for the binary files in an `_examples/_jsp` subdirectory under the `_pages` directory. This subdirectory would be created automatically by `ojspc` if it is run as in the above example.

At execution time, the JSP container would find the pre-translated binaries and would not have to perform translation, assuming that either the source file was not altered after pre-translation, or the JSP `main_mode` flag is set to `justrun`.

Note: OC4J JSP implementation details, such as use of underscores ("_") in output directory names, are subject to change from release to release. This documentation applies specifically to the current release.

Standard JSP Pre-Translation Without Execution

It is also possible to specify JSP pre-translation, without execution, when you invoke the page in the normal way. Accomplish this by enabling the standard `jsp_precompile` request parameter when invoking the JSP page from the browser.

Following is an example:

```
http://host[:port]/foo.jsp?jsp_precompile=true
```

or:

```
http://host[:port]/foo.jsp?jsp_precompile
```

(The `=true` is optional.)

Refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, for more information about this mode of operation.

Deployment of Binary Files Only

You can avoid exposing your JSP source, for proprietary or security reasons, by pre-translating the pages and deploying only the translated and compiled binary files. Pages that are pre-translated, either from previous execution in an on-demand translation scenario or by using `ojspc`, can be deployed to any standard J2EE environment. There are two aspects to this scenario:

- You must archive and deploy the binary files appropriately.
- In the target environment, the JSP container must be configured properly to run pages when the JSP source is not available.

Archiving and Deploying the Binary Files

You must take steps to create and archive the binary files in an appropriate hierarchy.

- If you pre-translate with `ojspc`, you must first set your current directory to the application root directory. After running `ojspc`, archive the output files using the `ojspc` output directory as the base directory for the archive. See "The ojspc Pre-Translation Utility" on page 6-13 for general information about this utility.
- If you are archiving binary files produced during previous execution in an on-demand translation environment, then archive the output directory structure, typically under the `_pages` directory.

In the target environment, place the archive JAR file in the `/WEB-INF/lib` directory, or restore the archived directory structure under the appropriate directory, typically under the `_pages` directory.

Configuring the JSP Container for Execution with Binary Files Only

If you have deployed binary files to an OC4J environment, set the JSP configuration parameter `main_mode` to the value `justrun` or the value `reload` to execute JSP pages without the original source.

Without this setting, the JSP translator will always look for the JSP source file to see if it has been modified more recently than the page implementation `.class` file, and abort with a "file not found" error if it cannot find the source file.

With `main_mode` set appropriately, the end user can invoke a page with the same URL that would be used if the source file were in place.

For how to set configuration parameters in the OC4J environment, see "Setting JSP Configuration Parameters in OC4J" on page 3-12.

7

JSP Tag Libraries

This chapter discusses custom tag libraries, covering the basic framework that vendors can use to provide their own libraries. It then concludes with a discussion of OC4J tag handler features, and a comparison of standard runtime tags versus vendor-specific compile-time tags. The chapter is organized as follows:

- Standard Tag Library Framework
- OC4J JSP Tag Handler Features
- Compile-Time Tags

For complete information about the tag libraries provided with OC4J, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Standard Tag Library Framework

Standard JavaServer Pages technology allows vendors to create custom JSP tag libraries.

A tag library defines a collection of custom actions. The tags can be used directly by developers in manually coding a JSP page, or automatically by Java development tools. A tag library must be portable between different JSP container implementations.

For information beyond what is provided here regarding tag libraries and the standard JavaServer Pages tag library framework, refer to the following resources:

- Sun Microsystems *JavaServer Pages Specification, Version 1.1*
- Sun Microsystems Javadoc for the `javax.servlet.jsp.tagext` package, at the following Web site:

`http://java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/jsp/tagext/package-summary.html`

Overview of a Custom Tag Library Implementation

A custom tag library is imported into a JSP page using a `taglib` directive of the following general form:

```
<%@ taglib uri="URI" prefix="prefix" %>
```

Note the following general points about a tag library implementation and usage:

- The tags of a library are defined in a *tag library description* file, as described in "Tag Library Description Files" on page 7-10.
- The URI in the `taglib` directive specifies where to find the tag library description file, as described in "The taglib Directive" on page 7-13. It is possible to use *URI shortcuts*, as described in "Use of web.xml for Tag Libraries" on page 7-12.
- The prefix in the `taglib` directive is a string of your choosing that you use in your JSP page with any tag from the library.

  Assume the `taglib` directive specifies a prefix `oracust`:

  ```
  <%@ taglib uri="URI" prefix="oracust" %>
  ```

Further assume that there is a tag `mytag` in the library. You might use `mytag` as follows:

```
<oracust:mytag attr1="...", attr2="..." />
```

Using the `oracust` prefix informs the JSP translator that `mytag` is defined in the tag library description file that can be found at the URI specified in the above `taglib` directive.

- The entry for a tag in the tag library description file provides specifications about usage of the tag, including whether the tag uses attributes (as `mytag` does), and the names of those attributes.
- The semantics of a tag—the actions that occur as the result of using the tag—are defined in a *tag handler class*, as described in "Tag Handlers" on page 7-4. Each tag has its own tag handler class, and the class name is specified in the tag library description file.
- The tag library description file indicates whether a tag uses a body.

  As seen above, a tag without a body is used as in the following example:

  ```
  <oracust:mytag attr1="...", attr2="..." />
  ```

  By contrast, a tag with a body is used as in the following example:

  ```
  <oracust:mytag attr1="...", attr2="..." >
     ...body...
  </oracust:mytag>
  ```

- A custom tag action can create one or more server-side objects that are available for use by the tag itself or by other JSP scripting elements, such as scriptlets. These objects are known as *scripting variables*.

  Details regarding the scripting variables that a custom tag uses are defined in a *tag-extra-info* class. This is described in "Scripting Variables and Tag-Extra-Info Classes" on page 7-7.

  A tag can create scripting variables with syntax such as in the following example, which creates the object `myobj`:

  ```
  <oracust:mytag id="myobj" attr1="...", attr2="..." />
  ```

- The tag handler of a nested tag can access the tag handler of an outer tag, in case this is required for any of the processing or state management of the nested tag. See "Access to Outer Tag Handler Instances" on page 7-10.

The sections that follow provide more information about these topics.

Note: Beginning with Oracle9*i*AS 9.0.2, the OC4J JSP container properly handles tag attributes that represent object types (`java.lang.Object`). A string specified as the value of such an attribute is converted to a `java.lang.Object` instance and passed in to the corresponding setter method in the tag handler instance. This features complies with the JSP 1.2 specification.

Tag Handlers

A *tag handler* describes the semantics of the action that results from use of a custom tag. A tag handler is an instance of a Java class that implements one of two standard Java interfaces, depending on whether the tag processes a body of statements (between a start tag and an end tag).

Each tag has its own handler class. By convention, the name of the tag handler class for a tag `abc`, for example, is `AbcTag`.

The tag library description file of a tag library specifies the name of the tag handler class for each tag in the library. (See "Tag Library Description Files" on page 7-10.)

A tag handler instance is a server-side object used at request-time. It has properties that are set by the JSP container, including the page context object for the JSP page that uses the custom tag, and a parent tag handler object if the use of this custom tag is nested within an outer custom tag.

See "Sample Tag Handler Class: ExampleLoopTag.java" on page 7-15 for sample code of a tag handler class.

Note: The Sun Microsystems *JavaServer Pages Specification, Version 1.1* does not mandate whether multiple uses of the same custom tag within a JSP page should use the same or different tag handler instances—this is left to the discretion of JSP vendors. See "OC4J JSP Tag Handler Features" on page 7-19 for information about the Oracle implementation.

Custom Tag Body Processing

Custom tags, as with standard JSP tags, may or may not have a body. In the case of a custom tag, even when there is a body, it may not need special processing by the tag handler.

There are three possible situations:

- There is no body.

  In this case, there is just a single tag, as opposed to a start tag and end tag. Following is a general example:

  ```
  <oracust:abcdef attr1="...", attr2="..." />
  ```

- There is a body that does not need special handling by the tag handler.

  In this case, there is a start tag and end tag with a body of statements in between, but the tag handler does not process the body—body statements are passed through for normal JSP processing only. Following is a general example:

  ```
  <foo:if cond="<%= ... %>" >
  ...body executed if cond is true, but not processed by tag handler...
  </foo:if>
  ```

- There is a body that needs special processing by the tag handler.

  In this case also, there is a start tag and end tag with a body of statements in between; however, the tag handler must process the body.

  ```
  <oracust:ghijkl attr1="...", attr2="..." >
  ...body processed by tag handler...
  </oracust:ghijkl>
  ```

Integer Constants for Body Processing

The tag handling interfaces that are described in the following sections specify a `doStartTag()` method (further described below) that you must implement to return an appropriate `int` constant, depending on the situation. The possible return values are as follows:

- `SKIP_BODY` if there is no body or if evaluation and execution of the body should be skipped
- `EVAL_BODY_INCLUDE` if there is a body that does not require special processing by the tag handler
- `EVAL_BODY_TAG` if there is a body that requires special processing by the tag handler

Handlers for Tags That Do Not Process a Body

For a custom tag that does not have a body, or has a body that does not need special processing by the tag handler, the tag handler class must directly or indirectly implement the following standard interface:

- `javax.servlet.jsp.tagext.Tag`

The following standard support class implements the `Tag` interface and can be used as a base class:

- `javax.servlet.jsp.tagext.TagSupport`

The `Tag` interface specifies a `doStartTag()` method and a `doEndTag()` method. The tag developer provides code for these methods in the tag handler class, as appropriate, to be executed as the start tag and end tag, respectively, are encountered. The JSP page implementation class generated by the JSP translator includes appropriate calls to these methods.

Action processing—whatever you want the action tag to accomplish—is implemented in the `doStartTag()` method. The `doEndTag()` method implements any appropriate post-processing. In the case of a tag without a body, essentially nothing happens between the execution of these two methods.

The `doStartTag()` method returns an integer value. For a tag handler class implementing the `Tag` interface (either directly or indirectly), this value must be either `SKIP_BODY` or `EVAL_BODY_INCLUDE` (described in "Integer Constants for Body Processing" above). `EVAL_BODY_TAG` is illegal for a tag handler class implementing the `Tag` interface.

Handlers for Tags That Process a Body

For a custom tag with a body that requires special processing by the tag handler, the tag handler class must directly or indirectly implement the following standard interface:

- `javax.servlet.jsp.tagext.BodyTag`

The following standard support class implements the `BodyTag` interface and can be used as a base class:

- `javax.servlet.jsp.tagext.BodyTagSupport`

The `BodyTag` interface specifies a `doInitBody()` method and a `doAfterBody()` method in addition to the `doStartTag()` and `doEndTag()` methods specified in the `Tag` interface.

Just as with tag handlers implementing the `Tag` interface (described in the preceding section, "Handlers for Tags That Do Not Process a Body"), the tag developer implements the `doStartTag()` method for action processing by the tag, and the `doEndTag()` method for any post-processing.

The `doStartTag()` method returns an integer value. For a tag handler class implementing the `BodyTag` interface (directly or indirectly), this value must be either `SKIP_BODY` or `EVAL_BODY_TAG` (described in "Integer Constants for Body Processing" on page 7-5). `EVAL_BODY_INCLUDE` is illegal for a tag handler class implementing the `BodyTag` interface.

In addition to implementing the `doStartTag()` and `doEndTag()` methods, the tag developer, as appropriate, provides code for the `doInitBody()` method, to be invoked before the body is evaluated, and the `doAfterBody()` method, to be invoked after each evaluation of the body. The body could be evaluated multiple times, such as at the end of each iteration of a loop. The JSP page implementation class generated by the JSP translator includes appropriate calls to all of these methods.

After the `doStartTag()` method is executed, the `doInitBody()` and `doAfterBody()` methods are executed if the `doStartTag()` method returned `EVAL_BODY_TAG`.

The `doEndTag()` method is executed after any body processing, when the end tag is encountered.

For custom tags that must process a body, the `javax.servlet.jsp.tagext.BodyContent` class is available. This is a subclass of `javax.servlet.jsp.JspWriter` that you can use to process body evaluations so that they can be re-extracted later. The `BodyTag` interface includes a `setBodyContent()` method that the JSP container can use to give a `BodyContent` handle to a tag handler instance.

Scripting Variables and Tag-Extra-Info Classes

A custom tag action can create one or more server-side objects, known as *scripting variables*, that are available for use by the tag itself or by other scripting elements, such as scriptlets and other tags.

Details regarding scripting variables that a custom tag defines must be specified in a subclass of the standard `javax.servlet.jsp.tagext.TagExtraInfo` abstract class. This document refers to such a subclass as a *tag-extra-info class*.

The JSP container uses tag-extra-info instances during translation. The tag library description file, specified in the `taglib` directive that imports the library into a JSP page, specifies the tag-extra-info class to use, if applicable, for any given tag.

A tag-extra-info class has a `getVariableInfo()` method to retrieve names and types of the scripting variables that will be assigned during HTTP requests. The JSP translator calls this method during translation, passing it an instance of the standard `javax.servlet.jsp.tagext.TagData` class. The `TagData` instance specifies attribute values set in the JSP statement that uses the custom tag.

This section covers the following topics:

- Defining Scripting Variables
- Scripting Variable Scopes
- Tag-Extra-Info Classes and the getVariableInfo() Method

Defining Scripting Variables

Objects that are defined explicitly in a custom tag can be referenced in other actions through the page context object, using the object ID as a handle. Consider the following example:

```
<oracust:foo id="myobj" attr1="..." attr2="..." />
```

This statement results in the object `myobj` being available to any scripting elements between the tag and the end of the page. The `id` attribute is a translation-time attribute. The tag developer provides a tag-extra-info class that will be used by the JSP container. Among other things, the tag-extra-info class specifies what class to instantiate for the `myobj` object.

