# Oracle®

## JavaServer Pages Developer's Guide and Reference

Release 8.1.7

(OracleJSP 1.1.0.0.0)

July 2000

Part No.  A83726-01

ORACLE®

JavaServer Pages Developer’s Guide and Reference, Release 8.1.7

Part No. A83726-01



Primary Author: Brian Wright

Contributing Author: Michael Freedman

Contributors: Julie Basu, Alex Yiu, Sunil Kunisetty, YaQing Wang, Hal Hildebrand, Jasen Minton, Matthieu Devin, Jose Alberto Fernandez, Olga Peschansky, Jerry Schwarz, Clement Lai, Shinji Yoshida, Robert Pang, 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 Oracle's JSP Implementation



## 3 Basics

## 4 Key Considerations

## 5 OracleJSP Extensions



## 6 JSP Translation and Deployment

## 7 JSP Tag Libraries and the Oracle JML Tags

## 8 OracleJSP NLS Support



## 9 Sample Applications

## Index

# Send Us Your Comments

**JavaServer Pages Developer's Guide and Reference, Release 8.1.7**

**Part No. A83726-01**

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

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

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

- Electronic mail — jpgcomnt@us.oracle.com
- FAX - 650-506-7225. Attn: Java Platform Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Information Development Manager
  500 Oracle Parkway, Mailstop 4op978
  Redwood Shores, CA 94065
  USA

Please indicate if you would like a reply.

If you have problems with the software, please contact your local Oracle World Wide Support Center.

# Preface

This document introduces and explains Oracle's implementation of JavaServer Pages (JSP) technology, specified by Sun Microsystems. The document discusses standard features, as specified by Sun, as well as Oracle-specific extensions in the OracleJSP product.

---

**Note:** This document is intended for use with the OracleJSP release that accompanies Oracle8*i* release 8.1.7. This OracleJSP release can be referred to as either OracleJSP release 8.1.7 or OracleJSP release 1.1.0.0.0. It is an implementation of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*. Previous versions of OracleJSP implemented the *JavaServer Pages Specification, Version 1.0*.

---

## Intended Audience

This document is intended for developers interested in using OracleJSP to create 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 B)
- 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)

This document focuses on Oracle JSP extensions, and on features and logistics particular to running JSP pages in the Oracle Servlet Engine (the Web server and servlet container inside Oracle8*i*).

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. OracleJSP has special features for earlier servlet environments, however, and there is special discussion of these features as they relate to servlet 2.0 environments, particularly Apache/JServ.

## Document Structure

This document includes the following chapters and appendixes:

| | |
|---|---|
| Chapter 1, "General Overview" | This chapter reviews standard JSP 1.1 technology. (It is not intended as a complete reference.) |
| Chapter 2, "Overview of Oracle's JSP Implementation" | This chapter discusses support for OracleJSP in both Oracle and non-Oracle JSP environments, and introduces Oracle JSP extensions and features. |
| Chapter 3, "Basics" | This chapter introduces basic JSP programming considerations and provides a starter sample for database access. |
| Chapter 4, "Key Considerations" | This chapter discusses a variety of general programming and configuration issues the developer should be aware of. It also covers considerations specific to the OSE and Apache/JServ environments. |
| Chapter 5, "OracleJSP Extensions" | This chapter covers Oracle JSP extensions—both Oracle-specific extensions and extensions that are portable to other JSP environments. |

| | |
|---|---|
| Chapter 6, "JSP Translation and Deployment" | This chapter focuses on procedures and logistics in deploying JSP pages to Oracle8*i* to run in the Oracle Servlet Engine, but also covers general JSP translation and deployment features and issues. |
| Chapter 7, "JSP Tag Libraries and the Oracle JML Tags" | This chapter introduces the basic JSP 1.1 framework for custom tag libraries and also provides an overview and tag descriptions for the JSP 1.1 (runtime) implementation of the Oracle JML sample tag library. |
| Chapter 8, "OracleJSP NLS Support" | This chapter discusses both standard and Oracle-specific features for National Language Support. |
| Chapter 9, "Sample Applications" | This chapter contains a set of sample applications covering both standard JSP technology and Oracle extensions. |
| Appendix A, "General Installation and Configuration" | This appendix covers OracleJSP required and optional files, configuration steps for non-Oracle environments such as Apache/JServ and Tomcat, and OracleJSP configuration parameters for on-demand translation. |
| Appendix B, "Servlet and JSP Technical Background" | This appendix provides a brief background of servlet technology and introduces the standard JSP interfaces for translated pages. |
| Appendix C, "Compile-Time JML Tag Support" | This chapter provides an overview of the compile-time implementation of the Oracle JML sample tag library (as supported in pre-JSP 1.1 releases), and documents tags not supported in the runtime implementation documented in Chapter 7. |

## Related Documents

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

- *Oracle8i Java Developer's Guide*

  This book introduces the basic concepts of Java in Oracle8*i* and provides general information about server-side configuration and functionality.

Information that pertains to the Oracle Java platform as a whole, rather than to a particular product (such as JDBC, SQLJ, or EJBs) is in this book.

- *Oracle8i Oracle Servlet Engine User's Guide*

  This book documents use of the Oracle Servlet Engine, the Web server and servlet container inside Oracle8*i*.

- *Oracle8i Java Tools Reference*

  This book documents Java-related tools and utilities for use with Oracle8*i* or in deploying applications to Oracle8*i* (such as the Oracle8*i* session shell and `loadjava` tools).

- *Oracle8i JDBC Developer's Guide and Reference*

  This book covers programming syntax and features of Oracle's implementation of the JDBC standard (for Java Database Connectivity). This includes an overview of the Oracle JDBC drivers, details of Oracle's implementation of JDBC 1.22 and 2.0 features, and discussion of Oracle JDBC type extensions and performance extensions.

- *Oracle8i JPublisher User's Guide*

  This book describes how to use the Oracle JPublisher utility to translate object types and other user-defined types to Java classes. If you are developing SQLJ or JDBC applications that use object types, VARRAY types, nested table types, or object reference types, then JPublisher can generate custom Java classes to map to them.

- *Oracle8i SQLJ Developer's Guide and Reference*

  This book covers the use of SQLJ to embed static SQL operations directly into Java code, covering SQLJ language syntax and SQLJ translator options and features. Both standard SQLJ features and Oracle-specific SQLJ features are described.

- *Oracle8i Java Stored Procedures Developer's Guide*

  This book discusses Java stored procedures—programs that run directly in the Oracle8*i* server. With stored procedures (functions, procedures, database triggers, and SQL methods), Java developers can implement business logic at the server level, thereby improving application performance, scalability, and security.

- *Oracle8i Enterprise JavaBeans Developer's Guide and Reference*

  This book describes Oracle's Enterprise JavaBeans implementation and extensions.

- *Oracle8i CORBA Developer's Guide and Reference*

  This book describes Oracle's CORBA implementation and extensions.

The following documentation is for Oracle products that incorporate OracleJSP. You may want to refer to them for JSP information, including installation and configuration, for those products:

- *Oracle Internet Application Server 8i Documentation Library*
- *Oracle Application Server, Release 4.0.8.2*
  *Developer's Guide: JServlet and JSP Applications*
- Oracle JDeveloper online help, particularly the topic "Creating JSP Pages for Business Components"
- *Oracle Web-to-go Implementation Guide*

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

- *Oracle8i Application Developer's Guide - XML*

  This book provides information about the Oracle XML-SQL Utility. Some of this is relevant to XML-related support provided by OracleJSP.

- *Oracle8i Application Developer's Guide - Fundamentals*

  This book introduces basic design concepts and programming features in using an Oracle8*i* database and creating database access applications.

- *Oracle8i National Language Support Guide*

  This book contains information about NLS environment variables, character sets, and territory and locale settings. In addition, it contains an overview of common NLS issues, typical scenarios, and NLS considerations for OCI and SQL programmers.

- *Oracle8i Supplied PL/SQL Packages Reference*

  This book documents PL/SQL packages available as part of the Oracle8*i* server, some of which may be useful to call from JDBC applications.

- *PL/SQL User's Guide and Reference*

  PL / This book explains the concepts and features of PL / SQL, Oracle's procedural language extension to SQL.

- *Oracle8i SQL Reference*

  This book contains a complete description of the content and syntax of the SQL commands and features used to manage information in an Oracle database.

- *Net8 Administrator's Guide*

  This book contains information about the Oracle8 Connection Manager and Net8 network administration in general.

- *Oracle Advanced Security Administrator's Guide*

  This book describes features of the Oracle Advanced Security Option (formerly known as ANO or ASO).

- *Oracle8i Reference*

  This book contains general reference information about the Oracle8*i* server.

- *Oracle8i Error Messages*

  This book contains information about error messages that can be passed by the Oracle8*i* server.

## Additional Resources

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://technet.oracle.com/tech/java/servlets/`

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

  `http://technet.oracle.com/support/bboard/discussions.htm`

The following resources are available from Sun Microsystems:

- Javasoft Web site for JavaServer Pages:

  `http://www.javasoft.com/products/jsp/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
  ```

## Document Conventions

The following conventions are used in this document:

| Convention | Meaning |
|---|---|
| *italicized regular text* | Italicized regular text is used for emphasis or to indicate a term that is being defined or will be defined shortly. |
| . . . | Horizontal ellipsis points in sample code indicate the omission of a statement or statements or part of a statement. This is done when you would normally expect additional statements or code to appear, but such statements or code would not be related to the example. |
| `code text` | Code text within regular text indicates commands, option names, parameter names, Java syntax, class names, object names, method names, variable names, Java types, Oracle datatypes, file names, and directory names. |
| *`italicized_code_text`* | Italicized code text in a program statement indicates something that must be provided by the user. |
| [*`italicized_code_text`*] | Square brackets enclosing italicized code text in a program statement indicates something that can *optionally* be provided by the user. |

# 1

# General Overview

This chapter reviews standard features and functionality of JavaServer Pages technology. For further information, consult the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

(For an overview of Oracle-specific OracleJSP features, see Chapter 2, "Overview of Oracle's JSP Implementation". Also note that Appendix B, "Servlet and JSP Technical Background", provides related background on standard servlet and JSP technology.)

The following topics are covered here:

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

# 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 (typically on demand, but sometimes in advance), and it processes HTTP requests and generates responses similarly to any other servlet. JSP technology offers a more convenient way to code the servlet.

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.

## 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 1-10.)

```
<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 B-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 B-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 is the case 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 1-18.

## 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, see "OracleJSP Support for XML and XSL" on page 5-9.

# JSP Execution

This section provides a top-level look at how a JSP 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 B-3.)

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

## 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 subclasses a class (provided by the JSP container) that implements the `javax.servlet.jsp.HttpJspPage` interface (described in "Standard JSP Interfaces and Methods" on page B-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 `javax.servlet.jsp.HttpJspPage` interface and generating code for its service method.

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

The servlet (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 subclassed 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 Request a JSP Page

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

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

If it uses port 8080 of the Web server, you can request it with the following URL:

`http://`*`hostname`*`:8080/myapp/dir1/HelloWorld.jsp`

(The application root directory is specified in the servlet context of the application. "Servlet Contexts" on page B-6 summarizes servlet contexts.)

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.

### 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 1-18 discusses the `jsp:include` and `jsp:forward` statements.)

# 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*, which convey information regarding the JSP page as a whole
- *scripting elements*, which are Java coding elements such as declarations, expressions, scriptlets, and comments
- *objects* and *scopes*, where 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*, which 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 5-9.

---

## 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 20kb (the default is 8kb):

  ```
  <%@ 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 clears the buffer (not outputting it to the browser).
- In OracleJSP, `java` is the default language setting. It is good programming practice to set it explicitly, however.

---

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

  Example:

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

  The `include` directive can specify either a page-relative or context-relative location (see "Requesting a JSP Page" on page 1-8 for related discussion).

---

**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 4-9.
- The `include` directive can be used only within the same servlet context.

---

- `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.

  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>
```

As you can see, this example uses XML-style start-tag and end-tag syntax.

JSP tag libraries and tag library description files are introduced later in this chapter, in "Tag Libraries" on page 1-23, and discussed in detail in Chapter 7, "JSP Tag Libraries and the Oracle JML Tags".

## 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 semi-colon, and is contained within `<%=...%>` tags.

  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, may 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 (one of which includes a JSP expression, which does *not* require a semi-colon). 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, assuming the use of a JavaBean instance, `pageBean`, and assuming that some object, `empgr`, 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 4-14.

---

- *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.

Scopes are discussed below, in "Object Scopes".

### Explicit Objects

Explicit objects are typically JavaBean instances 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 1-18, but an example is also shown here:

```
<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 the next section, "Object Scopes".

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

### Object Scopes

Objects in a JSP, 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 statement, you can explicitly set the scope with the following syntax (as in the example in the preceding section, "Explicit Objects"):

scope="*scopevalue*"

There are four possible scopes:

- `scope="page"`—The object is accessible only from within the JSP page where it was created.

  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.
- `scope="session"`—The object is accessible from any JSP page sharing the same HTTP session as the JSP page that created the object.
- `scope="application"`—The object is accessible from any JSP page used in the same Web application (within any single Java virtual machine) as the JSP page that created the object.

### 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 mechanism 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 location:

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

- `page`

  This is an instance of the JSP page implementation class that was created when the page was translated; `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 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. For example, in the Oracle Servlet Engine architecture, each user runs in his or her own JVM.)

- `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 this 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 3-18.

### 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>
```

## 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 begin with `<jsp:` syntax. Although the tags described earlier in this chapter that begin with `<%` syntax are sufficient to code a JSP page, the `<jsp:` tags provide additional functionality and convenience.

Action elements also use syntax similar to that of XML statements, with similar "begin" and "end" tags such as in the following example:

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

Or, where there is no body, the action statement is terminated with an empty tag:

```
<jsp:sampletag 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` action 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` action sets one or more bean properties. (The bean must have been previously specified in a `useBean` action.) 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 `useBean` example) to a value of "Smith":

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

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="*" />
```

- `jsp:getProperty`

  The `jsp:getProperty` action 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` action.) 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 the `jsp:param` action in conjunction with `jsp:include`, `jsp:forward`, or `jsp:plugin` actions (described below).