The JSP container enters `myobj` into the page context object, where it can later be obtained by other tags or scripting elements using syntax such as the following:

```
<oracust:bar ref="myobj" />
```

The `myobj` object is passed through the tag handler instances for the `foo` and `bar` tags. All that is required is knowledge of the name of the object (`myobj`).

Note that `id` and `ref` are merely sample attribute names; there are no special predefined semantics for these attributes. It is up to the tag handler to define attribute names and create and retrieve objects in the page context.

Scripting Variable Scopes

Specify the scope of a scripting variable in the tag-extra-info class of the tag that creates the variable. It can be one of the following `int` constants:

- `NESTED`—if the scripting variable is available between the start tag and end tag of the action that defines it
- `AT_BEGIN`—if the scripting variable is available from the start tag until the end of the page
- `AT_END`—if the scripting variable is available from the end tag until the end of the page

Tag-Extra-Info Classes and the getVariableInfo() Method

You must create a tag-extra-info class for any custom tag that creates scripting variables. The class describes the scripting variables and must be a subclass of the standard `javax.servlet.jsp.tagext.TagExtraInfo` abstract class.

The key method of the `TagExtraInfo` class is `getVariableInfo()`, which is called by the JSP translator and returns an array of instances of the standard `javax.servlet.jsp.tagext.VariableInfo` class (one array instance for each scripting variable the tag creates).

The tag-extra-info class constructs each `VariableInfo` instance with the following information regarding the scripting variable:

- its name
- its Java type (cannot be a primitive type)
- a boolean indicating whether it is a newly declared variable
- its scope

Important: As of the current OC4J JSP implementation, the `getVariableInfo()` method can return either a fully qualified class name (FQCN) or a partially qualified class name (PQCN) for the Java type of the scripting variable. FQCNs were required in previous releases, and are still preferred in order to avoid confusion in case there are duplicate class names between packages.

Note that primitive types are not supported.

See "Sample Tag-Extra-Info Class: ExampleLoopTagTEI.java" on page 7-16 for sample code of a tag-extra-info class.

Access to Outer Tag Handler Instances

Where nested custom tags are used, the tag handler instance of the nested tag has access to the tag handler instance of the outer tag, which may be useful in any processing and state management performed by the nested tag.

This functionality is supported through the static `findAncestorWithClass()` method of the `javax.servlet.jsp.tagext.TagSupport` class. Even though the outer tag handler instance is not named in the page context object, it is accessible because it is the closest enclosing instance of a given tag handler class.

Consider the following JSP code example:

```
<foo:bar1 attr="abc" >
   <foo:bar2 />
</foo:bar1>
```

Within the code of the `bar2` tag handler class (class `Bar2Tag`, by convention), you can have a statement such as the following:

```
Tag bar1tag = TagSupport.findAncestorWithClass(this, Bar1Tag.class);
```

The `findAncestorWithClass()` method takes the following as input:

- the `this` object that is the class handler instance from which `findAncestorWithClass()` was called (a `Bar2Tag` instance in the example)
- the name of the `bar1` tag handler class (presumed to be `Bar1Tag` in the example), as a `java.lang.Class` instance

The `findAncestorWithClass()` method returns an instance of the appropriate tag handler class, in this case `Bar1Tag`, as a `javax.servlet.jsp.tagext.Tag` instance.

It is useful for a `Bar2Tag` instance to have access to the outer `Bar1Tag` instance in case the `Bar2Tag` needs the value of a `bar1` tag attribute or needs to call a method on the `Bar1Tag` instance.

Tag Library Description Files

A *tag library description* (TLD) file is an XML-style document that contains information about a tag library and individual tags of the library. The name of a TLD file has the `.tld` extension.

A JSP container uses the TLD file in determining what action to take when it encounters a tag from the library.

A tag entry in the TLD file includes the following:

- name of the custom tag
- name of the corresponding tag handler class
- name of the corresponding tag-extra-info class (if applicable)
- information indicating how the tag body (if any) should be processed
- information about each attribute of the tag (the parameters that you specify whenever you use the custom tag), including its name, whether it is required, and whether it can accept real-time expressions as values

Here is a sample TLD file entry for the tag `myaction`:

```
<tag>
  <name>myaction</name>
  <tagclass>examples.MyactionTag</tagclass>
  <teiclass>examples.MyactionTagExtraInfo</teiclass>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>attr1</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>attr2</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

According to this entry, the tag handler class is `MyactionTag` and the tag-extra-info class is `MyactionTagExtraInfo`. The attribute `attr1` is required; the attribute `attr2` is optional and can take a real-time expression as its value.

The `bodycontent` parameter indicates how the tag body (if any) should be processed. There are three valid values:

- A value of `empty` indicates that the tag uses no body.
- A value of `JSP` indicates that the tag body should be processed as JSP source and translated.

- A value of `tagdependent` indicates that the tag body should not be translated. Any text in the body is treated as static text.

The `taglib` directive in a JSP page informs the JSP container where to find the TLD file. (See "The taglib Directive" on page 7-13.)

For more information about tag library description files, see the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

Use of web.xml for Tag Libraries

The Sun Microsystems *Java Servlet Specification, Version 2.2* (and higher) describes a standard deployment descriptor for servlets—the `web.xml` file. JSP pages can use this file in specifying the location of a JSP tag library description file.

For JSP tag libraries, the `web.xml` file can include a `<taglib>` element and two subelements:

- `<taglib-uri>`
- `<taglib-location>`

The `taglib-location` subelement indicates the application-relative location (by starting with "/") of the tag library description file.

The `<taglib-uri>` subelement indicates a "shortcut" URI to use in `taglib` directives in your JSP pages, with this URI being mapped to the TLD file location specified in the accompanying `<taglib-location>` subelement. (The term URI, *universal resource indicator*, is somewhat equivalent to the term URL, *universal resource locator*, but is more generic.)

Following is a sample `web.xml` entry for a tag library description file:

```
<taglib>
   <taglib-uri>/oracustomtags</taglib-uri>
   <taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

This entry makes `/oracustomtags` equivalent to `/WEB-INF/oracustomtags/tlds/MyTLD.tld` in `taglib` directives in your JSP pages. See "Using a Shortcut URI for the TLD File" on page 7-13 for an example.

See the Sun Microsystems *Java Servlet Specification, Version 2.2* or *Version 2.3*, and the Sun Microsystems *JavaServer Pages Specification, Version 1.1* for more information about the `web.xml` deployment descriptor and its use for tag library description files.

The taglib Directive

Import a custom library into a JSP page using a `taglib` directive, of the following form:

```
<%@ taglib uri="URI" prefix="prefix" %>
```

For the URI, you have the following options:

- Specify a shortcut URI, as defined in a `web.xml` file. (See "Use of web.xml for Tag Libraries" on page 7-12.)
- Fully specify the tag library description (TLD) file name and location.

Using a Shortcut URI for the TLD File

Assume the following `web.xml` entry for a tag library defined in the tag library description file `MyTLD.tld`:

```
<taglib>
   <taglib-uri>/oracustomtags</taglib-uri>
   <taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

Given this example, the following directive in your JSP page results in the JSP container finding the `/oracustomtags` URI in `web.xml` and, therefore, finding the accompanying name and location of the tag library description file (`MyTLD.tld`):

```
<%@ taglib uri="/oracustomtags" prefix="oracust" %>
```

This statement allows you to use any of the tags of this custom tag library in a JSP page.

Fully Specifying the TLD File Name and Location

If you do not want your JSP application to depend on a `web.xml` file for its use of a tag library, the `taglib` directive can fully specify the name and location of the tag library description file, as follows:

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.tld" prefix="oracust" %>
```

The location is specified as an application-relative location (by starting with "/", as in this example). See "Requesting a JSP Page" on page 1-25 for related discussion.

Alternatively, you can specify a `.jar` file instead of a `.tld` file in the `taglib` directive, where the `.jar` file contains a tag library description file. The tag library description file must be located and named as follows when you create the JAR file:

```
META-INF/taglib.tld
```

Then the `taglib` directive might be as follows, for example:

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.jar" prefix="oracust" %>
```

End-to-End Example: Defining and Using a Custom Tag

This section provides an end-to-end example of the definition and use of a custom tag, `loop`, that is used to iterate through the tag body a specified number of times.

Included in the example are the following:

- JSP source code for a page that uses the tag
- source code for the tag handler class
- source code for the tag-extra-info class
- the tag library description file

Note: Sample code here uses extended datatypes in the `oracle.jsp.jml` package. There is an overview of these types in "Extended Type JavaBeans" on page 2-11. For more information, refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Sample JSP Page: exampletag.jsp

Following is a sample JSP page, `exampletag.jsp`, that uses the `loop` tag, specifying that the outer loop be executed five times and the inner loop three times:

```
<%@ taglib uri="/WEB-INF/exampletag.tld" prefix="foo" %>
<% int num=5; %>
<br>
<pre>
<foo:loop index="i" count="<%=num%>">
body1here: i expr: <%=i%>
           i property: <jsp:getProperty name="i" property="value" />
  <foo:loop index="j" count="3">
  body2here: j expr: <%=j%>
```

```
  i property: <jsp:getProperty name="i" property="value" />
  j property: <jsp:getProperty name="j" property="value" />
  </foo:loop>
</foo:loop>
</pre>
```

Sample Tag Handler Class: ExampleLoopTag.java

This section provides source code for the tag handler class, `ExampleLoopTag`. Note the following:

- The `doStartTag()` method returns the integer constant `EVAL_BODY_TAG`, so that the tag body (essentially, the loop) is processed.
- After each pass through the loop, the `doAfterBody()` method increments the counter. It returns `EVAL_BODY_TAG` if there are more iterations left, and `SKIP_BODY` after the last iteration.

Here is the code:

```
package examples;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.Hashtable;
import java.io.Writer;
import java.io.IOException;
import oracle.jsp.jml.JmlNumber;

public class ExampleLoopTag
    extends BodyTagSupport
{

    String index;
    int count;
    int i=0;
    JmlNumber ib=new JmlNumber();

    public void setIndex(String index)
    {
      this.index=index;
    }
    public void setCount(String count)
    {
      this.count=Integer.parseInt(count);
```

```
    }

    public int doStartTag() throws JspException {
        return EVAL_BODY_TAG;
    }

    public void doInitBody() throws JspException {
        pageContext.setAttribute(index, ib);
        i++;
        ib.setValue(i);
    }

    public int doAfterBody() throws JspException {
        try {
            if (i >= count) {
                bodyContent.writeOut(bodyContent.getEnclosingWriter());
                return SKIP_BODY;
            } else
                pageContext.setAttribute(index, ib);
            i++;
            ib.setValue(i);
            return EVAL_BODY_TAG;
        } catch (IOException ex) {
            throw new JspTagException(ex.toString());
        }
    }
}
```

Sample Tag-Extra-Info Class: ExampleLoopTagTEI.java

This section provides the source code for the tag-extra-info class that describes the scripting variable used by the `loop` tag.

A `VariableInfo` instance is constructed that specifies the following for the variable:

- The variable name is according to the `index` attribute.
- The variable is of the type `oracle.jsp.jml.JmlNumber`, which you must specify as a fully qualified class name.
- The variable is newly declared.
- The variable scope is `NESTED`.

In addition, the tag-extra-info class has an `isValid()` method that determines whether the `count` attribute is valid—it must be an integer.

```
package examples;

import javax.servlet.jsp.tagext.*;

public class ExampleLoopTagTEI extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data) {
        return new VariableInfo[]
            {
                new VariableInfo(data.getAttributeString("index"),
                                 "oracle.jsp.jml.JmlNumber",
                                 true,
                                 VariableInfo.NESTED)
            };
    }

    public boolean isValid(TagData data)
    {
      String countStr=data.getAttributeString("count");
      if (countStr!=null)   // for request-time case
      {
        try {
          int count=Integer.parseInt(countStr);
        }
        catch (NumberFormatException e)
        {
          return false;
        }
      }
      return true;
    }
}
```

Sample Tag Library Description File: exampletag.tld

This section presents the tag library description (TLD) file for the tag library. In this example, the library consists of only the one tag, `loop`.

This TLD file specifies the following for the `loop` tag:

- the tag handler class: `examples.ExampleLoopTag`
- the tag-extra-info class: `examples.ExampleLoopTagTEI`

- a `bodycontent` specification with a value of `JSP`

  This means the JSP translator should process and translate the body code.

- attributes `index` and `count`, both mandatory

  The `count` attribute can be a request-time JSP expression.

Here is the TLD file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>simple</shortname>
  <info>
        A simple tag library for the examples
  </info>
  <!-- example tag -->
  <!-- for loop -->
  <tag>
    <name>loop</name>
    <tagclass>examples.ExampleLoopTag</tagclass>
    <teiclass>examples.ExampleLoopTagTEI</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>for loop</info>
    <attribute>
        <name>index</name>
        <required>true</required>
    </attribute>
    <attribute>
        <name>count</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>

</taglib>
```

OC4J JSP Tag Handler Features

This section describes OC4J JSP features for tag handler pooling and code generation size reduction.

Disabling or Enabling Tag Handler Instance Pooling

You can specify whether JSP tag handler instances are pooled in a particular JSP page, always in the `application` scope, by setting the `oracle.jsp.tags.reuse` attribute in the JSP page context. Set it to `true` to enable pooling, or to `false` to disable pooling. For example:

```
pageContext.setAttribute("oracle.jsp.tags.reuse", new Boolean(true));
```

You can use separate settings in different pages, or even in different sections of the same page.

The default is according to the setting of the `tags_reuse_default` JSP configuration parameter, or `true` if there is no `tags_reuse_default` setting. See "JSP Configuration Parameters" on page 3-4 for further information about this parameter and how to set it.

Note: The JSP container also supports tag handler instance pooling in the JServ environment. In that environment the default setting is `false`.

Tag Handler Code Generation

The JSP implementation in Oracle9iAS release 2 reduces the code generation size for custom tag usage. In addition, there is a JSP configuration flag, `reduce_tag_code`, that you can set to `true` for even further size reduction.

Be aware, however, that when this flag is enabled, the code generation pattern does not maximize tag handler reuse. Although you can still improve performance by setting `tags_reuse_default` to `true` as described in "Disabling or Enabling Tag Handler Instance Pooling" above, the effect is not maximized when `reduce_tag_code` is also true.

See "JSP Configuration Parameters" on page 3-4 for further information about these parameters and how to set them.

Compile-Time Tags

Standard tag libraries, as described in the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, use a runtime support mechanism. They are typically portable, not requiring any particular JSP container.

It is also possible, however, for vendors to support custom tags through vendor-specific functionality in their JSP translators. Such tags are not portable to other containers.

It is generally advisable to develop standard, portable tags that use the runtime mechanism, but there may be scenarios where tags using a compile-time mechanism are appropriate, as discussed in this section.

General Compile-Time Versus Runtime Considerations

The JSP 1.1 specification describes a runtime support mechanism for custom tag libraries. This mechanism, using an XML-style tag library description file to specify the tags, is covered in "Standard Tag Library Framework" on page 7-2.

Creating and using a tag library that adheres to this model assures that the library will be portable to any standard JSP environment.

There are, however, reasons to consider compile-time implementations:

- A compile-time implementation may produce more efficient code.
- A compile-time implementation allows the developer to catch errors during translation and compilation, instead of the end-user seeing them at runtime.

In the future, Oracle may offer a general framework for creating custom tag libraries with compile-time tag implementations. Because such implementations would depend on the OC4J JSP translator, they would not be portable to other JSP environments.

JSP Compile-Time Versus Runtime JML Library

OC4J provides a portable tag library called the JSP Markup Language (JML) library. This library uses the standard JSP 1.1 runtime mechanism.

However, the JML tags are also supported through a compile-time mechanism. This is because the tags were first introduced with JSP implementations that preceded the JSP 1.1 specification, when the runtime mechanism was introduced. The compile-time tags are still supported for backward compatibility.

The general advantages and disadvantages of compile-time implementations apply to the Oracle JML tag library as well. There may be situations where it is advantageous to use the compile-time JML implementation. There are also a few additional tags in that implementation, and some additional expression syntax that is supported.

Both the runtime version and the compile-time version of the JML library are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

8

JSP Globalization Support

The JSP container in OC4J provides standard globalization support (also known as National Language Support, or NLS) according to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, and also offers extended support for servlet environments that do not support multibyte parameter encoding.

Standard Java support for localized content depends on the use of Unicode 2.0 for uniform internal representation of text. Unicode is used as the base character set for conversion to alternative character sets.

This chapter describes key aspects of JSP support for globalization and internationalization. The following topics are covered:

- Content Type Settings
- JSP Support for Multibyte Parameter Encoding

Note: For detailed information about Oracle9*i*AS Globalization Support, see the *Oracle9i Application Server Globalization Support Guide*.

Content Type Settings

This section covers standard ways to statically or dynamically specify the content type for a JSP page. It also discusses an Oracle extension method that allows you to specify a non-IANA (Internet Assigned Numbers Authority) character set for the JSP writer object. The section is organized as follows:

- Content Type Settings in the page Directive
- Dynamic Content Type Settings
- Oracle Extension for the Character Set of the JSP Writer Object

Content Type Settings in the page Directive

You can use the standard `page` directive `contentType` parameter to set the MIME type and to optionally set the character encoding for a JSP page. The MIME type applies to the HTTP response at runtime. The character encoding, if set, applies to both the page text during translation and the HTTP response at runtime.

Use the following syntax for the `page` directive:

```
<%@ page ... contentType="TYPE; charset=character_set" ... %>
```

or, to set the MIME type while using the default character set:

```
<%@ page ... contentType="TYPE" ... %>
```

`TYPE` is an IANA MIME type; `character_set` is an IANA character set. (When specifying a character set, the space after the semicolon is optional.)

For example:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
```

or:

```
<%@ page language="java" contentType="text/html" %>
```

The default MIME type is `text/html`. The IANA maintains a registry of MIME types at the following site:

`ftp://www.isi.edu/in-notes/iana/assignments/media-types/media-types`

The default character encoding is `ISO-8859-1` (also known as Latin-1). The IANA maintains a registry of character encodings at the following site (use the indicated "preferred MIME name" if one is listed):

`http://www.iana.org/assignments/character-sets`

There is no JSP requirement to use an IANA character set as long as you use a character set that Java and the Web browser support, but the IANA site lists the most common character sets. Using the preferred MIME names they document is recommended.

The parameters of a `page` directive are static. If a page discovers during execution that a different setting is necessary for the response, it can do one of the following:

- Use the servlet response object API to set the content type during execution, as described in "Dynamic Content Type Settings" on page 8-4.
- Forward the request to another JSP page or to a servlet.

Notes:

- The `page` directive that sets `contentType` should appear as early as possible in the JSP page.
- A JSP page written in a character set other than `ISO-8859-1` must set the appropriate character set in a `page` directive. It cannot be set dynamically, because the page has to be aware of the setting during translation. Dynamic settings are for runtime only.
- The JSP 1.1 specification assumes that a JSP page is written in the same character set that it will use to deliver its content.
- This document, for simplicity, assumes the typical case that the page text, request parameters, and response parameters all use the same encoding (although other scenarios are technically possible). Request parameter encoding is controlled by the browser, although Netscape and Internet Explorer browsers follow the setting you specify for the response parameters.

Dynamic Content Type Settings

For situations where the appropriate content type for the HTTP response is not known until runtime, you can set it dynamically in the JSP page. The standard `javax.servlet.ServletResponse` interface specifies the following method for this purpose:

```
public void setContentType(java.lang.String contenttype)
```

(The implicit `response` object of a JSP page is a `javax.servlet.http.HttpServletResponse` instance, where the `HttpServletResponse` interface extends the `ServletResponse` interface.)

The `setContentType()` method input, like the `contentType` setting in a `page` directive, can include a MIME type only, or both a character set and a MIME type. For example:

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

or:

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

As with a `page` directive, the default MIME type is `text/html` and the default character encoding is `ISO-8859-1`.

This method has no effect on interpreting the text of the JSP page during translation. If a particular character set is required during translation, that must be specified in a `page` directive, as described in "Content Type Settings in the page Directive" on page 8-2.

Be aware of the following important usage notes:

- The JSP page *cannot* be unbuffered if you are using the `setContentType()` method. It is buffered by default; do not set `buffer="none"` in a `page` directive.
- The `setContentType()` call must appear early in the page, before any output to the browser or any `jsp:include` command (which flushes the JSP buffer to the browser).
- In servlet 2.2 environments, the `response` object has a `setLocale()` method that sets a default character set based on the specified locale, overriding any previous character set. For example, the following method call results in a character set of `Shift_JIS`:

  ```
  response.setLocale(new Locale("ja", "JP"));
  ```

Oracle Extension for the Character Set of the JSP Writer Object

In standard usage, the character set of the content type of the `response` object, as determined by the `page` directive `contentType` parameter or the `response.setContentType()` method, automatically becomes the character set of the JSP writer object as well. The JSP writer object is a `javax.servlet.jsp.JspWriter` instance.

There are some character sets, however, that are not recognized by IANA and therefore cannot be used in a standard content type setting. For this reason, OC4J provides the static `setWriterEncoding()` method of the `oracle.jsp.util.PublicUtil` class:

```
public static void setWriterEncoding(JspWriter out, String encoding)
```

You can use this method to specify the character set of the JSP writer directly, overriding the character set of the `response` object. The following example uses `Big-5` as the character set of the content type, but specifies `MS950`, a non-IANA Hong Kong dialect of `Big-5`, as the character set of the JSP writer:

```
<%@ page contentType="text/html; charset=Big-5" %>
<% oracle.jsp.util.PublicUtil.setWriterEncoding(out, "MS950"); %>
```

Note: Use the `setWriterEncoding()` method as early as possible in the JSP page.

JSP Support for Multibyte Parameter Encoding

The Sun Microsystems servlet 2.3 specification documents a method, `setCharacterEncoding()`, that is useful in case the default encoding of the servlet container is not suitable for multibyte request parameters and bean property settings, such as for a `getParameter()` call in Java code or a `jsp:setProperty` tag to set a bean property in JSP code.

The `setCharacterEncoding()` method and equivalent Oracle extensions affect parameter names and values, specifically:

- request object `getParameter()` method output
- request object `getParameterValues()` method output
- request object `getParameterNames()` method output
- `jsp:setProperty` settings for bean property values

This section covers the following topics:

Standard setCharacterEncoding() Method

Effective with the servlet 2.3 specification, the `setCharacterEncoding()` method is specified in the `javax.servlet.ServletRequest` interface as the standard mechanism for specifying a non-default character encoding for reading HTTP requests. The signature of this method is as follows:

```
void setCharacterEncoding(java.lang.String enc)
                          throws java.io.UnsupportedEncodingException
```

The `enc` parameter is a string specifying the name of the desired character encoding, and overrides the default character encoding. Call this method before reading request parameters or reading input through the `getReader()` method (also specified in the `ServletRequest` interface).

There is also a corresponding getter method:

```
String getCharacterEncoding()
```

Overview of Oracle Extensions for Older Servlet Environments

In pre-2.3 servlet environments, the `setCharacterEncoding()` method is not available. For such environments, particularly the JServ servlet 2.0 environment, Oracle provides two alternative mechanisms:

- `oracle.jsp.util.PublicUtil.setReqCharacterEncoding()` static method (preferred)
- `translate_params` configuration parameter (or equivalent code)

For information about these mechanisms, see "Multibyte Parameter Encoding in JServ" on page B-21.

A

Servlet and JSP Technical Background

This appendix provides technical background on servlets and JavaServer Pages. Although this document is written for users who are well grounded in servlet technology, the servlet information here may be a useful refresher for some.

Standard JavaServer Pages interfaces, implemented automatically by generated JSP page implementation classes, are briefly discussed as well. Most readers, however, will not require this information.

The following topics are covered:

- Background on Servlets
- Web Application Hierarchy
- Standard JSP Interfaces and Methods

Note: For more information about servlets and the OC4J servlet container, refer to the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*.

Background on Servlets

Because JSP pages are translated into Java servlets, a brief review of servlet technology may be helpful. Refer to the Sun Microsystems *Java Servlet Specification, Version 2.2* (or higher) for more information about the concepts discussed here.

For information about the methods this section discusses, refer to Sun Microsystems Javadoc at the following locations (for servlet 2.2 and 2.3, respectively):