  For `jsp:forward` and `jsp:include` statements, a `jsp:param` action 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" />
```

---

**Note:** The `jsp:param` tag is not supported for `jsp:include` or `jsp:forward` in the JSP 1.0 specification.

---

- `jsp:include`

  The `jsp:include` action 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).

  As of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, you must set `flush` to `true`, which results in the buffer being flushed to the browser when a `jsp:include` action is executed. (The `flush` attribute is mandatory, but a setting of `false` is currently invalid.)

  You can also have an action body with `jsp:param` settings, 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` action, 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 4-9.
- The `jsp:include` action can be used only within the same servlet context.

- `jsp:forward`

  The `jsp:forward` action 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 (you cannot set `buffer="none"`) to use a `jsp:forward` action. 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` settings, as shown in the second example.

  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` action can be used only within the same servlet context.

---

- `jsp:plugin`

  The `jsp:plugin` action 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 codebase, 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` actions 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 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` action statement as well. Use of these parameters is according to the general HTML specification.

## Tag Libraries

In addition to the standard JSP tags discussed previously in this section, 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, in 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 1-10.

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

- 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 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* 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 information about the sample tag library provided with OracleJSP, see "Overview of the JSP Markup Language (JML) Sample Tag Library" on page 7-20

For further information, see the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

# 2

# Overview of Oracle's JSP Implementation

As of Oracle8*i* release 8.1.7 (OracleJSP 1.1.0.0.0), the Oracle implementation of JavaServer Pages is a complete implementation of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

This chapter introduces features of OracleJSP as well as discussing support for OracleJSP in various environments, particularly the Oracle Servlet Engine (OSE). OSE is the Web server (more precisely, the servlet container) provided with Oracle8*i* JServer.

For an overview of standard JavaServer Pages features, see Chapter 1, "General Overview".

The following topics are covered here:

# Portability and Functionality Across Servlet Environments

Oracle's JavaServer Pages implementation is highly portable across server platforms and servlet environments. It also supplies a framework for Web applications in older servlet environments, where servlet context behavior was not yet sufficiently defined.

## OracleJSP Portability

OracleJSP can run on any servlet environment that complies with version 2.0 or higher of the Sun Microsystems *Java Servlet Specification*. This is in contrast to most JSP implementations, which require a servlet 2.1(b) or higher implementation. As the next section explains, OracleJSP provides functionality equivalent to the missing servlet functionality in older servlet environments.

Furthermore, the OracleJSP container is independent of the server environment and its servlet implementation. This is in contrast to vendors who deliver their JSP implementation as part of their servlet implementation instead of as a standalone product.

This portability makes it much easier to run OracleJSP in both your development environment and the target environment, as opposed to having to use a different JSP implementation on your development system because of any server or servlet platform limitations. There are usually benefits to developing on a system with the same JSP container as the target server; realistically speaking, there is usually some variation between environments.

## OracleJSP Extended Functionality for Servlet 2.0 Environments

Because of interdependence between servlet specifications and JSP functionality, Sun Microsystems has tied versions of the *JavaServer Pages Specification* to versions of the *Java Servlet Specification*. According to Sun, JSP 1.0 requires a servlet 2.1(b) implementation, and JSP 1.1 requires a servlet 2.2 implementation.

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 servlet 2.1 specification allowed, but did not mandate, a separate servlet context for each application. The servlet 2.1(b) and servlet 2.2 specifications mandated separate servlet contexts. (For background information about servlets and servlet contexts, see "Background on Servlets" on page B-2.)

The OracleJSP container, however, offers functionality that emulates the application support provided with the servlet 2.1(b) specification. This allows a full application framework in a servlet 2.0 environment such as Apache/JServ. This includes providing applications with distinct `ServletContext` and `HttpSession` objects.

This extended support is provided through a file, `globals.jsa`, that acts as a JSP application marker, application and session event handler, and centralized location for application-global declarations and directives. (For information, see "Overview of globals.jsa Functionality" on page 5-37.)

Because of this extended functionality, OracleJSP is not limited by the underlying servlet environment.

# Support for OracleJSP in Oracle Environments

This section provides brief overviews of Oracle environments that support and provide OracleJSP, covering the following topics:

- Overview of the Oracle Servlet Engine (OSE)
- Overview of the Oracle Internet Application Server
- Role of the Oracle HTTP Server, Powered by Apache
- Oracle Web Application Database Access Strategies
- Overview of Other Oracle JSP Environments

The Oracle Servlet Engine (OSE), a Web server and servlet container that runs inside the Oracle8*i* database, supports a JSP pre-translation model. JSP pages are translated into servlets prior to or during deployment into the database and subsequently run in the database address space.

For the other Oracle environments, the OracleJSP container supports the typical on-demand translation model, typically translating the pages at runtime. OracleJSP is designed to run effectively in either situation and provide consistent semantics regardless of your choice of server.

## Overview of the Oracle Servlet Engine (OSE)

If your JSP pages are intended to access an Oracle8*i* database, you can execute them directly in the database—the Oracle Servlet Engine (OSE), which is included with Oracle8*i* JServer, incorporates the OracleJSP container. This reduces communication overhead compared to JSP execution in a middle tier. Access to the database is through the Oracle JDBC server-side internal driver.

The OSE execution model requires the developer to take some special steps to deploy the JSP pages into the Oracle8*i* database. This includes translating the pages, loading them into the server, and "publishing" them to make them available for execution.

During installation of Oracle8*i* release 8.1.7, the Oracle HTTP Server, powered by Apache, is set as the default Web server, acting as a front-end for JSP and servlet applications running in OSE. Refer to your installation instructions if you want to change this setting.

Oracle Servlet Engine release 8.1.7 supports the servlet 2.2 and JSP 1.1 specifications, incorporating OracleJSP release 8.1.7 (1.1.0.0.0).

## Overview of the Oracle Internet Application Server

The Oracle Internet 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.

To deliver this range of content and services, Oracle Internet Application Server release 1.0.x incorporates the Oracle HTTP Server release 1.0.0 (powered by Apache), the *i*Cache for database caching and applications in the middle tier, Oracle Forms Services and Oracle Reports Services to support Oracle Forms-based applications and reports generation, and various business logic runtime environments that support Enterprise JavaBeans, stored procedures, and Oracle Business Components for Java.

For database access, the Oracle HTTP Server can route HTTP requests to servlets or JSP pages running in either of the following scenarios:

- directly in Oracle8*i* (routing is through the Apache `mod_ose` module)

  In this scenario, database access is through the JDBC server-side internal driver (using either JDBC or SQLJ code).

- in the Apache/JServ environment (routing is through the Apache `mod_jserv` module)

  In this scenario, database access is through client-side/middle-tier JDBC drivers (using either JDBC or SQLJ code).

Oracle Internet Application Server release 1.0.x provides servlet and JSP environments as follows:

- Both release 1.0.0 and 1.0.1 include the Apache/JServ servlet environment, supporting the servlet 2.0 specification.
- Release 1.0.0 includes OracleJSP release 1.0.0.6.1, supporting the JSP 1.0 specification.
- Release 1.0.1 includes OracleJSP release 1.1.0.0.0, supporting the JSP 1.1 specification.

See the *Oracle Internet Application Server 8i Documentation Library* for more information about the Internet Application Server.

---

**Note:** Future releases of Oracle Internet Application Server may replace the Apache/JServ environment with an alternative servlet environment.

---

## Role of the Oracle HTTP Server, Powered by Apache

Oracle HTTP Server release 1.0.0, powered by the Apache Web server, is included with Oracle Internet Application Server release 1.0.x and Oracle8*i* release 8.1.7 as the HTTP entry point for Web applications accessing the Oracle8*i* database.

You can employ the Oracle HTTP Server to access Oracle8*i* from applications running either inside or outside the database. It accesses the database through appropriate Apache add-on modules.

The remainder of this section covers the following topics:

- Use of Apache Mods
- More About mod_ose
- More About mod_jserv

---

**Notes:**

- The Oracle Servlet Engine itself can function as a Web server; however, it is advisable to use it as a servlet container in conjunction with the Oracle HTTP Server, particularly for applications with static HTML. The `mod_ose` module allows access to OSE through the Oracle HTTP Server.
- In the Oracle Internet Application Server framework, you can use the Oracle HTTP Server to access either the Oracle *i*Cache (for read-only data that has been cached) or the back-end Oracle8*i* database. For Internet Application Server releases 1.0.0 and 1.0.1, however, you cannot use the `mod_ose`/OSE scenario because those releases of *i*Cache do not yet include the Oracle Servlet Engine.

---

### Use of Apache Mods

In using the Oracle HTTP Server, powered by Apache, dynamic content is delivered through various Apache *mod* components provided either by Apache or by other vendors such as Oracle. (Static content is typically delivered from the file system.)

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.)

The following Apache mods are of interest to OracleJSP developers:

- `mod_ose` is provided by Oracle for JSP pages and servlets that have been deployed into the Oracle8*i* database and will be executed by the Oracle Servlet Engine inside the database address space.
- `mod_jserv` is provided by Apache and can be used in accessing the Oracle8*i* database from JSP pages or servlets running in the Apache/JServ servlet environment in a middle-tier JVM.

> **Note:** Many additional Apache "mod" components are available for use in an Apache environment, provided by Apache for general use or by Oracle for Oracle-specific use, but they are not relevant for JSP applications.

### More About mod_ose

The `mod_ose` component, supplied by Oracle, delegates HTTP requests to JSP pages or servlets running in OSE. It communicates with OSE using HTTP over the Net8 protocol and can handle either stateless or stateful requests. Each virtual domain configured in the Oracle HTTP Server is associated with a database connection string (a Net8 name-value list) that indicates where to make a connection to execute the request. The connection uses Net8 directly, providing the same load balancing and hot backup functionality as OCI.

If an application running in an Oracle Internet Application Server framework uses `mod_ose`, then the Internet Application Server's Apache/JServ servlet 2.0 environment is not involved. The Oracle Servlet Engine's own servlet 2.2 environment is used instead.

JSP applications and servlets running in OSE use the Oracle JDBC server-side internal driver for rapid database access. For an overview of OSE, see "Overview of the Oracle Servlet Engine (OSE)" on page 2-4.

You can use the JServer session shell `exportwebdomain` command to configure `mod_ose` to find published servlets and JSP pages in the database.

See the *Oracle8i Oracle Servlet Engine User's Guide* for more information about `mod_ose` and for information about the `exportwebdomain` command.

### More About mod_jserv

The `mod_jserv` component, supplied by Apache, delegates HTTP requests to JSP pages or servlets running in the Apache/JServ servlet container in a middle-tier JVM. Oracle Internet Application Server release 1.0.x includes the Apache/JServ servlet container, which supports the servlet 2.0 specification, and either JDK 1.1.8 or 1.2.2. The middle-tier environment may or may not be on the same physical host as the back-end Oracle8*i* database.

Communication between `mod_jserv` and middle-tier JVMs uses a proprietary Apache/JServ protocol over TCP/IP. The `mod_jserv` component can delegate requests to multiple JVMs in a pool for load balancing.

JSP applications running in middle-tier JVMs use the Oracle JDBC OCI driver or Thin driver to access the database.

Servlet 2.0 environments (as opposed to servlet 2.1 or 2.2 environments) have issues that require special consideration. See "Considerations for Apache/JServ Servlet Environments" on page 4-34.

Refer to Apache documentation for `mod_jserv` configuration information. (This documentation is provided with Oracle8*i* and with the Oracle Internet Application Server.)

## Oracle Web Application Database Access Strategies

Developers who are targeting the Oracle8*i* database from JSP applications have a number of options, including the following:

1. Run in the Apache/JServ servlet container through the Oracle HTTP Server (using `mod_jserv`).
2. Run in the Oracle Servlet Engine through the Oracle HTTP Server (using `mod_ose`).
3. Run in the Oracle Servlet Engine and use it as the Web server directly (although it is generally recommended to use Oracle HTTP Server).

---

**Note:** When you use the Oracle HTTP Server, be aware that the Apache/JServ servlet container has a different default doc root for static files than the Oracle Servlet Engine has. See "Doc Root for Oracle Internet Application Server Versus Oracle Servlet Engine" on page 6-67.

---

Running in Apache/JServ, because it uses a standard JVM (currently JDK 1.2.2 or 1.1.8), is necessary if you want to use the JDBC OCI driver or if the application requires Java features not available in the Oracle8*i* JServer environment (JNI, for example).

However, running in Apache/JServ has the disadvantage of requiring a pool of multiple JVMs that must be configured manually. (For more information, refer to the Apache `mod_jserv` documentation provided with Oracle8*i* or the Oracle Internet Application Server.)

If you do not require Java features such as JNI, especially for applications that are SQL-intensive, running in OSE is usually preferable for the following reasons:

- OSE offers speedier database access, because it runs in the Oracle8*i* address space and uses the Oracle JDBC server-side internal driver.
- OSE offers greater security.
- OSE offers stronger support for statefulness.

Although employing the Oracle Servlet Engine directly as the Web server is feasible, and may be preferable in some situations, accessing it through the Oracle HTTP Server and `mod_ose` is the typical scenario and is recommended.

In particular, Oracle HTTP Server and `mod_ose` can handle the following situations that OSE by itself cannot:

- database access through a Net8-certified firewall
- implementation of a fault-tolerant system using multiple databases
- database access through port 80

  This is typically not possible when using OSE as a Web server directly. In a UNIX environment, for example, port 80 is accessible only from the `root` account, and end-users do not have `root` access.
- connection pooling for stateless applications so that session startup overhead is mostly avoided

The default installation of Oracle8*i* release 8.1.7 uses the Oracle HTTP Server as the front-end Web server for JSP pages and servlets that run in OSE.

## Overview of Other Oracle JSP Environments

In addition to the Oracle Servlet Engine and Oracle Internet Application Server, the following Oracle environments support OracleJSP:

- Oracle Application Server
- Oracle Web-to-go
- Oracle JDeveloper

### Oracle Application Server

Oracle Application Server (OAS) is a scalable, standards-based middle-tier environment for application logic, offering database integration in supporting business applications in both corporate and e-business environments.

New customers will presumably use Oracle Internet Application Server, discussed previously, instead of OAS. For existing OAS customers, however, Oracle Application Server release 4.0.8.2 includes a servlet 2.1 environment and OracleJSP release 1.0.0.6.0 (supporting the JSP 1.0 specification).

Refer to the *Oracle Application Server Developer's Guide: JServlet and JSP Applications* for more information.

### Oracle Web-to-go

Oracle Web-to-go, a component of Oracle8*i* Lite, consists of a collection of modules and services that facilitate development, deployment, and management of mobile Web applications.

Web-to-go lets developers extend Web-based applications to intermittently connected users without coding the infrastructure required for replication, synchronization, and other networking issues. Unlike traditional mobile computing technologies, which rely on custom or proprietary application-programming interfaces (APIs), Web-to-go uses industry-standard Internet technologies.

Web-to-go release 1.3 provides a servlet 2.1 environment and OracleJSP release 1.0.0.6.1 (supporting the JSP 1.0 specification). Future releases will offer a servlet 2.2 environment and OracleJSP 1.1.x.

Refer to the *Oracle Web-to-go Implementation Guide* for more information.

### Oracle JDeveloper

JDeveloper is a Java development tool, rather than a "platform" like the other Oracle products listed here, but incorporates a Web listener, servlet runner, and the OracleJSP container for execution and testing.

See "Oracle JDeveloper Support for OracleJSP" on page 2-23 for more information.

JDeveloper version 3.1 provides a servlet 2.1 environment and OracleJSP release 1.0.0.6.1 (supporting the JSP 1.0 specification). Future releases will offer a servlet 2.2 environment and OracleJSP 1.1.x.

# Support for OracleJSP in Non-Oracle Environments

You should be able to install and run the OracleJSP container on any server environment supporting servlet specification 2.0 or higher. In particular, OracleJSP has been tested in the following environments as of release 8.1.7:

- Apache Software Foundation Apache/JServ 1.1

  This is a Web server and servlet 2.0 environment without a JSP environment. To run JSP pages, you must install a JSP environment on top of it.

- Sun Microsystems JSWDK 1.0 (JavaServer Web Developer's Kit)

  This is a Web server with the servlet 2.1 and JavaServer Pages 1.0 reference implementations. You can, however, install OracleJSP on top of the JSWDK servlet environment to replace the original JSP environment.

- Apache Software Foundation Tomcat 3.1

  This cooperative effort between Sun Microsystems and the Apache Software Foundation is a Web server with the servlet 2.2 and JavaServer Pages 1.1 reference implementations. You can, however, install OracleJSP on top of the Tomcat servlet environment to replace the original JSP environment. You can also run Tomcat in conjunction with the Apache Web server instead of using the Tomcat Web server.

# Overview of OracleJSP Programmatic Extensions

This section is an overview of extended programming features supported by OracleJSP.

OracleJSP provides the following extended functionality through custom tag libraries and custom JavaBeans, all of which are portable to other JSP environments:

- extended datatypes implemented as JavaBeans that can have a specified scope
- integration with XML and XSL
- database-access JavaBeans
- the Oracle JSP Markup Language (JML) custom tag library, which reduces the level of Java proficiency required for JSP development
- a custom tag library for SQL functionality

OracleJSP also provides the following Oracle-specific extensions:

- support for SQLJ, a standard syntax for embedding SQL statements directly into Java code
- extended NLS support
- `JspScopeListener` for event handling
- `globals.jsa` file for application support

Discussion of these topics is followed by a brief description of how OracleJSP pages can interact with Oracle PL/SQL Server Pages.

## Overview of Portable OracleJSP Extensions

The Oracle extensions discussed in this section are implemented either through the OracleJSP JML sample tag library or through custom JavaBeans. They are portable to other JSP environments.

### OracleJSP Extended Datatypes

JSP pages generally rely on core Java datatypes in representing scalar values. Neither of the following standard approaches to this is fully suitable for use in JSP pages:

- primitive types such as `int`, `float`, and `double`
- wrapper classes in the standard `java.lang` package, such as `Integer`, `Float`, and `Double`

Primitive values 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 type values are objects, so they can theoretically be stored in a JSP scope object. However, they cannot be declared 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, OracleJSP provides the `JmlBoolean`, `JmlNumber`, `JmlFPNumber`, and `JmlString` JavaBean classes in package `oracle.jsp.jml` to wrap the most common Java datatypes.

See "JML Extended Datatypes" on page 5-2 for more information.

### 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.

OracleJSP provides special tags in its sample JML tag library to specify that all or part of a JSP page should be transformed through an XSL stylesheet before it is output. You can use this JML tag multiple times in a single JSP page if you want to specify different style sheets for different portions of the page. Note that the JML tag library is portable to other JSP environments.

In addition, the OracleJSP translator supports XML-alternative syntax as specified in the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

See "OracleJSP Support for XML and XSL" on page 5-9 for more information.

### Custom Database-Access JavaBeans

OracleJSP supplies a set of custom JavaBeans for use in accessing an Oracle database. The following beans are provided in the `oracle.jsp.dbutil` package:

- `ConnBean` opens a simple database connection.
- `ConnCacheBean` uses Oracle's connection caching implementation for database connections.

- `DBBean` executes a database query.
- `CursorBean` provides general DML support for `UPDATE`, `INSERT`, and `DELETE` statements, as well as queries.

See "Oracle Database-Access JavaBeans" on page 5-13 for more information.

### OracleJSP SQL Custom Tag Library

With release 8.1.7, OracleJSP provides a custom tag library for SQL functionality. The following tags are provided:

- `dbOpen`—Open a database connection.
- `dbClose`—Close a database connection.
- `dbQuery`—Execute a query.
- `dbCloseQuery`—Close the cursor for a query.
- `dbNextRow`—Move to the next row of the result set.
- `dbExecute`—Execute any SQL DML or DDL statement.

See "OracleJSP Tag Library for SQL" on page 5-24 for more information.

### Oracle 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 and in many cases the only language considered. 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.

OracleJSP provides custom tags as an alternative—the JSP Markup Language (JML). The Oracle JML sample 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, control flow, 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` and `jml:print` tags:

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

For more information, see "Overview of the JSP Markup Language (JML) Sample Tag Library" on page 7-20.

---

**Note:** OracleJSP versions 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. For information, see Appendix C, "Compile-Time JML Tag Support".

---

## Overview of Oracle-Specific Extensions

The OracleJSP extensions listed in this section are not portable to other JSP environments.

### SQLJ Support in OracleJSP

Dynamic server pages commonly include data extracted from databases; however, JavaServer Pages technology does not offer built-in support to facilitate database access. JSP developers typically must 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. OracleJSP and the OracleJSP translator support SQLJ programming in JSP scriptlets.

SQLJ statements are indicated by the `#sql` token. You can trigger the OracleJSP translator to invoke the Oracle SQLJ translator by using the file name extension `.sqljsp` for the JSP source code file.

For more information, see "OracleJSP Support for Oracle SQLJ" on page 5-33.

### Extended NLS Support in OracleJSP

OracleJSP provides extended NLS support for servlet environments that cannot encode multibyte request parameters and bean property settings.

For such environments, OracleJSP offers the `translate_params` configuration parameter, which can be enabled to direct OracleJSP to override the servlet container and do the encoding itself.

For more information, see "OracleJSP Extended Support for Multibyte Parameter Encoding" on page 8-5.

### JspScopeListener for Event Handling

OracleJSP 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. The Oracle `JspScopeListener` can handle page-based, request-based, and application-based events as well.

For more information, see "OracleJSP Event Handling—JspScopeListener" on page 5-32.

### globals.jsa File for Application Support (Servlet 2.0)

For servlet 2.0 environments, where servlet contexts are not fully defined, OracleJSP defines a file, `globals.jsa`, to extend servlet application support.

Within any single Java virtual machine, there can be a `globals.jsa` file for each application (or, equivalently, for each servlet context). This file supports the concept of Web applications through use as an application location marker. Based on `globals.jsa` functionality, the OracleJSP container can also mimic servlet context and HTTP session behavior for servlet environments, where such behavior is not sufficiently defined.

The `globals.jsa` file also provides a vehicle for global Java declarations and JSP directives across all JSP pages of an application.

## Use of OracleJSP with Oracle PL/SQL Server Pages

Oracle provides a product called *PL/SQL Server Pages* (PSP). PSP technology allows embedded PL/SQL scriptlets and stored procedure calls within an HTML page, offering development advantages similar to those offered by JSP technology: namely, that coding the dynamic portion of the page and the static portion of the page can be largely separate development efforts. An HTML expert can code the static part of the page and a PL/SQL expert can code the dynamic part of the page.

The syntax used to distinguish PL/SQL scriptlets in a PSP page is identical to that used to distinguish Java scriptlets in a JSP page.

The remainder of this section discusses support for JSP-PSP interaction, and includes some background on PSP URLs.

For general information about PL/SQL Server Pages, see the *Oracle8i Application Developer's Guide - Fundamentals*.

### Supported Interaction between JSP Pages and PSP Pages

When an end-user runs a PSP application, PSP pages are translated into stored procedures for execution by the PL/SQL gateway in producing output to the Web browser. Because the PL/SQL gateway in Oracle8*i* executes in a servlet wrapper, JSP pages running in the Oracle Servlet Engine can interact with PSP pages as follows:

- You can dynamically include a PSP page from a JSP page (`jsp:include`).
- You can dynamically forward to a PSP page from a JSP page (`jsp:forward`).

However, PSP pages do not have the functionality to dynamically include or forward to a JSP page. Additionally, you cannot statically include a PSP page from a JSP page (the `<%@ include %>` directive to include a file during translation).

### PSP Page URLs

Each PSP page, when loaded and compiled in the database, becomes a PL/SQL stored procedure. The name of the stored procedure for a PSP page is either explicitly declared in the page, using `<%@ plsql procedure="proc-name" %>` syntax, or is derived from the name of the PSP file.

Given the name of the PL/SQL stored procedure, the URL is determined according to the following general syntax:

`http://`*`host`*`[:`*`port`*`]/`*`some-prefix/dad/`*`[`*`schema.`*`]`*`proc-name`*

Where <*`some-prefix`*> is `plsql` for the embedded PL/SQL module, and <*`dad`*> is the database access descriptor to run the stored procedure.

For more information, see the *Oracle8i Application Developer's Guide - Fundamentals*.

# Summary of OracleJSP Releases and Feature Sets

OracleJSP release 1.1.0.0.0, which fully supports the JSP 1.1 specification, is provided with Oracle8*i* release 8.1.7. In this document, "OracleJSP release 8.1.7" is synonymous with "OracleJSP release 1.1.0.0.0".

Some other Oracle platforms supporting OracleJSP have not yet incorporated the latest OracleJSP release, however—they integrate OracleJSP release 1.0.0.6.1 or 1.0.0.6.0, which were JSP 1.0 implementations.

## OracleJSP Releases Provided with Oracle Platforms

Table 2–1 summarizes which OracleJSP releases are provided with which Oracle platform releases as of this writing.

The "OracleJSP Feature Notes" column refers to OracleJSP release 1.1.0.0.0 features documented in this document that are limited in the OracleJSP release noted for the particular Oracle platform, or have special significance for the platform. For more information, see "OracleJSP Feature Notes for Release 1.0.0.6.x" on page 2-20.

***Table 2–1 Oracle Platform Releases and OracleJSP Releases***

| Oracle Platform | Servlet Environment | OracleJSP Release | OracleJSP Feature Notes |
|---|---|---|---|
| Oracle Servlet Engine (Oracle8*i*), release 8.1.7 | servlet 2.2 | OracleJSP 1.1.0.0.0 (JSP 1.1) | n/a |
| Oracle Internet Application Server, release 1.0.1 | servlet 2.0 (Apache/JServ) | OracleJSP 1.1.0.0.0 (JSP 1.1) | n/a |
| Oracle Internet Application Server, release 1.0.0 | servlet 2.0 (Apache/JServ) | OracleJSP 1.0.0.6.0 (JSP 1.0) | `globals.jsa`<br>config params<br>JML restrictions |
| Oracle Application Server, release 4.0.8.2 | servlet 2.1 | OracleJSP 1.0.0.6.0 (JSP 1.0) | config params<br>JML restrictions |
| Oracle Web-to-go, release 1.3 | servlet 2.1 | OracleJSP 1.0.0.6.1 (JSP 1.0) | config params<br>JML restrictions |
| Oracle JDeveloper, release 3.1 | servlet 2.1 | OracleJSP 1.0.0.6.1 (JSP 1.0) | config params<br>JML restrictions |

It is possible to download, incorporate, and use more recent OracleJSP versions with the above Oracle platforms; the OracleJSP versions documented are the versions that are supplied as part of the product.

To verify the OracleJSP release being used in a particular environment, retrieve the release number from the implicit `application` object in a JSP page, as follows:

```
<%= application.getAttribute("oracle.jsp.versionNumber") %>
```

## OracleJSP Feature Notes for Release 1.0.0.6.x

The following points describe the significance of the "OracleJSP Feature Notes" column in Table 2–1 above, regarding OracleJSP release 1.0.0.6.x.

- The servlet 2.0 specification did not provide a complete framework for Web applications. For servlet 2.0 environments such as Apache/JServ and Oracle Internet Application Server (which uses Apache/JServ), all releases of OracleJSP offer extensions through the `globals.jsa` mechanism to support a more complete application framework. See "OracleJSP Application and Session Support for Servlet 2.0" on page 5-37 for more information.
- Some OracleJSP configuration parameters that are supported in release 1.1.0.0.0 were not yet supported in release 1.0.0.6.x. See "Configuration Parameters Summary Table" on page A-15.
- Release 1.0.0.6.x of OracleJSP complied with the JSP 1.0 specification, not the JSP 1.1 specification, so could not support the JSP 1.1 custom tag library mechanism. As a result, these OracleJSP releases supported JML tags through an Oracle-specific compile-time implementation, using extensions to the OracleJSP translator.

  Use of JML in OracleJSP release 1.0.0.6.x requires a `taglib` directive (as specified for JSP 1.1 and supported by OracleJSP 1.1.0.0.0), but the directive must specify the class that contains the library, as follows:

  ```
  <%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>
  ```

  By contrast, when using a JSP implementation that complies with the JSP 1.1 specification, such as OracleJSP 1.1.0.0.0, the `taglib` directive specifies the tag library description file (in a `.tld` file or `.jar` file), as follows:

  ```
  <%@ taglib uri="/WEB-INF/tlds/jmltags.tld" prefix="jml" %>
  ```

  For information about the JML compile-time implementation, see Appendix C, "Compile-Time JML Tag Support".

# OracleJSP Execution Models

As mentioned earlier, you can use the OracleJSP framework in a variety of server environments. OracleJSP offers two distinct execution models:

- In environments other than the Oracle Servlet Engine, the OracleJSP container typically translates pages on demand before triggering their execution, as is also true with most other vendors' JSP implementations.
- In the Oracle Servlet Engine environment—for JSP pages running in the Oracle8*i* database—the developer translates the pages in advance and loads them into the Oracle8*i* database 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

OracleJSP uses the typical on-demand translation model for all server environments that support OracleJSP, other than the Oracle Servlet Engine. This includes using OracleJSP with the Apache Web server with JServ, for example, as well as various Oracle environments.

When a JSP page is requested from a Web server that incorporates the OracleJSP container, the servlet `oracle.jsp.JspServlet` is instantiated and invoked (assuming proper Web server configuration). This servlet can be thought of as the front-end of the OracleJSP container.

`JspServlet` locates the JSP page, translates and compiles it if necessary (if the page implementation 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`. The steps to accomplish this for Apache/JServ, the Sun Microsystems JWSDK, and Tomcat are discussed in detail in "Configuration of Web Server and Servlet Environment to Run OracleJSP" on page A-7.

## Oracle Servlet Engine Pre-Translation Model

JSP pages intended to run in the Oracle Servlet Engine (OSE), the Web server and servlet container within Oracle8*i*, are pre-translated and deployed into Oracle8*i* as working servlets. OSE incorporates the OracleJSP runtime.

### Deployment Steps to Run JSP Pages in the Oracle Servlet Engine

Perform the following steps to deploy JSP pages into the Oracle8*i* database:

1. Pre-translate the JSP pages (typically including compilation). The page implementation classes produced by the JSP translator are essentially working servlets.
2. Load the translated JSP pages into the Oracle8*i* database.
3. Optionally "hotload" the generated page implementation classes.
4. "Publish" the JSP pages to make them accessible from the database for execution.

Command-line tools are available to translate, load, and publish the pages. The translator creates the page implementation class in a `.java` file and compiles it into a `.class` file.

Hotloading can be enabled and accomplished through additional steps. This is a feature that allows more efficient use of literal strings such as the generated HTML tags in a page implementation class.

Deployment to Oracle8*i* can be performed with the translation being done either in the server or on the client. For more information about these scenarios and the steps involved, see "Deployment to Oracle8i with Server-Side Translation" on page 6-41 and "Deployment to Oracle8i with Client-Side Translation" on page 6-54.

### Oracle Servlet Engine JSP Container

The Oracle Servlet Engine incorporates its own OracleJSP container, which consists of most of the overall OracleJSP container without the OracleJSP translator (because any JSP page that runs in the OSE environment is pre-translated).

The OSE includes front-end JSP processing, with functionality similar to `JspServlet` in the on-demand translation model.

The front-end component finds and executes JSP pages according to a servlet path (often referred to as a "virtual path") that was entered in the Oracle8*i* JNDI name space during publishing. You specify a servlet path name when you publish the JSP page.

## Oracle JDeveloper Support for OracleJSP

Visual Java programming tools are beginning to support JSP coding. In particular, Oracle JDeveloper supports OracleJSP and includes the following features (as of JDeveloper release 3.1):

- integration of the OracleJSP 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 "Deployment of JSP Pages with JDeveloper" on page 6-71 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 the your browser) or to the server log (through the `log()` method of the implicit `application` object).

For information about JDeveloper, refer to their online help, particularly the topic "Creating JSP Pages for Business Components".

# 3

# Basics

This chapter discusses basic issues such as applications and sessions, JSP-servlet interaction, resource management, and application roots and doc roots. This is followed by a JSP "starter sample" for database access.

The following topics are included:

# Preliminary Considerations

This section discusses a few issues to be aware of before you start developing. The following topics are covered:

- Installation and Configuration Overview
- Development Environments Versus Deployment Environments
- Client-Side Considerations

## Installation and Configuration Overview

Installation and configuration, primarily for key non-Oracle environments, is covered in Appendix A, "General Installation and Configuration".

For installation and configuration of Oracle environments that support OracleJSP, consult the documentation for the particular Oracle product.

Within Oracle8*i*, the Oracle Servlet Engine (OSE) incorporates OracleJSP and is provided with Oracle8*i* JServer.

## Development Environments Versus Deployment Environments

JSP developers targeting a non-Oracle environment, such as Apache/JServ, typically develop in the same environment as the target environment. In this case, the installation and configuration instructions in Appendix A, "General Installation and Configuration" apply to both the development environment and the deployment environment, although some of the configuration parameters are of interest only during development.

JSP developers targeting the Oracle Servlet Engine or some other Oracle environment have at least two development options:

- Use Oracle JDeveloper for development and deployment.

  JDeveloper incorporates OracleJSP and a servlet container for use in testing during development. It also incorporates features to help you deploy the finished product to the target location.

  See "Oracle JDeveloper Support for OracleJSP" on page 2-23 for an introduction to OracleJSP support in JDeveloper. Refer to JDeveloper documentation for installation and configuration instructions.

- Develop and test in a non-Oracle environment such as Apache/JServ before deploying to the target Oracle environment for final testing and end use.

In this case, the information in Appendix A is presumably of interest to you for your development environment.

After testing in the development environment, you can pre-translate the JSP pages and deploy them to the Oracle8*i* database using command-line tools available with the OracleJSP installation. The OracleJSP command-line translator has options that are equivalent to relevant translation-time configuration parameters. For information, see "The ojspc Pre-Translation Tool" on page 6-23 and "Deployment to Oracle8i with Client-Side Translation" on page 6-54.

For information about installing and configuring any of the Oracle environments that support OracleJSP, refer to the documentation for the particular product.

## Client-Side Considerations

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 or data server.

# Application Root and Doc Root Functionality

This section provides an overview of application roots and doc roots, distinguishing between servlet 2.2 functionality and servlet 2.0 functionality.

## Application Roots in Servlet 2.2 Environments

As mentioned earlier, the servlet 2.2 specification provides for each 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 application. This is the *application root*. Each application has its own application root.

This is similar to how a Web server uses a *doc root* as the root location for HTML pages and other files belonging to a Web application.

For an application in a servlet 2.2 environment, there is a one-to-one mapping between the application root (for servlets and JSP pages) and the doc root (for static files, such as HTML files)—they are essentially the same thing.

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.

For example, consider an application with the application root `/home/dir/mybankappdir`, which is mapped to the context path `mybank`. Further assume the application includes a servlet whose servlet path is `loginservlet`. This servlet can be invoked 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/mybankappdir/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/mydefaultdir`, and a servlet with the servlet path `myservlet` uses the default context. Its URL would be as in the following example. (Again, the application root directory name itself is not visible to the user.)

```
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/mydefaultdir/dir2/def.html`:

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

## OracleJSP Application Root Functionality in Servlet 2.0 Environments

Apache/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.

For Apache, the doc root is typically some `.../htdocs` directory. In addition, it is possible to specify "virtual" doc roots through `alias` settings in the `httpd.conf` configuration file.

In a servlet 2.0 environment, OracleJSP offers the following functionality regarding doc roots and application roots:

- By default, OracleJSP uses the doc root as an application root.
- Through the OracleJSP `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 5-37.)

# Overview of JSP Applications and Sessions

This section provides a brief overview of how JSP applications and sessions are supported by OracleJSP.

## General OracleJSP Application and Session Support

OracleJSP uses underlying servlet mechanisms for managing applications and sessions. For information about these mechanisms, see "Servlet Sessions" on page B-4 and "Servlet Contexts" on page B-6. For servlet 2.1 and servlet 2.2 environments, these underlying mechanisms are sufficient, providing a distinct servlet context and session object for each JSP application.

Using the servlet mechanisms becomes problematic, however, in a servlet 2.0 environment such as Apache/JServ. The concept of a Web application was not well defined in the servlet 2.0 specification, so in a servlet 2.0 environment there is only one servlet context per servlet container. Additionally, there is one session object only per servlet container. However, for Apache/JServ and other servlet 2.0 environments, OracleJSP provides extensions to optionally allow distinct servlet contexts and session objects for each application. (This is unnecessary for Web servers hosting just a single application.)

---

**Note:** For additional information relevant to Apache/JServ and other servlet 2.0 environments, see "Considerations for Apache/JServ Servlet Environments" on page 4-34 and "Overview of globals.jsa Functionality" on page 5-37.

---

## JSP Default Session Requests

Generally speaking, servlets do *not* request an HTTP session by default. However, JSP page implementation classes *do* request an HTTP session by default. You can override this by setting the `session` parameter to `false` in a JSP `page` directive, as follows:

```
<%@ page ... session="false" %>
```

# 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 4-19.

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 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 1-18.) 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 Apache/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 Apache/JServ" on page 4-35.

## 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).

A `jsp:param` tag is used 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 1-18.

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 3-9.

**Note:** The `jsp:param` tag was introduced in the JSP 1.1 specification.

## 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 the next section, "Passing Data Between a JSP Page and a Servlet", 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` actions. 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.

Here is an 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.

  Here is an 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:**

- This functionality was introduced in the servlet 2.1 specification. Be aware that the semantics are different between the servlet 2.1 specification and the servlet 2.2 specification—in a servlet 2.1 environment a given attribute can be set only once.
- Mechanisms discussed in this section can be used 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

```
<%-------------------------------------------------------------
   Copyright © 1999, Oracle Corporation. All rights reserved.
--------------------------------------------------------------%>

<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

The `javax.servlet.http` package offers a standard mechanism for managing session resources. Additionally, Oracle provides extensions for managing application, session, page, and request resources.

## 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-scoped resources. Through this mechanism, a session-scoped 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 Database Access" on page 3-21 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).

---

**Note:** OracleJSP provides extensions for additional resource management, allowing you to program JavaBeans to manage page-scoped, request-scoped, or application-scoped resources as well as session-scoped resources. See "OracleJSP Event Handling—JspScopeListener" on page 5-32.

---

The next section, "JDBCQueryBean JavaBean Code", 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 3-15), 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-scoped 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

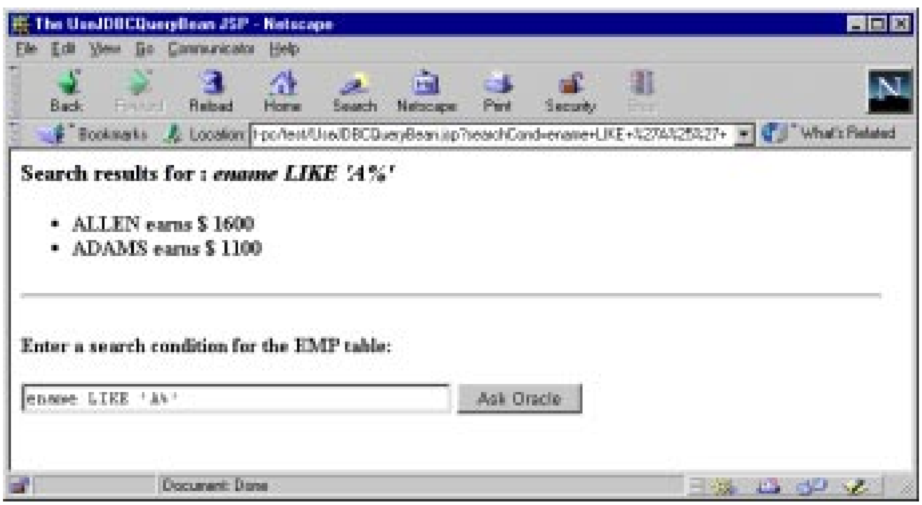
The following JSP page uses the `JDBCQueryBean` JavaBean defined in the preceding section ("JDBCQueryBean JavaBean Code"), invoking the bean with `session` scope. It uses `JDBCQueryBean` to display employee names that match a search condition entered by the user.

`JDBCQueryBean` gets the search condition through the `jsp:setProperty` command 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 is what 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 Extensions for Resource Management

Oracle provides the following extensions for managing application and session resources as well as page and request resources:

- `JspScopeListener`—for managing application, session, page, or request resources

  For information, see "OracleJSP Event Handling—JspScopeListener" on page 5-32.

- `globals.jsa` application and session events—for start and end events for applications and sessions, typically in a servlet 2.0 environment such as Apache/JServ

  See "The globals.jsa Event Handlers" on page 5-42 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 a simple 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 1-10. )

In a servlet 2.2 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* 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 1-16.)

See the next section, "JSP Error Page Example", for an example of error page usage.

---

**Note:** There is ambiguity in the JSP 1.1 specification regarding exception types that can be handled through the JSP mechanism.

A page implementation class generated by the OracleJSP 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 OracleJSP container to the servlet container.

The ambiguity is expected to be addressed in the JSP 1.2 specification. OracleJSP behavior will be modified appropriately in a future release.

---

## 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 Database Access

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

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

---

**Notes:**

- Oracle JDBC provides several driver alternatives: 1) JDBC OCI drivers 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 *Oracle8i JDBC Developer's Guide and Reference*.
- OracleJSP also supports SQLJ (embedded SQL in Java) for static SQL operations and provides custom JavaBeans and custom SQL tags for database access. These features are discussed in Chapter 5, "OracleJSP Extensions".

---

The following example creates a query dynamically, from search conditions that 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 instead of SQLJ because the query is formed dynamically. SQLJ is for static 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).

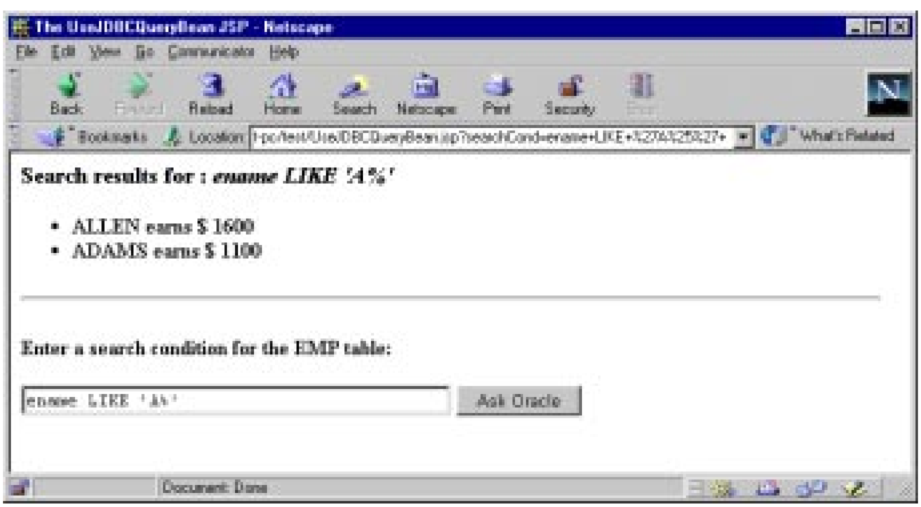
**Notes:**

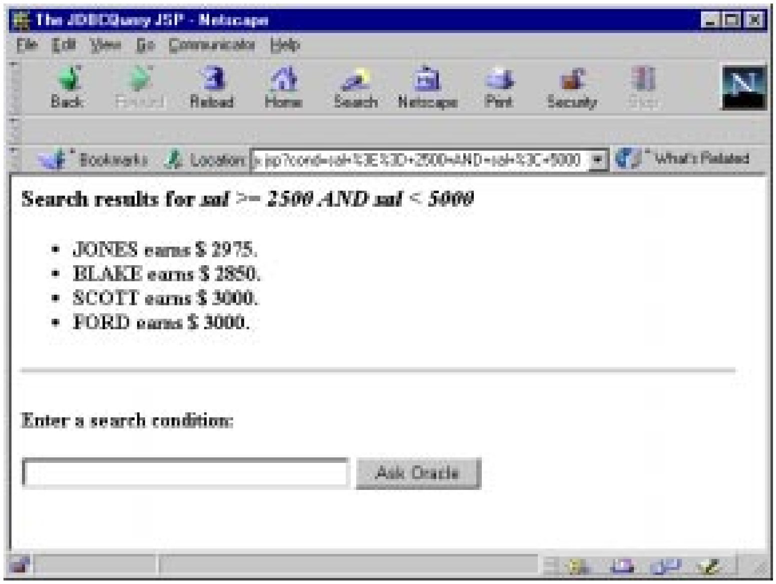
- Another approach to this example would be to define the `runQuery()` method in `<%...%>` scriptlet syntax instead of `<%!...%>` declaration syntax.
- This example uses the JDBC OCI driver, which requires an Oracle client installation. If you want to run this sample, use an appropriate JDBC driver and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




The JDBCQuery JSP - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: What's Related
Search results for sal >= 2500 AND sal < 5000
JONES earns $ 2975.
BLAKE earns $ 2850.
SCOTT earns $ 3000.
FORD earns $ 3000.
Enter a search condition:
Ask Oracle

# 4

# Key Considerations

This chapter discusses important programming, configurational, and runtime considerations, as well as special considerations for particular execution environments. The following topics are covered:

# General JSP Programming Strategies, Tips, and Traps

This section discusses issues you should consider when programming JSP pages that will run in the OracleJSP container, regardless of the particular target environment.

---

**Note:** In addition to being aware of the topics in this section, you should be aware of OracleJSP 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 logic in the JSP page itself, 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 Database Access" on page 3-21, although illustrative, is probably not an ideal design. Database 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.

## Use of Enterprise JavaBeans in JSP Pages

To use an Enterprise JavaBean (EJB) in a JSP page, choose either of the following approaches:

- Use a JavaBean wrapper for the EJB and call the JavaBean from the JSP page as you would any other JavaBean.
- Call the EJB directly from the JSP page.

This section provides two examples of calling an EJB from a JSP page—one where the JSP page runs in a middle-tier environment, and one where it runs in the Oracle Servlet Engine.

These examples point out some significant advantages in using the Oracle Servlet Engine.

For general information about the Oracle EJB implementation, see the *Oracle8i Enterprise JavaBeans Developer's Guide and Reference*.

### Calling an EJB from a JSP Page in the Middle Tier

The following JSP page calls an EJB from a middle-tier environment such as the Oracle Internet Application Server. In this case, the service URL is specified as `sess_iiop://localhost:2481:ORCL` ( you may need to modify it to use your own hostname, IIOP port number and Oracle instance name). The JNDI naming context is set up through the `new InitialContext(env)` construction, where `env` is a hashtable defining the parameters for the context. Once the initial context (`ic`) is created, the code looks up the EJB home object using the service URL and the JNDI name for the EJB:

```
EmployeeHome home = (EmployeeHome) ic.lookup (surl + "/test/employeeBean");
```

Then the `home.create()` method is called to create an instance of the bean, and the bean's `query()` method is called to get the name and salary for the employee whose number was entered through the HTML form in the JSP page.

Following is the sample code:

```
<HTML>
<%@ page import="employee.Employee, employee.EmployeeHome,
employee.EmpRecord, oracle.aurora.jndi.sess_iiop.ServiceCtx,
javax.naming.Context, javax.naming.InitialContext, java.util.Hashtable"
%>

<HEAD> <TITLE> The CallEJB JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">
<BR>
<%  String empNum = request.getParameter("empNum");
    String surl = request.getParameter("surl");
    if (empNum != null) {
      try {
          Hashtable env = new Hashtable();
          env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
          env.put(Context.SECURITY_PRINCIPAL, "scott");
          env.put(Context.SECURITY_CREDENTIALS, "tiger");
          env.put(Context.SECURITY_AUTHENTICATION,
                  ServiceCtx.NON_SSL_LOGIN);
          Context ic = new InitialContext (env);
          EmployeeHome home = (EmployeeHome)ic.lookup (surl +
                           "/test/employeeBean");
          Employee testBean = home.create();
```

```
          EmpRecord empRec = testBean.query (Integer.parseInt(empNum));

%>
<h2><BLOCKQUOTE><BIG><PRE>
        Hello, I'm an EJB in Oracle8i.
        Employee <%= empRec.ename %> earns $ <%= empRec.sal %>
<%  } catch (Exception e) { %>
        Error occurred: <%= e %>
<%   }
   } %>
</PRE></BIG></BLOCKQUOTE></h2>
      <HR>

<P><B>Enter an employee number and EJB service URL:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="empNum" SIZE=10 value="7654">
<INPUT TYPE=text NAME="surl" SIZE=40 value="sess_iiop://localhost:2481:ORCL">
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>
```

## Calling an EJB from a JSP Page in the Oracle Servlet Engine

If you are deploying the JSP page to the Oracle Servlet Engine in Oracle8*i*, the EJB lookup and invocation is much simpler and highly optimized. In this case, the bean lookup is done locally within the Oracle8*i* JNDI namespace. An explicit service URL specification is not required. The naming context is initialized for the current session with the simple call :

```
Context ic = new InitialContext();
```

Note that the constructor in this case does not require any arguments, unlike the middle-tier example. The bean is looked up using just its JNDI name (without the service URL):

```
EmployeeHome home = (EmployeeHome)ic.lookup ("/test/employeeBean");
```

Following is the sample code:

```
<HTML>
<%@ page import="employee.Employee, employee.EmployeeHome,
employee.EmpRecord, oracle.aurora.jndi.sess_iiop.ServiceCtx,
javax.naming.Context, javax.naming.InitialContext,
java.util.Hashtable"  %>
```

```
<HEAD> <TITLE> The CallEJB JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">
<BR>
<%  String empNum = request.getParameter("empNum");
    if (empNum != null) {
      try {
        Context ic = new InitialContext();
        EmployeeHome home = (EmployeeHome)ic.lookup("/test/employeeBean");
        Employee testBean = home.create();
        EmpRecord empRec =  testBean.query (Integer.parseInt(empNum));
%>
<h2><BLOCKQUOTE><BIG><PRE>
        Hello, I'm an EJB in Oracle8i.
        Employee <%= empRec.ename %> earns $ <%= empRec.sal %>
<%  } catch (Exception e) { %>
        Error occurred: <%= e %>
<%  }
  } %>
</PRE></BIG></BLOCKQUOTE></h2>
      <HR>

<P><B>Enter an employee number URL:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="empNum" SIZE=10 value="7654">
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>
```

## Use of JDBC Performance Enhancement Features

You can use the following performance enhancement features in JSP applications executed by OracleJSP:

- caching database connections (through Oracle extensions)
- caching JDBC statements (through Oracle extensions)
- batching update statements (through Oracle extensions)
- prefetching rows during a query (through Oracle extensions)
- caching rowsets (through Sun Microsystems extensions)

Most of these performance features are supported by the `ConnBean` and `ConnCacheBean` database-access JavaBeans (but not by `DBBean`). "Oracle Database-Access JavaBeans" on page 5-13 describes these beans.

### 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` database-access JavaBean provided with OracleJSP. This is probably how most OracleJSP developers will use connection caching. For information, see "ConnCacheBean for Connection Caching" on page 5-16.

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` statements or directly through the class setter methods.

For examples of using `OracleConnectionCacheImpl` directly, see "Connection Caching—ConnCache3.jsp and ConnCache1.jsp" on page 9-16.

For information about the Oracle JDBC connection caching scheme and the `OracleConnectionCacheImpl` class, see the *Oracle8i JDBC Developer's Guide and Reference*.

### JDBC Statement Caching

Statement caching, an Oracle JDBC extension that is available with release 8.1.7, 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` database-access JavaBeans that are provided with OracleJSP. Each of these beans has a `stmtCacheSize` property that can be set through a `jsp:setProperty` statement or the bean's `setStmtCacheSize()` method. For information, see "ConnBean for a Database Connection" on page 5-14 and "ConnCacheBean for Connection Caching" on page 5-16.

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 *Oracle8i 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 `executeBatch()` 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 10 operations will be sent to the database and processed in one trip.

OracleJSP supports Oracle JDBC update batching directly, through the `executeBatch` property of the `ConnBean` database-access JavaBean. You can set this property through a `jsp:setProperty` statement or through the bean's setter method. If you use `ConnCacheBean` instead, you can enable update batching through Oracle JDBC functionality in the connection and statement objects you create. See "ConnBean for a Database Connection" on page 5-14 and "ConnCacheBean for Connection Caching" on page 5-16 for information about these JavaBeans.

For more information about Oracle JDBC update batching, see the *Oracle8i 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.

OracleJSP supports Oracle JDBC row prefetching directly, through the `preFetch` property of the `ConnBean` database-access JavaBean. You can set this property through a `jsp:setProperty` statement or through the bean's setter method. If you use `ConnCacheBean` instead, you can enable row prefetching through Oracle JDBC functionality in the connection and statement objects you create. See "ConnBean for a Database Connection" on page 5-14 and "ConnCacheBean for Connection Caching" on page 5-16 for information about these JavaBeans.

For more information about Oracle JDBC row prefetching, see the *Oracle8i JDBC Developer's Guide and Reference*.

### Rowset Caching

A cached rowset provides a disconnected, serializable, and scrollable container for data retrieved from the database. 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.

As of release 8.1.7, Oracle JDBC does not provide a cached rowset implementation; however, a reference implementation is available from Sun Microsystems. Download the file `rowset.jar` from the Sun Web site, include it in your Web server classpath, and import the package `sun.jdbc.rowset.*` in your code. Then use code to create and populate a cached rowset, such as in the following example:

```
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.

## Static Includes Versus Dynamic Includes

The `include` directive, described in "Directives" on page 1-10, 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` action, described in "JSP Actions and the <jsp: > Tag Set" on page 1-18, 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 of you who are 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:** Both static includes and dynamic includes can be used only within 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 each 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. (There is a 64K size limit on the service method of the generated page implementation class—see "Workarounds for Large Static Content in JSP Pages" on page 4-13.)

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 4-12.)

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.

## 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.
- 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 sample tag library provided with OracleJSP. This library includes tags that support logic equivalent to Java loops and conditionals. (See "Overview of the JSP Markup Language (JML) Sample Tag Library" on page 7-20 for information.)

### 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 library's tags):

```
<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 3-12.)

```
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.

## 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` command (`jsp:include`) to include its output in the JSP page output at runtime. See "JSP Actions and the <jsp: > Tag Set" on page 1-18 for information about the `jsp:include` command.

  > **Important:** A static `<%@ include... %>` command 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.

  OracleJSP will do this for you if you enable the `external_resource` configuration parameter. This parameter is documented in "OracleJSP Configuration Parameters (Non-OSE)" on page A-15.

  For deployment to Oracle8*i*, the `-extres` and `-hotload` options of the `ojspc` pre-translation tool, and the `-hotload` option of the `publishjsp` session shell command, 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 64K byte 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 1-12, 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.
- A variable that is declared in `<% ... %>` JSP scriptlet syntax is local to the service method of the page implementation class.

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 OracleJSP 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 Example 2, 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

OracleJSP (and JavaServer Pages implementations in general) preserves 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 (nowhitsp.jsp)

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 comprise a single wrap-around line of code.)

```
<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 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 (whitesp.jsp)

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

```
<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 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 of course 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.

# Key OracleJSP Configuration Issues

This section covers important effects of how you set key `page` directive parameters and OracleJSP configuration parameters. The discussion focuses on JSP page optimization, and classpath and class loader issues. The following topics are covered:

- Optimization of JSP Execution
- Classpath and Class Loader Issues (Non-OSE Only)

## Optimization of JSP Execution

There are settings you can consider to optimize JSP performance, including the following:

- Unbuffering a JSP Page
- Not Checking for Retranslation (Non-OSE Only) ("developer mode")
- Not Using an HTTP Session

### Unbuffering a JSP Page

By default, a JSP page uses an area of memory known as a *page buffer*. This buffer (8KB by default) is required if the page uses dynamic NLS 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 an output step (output goes straight to the browser instead of going through the buffer first).

### Not Checking for Retranslation (Non-OSE Only)

When OracleJSP executes a JSP page, by default it will check 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.

If comparing timestamps is unnecessary (as is the case in a typical deployment environment, where source code will not change), you can avoid the timestamp comparison by disabling the OracleJSP `developer_mode` flag (`developer_mode=false`).

The default setting is `true`. For information about how to set this flag in the Apache/JServ, JSWDK, and Tomcat environments, see "OracleJSP Configuration Parameter Settings" on page A-25.

### 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 avoid using a session through the following `page` directive:

```
<%@ 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 B-4.)

## Classpath and Class Loader Issues (Non-OSE Only)

OracleJSP uses its own classpath, distinct from the Web server classpath, and by default uses its own class loader to load classes from this classpath. This has significant advantages and disadvantages.

The OracleJSP classpath combines the following elements:

- the OracleJSP default classpath
- additional classpaths you specify in the OracleJSP `classpath` configuration parameter

If there are classes you want loaded by the OracleJSP class loader instead of the system class loader, use the OracleJSP `classpath` configuration parameter, or place the classes in the OracleJSP default classpath. See "Advantages and Disadvantages of the OracleJSP Class Loader" on page 4-23 for related discussion.

### OracleJSP Default Classpath

Oracle JSP defines standard locations on the Web server for locating `.class` files and `.jar` files for classes (such as JavaBeans) that it requires. OracleJSP will find files in these locations without any Web server classpath configuration.

These locations are as follows and are relative to the application root:

```
/WEB-INF/classes
/WEB-INF/lib
/_pages
```

**Important:** If you want classes in the `WEB-INF` directories to be loaded by the system class loader instead of the OracleJSP class loader, place the classes somewhere in the Web server classpath as well. The system class loader takes priority—any class that is placed in both classpaths will always be loaded by the system class loader.

The `classes` directory is for individual Java `.class` files. These classes should be stored 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. OracleJSP 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 application root directory can be located in any of the following locations (as applicable, depending on your Web server and servlet environment), listed in the order they are searched:

- the Web server directory this application is mapped to
- the Web server document root directory
- the directory containing the `globals.jsa` file (where applicable, typically in a servlet 2.0 environment)

**Notes:**

- Some Web servers, particularly those supporting the servlet 2.0 specification, do not offer full application support such as complete servlet context functionality. In this case, or when application mapping is not used, the default application is the server itself, and the application root is the Web server document root.
- For older servlet environments, the `globals.jsa` file is an Oracle extension that can be used as an application marker to establish an application root. See "OracleJSP Application and Session Support for Servlet 2.0" on page 5-37.

### OracleJSP classpath Configuration Parameter

Use the OracleJSP `classpath` configuration parameter to add to the OracleJSP classpath.

For more information about this parameter, see "OracleJSP Configuration Parameters (Non-OSE)" on page A-15.

For information about how to set this parameter in the Apache/JServ, JSWDK, and Tomcat environments, see "OracleJSP Configuration Parameter Settings" on page A-25.

### Advantages and Disadvantages of the OracleJSP Class Loader

Using the OracleJSP class loader results in the following advantages and disadvantages:

- limited access to OracleJSP-loaded classes from classes loaded by any other class loader

  When a class is loaded by the OracleJSP class loader , its definition exists in the OracleJSP class loader only. Classes loaded by the system class loader or any other class loader, including any servlets, would have only limited access. The classes loaded by another class loader could not cast the OracleJSP-loaded class or call methods on it. This may be desirable or undesirable, depending on your situation.

- automatic class reloading

  By default, the OracleJSP class loader will automatically reload a class in the OracleJSP classpath whenever the class file or JAR file has been modified since it was last loaded. (For a JSP page, for example, this can happen as a result of dynamic retranslation, which occurs by default if the `.jsp` source file for a page has a more recent timestamp than its corresponding page implementation `.class` file).

This is usually only advantageous in a development environment. In a typical deployment environment, the source, class, and JAR files will not change, and it is inefficient to check them for changes.

See "Dynamic Class Reloading" on page 4-26 for more information.

It follows that in a deployment environment, you will typically *not* want to use the OracleJSP classpath. By default, the `classpath` parameter is empty.

# OracleJSP Runtime Considerations (Non-OSE Only)

This section describes conditions under which OracleJSP retranslates pages, reloads pages, and reloads classes during runtime. This discussion does not apply to JSP pages running in the Oracle Servlet Engine.

## Dynamic Page Retranslation

As a Web application is running, the OracleJSP container by default will automatically retranslate and reload a JSP page whenever the page source is modified.

OracleJSP checks whether the last-modified time of the page implementation class file, as indicated in the OracleJSP in-memory cache, is older than the last-modified time of the JSP page source file.

You can avoid the overhead of OracleJSP checking timestamps for retranslation by setting the OracleJSP `developer_mode` flag to `false`. This is advantageous in a deployment environment, where source and class files will typically not change. For more information about this flag, see "OracleJSP Configuration Parameters (Non-OSE)" on page A-15. For how to set it, see "OracleJSP Configuration Parameter Settings" on page A-25.

---

**Notes:**

- Because of the usage of in-memory values for the class file last-modified time, note that removing a page implementation class file from the file system will *not* cause OracleJSP to retranslate the associated JSP page source. OracleJSP will only retranslate when the JSP page source file timestamp changes.
- The class file will be regenerated when the 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.

---

## Dynamic Page Reloading

The OracleJSP container will automatically reload a JSP page (in other words, reload the generated page implementation class) under the following circumstances.

- the page is retranslated

  (See the previous section, "Dynamic Page Retranslation".)

- a Java class that is called by the page and was loaded by the OracleJSP class loader (and not the system class loader) is modified

  (See the following section, "Dynamic Class Reloading".)

- any page in the same application is reloaded

  A JSP pages is associated with the overall Web application within which it runs (even JSP pages not associated with a particular application are considered to be part of a "default application").

  Whenever a JSP page is reloaded, all JSP pages in the application are reloaded.

**Notes:**

- OracleJSP does *not* reload a page just because a statically included file has changed. (Statically included files, included through `<%@ include %>` syntax, are included during translation-time.)
- Page reloading and page retranslation are not the same thing. Reloading does not imply retranslation.

## Dynamic Class Reloading

By default, before OracleJSP dispatches a request that will execute a Java class that was loaded by the OracleJSP class loader, it checks to see if the class file has been modified since it was first loaded. If the class has been modified, then the OracleJSP class loader reloads it.

This applies only to classes in the OracleJSP classpath, which includes the following:

- JAR files in the `WEB-INF/lib` directory
- `.class` files in the `WEB-INF/classes` directory
- classes in paths specified through the OracleJSP `classpath` configuration parameter
- generated `.class` files in the `_pages` output directory

As mentioned in the preceding section, "Dynamic Page Reloading", reloading a class results in the dynamic reloading of JSP pages that reference that class.

---

**Important:**

- Remember that classes must be in the JSP classpath, not the system classpath, to be dynamically reloaded. If they are in the system classpath as well, the system class loader may take precedence in some circumstances, possibly interfering with JSP automatic-reloading functionality.
- Dynamic class reloading can be expensive in terms of CPU usage. You can disable this feature by setting the OracleJSP `developer_mode` parameter to `false`. This is appropriate in deployment environments where classes are not expected to change.

---

For information about the `classpath` and `developer_mode` configuration parameters and how to set them, see "OracleJSP Configuration Parameters (Non-OSE)" on page A-15 and "OracleJSP Configuration Parameter Settings" on page A-25.

# Considerations for the Oracle Servlet Engine

The Oracle Servlet Engine (OSE) is integrated with the Oracle8*i* JServer environment. To run in OSE, a JSP page must be loaded and published in the database. The details of deploying JSP pages into Oracle8*i* are discussed in Chapter 6, "JSP Translation and Deployment". This section discusses special programming considerations for the OSE environment and provides an overview of key OSE characteristics.

A JSP application can run in OSE by using the Oracle HTTP Server, powered by Apache, as a front-end Web server (generally recommended), or by using OSE as the Web server directly. See "Oracle Web Application Database Access Strategies" on page 2-8. When installing Oracle8*i* release 8.1.7, Oracle HTTP Server is set as the default Web server. Refer to your installation instructions if you want to change this setting.

It is assumed that JSP pages running in the Oracle Servlet Engine are intended for database access, so some background is provided on database connections in JServer.

JSP code is generally completely portable between OSE and other environments where OracleJSP is used. The exception is that connecting in JServer through the JDBC server-side internal driver is different (for example, does not require a connect string), as mentioned in "JServer Connections" on page 4-29.

Aside from the use of any JServer database connection code or other JServer-specific features, JSP pages written for OSE are portable to other environments running OracleJSP. The original code has to be modified and re-translated only if JServer-specific features were used.

The following topics are covered here:

---

**Notes:** This section discusses development considerations in targeting OSE. For deployment considerations, including hotloaded classes and client-side versus server-side translation, see "Overview of Features and Logistics in Deployment to Oracle8i" on page 6-12.

---

## Introduction to the JServer JVM and JDBC Server-Side Internal Driver

Each Oracle8*i* JServer database session invokes its own dedicated Java virtual machine. This one-to-one correspondence between sessions and JVMs is important to keep in mind.

Any Java program running inside JServer in the target Oracle8*i* database typically uses the JDBC *server-side internal driver* to access the local SQL engine. This driver is intrinsically tied to the Oracle8*i* database and to the JVM. The driver runs as part of the same process as the database. It also runs within a default database session—the same session in which the JVM was invoked.

The server-side internal driver is optimized to run within the database server and provide direct access to SQL data and PL/SQL subprograms on the local database. The entire JVM operates in the same address space as the database and the SQL engine. Access to the SQL engine is a function call—there is no network. This enhances the performance of your JDBC programs and is much faster than executing a remote Net8 call to access the SQL engine.

## JServer Connections

Because the JDBC server-side internal driver runs within a default database session, you are already "connected" to the database implicitly. There are two JDBC methods you can use to access the default connection:

- Use the Oracle-specific `defaultConnection()` method of the `OracleDriver` class. (This returns the same connection object each time it is called.)
- Use the static `DriverManager.getConnection()` method, with either `jdbc:oracle:kprb` or `jdbc:default:connection` as the URL string. (This returns a different connection object each time it is called.)

Using the `defaultConnection()` method is generally recommended.

It is also possible to use the server-side Thin driver for an internal connection (a connection to the database in which your Java code is running), but this is not typical.

**Notes:**

- Alternatively, you can connect using custom JavaBeans provided with OracleJSP. See "Oracle Database-Access JavaBeans" on page 5-13.
- You are not required to register the `OracleDriver` class for connecting with the server-side internal driver, although there is no harm in doing so. This is true whether you are using `getConnection()` or `defaultConnection()` to make the connection.

For more information about server-side connections through Oracle JDBC, see the *Oracle8i JDBC Developer's Guide and Reference.*

### Connecting with the OracleDriver Class defaultConnection() Method

The `oracle.jdbc.driver.OracleDriver` class `defaultConnection()` method is an Oracle extension you can use to make an internal database connection. This method always returns the same connection object. Even if you invoke this method multiple times, assigning the resulting connection object to different variable names, a single connection object is reused.

The `defaultConnection()` method does not take a connect string. For example:

```
import java.sql.*;
import oracle.jdbc.driver.*;

class JDBCConnection
{
  public static Connection connect() throws SQLException
  {
    Connection conn = null;
    try {
      // connect with the server-side internal driver
         OracleDriver ora = new OracleDriver();
         conn = ora.defaultConnection();
      }

    } catch (SQLException e) {...}
    return conn;
  }
}
```

Note that there is no `conn.close()` call in the example. When JDBC code is running inside the target server, the connection is an implicit data channel, not an explicit connection instance as from a client. It should typically not be closed.

If you do call the `close()` method, be aware of the following:

- All connection instances obtained through the `defaultConnection()` method, which actually all reference the same connection object, will be closed and unavailable for further use, with state and resource cleanup as appropriate. Executing `defaultConnection()` afterward would result in a new connection object and, therefore, a new transaction.
- Even though the connection object is closed, the implicit connection to the database will not be closed.

### Connecting with the DriverManager.getConnection() Method

Instead of using the `defaultConnection()` method to make an internal database connection, you can use the static `DriverManager.getConnection()` method with either of the following connect strings:

```
Connection conn = DriverManager.getConnection("jdbc:oracle:kprb:");
```

or:

```
Connection conn = DriverManager.getConnection("jdbc:default:connection:");
```

Any user name or password you include in the URL string is ignored in connecting to the server default connection.

The `DriverManager.getConnection()` method returns a new Java `Connection` object every time you call it. Note that although the method is not creating a new physical connection (only a single implicit connection is used), it is returning a new object.

The fact that `DriverManager.getConnection()` returns a new connection object every time you call it is significant if you are working with object maps, known as "type maps". A type map, for mapping Oracle SQL object types to Java classes, is associated with a specific `Connection` object and with any state that is part of the object. If you want to use multiple type maps as part of your program, then you can call `getConnection()` to create a new `Connection` object for each type map. For general information about type maps, see the *Oracle8i JDBC Developer's Guide and Reference.*

### Connecting with the Server-Side Thin Driver

The Oracle JDBC server-side Thin driver is generally intended for connecting to one database from within another database. It is possible, however, to use the server-side Thin driver for an internal connection. Specify a connect string as you would for any usage of the Oracle JDBC Thin driver.

This feature offers the possible advantage of code portability between the Oracle Servlet Engine and other servlet environments; however, the server-side internal driver offers more efficient performance.

### No Auto-Commit in Server-Side Internal Driver

The JDBC auto-commit feature is disabled in the server-side internal driver. You must commit or roll back changes manually.

### No Connection Pooling or Caching with Server-Side Internal Driver

Connection pooling and caching is not applicable when using the server-side internal driver, because it uses a single implicit database connection. Attempts to use these features through the internal driver may actually degrade performance.

## Use of JNDI by the Oracle Servlet Engine

The Oracle Servlet Engine uses a JNDI mechanism to look up "published" JSP pages and servlets, although this mechanism is generally invisible to the JSP developer or user. Publishing a JSP page, which you accomplish during deployment to OSE, involves either running the Oracle session-shell `publishjsp` command (for deployment with server-side translation) or running the session-shell `publishservlet` command (for deployment with client-side translation).

The `publishservlet` command requires you to specify a virtual path name and a servlet name for the page implementation class. The virtual path name is then used to invoke the page through a URL, or to include or forward to the page from any other page running in OSE.

The `publishjsp` command can either take a virtual path name and servlet name on the command line, or will infer them from the JSP source file name and directory path that you specify.

Both the servlet name and the virtual path name are entered into the JServer JNDI namespace, but the JSP developer or user need only be aware of the virtual path name.

For more information about publishing a JSP page for OSE, see "Translating and Publishing JSP Pages in Oracle8i (Session Shell publishjsp)" on page 6-42 (for deployment with server-side translation) or "Publishing Translated JSP Pages in Oracle8i (Session Shell publishservlet)" on page 6-63 (for deployment with client-side translation).

For general information about how the Oracle Servlet Engine uses JNDI, see the *Oracle8i Oracle Servlet Engine User's Guide*.

## Equivalent Code for OracleJSP Runtime Configuration Parameters

Some OracleJSP configuration parameters take effect during translation; others take effect during runtime. When you deploy JSP pages to the Oracle8*i* database to run in the Oracle Servlet Engine, you can make appropriate translation-time settings through command-line options of the OracleJSP pre-translation tool.

At runtime, however, the Oracle Servlet Engine does not support execution-time configuration parameters. The most significant runtime parameter is `translate_params`, which relates to NLS. For a discussion of equivalent code, see "Code Equivalent to the translate_params Configuration Parameter" on page 8-6.

# Considerations for Apache/JServ Servlet Environments

There are special considerations in running OracleJSP in Apache/JServ-based platforms, including Oracle Internet Application Server release 1.0.x, because this is a servlet 2.0 environment. The servlet 2.0 specification lacked support for some significant features that are available in servlet 2.1 and 2.2 environments.

For information about how to configure an Apache/JServ environment for OracleJSP, see the following sections:

- "Add OracleJSP-Related JAR and ZIP Files to Web Server Classpath" on page A-8
- "Map JSP File Name Extensions to Oracle JspServlet" on page A-11
- "Setting OracleJSP Parameters in Apache/JServ" on page A-25

(If you use Apache/JServ through an Oracle platform, see the installation and configuration documentation for that platform instead.)

The rest of this section, after summarizing the use of Apache/JServ by the Oracle Internet Application Server, discusses the following Apache-specific considerations:

- Dynamic Includes and Forwards in Apache/JServ
- Application Framework for Apache/JServ
- JSP and Servlet Session Sharing
- Directory Alias Translation

## Use of Apache/JServ in the Oracle Internet Application Server

As of Oracle Internet Application Server release 1.0.0 and release 1.0.1, this product uses Apache/JServ as its servlet environment.

As in any Apache/JServ or other servlet 2.0 environment, there are special considerations relating to servlet and JSP usage when using Oracle Internet Application Server release 1.0.x. These are detailed in the sections that follow.

(The Oracle Internet Application Server includes the Oracle HTTP Server, powered by Apache, as its Web server. Be aware that if you use the Oracle HTTP Server `mod_ose` Apache mod to run your JSP application in the Oracle Servlet Engine, you are using the OSE servlet 2.2 environment, not the Internet Application Server Apache/JServ servlet 2.0 environment.)

---

**Note:** Future releases of the Oracle HTTP Server and Oracle Internet Application Server may use a servlet environment other than Apache/JServ.

---

For a brief overview of the Oracle Internet Application Server and its use of the Oracle HTTP Server, see "Support for OracleJSP in Oracle Environments" on page 2-4.

## Dynamic Includes and Forwards in Apache/JServ

JSP dynamic includes (the `jsp:include` action) and forwards (the `jsp:forward` action) rely on request dispatcher functionality that is present in servlet 2.1 and 2.2 environments, but not in servlet 2.0 environments.

OracleJSP, 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 Apache/JServ and other servlet 2.0 environments.

This OracleJSP functionality does not, however, allow dynamic forwards or includes to servlets. (Servlet execution is controlled by the JServ or other servlet container, not the OracleJSP container.)

If you want to include or forward to a servlet, 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 an Apache/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 2.1 or 2.2 environment.

**Notes:**

- 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 `include` or `forward` 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.) Alternatively, you can use the `HTTPClient` class from Innovation GmbH. Oracle8*i* JServer provides a modified version of this class that supports SSL, directly or through a proxy, when you use `https://` for the URL. (See `www.innovation.ch/java/HTTPClient` for general information about this class. Click "Getting Started" for information that includes how to replace the JDK HTTP client with the `HTTPClient` class.) Details of these alternatives are outside the scope of this document, however, and this approach is generally not recommended.

## Application Framework for Apache/JServ

The servlet 2.0 specification does not provide the full servlet context framework for application support that is provided in later specifications.

For servlet 2.0 environments, including Apache/JServ, OracleJSP supplies its own application framework using a file, `globals.jsa`, that you can use as an application marker.

For more information, see "Distinct Applications and Sessions Through globals.jsa" on page 5-38.

## JSP and Servlet Session Sharing

To share HTTP session information between JSP pages and servlets in an Apache/JServ environment, you must configure your environment so that `oracle.jsp.JspServlet` (the servlet that acts as the front-end of the OracleJSP container) is in the same zone as the servlet or servlets with which you want your

JSP pages to share a session. 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, for applications that use a `globals.jsa` file, the OracleJSP configuration parameter `session_sharing` should be set to `true` (the default) for JSP session data to be accessible to servlets. See these sections for related information:

- "OracleJSP Application and Session Support for Servlet 2.0" on page 5-37
- "OracleJSP Configuration Parameters (Non-OSE)" on page A-15
- "OracleJSP Configuration Parameter Settings" on page A-25

## Directory Alias Translation

Apache 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. (An implicit application is created for the Web server document root and each aliasing root. )

Consider the following sample `httpd.conf` entry:

```
Alias /icons/ "/apache/apache139/icons/"
```

This command should result 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.

OracleJSP provides an Apache-specific configuration parameter, `alias_translation`, that works around this limitation when you set `alias_translation=true` (the default setting is `false`).

For information about how to set OracleJSP configuration parameters in an Apache/JServ environment, see "Setting OracleJSP Parameters in Apache/JServ" on page A-25.

# 5

# OracleJSP Extensions

This chapter discusses extended functionality offered by OracleJSP, covering the following topics:

- Portable OracleJSP Programming Extensions
- Oracle-Specific Programming Extensions
- OracleJSP Application and Session Support for Servlet 2.0

Portable extensions are provided through Oracle's JSP Markup Language (JML) custom tags, JML extended datatypes, SQL custom tags, and database-access JavaBeans. You can use these features in other JSP environments.

Non-portable extensions are those that require OracleJSP for translation and execution.

Extended application and session support for servlet 2.0 environments is supplied through Oracle `globals.jsa` functionality and also requires OracleJSP.

# Portable OracleJSP Programming Extensions

The Oracle extensions documented in this section are implemented either through the Oracle JSP Markup Language (JML) sample tag library or through custom JavaBeans. These extensions are portable to any standard JSP environment. This includes the following:

- JML extended datatypes
- XML and XSL support (including JML tags)
- database-access JavaBeans
- JML SQL tags

---

**Important:** To use any of the JML functionality, see "Overview of the JSP Markup Language (JML) Sample Tag Library" on page 7-20.

---

## JML Extended Datatypes

To work around shortcomings for JSP usage in the Java primitive datatypes and `java.lang` wrapper types (as discussed in "OracleJSP Extended Datatypes" on page 2-13), OracleJSP provides the following JavaBean classes in the `oracle.jsp.jml` package to act as wrappers for the most common Java datatypes:

- `JmlBoolean` to represent a `boolean` value
- `JmlNumber` to represent an `int` value
- `JmlFPNumber` to represent a `double` value
- `JmlString` to represent a `String` value

Each of these classes has a single attribute, `value`, and includes methods to get the value, set the value from input in various formats, test whether the value is equal to a value specified in any of several formats, and convert the value to a string.

Alternatively, instead of using the `getValue()` and `setValue()` methods, you can use the `jsp:getProperty` and `jsp:setProperty` tags, as with any other bean.

The following example creates a `JmlNumber` instance called `count` that has `application` scope.

```
<jsp:useBean id="count" class="oracle.jsp.jml.JmlNumber" scope="application" />
```

Later, assuming that the value has been set elsewhere, you can access it as follows:

```
<h3> The current count is <%=count.getValue() %> </h3>
```

The following example creates a `JmlNumber` instance called `maxSize` that has `request` scope, and sets it using `setProperty`:

```
<jsp:useBean id="maxSize" class="oracle.jsp.jml.Number" scope="request" >
    <jsp:setProperty name="maxSize" property="value" value="<%= 25 %>" />
</jsp:useBean>
```

The remainder of this section documents the public methods of the four extended datatype classes, followed by an example.

### Type JmlBoolean

A `JmlBoolean` object represents a Java `boolean` value.

The `getValue()` and `setValue()` methods get or set the bean's `value` property as a Java `boolean` value.

- `boolean getValue()`
- `void setValue(boolean)`

The `setTypedValue()` method has several signatures and can set the bean's `value` property from a string (such as "true" or "false"), a `java.lang.Boolean` value, a Java `boolean` value, or a `JmlBoolean` value. For the string input, conversion of the string is performed according to the same rules as for the standard `java.lang.Boolean.valueOf()` method.

- `void setTypedValue(String)`
- `void setTypedValue(Boolean)`
- `void setTypedValue(boolean)`
- `void setTypedValue(JmlBoolean)`

The `equals()` method tests whether the bean's `value` property is equal to the specified Java `boolean` value.

- `boolean equals(boolean)`

The `typedEquals()` method has several signatures and tests whether the bean's `value` property has a value equivalent to a specified string (such as "true" or "false"), `java.lang.Boolean` value, or `JmlBoolean` value.

- `boolean typedEquals(String)`

- `boolean typedEquals(Boolean)`
- `boolean typedEquals(JmlBoolean)`

The `toString()` method returns the bean's `value` property as a `java.lang.String` value, either "true" or "false".

- `String toString()`

### Type JmlNumber

A `JmlNumber` object represents a 32-bit number equivalent to a Java `int` value.

The `getValue()` and `setValue()` methods get or set the bean's `value` property as a Java `int` value.

- `int getValue()`
- `void setValue(int)`

The `setTypedValue()` method has several signatures and can set the bean's `value` property from a string, a `java.lang.Integer` value, a Java `int` value, or a `JmlNumber` value. For the string input, conversion of the string is performed according to the same rules as for the standard `java.lang.Integer.decode()` method.

- `void setTypedValue(String)`
- `void setTypedValue(Integer)`
- `void setTypedValue(int)`
- `void setTypedValue(JmlNumber)`

The `equals()` method tests whether the bean's value property is equal to the specified Java `int` value.

- `boolean equals(int)`

The `typedEquals()` method has several signatures and tests whether the bean's `value` property has a value equivalent to a specified string (such as "1234"), `java.lang.Number` value, or `JmlNumber` value.

- `boolean typedEquals(String)`
- `boolean typedEquals(Integer)`
- `boolean typedEquals(JmlNumber)`

The `toString()` method returns the bean's `value` property as an equivalent `java.lang.String` value (such as "1234"). This method has the same functionality as the standard `java.lang.Integer.toString()` method.

- `String toString()`

### Type JmlFPNumber

A `JmlFPNumber` object represents a 64-bit floating point number equivalent to a Java `double` value.

The `getValue()` and `setValue()` methods get or set the bean's `value` property as a Java `double` value.

- `double getValue()`
- `void setValue(double)`

The `setTypedValue()` method has several signatures and can set the bean's `value` property from a string (such as "3.57"), a `java.lang.Integer` value, a Java `int` value, a `java.lang.Float` value, a Java `float` value, a `java.lang.Double` value, a Java `double` value, or a `JmlFPNumber` value. For the string input, conversion of the string is according to the same rules as for the standard `java.lang.Double.valueOf()` method.

- `void setTypedValue(String)`
- `void setTypedValue(Integer)`
- `void setTypedValue(int)`
- `void setTypedValue(Float)`
- `void setTypedValue(float)`
- `void setTypedValue(Double)`
- `void setTypedValue(double)`
- `void setTypedValue(JmlFPNumber)`

The `equals()` method tests whether the bean's `value` property is equal to the specified Java `double` value.

- `boolean equals(double)`

The `typedEquals()` method has several signatures and tests whether the bean's `value` property has a value equivalent to a specified string (such as "3.57"), `java.lang.Integer` value, Java `int` value, `java.lang.Float` value, Java `float` value, `java.lang.Double` value, Java `double` value, or `JmlFPNumber` value.

- `boolean typedEquals(String)`
- `boolean typedEquals(Integer)`
- `boolean typedEquals(int)`
- `boolean typedEquals(Float)`
- `boolean typedEquals(float)`
- `boolean typedEquals(Double)`
- `boolean typedEquals(JmlFPNumber)`

The `toString()` method returns the bean's value property as a `java.lang.String` value (such as "3.57"). This method has the same functionality as the standard `java.lang.Double.toString()` method.

- `String toString()`

### Type JmlString

A `JmlString` object represents a `java.lang.String` value.

The `getValue()` and `setValue()` methods get or set the bean's `value` property as a `java.lang.String` value. If the input in a `setValue()` call is null, then the value property is set to an empty (zero-length) string.

- `String getValue()`
- `void setValue(String)`

The `toString()` method is functionally equivalent to the `getValue()` method.

- `String toString()`

The `setTypedValue()` method sets the bean's `value` property according to the specified `JmlString` value. If the `JmlString` value is null, then the value property is set to an empty (zero-length) string.

- `void setTypedValue(JmlString)`

The `isEmpty()` method tests whether the bean's `value` property is an empty (zero-length) string: ""

- `boolean isEmpty()`

The `equals()` method has two signatures and tests whether the bean's `value` property is equal to a specified `java.lang.String` value or `JmlString` value.

- `boolean equals(String)`
- `boolean equals(JmlString)`

### JML Datatypes Example

This example illustrates use of JML datatype JavaBeans for management of simple datatypes at scope. The page declares four session objects—one for each JML type. The page presents a form that allows you to enter values for each of these types. Once new values are submitted, the form displays both the new values and the previously set values. In the process of generating this output, the page updates the session objects with the new form values.

```
<jsp:useBean id = "submitCount" class = "oracle.jsp.jml.JmlNumber" scope = "session" />

<jsp:useBean id = "bool" class = "oracle.jsp.jml.JmlBoolean" scope = "session" >
        <jsp:setProperty name = "bool" property = "value" param = "fBoolean" />
</jsp:useBean>

<jsp:useBean id = "num" class = "oracle.jsp.jml.JmlNumber" scope = "session" >
        <jsp:setProperty name = "num" property = "value" param = "fNumber" />
</jsp:useBean>

<jsp:useBean id = "fpnum" class = "oracle.jsp.jml.JmlFPNumber" scope = "session" >
        <jsp:setProperty name = "fpnum" property = "value" param = "fFPNumber" />
</jsp:useBean>

<jsp:useBean id = "str" class = "oracle.jsp.jml.JmlString" scope = "session" >
        <jsp:setProperty name = "str" property = "value" param = "fString" />
</jsp:useBean>


<HTML>

<HEAD>
        <META HTTP-EQUIV="Content-Type" CONTENT="text/html;CHARSET=iso-8859-1">
        <META NAME="GENERATOR" Content="Visual Page 1.1 for Windows">
        <TITLE>OracleJSP Extended Datatypes Sample</TITLE>
</HEAD>

<BODY BACKGROUND="images/bg.gif" BGCOLOR="#FFFFFF">
```

```
<% if (submitCount.getValue() > 1) { %>
        <h3> Last submitted values </h3>
        <ul>
                <li> bool: <%= bool.getValue() %>
                <li> num: <%= num.getValue() %>
                <li> fpnum: <%= fpnum.getValue() %>
                <li> string: <%= str.getValue() %>
        </ul>
<% }

   if (submitCount.getValue() > 0) { %>

        <jsp:setProperty name = "bool" property = "value" param = "fBoolean" />
        <jsp:setProperty name = "num" property = "value" param = "fNumber" />
        <jsp:setProperty name = "fpnum" property = "value" param = "fFPNumber" />
        <jsp:setProperty name = "str" property = "value" param = "fString" />

        <h3> New submitted values </h3>
        <ul>
                <li> bool: <jsp:getProperty name="bool" property="value" />
                <li> num: <jsp:getProperty name="num" property="value" />
                <li> fpnum: <jsp:getProperty name="fpnum" property="value" />
                <li> string: <jsp:getProperty name="str" property="value" />
        </ul>
<% } %>

<jsp:setProperty name = "submitCount" property = "value" value = "<%= submitCount.getValue() + 1
%>" />

<FORM ACTION="index.jsp" METHOD="POST" ENCTYPE="application/x-www-form-urlencoded">
<P> <pre>
 boolean test: <INPUT TYPE="text" NAME="fBoolean" VALUE="<%= bool.getValue() %>" >
  number test: <INPUT TYPE="text" NAME="fNumber" VALUE="<%= num.getValue() %>" >
fpnumber test: <INPUT TYPE="text" NAME="fFPNumber" VALUE="<%= fpnum.getValue() %>" >
  string test: <INPUT TYPE="text" NAME="fString" VALUE= "<%= str.getValue() %>" >
</pre>

<P> <INPUT TYPE="submit">

</FORM>

</BODY>

</HTML>
```

## OracleJSP Support for XML and XSL

JSP technology can be used to produce dynamic XML pages as well as dynamic HTML pages. OracleJSP supports the use of XML and XSL technology with JSP pages in two ways:

- The OracleJSP translator includes logic to recognize standard XML-alternative JSP syntax.
- OracleJSP provides JML tags to apply an XSL stylesheet to the JSP output stream.

Additionally, the `oracle.xml.sql.query.OracleXMLQuery` class is provided with Oracle8*i* as part of the XML-SQL utility for XML functionality in database queries. This class requires file `xsu12.jar` (for JDK 1.2.x) or `xsu111.jar` (for JDK 1.1.x), which is also required for XML functionality in the OracleJSP database-access JavaBeans, and which is provided with Oracle8*i* release 8.1.7.

For a JSP sample using `OracleXMLQuery`, see "XML Query—XMLQuery.jsp" on page 9-36.

For information about the `OracleXMLQuery` class and other XML-SQL utility features, refer to the *Oracle8i Application Developer's Guide - XML*.

### 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.

OracleJSP does not use this DTD directly or require you to use a `jsp:root` tag, but the OracleJSP 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:<br>`<%@ page ... %>`<br>`<%@ include ... %>` | Such as:<br>`<jsp:directive.page ... />`<br>`<jsp:directive.include ... />` |
| `<%! ... %>` (declaration) | `<jsp:declaration>`<br>*...declarations go here...*<br>`</jsp:declaration>` |
| `<%= ... %>` (expression) | `<jsp:expression>`<br>*...expression goes here...*<br>`</jsp:expression>` |
| `<% ... %>` (scriptlet) | `<jsp:scriptlet>`<br>*...code fragment goes here...*<br>`</jsp:scriptlet>` |

JSP action tags, such as `jsp:useBean`, for the most part already use syntax that complies with XML. Changes due to quoting conventions or for request-time attribute expressions may be necessary, however.

## JML Tags for XSL Stylesheets

Many uses of XML and XSL for dynamic pages require an XSL transformation to occur in the server before results are returned to the client.

OracleJSP provides two synonymous JML tags to simplify this process. Use either the JML `transform` tag or the JML `styleSheet` tag (their effects are identical), as in the following example:

```
<jml:transform href="xslRef" >

   ...Tag body contains regular JSP commands and static text that
   produce the XML code that the stylesheet is to be applies to...