```
http://java.sun.com/products/servlet/2.2/javadoc/index.html
```

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

Review of Servlet Technology

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

Prior to servlets, CGI (Common Gateway Interface) 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.

Servlet technology, in addition to improved scalability, offers the well-known Java advantages of object orientation, platform independence, security, and robustness. Servlets can use all standard Java APIs, including the JDBC API (for Java database connectivity, of particular interest to database programmers).

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

The Servlet Interface

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

For Web applications, the `Servlet` interface can be implemented by extended the standard `javax.servlet.http.HttpServlet` abstract class. The `HttpServlet` class includes the following methods:

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

A servlet class that subclasses `HttpServlet` must implement some of these methods, as appropriate. Each method takes as input a standard `javax.servlet.http.HttpServletRequest` instance and a standard `javax.servlet.http.HttpServletResponse` instance.

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

Servlet Containers

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

When a servlet is called (such as when a servlet is specified by URL), the Web server passes the HTTP request to the servlet container. The container, in turn, passes the request to the servlet. In the course of managing a servlet, a simple container performs the following:

- It creates an instance of the servlet and calls its `init()` method to initialize it.
- It calls the `service()` method of the servlet.
- 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.

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

Servlet Sessions

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

HttpSession Interface

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

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

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

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

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

- `public Object getAttribute(String name)`

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

Note: Older servlet implementations use `putValue()` and `getValue()` instead of `setAttribute()` and `getAttribute()`, with the same signatures.

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

- `public boolean invalidate()`

  This method immediately invalidates the session and unbinds any objects from it.

- `public boolean setMaxInactiveInterval(int interval)`

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

- `public boolean isNew()`

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

- `public boolean getCreationTime()`

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

- `public boolean getLastAccessedTime()`

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

Session Tracking

The `HttpSession` interface supports alternative mechanisms for tracking sessions. Each involves some way to assign a *session ID*. A session ID is an intermediate handle that is assigned and used by the servlet container. Multiple sessions by the same user can share the same session ID, if appropriate.

The following session-tracking mechanisms are supported:

- cookies

  The servlet container sends a cookie to the client, which returns the cookie to the server upon each HTTP request. This associates the request with the session ID indicated by the cookie. `JSESSIONID` must be the name of the cookie.

  This is the most frequently used mechanism and is supported by any servlet container that adheres to the servlet 2.2 or higher specification.

- URL rewriting

  The servlet container appends a session ID to the URL path. The name of the path parameter must be `jsessionid`, as in the following example:

  `http://`*`host`*`[:`*`port`*`]/myapp/index.html?jsessionid=6789`

  This is the most frequently used mechanism where clients do not accept cookies.

- SSL Sessions

  SSL (Secure Sockets Layer, used in the HTTPS protocol) includes a mechanism to take multiple requests from a client and define them as belonging to a single session. Some servlet containers use the SSL mechanism for their own session tracking as well.

Servlet Contexts

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

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

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

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

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

The `ServletContext` interface specifies methods that allow a servlet to communicate with the servlet container that runs it, which is one of the ways that the servlet can retrieve application-level environment and state information.

Note: In early versions of the servlet specification, the concept of servlet contexts was not sufficiently defined. Beginning with version 2.1(b), however, the concept was further clarified, and it was specified that an HTTP session object could not exist across multiple servlet context objects.

Application Lifecycle Management Through Event Listeners

The *Java Servlet Specification, Version 2.2* (and higher) provides limited application lifecycle management through the standard Java event-listener mechanism. HTTP session objects can use event listeners to make objects stored in the session object aware of when they are added or removed. Because the typical reason for removing objects within a session object is that the session has become invalid, this mechanism allows the developer to manage session-based resources. Similarly, the event-listener mechanism also allows the managing of page-based and request-based resources.

Unfortunately, servlet context objects do not support this sort of notification. Standard servlet application support does not provide a way to manage application-based resources.

Servlet Invocation

A servlet, like an HTML page, is invoked through a URL. The servlet is launched according to how servlets are mapped to URLs in the Web server implementation. Following are the possibilities:

- A specific URL can be mapped to a specific servlet class.
- An entire directory can be mapped so that any class in the directory is executed as a servlet. For example, the special `/servlet` directory can be mapped so that any URL of the form `/servlet/`*`servlet_name`* executes a servlet.
- A file name extension can be mapped so that any URL specifying a file whose name includes that extension executes a servlet.

This mapping would be specified as part of the Web server configuration. In OC4J, this is according to settings in the `global-web-application.xml` file.

Web Application Hierarchy

The entities relating to a Web application (which consists of some combination of servlets and JSP pages) do not follow a simple hierarchy but can be considered in the following order:

1. servlet objects (including page implementation objects)

   There is a servlet object for each servlet and for each JSP page implementation in a running application (and possibly more than one object, depending on whether a single-thread or multiple-thread execution model is used). A servlet object processes request objects from a client and sends response objects back to the client. A JSP page, as with servlet code, specifies how to create the response objects.

   You can think of multiple servlet objects as being within a single request object in some circumstances, such as when one page or servlet "includes" or forwards to another.

   A user will typically access multiple servlet objects in the course of a session, with the servlet objects being associated with the session object.

   Servlet objects, as well as page implementation objects, indirectly implement the standard `javax.servlet.Servlet` interface. For servlets in a Web application, this is accomplished by subclassing the standard `javax.servlet.http.HttpServlet` abstract class. For JSP page implementation classes, this is accomplished by implementing the standard `javax.servlet.jsp.HttpJspPage` interface.

2. request and response objects

   These objects represent the individual HTTP requests and responses that are generated as a user runs an application.

   A user will typically generate multiple requests and receive multiple responses in the course of a session. The request and response objects are not "contained in" the session, but are associated with the session.

   As a request comes in from a client, it is mapped to the appropriate servlet context object (the one associated with the application the client is using) according to the virtual path of the URL. The virtual path will include the root path of the application.

   A request object implements the standard `javax.servlet.http.HttpServletRequest` interface.

A response object implements the standard `javax.servlet.http.HttpServletResponse` interface.

3. session objects

   Session objects store information about the user for a given session and provide a way to identify a single user across multiple page requests. There is one session object per user.

   There may be multiple users of a servlet or JSP page at any given time, each represented by their own session object. All these session objects, however, are maintained by the servlet context that corresponds to the overall application. In fact, you can think of each session object as representing an instance of the Web application associated with a common servlet context.

   Typically, a session object will sequentially make use of multiple request objects, response objects, and page or servlet objects, and no other session will use the same objects; however, the session object does not "contain" those objects per se.

   A session lifecycle for a given user starts with the first request from that user. It ends when the user session terminates (such as when the user quits the application) or there is a timeout.

   HTTP session objects implement the `javax.servlet.http.HttpSession` interface.

   Note: Prior to the 2.1(b) version of the servlet specification, a session object could span multiple servlet context objects.

4. servlet context object

   A servlet context object is associated with a particular path in the server. This is the base path for modules of the application associated with the servlet context, and is referred to as the *application root*.

   There is a single servlet context object for all sessions of an application in any given JVM, providing information from the server to the servlets and JSP pages that form the application. The servlet context object also allows application sessions to share data within a secure environment isolated from other applications.

   The servlet container provides a class that implements the standard `javax.servlet.ServletContext` interface, instantiates this class the first time a user requests an application, and provides this `ServletContext` object with the path information for the location of the application.

The servlet context object typically has a pool of session objects to represent the multiple simultaneous users of the application.

A servlet context lifecycle starts with the first request (from any user) for the corresponding application. The lifecycle ends only when the server is shut down or otherwise terminated.

For additional introductory information about servlet contexts, see "Servlet Contexts" on page A-6.

5. servlet configuration object

   The servlet container uses a servlet configuration object to pass information to a servlet when it is initialized—the `init()` method of the `Servlet` interface takes a servlet configuration object as input.

   The servlet container provides a class that implements the standard `javax.servlet.ServletConfig` interface and instantiates it as necessary. Included within the servlet configuration object is a servlet context object (also instantiated by the servlet container).

Standard JSP Interfaces and Methods

Two standard interfaces, both in the `javax.servlet.jsp` package, are available to be implemented in code that is generated by a JSP translator:

- `JspPage`
- `HttpJspPage`

`JspPage` is a generic interface that is not intended for use with any particular protocol. It extends the `javax.servlet.Servlet` interface.

`HttpJspPage` is an interface for JSP pages using the HTTP protocol. It extends `JspPage` and is typically implemented directly and automatically by any servlet class generated by a JSP translator.

`JspPage` specifies the following methods for use in initializing and terminating instances of the generated class:

- `jspInit()`
- `jspDestroy()`

If you want any special initialization or termination functionality, you must provide a JSP declaration to override the relevant method, as in the following example:

```
<%! void jspInit()
    {
        ...your implementation code...
    }
%>
```

`HttpJspPage` adds specification for the following method:

- `_jspService()`

Code for this method is typically generated automatically by the translator and includes the following:

- code from scriptlets in the JSP page
- code resulting from any JSP directives
- any static content of the page

(JSP directives provide information for the page, such as specifying the Java language for scriptlets and providing package imports. See "Directives" on page 1-7.)

As with the `Servlet` methods, the `_jspService()` method takes an `HttpServiceRequest` instance and an `HttpServiceResponse` instance as input.

The `JspPage` and `HttpJspPage` interfaces inherit the following methods from the `Servlet` interface:

- `init()`
- `destroy()`
- `service()`
- `getServletConfig()`
- `getServletInfo()`

Refer back to "The Servlet Interface" on page A-3 for a discussion of the `Servlet` interface and its key methods.

B

The Apache JServ Environment

The primary Web application environment supplied with Oracle9*i* Application Server release 2 is Oracle9*i*AS Containers for J2EE (OC4J). In addition, an Apache JServ servlet environment is provided, and was the primary servlet environment in earlier releases of Oracle9*i* Application Server.

For those who use the JServ environment (presumably for backward compatibility), there are special considerations relating to servlet and JSP usage, as with any servlet 2.0 environment. This appendix covers these considerations.

The following topics are discussed:

- Getting Started in a JServ Environment
- Considerations for the JServ Environment
- JSP Application and Session Support for JServ
- Samples Using globals.jsa for Servlet 2.0 Environments

Getting Started in a JServ Environment

This section provides information about configuring JServ to run JSP pages, covering the following topics:

- Adding Files to the Apache JServ Web Server Classpath
- Mapping JSP File Name Extensions for JServ
- JSP Configuration Parameters for JServ
- Setting JSP Parameters in JServ

Adding Files to the Apache JServ Web Server Classpath

To add files to the Web server classpath in a JServ environment, insert appropriate `wrapper.classpath` commands into the `jserv.properties` file in the JServ `conf` directory. Note that `jsdk.jar` should already be in the classpath. This file is from the Sun Microsystems JSDK 2.0 and provides servlet 2.0 versions of the `javax.servlet.*` packages that are required by JServ. Additionally, files for your JDK environment should already be in the classpath.

The following example (which happens to use UNIX directory paths) includes files for JSP, JDBC, and SQLJ. Replace `[Oracle_Home]` with your Oracle Home path.

```
# servlet 2.0 APIs (required by JServ, from Sun JSDK 2.0):
wrapper.classpath=jsdk2.0/lib/jsdk.jar
#
# servlet 2.2 APIs (required and provided by OC4J):
wrapper.classpath=[Oracle_Home]/ojsp/lib/servlet.jar
# JSP packages:
wrapper.classpath=[Oracle_Home]/ojsp/lib/ojsp.jar
wrapper.classpath=[Oracle_Home]/ojsp/lib/ojsputil.jar
# XML parser (used for servlet 2.2 web deployment descriptor):
wrapper.classpath=[Oracle_Home]/ojsp/lib/xmlparserv2.jar
# JDBC libraries for Oracle database access (JDK 1.2.x environment):
wrapper.classpath=[Oracle_Home]/ojsp/lib/classes12.zip
# SQLJ translator (optional):
wrapper.classpath=[Oracle_Home]/ojsp/lib/translator.zip
# SQLJ runtime (optional) (for JDK 1.2.x enterprise edition):
wrapper.classpath=[Oracle_Home]/ojsp/lib/runtime12.zip
```

Note: If `servlet.jar` (provided with OC4J for servlet 2.2 versions of `javax.servlet.*` packages) is in your classpath in a JServ environment, `jsdk.jar` must precede it.

Now consider an example where you have the following `useBean` command:

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session" />
```

You can add the following `wrapper.classpath` command to the `jserv.properties` file. (This example happens to be for a Windows NT environment.)

```
wrapper.classpath=D:\Apache\Apache1.3.9\beans\
```

And then `JDBCQueryBean.class` should be located as follows:

```
D:\Apache\Apache1.3.9\beans\mybeans\JDBCQueryBean.class
```

Mapping JSP File Name Extensions for JServ

In a JServ environment, mapping each JSP file name extension to `oracle.jsp.JspServlet` (the JSP front-end servlet for JServ) requires an `ApJServAction` command in either the `jserv.conf` file or the `mod_jserv.conf` file. These configuration files are in the JServ `conf` directory.

(In older versions, you must instead update the `httpd.conf` file in the Apache `conf` directory. In newer versions, the `jserv.conf` or `mod_jserv.conf` file is "included" into `httpd.conf` during execution—look at the `httpd.conf` file to see which one it includes.)

Following is an example (which happens to use UNIX syntax):

```
# Map file name extensions (.sqljsp and .SQLJSP are optional).
ApJServAction .jsp /servlets/oracle.jsp.JspServlet
ApJServAction .JSP /servlets/oracle.jsp.JspServlet
ApJServAction .sqljsp /servlets/oracle.jsp.JspServlet
ApJServAction .SQLJSP /servlets/oracle.jsp.JspServlet
```

The path you use in this command for `oracle.jsp.JspServlet` is not a literal directory path in the file system. The path to specify depends on your JServ servlet configuration—how the servlet zone is mounted, the name of the zone properties file, and the file system directory that is specified as the repository for the servlet.

("Servlet zone" is a JServ term that is similar conceptually to "servlet context".) Consult your JServ documentation for more information.

JSP Configuration Parameters for JServ

This section describes the configuration parameters supported by the Oracle `JspServlet` for the JServ environment.

For general information about JSP configuration parameters for Oracle9*i*AS, see "JSP Configuration Parameters" on page 3-4.

Configuration Parameter Summary Table for JServ

Table B–1 summarizes the configuration parameters supported by the original Oracle JSP front-end servlet, `oracle.jsp.JspServlet`. This is the front-end used for the JServ environment. (OC4J uses the front-end servlet `oracle.jsp.runtimev2.JspServlet`.) For each parameter, the table notes the following:

- the correspondence (if any) to OC4J configuration parameters
- equivalent or related `ojspc` options for pre-translation
- a brief description of the option
- the default value
- whether it is used at compile-time or runtime

Notes:

- See "The ojspc Pre-Translation Utility" on page 6-13 for information about the equivalent `ojspc` options.
- The `main_mode`, `precompile_check`, and `static_text_in_chars` parameters, offered for OC4J, are not available for JServ environments. For JServ, however, outputting static text as characters is the default.

Table B–1 JSP Configuration Parameters, JServ Environment

Parameter	Correspondence to Configuration Parameters in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
alias_translation	not applicable in OC4J	(n/a)	boolean; `true` to work around JServ limitations in directory aliasing for JSP page references	false	both
bypass_source	migrated to `main_mode` flag, `justrun` setting	(n/a)	boolean; `true` for the JSP container to ignore `FileNotFound` exceptions on `.jsp` source; uses pre-translated and compiled code when source is not available	false	runtime
classpath	not applicable in OC4J	-addclasspath (related, but different functionality)	additional classpath entries for JSP class loading	null (no addl. path)	both
debug_mode	exists as is	(n/a)	boolean; `true` for the JSP container to print the stack trace when a runtime exception occurs	true	runtime
developer_mode	migrated to `main_mode` flag	(n/a)	boolean; `false` to *not* check timestamps to see if page retranslation and class reloading is necessary when a page is requested	true	runtime
emit_debuginfo	exists as is	-debug	boolean; `true` to generate a line map to the original `.jsp` file for debugging	false	compile-time

Table B–1 JSP Configuration Parameters, JServ Environment (Cont.)

Parameter	Correspondence to Configuration Parameters in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
external_resource	exists as is	-extres	boolean; `true` for the JSP translator to place all static content of the page into a separate Java resource file during translation	false	compile-time
javaccmd	exists as is	-noCompile	Java compiler command line—`javac` options, or alternative Java compiler run in a separate JVM (`null` means use JDK `javac` with default options)	null	compile-time
old_include_from_top	exists as is	-oldIncludeFromTop	boolean; `true` for page locations in nested `include` directives to be relative to the top-level page, for backward compatibility with Oracle JSP behavior prior to Oracle9*i*AS release 2	false	compile-time
reduce_tag_code	exists as is	-reduceTagCode	boolean; `true` for further reduction in the size of generated code for custom tag usage	false	compile-time

Table B–1 JSP Configuration Parameters, JServ Environment (Cont.)

Parameter	Correspondence to Configuration Parameters in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
req_time_introspection	exists as is	-reqTimeIntrospection	boolean; `true` to enable request-time JavaBean introspection whenever compile-time introspection is not possible	false	compile-time
send_error	not applicable in OC4J	(n / a)	boolean; `true` to output standard "404" errors for file-not-found, and "500" errors for compilation failure (instead of outputting customized messages)	false	runtime
session_sharing (for use with `globals.jsa`)	not applicable in OC4J	(n / a)	boolean; for applications using `globals.jsa`, `true` for JSP session data to be propagated to underlying servlet session	true	runtime
sqljcmd	exists as is	-S	SQLJ command line—`sqlj` options, or alternative SQLJ translator run in a separate JVM (`null` means use the Oracle SQLJ version provided with OC4J, with default option settings)	null	compile-time

Table B–1 JSP Configuration Parameters, JServ Environment (Cont.)

Parameter	Correspondence to Configuration Parameters in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
tags_reuse_default	exists as is	(n / a)	boolean; specifies a default setting for JSP tag handler pooling (`true` to enable by default; `false` to disable by default); this default setting can be overridden for any particular JSP page	false	runtime
translate_params	not applicable in OC4J	(n / a)	boolean; `true` to override servlet containers that do not perform multibyte encoding	false	runtime
unsafe_reload	not applicable in OC4J	(n / a)	boolean; `true` to *not* restart the application and sessions whenever a JSP page is retranslated and reloaded	false	runtime
xml_validate	exists as is	-xmlValidate	boolean; `true` for XML validation to be performed on the `web.xml` file and TLD files	false	compile-time

Configuration Parameter Descriptions for JServ

This section describes configuration parameters for the JServ environment in more detail.

alias_translation (boolean; default: `false`)

This parameter allows the OC4J JSP container to work around limitations in the way JServ handles directory aliasing. For information about the current limitations, see "JServ Directory Alias Translation" on page B-19.

You must set `alias_translation` to `true` for `httpd.conf` directory aliasing commands, such as the following example, to work properly in the JServ servlet environment:

```
Alias /icons/ "/apache/apache139/icons/"
```

bypass_source (boolean; default: `false`)

Normally, when a JSP page is requested, the JSP container throws a `FileNotFound` exception if it cannot find the corresponding `.jsp` source file, even if it can find the page implementation class. This is because, by default, the JSP container checks the page source to see if it has been modified since the page implementation class was generated.

Set this parameter to `true` for the JSP container to proceed and execute the page implementation class even if it cannot find the page source.

If `bypass_source` is enabled, the container still checks for retranslation if the source is available and is needed. One of the factors in determining whether it is needed is the setting of the `developer_mode` parameter.

Notes:

- The `bypass_source` option is useful in deployment environments that have the generated classes only, not the source. (For related discussion, see "Deployment of Binary Files Only" on page 6-33.)
- Oracle9*i* JDeveloper enables `bypass_source` so that you can translate and run a JSP page before you have saved the JSP source to a file.

---

classpath (fully qualified path; default: `null`)

Use this parameter to add classpath entries to the JSP default classpath for use during translation, compilation, or execution of JSP pages.

Overall, the JSP container loads classes from its own classpath (including entries from this `classpath` parameter), the system classpath, the Web server classpath, the page repository, and predefined locations relative to the root directory of the JSP application.

Be aware that classes that are loaded through the path specified in the `classpath` setting are loaded by the JSP class loader, not by the system class loader. During JSP execution, classes that are loaded by the JSP class loader cannot access (or be accessed by) classes that are loaded by the system class loader or by any other class loader.

Notes:

- Runtime automatic class reloading applies only to classes in the JSP classpath. This includes paths specified through this `classpath` parameter. (See "Dynamic Page Retranslation and Class Reloading" on page 5-25 for information about this feature.)
- When you pre-translate pages, the `ojspc -addclasspath` option offers some related, though different, functionality. See "Option Descriptions for ojspc" on page 6-18.

---

debug_mode (boolean; default: `true`)

This flag has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

developer_mode (boolean; default: `true`)

Set this flag to `false` to instruct the JSP container to *not* routinely compare the timestamp of the page implementation class to the timestamp of the `.jsp` source file when a page is requested. With `developer_mode=true`, the container checks every time to see if the source has been modified since the page implementation class was generated. If that is the case, the JSP translator retranslates the page. With `developer_mode=false`, the JSP container will check only upon the initial request for the page or application. For subsequent requests, it will simply re-execute the generated page implementation class.

This flag also affects dynamic class reloading for JavaBeans and other support classes called by a JSP page. With `developer_mode=true`, The JSP container checks to see if such classes have been modified since being loaded by the JSP class loader.

Oracle generally recommends setting `developer_mode` to `false`, particularly in a deployment environment where code is not likely to change and where performance is a significant issue.

Also see "Dynamic Page Retranslation and Class Reloading" on page 5-25.

emit_debuginfo (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

external_resource (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

javaccmd (compiler executable; default: `null`)

This has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

old_include_from_top (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

reduce_tag_code (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

req_time_introspection (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

send_error (boolean; default: `false`)

Set this flag to `true` to direct the JSP container to output generic "404" messages for file-not-found conditions, and generic "500" messages for compilation errors.

This is in contrast to outputting customized messages that provide more information (such as the name of the file not found). Some environments, such as JServ, do not allow output of a customized message if a "404" or "500" message is output.

session_sharing (boolean; default: `true`) (for use with `globals.jsa`)

When you use a `globals.jsa` file for an application, presumably in a servlet 2.0 environment, each JSP page uses a distinct JSP session wrapper attached to the single overall servlet session object provided by the servlet container.

In this situation, the `true` (default) setting of the `session_sharing` parameter results in JSP session data being propagated to the underlying servlet session. This allows servlets in the application to access the session data of JSP pages in the application.

If `session_sharing` is `false` (which parallels standard behavior in most JSP implementations), JSP session data is not propagated to the servlet session. As a result, application servlets would not be able to access JSP session data.

This parameter is meaningless if `globals.jsa` is not used. For information about `globals.jsa`, see "JSP Application and Session Support for JServ" on page B-28.

sqljcmd (SQLJ translator executable and options; default: `null`)

This has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

tags_reuse_default (boolean; default: `false`)

This has the same use in JServ as in OC4J, but in JServ is `false` by default. See "Configuration Parameter Descriptions" on page 3-7.

translate_params (boolean; default: `false`)

Set this flag to `true` to override servlet containers that do not encode multibyte (globalization support) request parameters or bean property settings. With this setting, the JSP container encodes request parameters and bean property settings. Otherwise, it returns the parameters from the servlet container unchanged.

For more information about the functionality and use of `translate_params`, including situations where it should not be used, see "Multibyte Parameter Encoding in JServ" on page B-21.

Note: It is preferable to use the `PublicUtil.setReqCharacterEncoding()` method instead of using the older `translate_params` parameter. See "The setReqCharacterEncoding() Method" on page B-21.

---

unsafe_reload (boolean; default: `false`)

By default, the JSP container restarts the application and sessions whenever a JSP page is dynamically retranslated and reloaded (which occurs when there is a `.jsp` source file with a more recent timestamp than the corresponding page implementation class).

Set this parameter to `true` to instruct the JSP container to *not* restart the application after dynamic retranslations and reloads. This avoids having existing sessions become invalid. A `true` setting is appropriate for deployment environments. The `false` (default) setting is appropriate for development environments.

For a given JSP page, this parameter has no effect after the initial request for the page if `developer_mode` is set to `false` (in which case the JSP container never retranslates after the initial request).

xml_validate (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "Configuration Parameter Descriptions" on page 3-7.

Setting JSP Parameters in JServ

Each Web application in a JServ environment has its own properties file, known as a *zone properties file*. In Apache terminology, a *zone* is essentially the same as a servlet context.

The name of the zone properties file depends on how you mount the zone. (See the Apache JServ documentation for information about zones and mounting.)

To set JSP configuration parameters in a JServ environment, set the `JspServlet initArgs` property in the application zone properties file, as in the following example (which happens to use UNIX syntax):

```
servlet.oracle.jsp.JspServlet.initArgs=developer_mode=false,
sqljcmd=sqlj -user=scott/tiger -ser2class=true,classpath=/mydir/myapp.jar
```

(This is a single wraparound line.)

The servlet path, `servlet.oracle.jsp.JspServlet`, also depends on how you mount the zone. It does not represent a literal directory path.

Be aware of the following:

- The effects of multiple `initArgs` commands are cumulative and overriding. For example, consider the following two commands (in order):

  ```
  servlet.oracle.jsp.JspServlet.initArgs=foo1=val1,foo2=val2
  servlet.oracle.jsp.JspServlet.initArgs=foo1=val3
  ```

  This combination is equivalent to the following single command:

  ```
  servlet.oracle.jsp.JspServlet.initArgs=foo1=val3,foo2=val2
  ```

  In the first two commands, the `val3` value overrides the `val1` value for `foo1`, but does not affect the `foo2` setting.

- Because `initArgs` parameters are comma-separated, there can be no commas within a parameter setting. Spaces and other special characters (such as "=" in this example) do not cause a problem, however.

Using ojspc for JServ

Using `ojspc` for a JServ environment requires a different command. Use `ojspc_jserv` instead of `ojspc`. This executes the same class, `oracle.jsp.tool.Jspc`, but is set up with a classpath appropriate for JServ.

The `ojspc -staticTextInChars` option has no effect for JServ, because in a JServ environment static text is output as characters by default. You cannot disable this.

Important: To use `ojspc_jserv` (as with `ojspc`), you must be using a Sun Microsystems JDK (version 1.1.8 or higher) and you must have `tools.jar` (for JDK 2.0 or higher) or `classes.zip` (for JDK 1.1.8) in your classpath.

Considerations for the JServ Environment

There are special considerations in running JSP pages in the JServ environment, because it is a servlet 2.0 environment. The servlet 2.0 specification lacks support for some significant features that are available in servlet 2.2 and higher environments.

For information about how to configure a JServ environment for JSP pages, see "Getting Started in a JServ Environment" on page B-2.

This section discusses the following considerations for the JServ environment:

- The mod_jserv Apache Mod
- JSP Container Features for Application Root Support in JServ
- Overview of Application and Session Framework for JServ
- JSP and Servlet Session Sharing in JServ
- Dynamic Includes and Forwards in JServ
- JServ Directory Alias Translation
- Multibyte Parameter Encoding in JServ

The mod_jserv Apache Mod

The `mod_jserv` component, supplied by Apache, delegates HTTP requests to JSP pages or servlets running in the JServ servlet container in a middle-tier JVM. The middle-tier environment may or may not be on the same physical host as the back-end Oracle9*i* database.

Communication between `mod_jserv` and middle-tier JVMs uses the proprietary Apache JServ protocol (AJP) over TCP / IP. The `mod_jserv` component can delegate requests to multiple JVMs in a pool for load balancing.

Refer to Apache documentation for `mod_jserv` configuration information. This documentation is provided with Oracle9*i*AS.

JSP Container Features for Application Root Support in JServ

JServ and other servlet 2.0 environments have no concept of application roots, because there is only a single application environment. The Web server doc root is effectively the application root. By default, JSP pages and servlets running in the JServ environment of the Oracle9*i* Application Server, which are routed through the Apache `mod_jserv` module provided with JServ, use the Apache JServ doc root.

This is typically some `.../htdocs` directory. In addition, it is possible to specify "virtual" doc roots through `alias` settings in the `httpd.conf` configuration file.

The OC4J JSP container does, however, offer additional functionality regarding doc roots and application roots in the JServ environment. Through the OC4J JSP `globals.jsa` mechanism, you can designate a directory under the doc root to serve as an application root for any given application. This is accomplished by placing a `globals.jsa` file as a marker in the desired directory. See "Overview of globals.jsa Functionality" on page B-28 for more information.

The application root directory can be located in any of the following locations, listed in the order they are searched:

1. the Web server directory the application is mapped to
2. the Web server document root directory
3. the directory containing the `globals.jsa` file

Overview of Application and Session Framework for JServ

Because the concept of a Web application is not well defined in the servlet 2.0 specification, in JServ there is only one servlet context per servlet container. Additionally, there is only one session object per servlet container.

OC4J, however, supports a special application framework for use in the JServ environment. It accomplishes this through a file, `globals.jsa`, that you can use as an application marker. This allows distinct servlet contexts and session objects for each application.

For more information, see "Distinct Applications and Sessions Through globals.jsa" on page B-29.

JSP and Servlet Session Sharing in JServ

To share HTTP session information between JSP pages and servlets in a JServ environment, you must configure your environment so that `oracle.jsp.JspServlet`, the servlet that acts as the front-end of the JSP container in a JServ environment, is in the same zone as the servlet or servlets that you want your JSP pages to share a session with. Consult your Apache documentation for more information.

To verify proper zone setup, some browsers allow you to enable a warning for cookies. In an Apache environment, the cookie name includes the zone name.

Additionally, when you use a `globals.jsa` file for an application, presumably in a servlet 2.0 environment such as JServ, each JSP page uses a distinct JSP session wrapper attached to the single overall servlet session object provided by the servlet container.

In this situation, the `true` (default) setting of the JSP `session_sharing` configuration parameter results in JSP session data being propagated to the underlying servlet session. This allows servlets in the application to access the session data of JSP pages in the application.

If `session_sharing` is `false` (which parallels standard behavior in most JSP implementations), JSP session data is not propagated to the servlet session. As a result, application servlets would not be able to access JSP session data.

This parameter is meaningless if `globals.jsa` is not used. For information about `globals.jsa`, see "JSP Application and Session Support for JServ" on page B-28.

Also see these sections for related information:

- "JSP Application and Session Support for JServ" on page B-28
- "JSP Configuration Parameters for JServ" on page B-4
- "Setting JSP Parameters in JServ" on page B-13

Dynamic Includes and Forwards in JServ

JSP dynamic includes (using the `jsp:include` tag) and forwards (using the `jsp:forward` tag) rely on request dispatcher functionality that is present in servlet 2.2 and higher environments, but not in servlet 2.0 environments.

The OC4J JSP container, however, provides extended functionality to allow dynamic includes and forwards from one JSP page to another JSP page or to a static HTML file in JServ and other servlet 2.0 environments.

This functionality for servlet 2.0 environments does not, however, allow dynamic forwards or includes to servlets. (Servlet execution is controlled by the JServ or other servlet container, not the JSP container.)

If you want to include or forward to a servlet in JServ, however, you can create a JSP page that acts as a wrapper for the servlet.

The following example shows a servlet, and a JSP page that acts as a wrapper for that servlet. In a JServ environment, you can effectively include or forward to the servlet by including or forwarding to the JSP wrapper page.

Servlet Code Presume that you want to include or forward to the following servlet:

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

public class TestServlet extends HttpServlet {

    public void init(ServletConfig config) throws ServletException
    {
      super.init(config);
      System.out.println("initialized");
    }

    public void destroy()
    {
      System.out.println("destroyed");
    }

    public void service
        (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><BODY>");
        out.println("TestServlet Testing");
        out.println("<H3>The local time is: "+ new java.util.Date());
        out.println("</BODY></HTML>");
    }
}
```

JSP Wrapper Page Code You can create the following JSP wrapper (`wrapper.jsp`) for the preceding servlet.

```
<%-- wrapper.jsp--wraps TestServlet for JSP include/forward --%>
<%@ page isThreadSafe="true" import="TestServlet" %>
<%!
  TestServlet s=null;
  public void jspInit() {
    s=new TestServlet();
    try {
      s.init(this.getServletConfig());
    } catch (ServletException se)
    {
```

```
      s=null;
    }
  }
  public void jspDestroy() {
    s.destroy();
  }
%>
<% s.service(request,response); %>
```

Including or forwarding to `wrapper.jsp` in a servlet 2.0 environment has the same effect as directly including or forwarding to `TestServlet` in a servlet 2.2 or higher environment.

Notes Regarding Dynamic Includes and Forwards

- Whether to set `isThreadSafe` to `true` or `false` in the wrapper JSP page depends on whether the original servlet is thread-safe.
- As an alternative to using a wrapper JSP page for this situation, you can add HTTP client code to the original JSP page (the one from which the `jsp:include` action or `jsp:forward` action is to occur). You can use an instance of the standard `java.net.URL` class to create an HTTP request from the original JSP page to the servlet. (Note that you cannot share session data or security credentials in this scenario.)

JServ Directory Alias Translation

Apache JServ supports directory aliasing by allowing you to create a "virtual directory" through an `Alias` command in the `httpd.conf` configuration file. This allows Web documents to be placed outside the default doc root directory.

Consider the following sample `httpd.conf` entry:

```
Alias /icons/ "/apache/apache139/icons/"
```

This command results in `icons` being usable as an alias for the `/apache/apache139/icons/` path. In this way, for example, the file `/apache/apache139/icons/art.gif`, could be accessed by the following URL:

`http://`*`host`*`[:`*`port`*`]/icons/art.gif`

Currently, however, this functionality does not work properly for servlets and JSP pages, because the Apache JServ `getRealPath()` method returns an incorrect value when processing a file under an alias directory.

The OC4J JSP container supports an Apache-specific configuration parameter, `alias_translation`, that works around this limitation when you set it to `true`. (The default setting is `false`.)

Be aware that setting `alias_translation=true` also results in the alias directory becoming the application root. Therefore, in a `jsp:include` or `jsp:forward` tag where the target file name starts with "/", the expected target file location will be relative to the alias directory.

Consider the following example, which results in all JSP and HTML files under `/private/foo` being effectively under the application `/mytest`:

```
Alias /mytest/ "/private/foo/"
```

And assume there is a JSP page located as follows:

```
/private/foo/xxx.jsp
```

The following `jsp:include` tag will work, because `xxx.jsp` is directly below the aliased directory, `/private/foo`, which is effectively the application root:

```
<jsp:include page="/xxx.jsp" flush="true" />
```

JSP pages in other applications or in the general doc root cannot forward to or include JSP pages or HTML files under the `/mytest` application. It is possible to forward to or include pages or HTML files only within the same application (per the servlet 2.2 specification).

Notes:

- An implicit application is created for the Web server document root and each aliasing root.
- For information about how to set JSP configuration parameters in a JServ environment, see "Setting JSP Parameters in JServ" on page B-13.

Multibyte Parameter Encoding in JServ

This section describes Oracle extensions to support multibyte request parameters and bean property settings in a JServ or other servlet 2.0 environment, such as for a `getParameter()` call in Java code or for a `jsp:setProperty` tag to set a bean property in JSP code. There are two mechanisms for this:

- `oracle.jsp.util.PublicUtil.setReqCharacterEncoding()` static method (preferred)
- `translate_params` configuration parameter (or equivalent code)

The discussion of `translate_params` is followed by a discussion of how to migrate away from its use when you move to an OC4J environment.

For general information about multibyte parameter encoding, see "JSP Support for Multibyte Parameter Encoding" on page 8-6.

The setReqCharacterEncoding() Method

For pre-2.3 servlet environments, Oracle provides a `setReqCharacterEncoding()` method that is useful in case the default encoding for the servlet container is not appropriate. Use this method to specify the encoding of multibyte request parameters and bean property settings, such as for a `getParameter()` call in Java code or a `jsp:setProperty` tag to set a bean property in JSP code. If the default encoding is already appropriate, then it is not necessary to use this method, and in fact using it may create some performance overhead in your application.

The `setReqCharacterEncoding()` method is a static method in the `PublicUtil` class of the `oracle.jsp.util` package, with the following signature:

```
public static void setReqCharacterEncoding
                  (HttpServletRequest req, String encoding)
                  throws java.io.UnsupportedEncodingException
```

This method affects parameter names and values, specifically:

- request object `getParameter()` method output
- request object `getParameterValues()` method output
- request object `getParameterNames()` method output
- `jsp:setProperty` settings for bean property values

When invoking the method, input a request object and a string that specifies the desired encoding, as follows:

```
oracle.jsp.util.PublicUtil.setReqCharacterEncoding(request, "EUC-JP");
```

Notes: The `setReqCharacterEncoding()` method is forward-compatible with the method `request.setCharacterEncoding(encoding)` of the servlet 2.3 API.

JSP translate_params Configuration Parameter

Set this boolean flag to `true` to override servlet containers that do not encode multibyte (globalization support) request parameters or bean property settings. (The default setting is `false`.) With a `true` setting, the JSP container decodes and encodes request parameters and bean property settings. Otherwise, it returns the parameters from the servlet container unchanged.

Note that you should *not* enable `translate_params` in any of the following circumstances:

- when the servlet container properly handles multibyte parameter encoding itself

  Setting `translate_params` to `true` in this situation may cause incorrect results. As of this writing, however, it is known that JServ does *not* properly handle multibyte parameter encoding.

- when the request parameters use a different encoding from what is specified for the response in the JSP `page` directive or `setContentType()` method

- when code with workaround functionality equivalent to what `translate_params` accomplishes is already present in the JSP page

  (See "Code Equivalent to the translate_params Configuration Parameter" on page B-23.)

Effect of translate_params in Overriding Non-Multibyte Servlet Containers

Setting `translate_params` to `true` overrides the insufficient functionality of servlet containers that cannot decode and encode multibyte request parameters and bean property settings. (For information about how to set JSP configuration parameters, see "Setting JSP Parameters in JServ" on page B-13.)

When this flag is enabled, the JSP container encodes the request parameters and bean property settings based on the character set of the `response` object, as indicated by the `response.getCharacterEncoding()` method.

Code Equivalent to the translate_params Configuration Parameter

There may be situations where you do not want to or cannot use the `translate_params` configuration parameter. It is useful to be aware of equivalent functionality that you can implement through scriptlet code in the JSP page, for example:

```
<%@ page contentType="text/html; charset=EUC-JP" %>
...
String paramName="XXYYZZ";       // where XXYYZZ is a multibyte string
paramName =
   new String(paramName.getBytes(response.getCharacterEncoding()), "ISO8859_1");
String paramValue = request.getParameter(paramName);
paramValue= new String(paramValue.getBytes("ISO8859_1"), "EUC-JP");
...
```

This code accomplishes the following:

- It sets `XXYYZZ` as the parameter name to search for. (Presume `XX`, `YY`, and `ZZ` are three Japanese characters.)
- It encodes the parameter name to `ISO-8859-1`, the servlet container character set, so that the servlet container can interpret it. (First a byte array is created for the parameter name, using the character encoding of the request object.)
- It gets the parameter value from the request object by looking for a match for the parameter name. (It is able to find a match because parameter names in the request object are also in `ISO-8859-1` encoding.)
- It encodes the parameter value to `EUC-JP` for further processing or output to the browser.

See the next two sections for a globalization sample that depends on `translate_params` being enabled, and one that contains the equivalent code so that it does not depend on the `translate_params` setting.

Globalization Sample Depending on translate_params

The following sample accepts a user name in Japanese characters and correctly outputs the name back to the browser. In a servlet environment that cannot encode multibyte request parameters, this sample depends on the JSP configuration setting of `translate_params=true`.

Presume XXYY is the parameter name (something equivalent to "user name" in Japanese) and AABB is the default value (also in Japanese).

(See the next section for a sample that has the code equivalent of the `translate_params` functionality, thereby not depending on the `translate_params` setting.)

```
<%@ page contentType="text/html; charset=EUC-JP" %>

<HTML>
<HEAD>
<TITLE>Hello</TITLE></HEAD>
<BODY>
<%
   //charset is as specified in page directive (EUC-JP)
   String charset = response.getCharacterEncoding();
%>
   <BR> encoding = <%= charset %> <BR>

<%

String paramValue = request.getParameter("XXYY");

if (paramValue == null || paramValue.length() == 0) { %>
   <FORM METHOD="GET">
   Please input your name: <INPUT TYPE="TEXT" NAME="XXYY" value="AABB" size=20>
<BR>
   <INPUT TYPE="SUBMIT">
   </FORM>
<% }
else
{ %>
   <H1> Hello, <%= paramValue %> </H1>
<% } %>
</BODY>
</HTML>
```

Following is the sample input:

and the sample output:

Globalization Sample Not Depending on translate_params

The following sample, as with the preceding sample, accepts a user name in Japanese characters and correctly outputs the name back to the browser. This sample, however, has the code equivalent of `translate_params` functionality, so does not depend on the `translate_params` setting.

> **Important:** If you use `translate_params`-equivalent code, do *not* also enable the `translate_params` flag. This may cause incorrect results.

Presume XXYY is the parameter name (something equivalent to "user name" in Japanese) and AABB is the default value (also in Japanese).

For an explanation of the critical code in this sample, see "Code Equivalent to the translate_params Configuration Parameter" on page B-23.

```
<%@ page contentType="text/html; charset=EUC-JP" %>

<HTML>
<HEAD>
<TITLE>Hello</TITLE></HEAD>
<BODY>
<%
   //charset is as specified in page directive (EUC-JP)
   String charset = response.getCharacterEncoding();
%>
   <BR> encoding = <%= charset %> <BR>
<%
String paramName = "XXYY";

paramName = new String(paramName.getBytes(charset), "ISO8859_1");

String paramValue = request.getParameter(paramName);

if (paramValue == null || paramValue.length() == 0) { %>
   <FORM METHOD="GET">
   Please input your name: <INPUT TYPE="TEXT" NAME="XXYY" value="AABB" size=20>
<BR>
   <INPUT TYPE="SUBMIT">
   </FORM>
<% }
else
{
```

```
   paramValue= new String(paramValue.getBytes("ISO8859_1"), "EUC-JP"); %>
   <H1> Hello, <%= paramValue %> </H1>
<% } %>
</BODY>
</HTML>
```

Migration Away from translate_params

The new "global includes" functionality in the OC4J JSP container in Oracle9*i*AS 9.0.2, described in "Oracle JSP Global Includes" on page 6-9, is useful in migrating applications that have previously used `translate_params` for globalization.

In this case, the globally included file can consist of a scriptlet similar to one of the following to achieve functionality that is equivalent to that of `translate_params`:

- Hardcode the request character set:

  ```
  <% request.setCharacterEncoding("desired_charset"); %>
  ```

or:

- Use the character set of the response as the character set of the request, where the character set of the response is specified in the `contentType` attribute of a JSP `page` directive:

  ```
  <% request.setCharacterEncoding(response.getCharacterEncoding()); %>
  ```

or:

- Use the character set of the response as the character set of the request, where the character set of the response is determined dynamically by Java logic:

  ```
  <% String yourCharSet = yourLogicToDetermineCharset();
     response.setContentType("text/html; charset="+yourCharSet);
     request.setCharacterEncoding(response.getCharcterEncoding());
     // NOTE: The relative ordering of response.setContentType()
     // and request.setCharacterEncoding() is important.
  %>
  ```

JSP Application and Session Support for JServ

OC4J supports a file, `globals.jsa`, as a mechanism for implementing the JSP specification in a servlet 2.0 environment. Web applications and servlet contexts are not fully defined in the servlet 2.0 specification.

This section discusses the `globals.jsa` mechanism and covers the following topics, including information about migrating away from `globals.jsa` when you move to an OC4J environment:

- Overview of globals.jsa Functionality
- Overview of globals.jsa Syntax and Semantics
- The globals.jsa Event-Handlers
- Global Declarations and Directives
- Migration from globals.jsa

For sample applications, see "Samples Using globals.jsa for Servlet 2.0 Environments" on page B-41.

Important: Use all lowercase for the `globals.jsa` file name. Mixed case works in a non-case-sensitive environment, but makes it difficult to diagnose resulting problems if you port the pages to a case-sensitive environment.

Overview of globals.jsa Functionality

Within any single Java virtual machine, you can use a `globals.jsa` file for each application (or, equivalently, for each servlet context). This file supports the concept of Web applications in the following areas:

- application deployment—through its role as an application location marker to define an application root
- distinct applications and sessions—through its use in providing distinct servlet context and session objects for each application
- application lifecycle management—through start and end events for sessions and applications

The `globals.jsa` file also provides a vehicle for global Java declarations and JSP directives across all JSP pages of an application.

Application Deployment through globals.jsa

To deploy a JSP application that does not incorporate servlets, copy the directory structure into the Web server, and create a file called `globals.jsa` to place at the application root directory.

The `globals.jsa` file can be of zero size. The JSP container will locate it, and its presence in a directory defines that directory (as mapped from the URL virtual path) as the root directory of the application.

The JSP container also defines default locations for JSP application resources. For example, application beans and classes in the application-relative `/WEB-INF/classes` and `/WEB-INF/lib` directories will automatically be loaded by the JSP classloader without the need for specific configuration.

Notes: For an application that *does* incorporate servlets, especially in a servlet environment preceding the servlet 2.2 specification, manual configuration is required as with any servlet deployment.

Distinct Applications and Sessions Through globals.jsa

The servlet 2.0 specification does not have a clearly defined concept of a Web application and there is no defined relationship between servlet contexts and applications, as there is in later servlet specifications. In a servlet 2.0 environment such as JServ, there is only one servlet context object per JVM. A servlet 2.0 environment also has only one session object.

The `globals.jsa` file, however, provides support for multiple applications and multiple sessions in a Web server, particularly for use in a servlet 2.0 environment.

Where a distinct servlet context object would not otherwise be available for each application, the presence of a `globals.jsa` file for an application allows the JSP container to provide the application with a distinct `ServletContext` object.

Additionally, where there would otherwise be only one session object (with either one servlet context or across multiple servlet contexts), the presence of a `globals.jsa` file allows the JSP container to provide a proxy `HttpSession` object to the application. This prevents the possibility of session variable-name collisions with other applications, although unfortunately it cannot protect application data from being inspected or modified by other applications. This is because `HttpSession` objects must rely on the underlying servlet session environment for some of their functionality.

Application and Session Lifecycle Management Through globals.jsa

An application must be notified when a significant state transition occurs. For example, applications often want to acquire resources when an HTTP session begins and release resources when the session ends, or restore or save persistent data when the application itself is started or terminated.

In standard servlet and JSP technology, however, only session-based events are supported.

For applications that use a `globals.jsa` file, this functionality is extended with the following four events:

- `session_OnStart`
- `session_OnEnd`
- `application_OnStart`
- `application_OnEnd`

You can write event handlers in the `globals.jsa` file for any of these events that the server should respond to.

The `session_OnStart` event and `session_OnEnd` event are triggered at the beginning and end of an HTTP session, respectively.

The `application_OnStart` event is triggered for any application by the first request for that application within any single JVM. The `application_OnEnd` event is triggered when the JSP container unloads an application.

For more information, see "The globals.jsa Event-Handlers" on page B-32.

Overview of globals.jsa Syntax and Semantics

This section is an overview of general syntax and semantics for a `globals.jsa` file.

Each event block in a `globals.jsa` file—a `session_OnStart` block, a `session_OnEnd` block, an `application_OnStart` block, or an `application_OnEnd` block—has an event start tag, an event end tag, and a body (everything between the start and end tags) that includes the event-handler code. The following example shows this pattern:

```
<event:session_OnStart>
   <% This scriptlet contains the implementation of the event handler %>
</event:session_OnStart>
```

The body of an event block can contain any valid JSP tags—standard tags as well as tags defined in a custom tag library.

The scope of any JSP tag in an event block, however, is limited to only that block. For example, a bean that is declared in a `jsp:useBean` tag within one event block must be redeclared in any other event block that uses it. You can avoid this restriction, however, through the `globals.jsa` global declaration mechanism—see "Global Declarations and Directives" on page B-37.

For details about each of the four event handlers, see "The globals.jsa Event-Handlers" on page B-32.

Important: Static text as used in a regular JSP page can reside in a `session_OnStart` block only. Event blocks for `session_OnEnd`, `application_OnStart`, and `application_OnEnd` can contain only Java scriptlets.

JSP implicit objects are available in `globals.jsa` event blocks as follows:

- The `application_OnStart` block has access to the `application` object.
- The `application_OnEnd` block has access to the `application` object.
- The `session_OnStart` block has access to the `application`, `session`, `request`, `response`, `page`, and `out` objects.
- The `session_OnEnd` block has access to the `application` and `session` objects.

Example of a Complete globals.jsa File This example shows you a complete `globals.jsa` file, using all four event handlers.

```
<event:application_OnStart>

   <%-- Initializes counts to zero --%>
   <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

</event:application_OnStart>

<event:application_OnEnd>

   <%-- Acquire beans --%>
   <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
```

```
   <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
   <% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>

</event:application_OnEnd>

<event:session_OnStart>

   <%-- Acquire beans --%>
   <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <%
      sessionCount.setValue(sessionCount.getValue() + 1);
      activeSessions.setValue(activeSessions.getValue() + 1);
   %>
   <br>
   Starting session #: <%=sessionCount.getValue() %> <br>
   There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>

</event:session_OnStart>

<event:session_OnEnd>

   <%-- Acquire beans --%>
   <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <%
      activeSessions.setValue(activeSessions.getValue() - 1);
   %>

</event:session_OnEnd>
```

The globals.jsa Event-Handlers

This section provides details about each of the four `globals.jsa` event-handlers.

application_OnStart

The `application_OnStart` block has the following general syntax:

```
<event:application_OnStart>
   <% This scriptlet contains the implementation of the event handler %>
</event:application_OnStart>
```

The body of the `application_OnStart` event handler is executed when the JSP container loads the first JSP page in the application. This usually occurs when the first HTTP request is made to any page in the application, from any client. Applications use this event to initialize application-wide resources, such as a database connection pool or data read from a persistent repository into application objects.

The event handler must contain only JSP tags (including custom tags)—it cannot contain static text.

Errors that occur in this event handler but are not processed in the event-handler code are automatically trapped by the JSP container and logged using the servlet context of the application. Event handling then proceeds as if no error had occurred.

Example: application_OnStart The following `application_OnStart` example is from the "A globals.jsa Example for Application Events—lotto.jsp" on page B-41. In this example, the generated lottery numbers for a particular user are cached for an entire day. If the user re-requests the picks, he or she gets the same set of numbers. The cache is recycled once a day, giving each user a new set of picks. To function as intended, the lotto application must make the cache persistent when the application is being shut down, and must refresh the cache when the application is reactivated.

The `application_OnStart` event handler reads the cache from the `lotto.che` file.

```
<event:application_OnStart>
<%
        Calendar today = Calendar.getInstance();
        application.setAttribute("today", today);
        try {
                FileInputStream fis = new FileInputStream
                            (application.getRealPath("/")+File.separator+"lotto.che");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Calendar cacheDay = (Calendar) ois.readObject();
                if (cacheDay.get(Calendar.DAY_OF_YEAR) == today.get(Calendar.DAY_OF_YEAR)) {
                        cachedNumbers = (Hashtable) ois.readObject();
                        application.setAttribute("cachedNumbers", cachedNumbers);
                }
                ois.close();
        } catch (Exception theE) {
                // catch all -- can't use persistent data
        }
%>
</event:application_OnStart>
```

application_OnEnd

The `application_OnEnd` block has the following general syntax:

```
<event:application_OnEnd>
   <% This scriptlet contains the implementation of the event handler %>
</event:application_OnEnd>
```

The body of the `application_OnEnd` event handler is executed when the JSP container unloads the JSP application. Unloading occurs whenever a previously loaded page is reloaded after on-demand dynamic re-translation (unless the JSP `unsafe_reload` configuration parameter is enabled), or when the JSP container, which itself is a servlet, is terminated by having its `destroy()` method called by the underlying servlet container. Applications use the `application_OnEnd` event to clean up application level resources or to write application state to a persistent store.

The event handler must contain only JSP tags (including custom tags)—it cannot contain static text.

Errors that occur in this event handler but are not processed in the event-handler code are automatically trapped by the JSP container and logged using the servlet context of the application. Event handling then proceeds as if no error had occurred.

Example: application_OnEnd The following `application_OnEnd` example is from the "A globals.jsa Example for Application Events—lotto.jsp" on page B-41. In this event handler, the cache is written to file `lotto.che` before the application is terminated.

```
<event:application_OnEnd>

<%
        Calendar now = Calendar.getInstance();
        Calendar today = (Calendar) application.getAttribute("today");
        if (cachedNumbers.isEmpty() ||
                   now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
                File f = new File(application.getRealPath("/")+File.separator+"lotto.che");
                if (f.exists()) f.delete();
                return;
        }

        try {
                FileOutputStream fos = new FileOutputStream
                            (application.getRealPath("/")+File.separator+"lotto.che");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(today);
```

```
                oos.writeObject(cachedNumbers);
                oos.close();
        } catch (Exception theE) {
                // catch all -- can't use persistent data
        }
%>

</event:application_OnEnd>
```

session_OnStart

The `session_OnStart` block has the following general syntax:

```
<event:session_OnStart>
   <% This scriptlet contains the implementation of the event handler %>
   Optional static text...
</event:session_OnStart>
```

The body of the `session_OnStart` event handler is executed when the JSP container creates a new session in response to a JSP page request. This occurs on a per client basis, whenever the first request is received for a session-enabled JSP page in an application.

Applications might use this event for the following purposes:

- to initialize resources tied to a particular client
- to control where a client starts in an application

Because the implicit `out` object is available to `session_OnStart`, this is the only `globals.jsa` event handler that can contain static text in addition to JSP tags.

The `session_OnStart` event handler is called before the code of the JSP page is executed. As a result, output from `session_OnStart` precedes any output from the page.

The `session_OnStart` event handler and the JSP page that triggered the event share the same `out` stream. The buffer size of this stream is controlled by the buffer size of the JSP page. The `session_OnStart` event handler does not automatically flush the stream to the browser—the stream is flushed according to general JSP rules. Headers can still be written in JSP pages that trigger the `session_OnStart` event.

Errors that occur in this event handler but are not processed in the event-handler code are automatically trapped by the JSP container and logged using the servlet context of the application. Event handling then proceeds as if no error had occurred.

Example: session_OnStart The following example makes sure that each new session starts on the initial page (`index.jsp`) of the application.

```
<event:session_OnStart>

   <% if (!page.equals("index.jsp")) { %>
         <jsp:forward page="index.jsp" />
   <% } %>

</event:session_OnStart>
```

session_OnEnd

The `session_OnEnd` block has the following general syntax:

```
<event:session_OnEnd>
   <% This scriptlet contains the implementation of the event handler %>
</event:session_OnEnd>
```

The body of the `session_OnEnd` event handler is executed when the JSP container invalidates an existing session. This occurs in either of the following circumstances:

- The application invalidates the session by calling the `session.invalidate()` method.
- The session expires ("times out") on the server.

Applications use this event to release client resources.

The event handler must contain only JSP tags (including tag library tags)—it cannot contain static text.

Errors that occur in this event handler but are not processed in the event-handler code are automatically trapped by the JSP container and logged using the servlet context of the application. Event handling then proceeds as if no error had occurred.

Example: session_OnEnd The following example decrements the "active session" count when a session is terminated.

```
<event:session_OnEnd>
     <%-- Acquire beans --%>
     <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
     <%
         activeSessions.setValue(activeSessions.getValue() - 1);
     %>
 </event:session_OnEnd>
```

Global Declarations and Directives

In addition to holding event handlers, a `globals.jsa` file can be used to globally declare directives and objects for the JSP application. You can include JSP directives, JSP declarations, JSP comments, and JSP tags that have a `scope` parameter (such as `jsp:useBean`).

This section covers the following topics:

Global JSP Directives

Directives used within a `globals.jsa` file serve a dual purpose:

- They declare the information that is required to process the `globals.jsa` file itself.
- They establish default values for succeeding pages.

A directive in a `globals.jsa` file becomes an implicit directive for all JSP pages in the application, although a `globals.jsa` directive can be overwritten for any particular page.

A `globals.jsa` directive is overwritten in a JSP page on an attribute-by-attribute basis. If a `globals.jsa` file has the following directive:

```
<%@ page import="java.util.*" bufferSize="10kb" %>
```

and a JSP page has the following directive:

```
<%@page bufferSize="20kb" %>
```

then this would be equivalent to the page having the following directive:

```
<%@ page import="java.util.*" bufferSize="20kb" %>
```

globals.jsa Declarations

If you want to declare a method or data member to be shared across any of the event handlers in a `globals.jsa` file, use a JSP `<%! ... %>` declaration within the `globals.jsa` file.

Note that JSP pages in the application do not have access to these declarations, so you cannot use this mechanism to implement an application library. Declaration support is provided in the `globals.jsa` file for common functions to be shared across event handlers.

Global JavaBeans

Probably the most common elements declared in `globals.jsa` files are global objects. Objects declared in a `globals.jsa` file become part of the implicit object environment of the `globals.jsa` event handlers and all the JSP pages in the application.

An object that is declared in a `globals.jsa` file (such as by a `jsp:useBean` tag) need not be redeclared in any of the individual JSP pages of the application.

You can declare a global object using any JSP tag or extension that has a `scope` parameter, such as the standard `jsp:useBean` tag or the JML `useVariable` tag. Globally declared objects must be of either `session` or `application` scope (not `page` or `request` scope).

Nested tags are supported. Thus, a `jsp:setProperty` tag can be nested in a `jsp:useBean` tag. (A translation error occurs if `jsp:setProperty` is used outside a `jsp:useBean` tag.)

globals.jsa Structure

When a global object is used in a `globals.jsa` event handler, the position of its declaration is important. Only those objects that are declared before a particular event handler are added as implicit objects to that event handler. For this reason, developers are advised to structure their `globals.jsa` file in the following sequence:

1. global directives
2. global objects
3. event handlers
4. `globals.jsa` declarations

Global Declarations and Directives Example

The sample `globals.jsa` file below accomplishes the following:

- It defines the JML tag library (in this case, the compile-time implementation) for the `globals.jsa` file, as well as for all subsequent pages. By including the `taglib` directive in the `globals.jsa` file, the directive does not have to be included in any of the individual JSP pages of the application.
- It declares three application variables for use by all pages (in the `jsp:useBean` statements).

For an additional example of using `globals.jsa` for global declarations, see "A globals.jsa Example for Global Declarations—index2.jsp" on page B-47.

```
<%-- Directives at the top --%>

   <%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>

<%-- Declare global objects here --%>

   <%-- Initializes counts to zero --%>
   <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<%-- Application lifecycle event handlers go here --%>

   <event:application_OnStart>
      <% This scriptlet contains the implementation of the event handler %>
   </event:application_OnStart>

   <event:application_OnEnd>
      <% This scriptlet contains the implementation of the event handler %>
   </event:application_OnEnd>

   <event:session_OnStart>
      <% This scriptlet contains the implementation of the event handler %>
   </event:session_OnStart>

   <event:session_OnEnd>
      <% This scriptlet contains the implementation of the event handler %>
   </event:session_OnEnd>

<%-- Declarations used by the event handlers go here --%>
```

Migration from globals.jsa

The OC4J JSP front-end servlet in Oracle9*i*AS release 2 no longer supports `globals.jsa`. If an existing application uses `globals.jsa`, you should migrate away from this usage. The following substitutions for `globals.jsa` functionality are recommended:

- Instead of using `globals.jsa` as an application marker, use standard WAR packaging to denote the application structure.
- Instead of using `globals.jsa` start-session, end-session, start-application, and end-application events, use standard servlet 2.3 listener functionality. For example, equivalent capabilities are offered through the standard `javax.servlet.ServletContextListener` and `javax.servlet.http.HttpSessionListener` interfaces.
- Instead of using `globals.jsa` for global variable declarations, make the declarations in a single source file and use "global include" functionality of the OC4J JSP engine, introduced in Oracle9*i*AS 9.0.2. See "Oracle JSP Global Includes" on page 6-9.

If you cannot migrate your code immediately, an application that uses `globals.jsa` can still run in OC4J if you use the previous `oracle.jsp.JspServlet` front-end servlet instead of the new `oracle.jsp.runtimev2.JspServlet` front-end. You can specify this in the `<servlet>` element in the application `web.xml` file, which overrides definitions in the OC4J `global-web-application.xml` file. This should be for short-term use only, however, given that the new front-end servlet supports many new features and configuration parameters and offers improved performance.

Samples Using globals.jsa for Servlet 2.0 Environments

This section has examples of how the Oracle `globals.jsa` mechanism can be used in servlet 2.0 environments to provide an application framework and application-based and session-based event handling. The following examples are provided:

- A globals.jsa Example for Application Events—lotto.jsp
- A globals.jsa Example for Application and Session Events—index1.jsp
- A globals.jsa Example for Global Declarations—index2.jsp

For information about `globals.jsa` usage, see "JSP Application and Session Support for JServ" on page B-28.

Note: The examples in this section base some of their functionality on application shutdown. Many servers do not allow an application to be shut down manually. In this case, `globals.jsa` cannot function as an application marker. However, you can cause the application to be automatically shut down and restarted (presuming `developer_mode=true`) by updating either the `lotto.jsp` source or the `globals.jsa` file. (The JSP container always terminates a running application before retranslating and reloading an active page.)

A globals.jsa Example for Application Events—lotto.jsp

This sample illustrates `globals.jsa` event handling through the `application_OnStart` and `application_OnEnd` event handlers. In this sample, numbers are cached on a per-user basis for the duration of the day. As a result, only one set of numbers is ever presented to a user for a given lottery drawing. In this sample, users are identified by their IP addresses.

Code has been written for `application_OnStart` and `application_OnEnd` to make the cache persistent across application shutdowns. The sample writes the cached data to a file as it is being terminated and reads from the file as it is being restarted (presuming the server is restarted the same day that the cache was written).

globals.jsa File for lotto.jsp

```
<%@ page import="java.util.*, oracle.jsp.jml.*" %>

<jsp:useBean id = "cachedNumbers" class = "java.util.Hashtable" scope = "application" />

<event:application_OnStart>

<%
        Calendar today = Calendar.getInstance();
        pageContext.setAttribute("today", today, PageContext.APPLICATION_SCOPE);
        try {
                FileInputStream fis = new FileInputStream
                            (application.getRealPath("/")+File.separator+"lotto.che");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Calendar cacheDay = (Calendar) ois.readObject();
                if (cacheDay.get(Calendar.DAY_OF_YEAR) == today.get(Calendar.DAY_OF_YEAR)) {
                        cachedNumbers = (Hashtable) ois.readObject();
                        pageContext.setAttribute(
                            "cachedNumbers", cachedNumbers, PageContext.APPLICATION_SCOPE);
                }
                ois.close();
        } catch (Exception theE) {
                // catch all -- can't use persistent data
        }
%>

</event:application_OnStart>

<event:application_OnEnd>

<%
        Calendar now = Calendar.getInstance();
        Calendar today = (Calendar) application.getAttribute("today");
        if (cachedNumbers.isEmpty() ||
                   now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
                File f = new File(application.getRealPath("/")+File.separator+"lotto.che");
                if (f.exists()) f.delete();
                return;
        }

        try {
                FileOutputStream fos = new FileOutputStream
                            (application.getRealPath("/")+File.separator+"lotto.che");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(today);
```

```
                oos.writeObject(cachedNumbers);
                oos.close();
        } catch (Exception theE) {
                // catch all -- can't use persistent data
        }
%>

</event:application_OnEnd>
```

lotto.jsp Source

```
<%@ page session = "false" %>
<jsp:useBean id = "picker" class = "oracle.jsp.sample.lottery.LottoPicker" scope = "page" />

<HTML>
<HEAD><TITLE>Lotto Number Generator</TITLE></HEAD>
<BODY BACKGROUND="images/cream.jpg" BGCOLOR="#FFFFFF">
<H1 ALIGN="CENTER"></H1>

<BR>

<!-- <H1 ALIGN="CENTER"> IP: <%= request.getRemoteAddr() %> <BR> -->
<H1 ALIGN="CENTER">Your Specially Picked</H1>
<P ALIGN="CENTER"><IMG SRC="images/winningnumbers.gif" WIDTH="450" HEIGHT="69" ALIGN="BOTTOM"
BORDER="0"></P>
<P>

<P ALIGN="CENTER">
<TABLE ALIGN="CENTER" BORDER="0" CELLPADDING="0" CELLSPACING="0">
<TR>
<%
        int[] picks;
        String identity = request.getRemoteAddr();

        // Make sure its not tomorrow
        Calendar now = Calendar.getInstance();
        Calendar today = (Calendar) application.getAttribute("today");
        if (now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
                System.out.println("New day....");
                cachedNumbers.clear();
                today = now;
                pageContext.setAttribute("today", today, PageContext.APPLICATION_SCOPE);
        }
```

```
        synchronized (cachedNumbers) {
                if ((picks = (int []) cachedNumbers.get(identity)) == null) {
                        picks = picker.getPicks();
                        cachedNumbers.put(identity, picks);
                }
        }
        for (int i = 0; i < picks.length; i++) {
%>
     <TD>
     <IMG SRC="images/ball<%= picks[i] %>.gif" WIDTH="68" HEIGHT="76" ALIGN="BOTTOM" BORDER="0">
     </TD>

<%
     }
%>
</TR>
</TABLE>

</P>

<P ALIGN="CENTER"><BR>
<BR>
<IMG SRC="images/playrespon.gif" WIDTH="120" HEIGHT="73" ALIGN="BOTTOM" BORDER="0">

</BODY>
</HTML>
```

A globals.jsa Example for Application and Session Events—index1.jsp

This example uses a `globals.jsa` file to process applications and session lifecycle events. It counts the number of active sessions, the total number of sessions, and the total number of times the application page has been hit. Each of these values is maintained at the `application` scope. The application page (`index1.jsp`) updates the page hit count on each request. The `globals.jsa session_OnStart` event handler increments the number of active sessions and the total number of sessions. The `globals.jsa session_OnEnd` handler decrements the number of active sessions by one.

The page output is simple. When a new session starts, the session counters are output. The page counter is output on every request. The final tally of each value is output in the `globals.jsa application_OnEnd` event handler.

Note the following in this example:

- When the counter variables are updated, access must be synchronized, because these values are maintained at `application` scope.
- The count values use the `oracle.jsp.jml.JmlNumber` extended datatype, which simplifies the use of data values at application scope. (For information about the JML extended datatypes, refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.)

globals.jsa File for index1.jsp

```
<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>

<event:application_OnStart>

      <%-- Initializes counts to zero --%>
      <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
      <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
       <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

      <%-- Consider storing pageCount persistently -- If you do read it here --%>

</event:application_OnStart>

<event:application_OnEnd>
      <%-- Acquire beans --%>
      <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
      <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />

      <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
       <% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>

      <%-- Consider storing pageCount persistently -- If you do write it here --%>

</event:application_OnEnd>

<event:session_OnStart>

      <%-- Acquire beans --%>
      <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
      <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

      <%
        synchronized (sessionCount) {
```

```
                sessionCount.setValue(sessionCount.getValue() + 1);
      %>
                <br>
                Starting session #: <%= sessionCount.getValue() %> <br>
      <%
        }
      %>

      <%
        synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() + 1);
      %>
                There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>
      <%
        }
      %>

</event:session_OnStart>

<event:session_OnEnd>

      <%-- Acquire beans --%>
      <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

      <%
         synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() - 1);
        }
      %>

</event:session_OnEnd>
```

index1.jsp Source

```
<%-- Acquire beans --%>
<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<%
   synchronized(pageCount) {
        pageCount.setValue(pageCount.getValue() + 1);
   }
%>
This page has been accessed <b> <%= pageCount.getValue() %> </b>  times.
<p>
```

A globals.jsa Example for Global Declarations—index2.jsp

This example uses a `globals.jsa` file to declare variables globally. It is based on the event handler sample in "A globals.jsa Example for Application and Session Events—index1.jsp" on page B-44, but differs in that the three application counter variables are declared globally. (In the original event-handler sample, by contrast, each event handler and the JSP page itself must provide `jsp:useBean` tags to locally declare the beans they access.)

Declaring the beans globally results in implicit declaration in all event handlers and the JSP page.

globals.jsa File for index2.jsp

```
<%-- globally declares variables and initializes them to zero --%>

<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<event:application_OnStart>

      <%-- Consider storing pageCount persistently -- If you do read it here --%>

</event:application_OnStart>

<event:application_OnEnd>

      <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
       <% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>

      <%-- Consider storing pageCount persistently -- If you do write it here --%>

</event:application_OnEnd>

<event:session_OnStart>

      <%
         synchronized (sessionCount) {
                sessionCount.setValue(sessionCount.getValue() + 1);
      %>

              <br>
              Starting session #: <%= sessionCount.getValue() %> <br>
```

```
      <%
        }
      %>

      <%
        synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() + 1);
      %>
                There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>
      <%
        }
      %>

</event:session_OnStart>

<event:session_OnEnd>

      <%
         synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() - 1);
        }
      %>

</event:session_OnEnd>
```

index2.jsp Source

```
<%-- pageCount declared in globals.jsa so active in all pages --%>

<%
   synchronized(pageCount) {
        pageCount.setValue(pageCount.getValue() + 1);
   }
%>

This page has been accessed <b> <%= pageCount.getValue() %> </b>  times.

<p>
```

C

Third Party Licenses

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

Apache HTTP Server

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

The Apache Software License

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

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

Apache JServ

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

Apache JServ Public License

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

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

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

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

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

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

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

Index

Symbols

A

B

C

D

E

F

G

H

I

J

M

N

O

P

R

S

T

U

V

W

X