</jml:transform >
```

(The `jml:` prefix is used by convention, but you can specify any prefix in your `taglib` directive.)

**Important:** If you will use any JML tags, refer to "Overview of the JSP Markup Language (JML) Sample Tag Library" on page 7-20.

Note the following regarding the `href` parameter:

- It can refer to either a static XSL stylesheet or a dynamically generated one. For example, it can refer to a JSP page or servlet that generates the stylesheet.
- It can be a fully qualified URL (`http://`*host*`[:`*port*`]/yourpath`), an application-relative JSP reference (starting with "`/`"), or a page-relative JSP reference (not starting with "`/`"). See "Indirectly Requesting a JSP Page" on page 1-9 for information about application-relative and page-relative paths.
- It can be dynamically specified. By default, the value of `href` is a static Java string. However, you can use standard JSP expression syntax to provide a dynamically computed value.

Typically, you would use the `transform` or `styleSheet` tag to transform an entire page. However, the tag applies only to what is in its body, between its start and end tags. Therefore, you can have distinct XSL blocks within a page, each block bounded by its own `transform` or `styleSheet` tag set, specifying its own `href` pointer to the appropriate stylesheet.

## XSL Example using jml:transform

This section provides a sample XSL stylesheet and a sample JSP page that uses the `jml:transform` tag to filter its output through the stylesheet. (This is a simplistic example—the XML in the page is static. A more realistic example might use the JSP page to dynamically generate all or part of the XML before performing the transformation.)

### Sample Stylesheet: hello.xsl

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="page">
   <html>
    <head>
     <title>
      <xsl:value-of select="title"/>
     </title>
```

```
    </head>
    <body bgcolor="#ffffff">
     <xsl:apply-templates/>
    </body>
   </html>
  </xsl:template>

  <xsl:template match="title">
   <h1 align="center">
    <xsl:apply-templates/>
   </h1>
  </xsl:template>

  <xsl:template match="paragraph">
   <p align="center">
    <i>
     <xsl:apply-templates/>
    </i>
   </p>
  </xsl:template>

</xsl:stylesheet>
```

### Sample JSP Page: hello.jsp

```
<%@ page session = "false" %>
<%@ taglib uri="/WEB-INF/jmltaglib.tld" prefix="jml" %>

<jml:transform href="style/hello.xsl" >

<page>
 <title>Hello</title>
 <content>
  <paragraph>This is my first XML/XSL file!</paragraph>
 </content>
</page>

</jml:transform>
```

This example results in the following output:


## Oracle Database-Access JavaBeans

OracleJSP supplies a set of custom JavaBeans for accessing an Oracle database. The following beans are included in the `oracle.jsp.dbutil` package:

- `ConnBean` opens a simple database connection.
- `ConnCacheBean` uses Oracle's 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.

For examples using these beans, see "Database-Access JavaBean Samples" on page 9-21.

All four beans implement the OracleJSP `JspScopeListener` interface for event notification. See "OracleJSP Event Handling—JspScopeListener" on page 5-32.

This section presumes a working knowledge of Oracle JDBC. Consult the *Oracle8i JDBC Developer's Guide and Reference* as necessary.

---

**Important:** To use the Oracle database-access JavaBeans, install the file `ojsputil.jar` and include it in your classpath. This file is provided with the OracleJSP installation. For XML-related methods and functionality, you will also need file `xsu12.jar` (for JDK 1.2.x) or `xsu111.jar` (for JDK 1.1.x), which is provided with Oracle8*i* release 8.1.7.

---

### ConnBean for a Database Connection

Use `oracle.jsp.dbutil.ConnBean` to establish a simple database connection (one that uses no connection pooling or caching).

---

**Notes:**

- For queries only, it is simpler to use `DBBean`, which has its own connection mechanism.
- To use connection caching, use `ConnCacheBean` instead.
- As with any JavaBean you use in a JSP page, you can set any of the `ConnBean` properties with a `jsp:setProperty` action instead of using the setter method directly.

---

`ConnBean` has the following properties:

- `user` (user ID for database schema)
- `password` (user password)
- `URL` (database connection string)
- `stmtCacheSize` (cache size for Oracle JDBC statement caching)

  Setting `stmtCacheSize` enables the Oracle JDBC statement caching feature. See "JDBC Statement Caching" on page 4-6 for a brief overview of statement caching features and limitations.
- `executeBatch` (batch size for Oracle JDBC update batching)

  Setting `executeBatch` enables Oracle JDBC update batching. See "Update Batching" on page 4-7 for a brief overview of this feature.

- `preFetch` (number of statements to prefetch in Oracle JDBC row prefetching)

  Setting `preFetch` enables Oracle JDBC row prefetching. Refer to "Row Prefetching" on page 4-8 for a brief overview of this feature.

`ConnBean` provides the following setter and getter methods for these properties:

- `void setUser(String)`
- `String getUser()`
- `void setPassword(String)`
- `String getPassword()`
- `void setURL(String)`
- `String getURL()`
- `void setStmtCacheSize(int)`
- `int getStmtCacheSize()`
- `void setExecuteBatch(int)`
- `int getExecuteBatch()`
- `void setPreFetch(int)`
- `int getPreFetch()`

Use the following methods to open and close a connection:

- `void connect()`—Establish a database connection using `ConnBean` property settings.
- `void close()`—Close the connection and any open cursors.

Use the following method to open a cursor and return a `CursorBean` object:

- `CursorBean getCursorBean(int, String)`

or:

- `CursorBean getCursorBean(int)`

  Input the following:

  – one of the following `int` constants to specify the type of JDBC statement you want: `CursorBean.PLAIN_STMT` (for a `Statement` object), `CursorBean.PREP_STMT` (for a `PreparedStatement` object), or `CursorBean.CALL_STMT` (for a `CallableStatement` object)

- a string specifying the SQL operation to execute (optional; alternatively, the SQL operation can be specified in the `CursorBean` method call that executes the statement)

See "CursorBean for DML and Stored Procedures" on page 5-20 for information about `CursorBean` functionality.

### ConnCacheBean for Connection Caching

Use `oracle.jsp.dbutil.ConnCacheBean` to use the Oracle JDBC connection caching mechanism (using JDBC 2.0 connection pooling) for your database connections. For a brief overview of connection caching, see "Database Connection Caching" on page 4-6.

---

**Notes:**

- To use simple connection objects (no pooling or caching), use `ConnBean` instead.
- `ConnCacheBean` extends `OracleConnectionCacheImpl`, which extends `OracleDataSource` (both in Oracle JDBC package `oracle.jdbc.pool`). See the *Oracle8i JDBC Developer's Guide and Reference* (release 8.1.6 or later) for more information about these Oracle JDBC classes.
- As with any JavaBean you use in a JSP page, you can set any of the `ConnCacheBean` properties with a `jsp:setProperty` action instead of using the setter method directly.
- Unlike `ConnBean`, when you use `ConnCacheBean`, you use normal `Connection` object functionality to create and execute statement objects.

---

`ConnCacheBean` has the following properties:

- `user` (user ID for database schema)
- `password` (user password)
- `URL` (database connection string)
- `maxLimit` (maximum number of connections allowed by this cache)
- `minLimit` (minimum number of connections existing for this cache; if you are using fewer than this number, then there will also be connections in the "idle pool" of the cache)

- `stmtCacheSize` (cache size for Oracle JDBC statement caching)

  Setting `stmtCacheSize` enables the Oracle JDBC statement caching feature. See "JDBC Statement Caching" on page 4-6 for a brief overview of Oracle JDBC statement caching features and limitations.

- `cacheScheme` (type of cache):
  - `DYNAMIC_SCHEME`—New pooled connections can be created above and beyond the maximum limit, but each one is automatically closed and freed as soon as the logical connection instance that it provided is no longer in use.
  - `FIXED_WAIT_SCHEME`—When the maximum limit is reached, any new connection waits for an existing connection object to be released.
  - `FIXED_RETURN_NULL_SCHEME`—When the maximum limit is reached, any new connection fails (`null` is returned) until connection objects have been released.

The `ConnCacheBean` class supports methods defined in the Oracle JDBC `OracleConnectionCacheImpl` class, including the following getter and setter methods for its properties:

- `void setUser(String)`
- `String getUser()`
- `void setPassword(String)`
- `String getPassword()`
- `void setURL(String)`
- `String getURL()`
- `void setMaxLimit(int)`
- `int getMaxLimit()`
- `void setMinLimit(int)`
- `int getMinLimit()`
- `void setStmtCacheSize(int)`
- `int getStmtCacheSize()`

- `void setCacheScheme(int)`

  Specify `ConnCacheBean.DYNAMIC_SCHEME`, `ConnCacheBean.FIXED_WAIT_SCHEME`, or `ConnCacheBean.FIXED_RETURN_NULL_SCHEME`.

- `int getCacheScheme()`

  Returns `ConnCacheBean.DYNAMIC_SCHEME`, `ConnCacheBean.FIXED_WAIT_SCHEME`, or `ConnCacheBean.FIXED_RETURN_NULL_SCHEME`.

The `ConnCacheBean` class also inherits properties and related getter and setter methods from the `oracle.jdbc.pool.OracleDataSource` class. This provides getter and setter methods for the following properties: `databaseName`, `dataSourceName`, `description`, `networkProtocol`, `portNumber`, `serverName`, and `driverType`. For information about these properties and their getter and setter methods, see the *Oracle8i JDBC Developer's Guide and Reference*.

Use the following methods to open and close a connection:

- `Connection getConnection()`—Get a connection from the connection cache using `ConnCacheBean` property settings.
- `void close()`—Close all connections and any open cursors.

Although the `ConnCacheBean` class does not support Oracle JDBC update batching and row prefetching directly, you can enable these features by calling the `setDefaultExecuteBatch(int)` and `setDefaultRowPrefetch(int)` methods of the `Connection` object that you retrieve from the `getConnection()` method. Alternatively, you can use the `setExecuteBatch(int)` and `setRowPrefetch(int)` methods of JDBC statement objects that you create from the `Connection` object (update batching is supported only in prepared statements). See "Update Batching" on page 4-7 and "Row Prefetching" on page 4-8 for brief overviews of these features.

### DBBean for Queries Only

Use `oracle.jsp.dbutil.DBBean` to execute queries only.

---

**Notes:**

- `DBBean` has its own connection mechanism; do not use `ConnBean`.
- Use `CursorBean` for any other DML operations (`UPDATE`, `INSERT`, `DELETE`, or stored procedure calls).
- As with any JavaBean you use in a JSP page, you can set any of the `DBBean` properties with a `jsp:setProperty` statement instead of using the setter method directly.

---

`DBBean` has the following properties:

- `user` (user ID for database schema)
- `password` (user password)
- `URL` (database connection string)

`DBBean` provides the following setter and getter methods for these properties:

- `void setUser(String)`
- `String getUser()`
- `void setPassword(String)`
- `String getPassword()`
- `void setURL(String)`
- `String getURL()`

Use the following methods to open and close a connection:

- `void connect()`—Establish a database connection using `DBBean` property settings.
- `void close()`—Close the connection and any open cursors.

Use either of the following methods to execute a query.

- `String getResultAsHTMLTable(String)`—Input a string with the `SELECT` statement.

This method returns a string with the HTML commands necessary to output the result set as an HTML table. SQL column names (or aliases) are used for the table column headers.

- `String getResultAsXMLString(String)`—Input a string with the `SELECT` statement.

  This method returns the result set as an XML string, using SQL names (or aliases) for the XML tags.

### CursorBean for DML and Stored Procedures

Use `oracle.jsp.dbutil.CursorBean` for `SELECT`, `UPDATE`, `INSERT`, or `DELETE` operations or stored procedure calls on a simple connection. It uses a previously defined `ConnBean` object for the connection.

You can specify a SQL operation in a `ConnBean` object `getCursorBean()` call, or through a call to one of the `create()`, `execute()`, or `executeQuery()` methods of a `CursorBean` object as described below.

`CursorBean` supports scrollable and updatable cursors, update batching, row prefetching, and query timeout limits. For information about these Oracle JDBC features, see the *Oracle8i JDBC Developer's Guide and Reference*, release 8.1.6 or later.

---

**Notes:**

- To use connection caching, use `ConnCacheBean` and normal `Connection` object functionality. Do not use `CursorBean`.
- As with any JavaBean you use in a JSP page, you can set any of the `CursorBean` properties with a `jsp:setProperty` action instead of using the setter method directly.

---

`CursorBean` has the following properties:

- `executeBatch` (batch size for Oracle JDBC update batching)

  Setting this property enables Oracle JDBC update batching.
- `preFetch` (number of statements to prefetch in Oracle JDBC row prefetching)

  Setting this property enables Oracle JDBC row prefetching.
- `queryTimeout` (number of seconds for the driver to wait for a statement to execute before issuing a timeout)

- `resultSetType` (scrollability of result set):
  - `TYPE_FORWARD_ONLY` (default)—A result set that can scroll only forward (using the `next()` method) and is not positionable.
  - `TYPE_SCROLL_INSENSITIVE`—A result set that can scroll forward or backward and is positionable, but is not sensitive to underlying database changes.
  - `TYPE_SCROLL_SENSITIVE`—A result set that can scroll forward or backward, is positionable, and is sensitive to underlying database changes.

  See the *Oracle8i JDBC Developer's Guide and Reference* for information about result set scrollability types.

- `resultSetConcurrency` (updatability of result set):
  - `CONCUR_READ_ONLY` (default)—A result set that is read-only (cannot be updated).
  - `CONCUR_UPDATABLE`—A result set that is updatable.

  See the *Oracle8i JDBC Developer's Guide and Reference* for information about updatable result sets.

You can set these properties with the following methods to enable Oracle JDBC features, as desired:

- `void setExecuteBatch(int)`
- `int getExecuteBatch()`
- `void setPreFetch(int)`
- `int getPreFetch()`
- `void setQueryTimeout(int)`
- `int getQueryTimeout()`
- `void setResultSetConcurrency(int)`

  Specify `CursorBean.CONCUR_READ_ONLY` or `CursorBean.CONCUR_UPDATABLE`.

- `int getResultSetConcurrency()`

  Returns `CursorBean.CONCUR_READ_ONLY` or `CursorBean.CONCUR_UPDATABLE`.

- `void setResultSetType(int)`

  Specify `CursorBean.TYPE_FORWARD_ONLY`, `CursorBean.TYPE_SCROLL_INSENSITIVE`, or `CursorBean.TYPE_SCROLL_SENSITIVE`.

- `int getResultSetType()`

  Returns `CursorBean.TYPE_FORWARD_ONLY`, `CursorBean.TYPE_SCROLL_INSENSITIVE`, or `CursorBean.TYPE_SCROLL_SENSITIVE`.

To execute a query once a `CursorBean` instance has been defined in a `jsp:useBean` statement, you can use `CursorBean` methods to create a cursor in one of two ways. You can use the following methods to create the cursor and supply a connection in separate steps:

- `void create()`
- `void setConnBean(ConnBean)`

Or you can combine the process into a single step:

- `void create(ConnBean)`

(Set up the `ConnBean` object as described in "ConnBean for a Database Connection" on page 5-14.)

Then use the following method to specify and execute a query. (This uses a JDBC plain `Statement` object behind the scenes.)

- `ResultSet executeQuery(String)`

  Specify a string with the `SELECT` statement.

Alternatively, if you want to format the result set as an HTML table or XML string, use either of the following methods instead of `executeQuery()`:

- `String getResultAsHTMLTable(String)`

  Returns a string with HTML statements to create an HTML table for the result set. Specify a string with the `SELECT` statement.

- `String getResultAsXMLString(String)`

  Returns the result set data in an XML string. Specify a string with the `SELECT` statement.

To execute an `UPDATE`, `INSERT`, or `DELETE` statement once a `CursorBean` instance has been defined in a `jsp:useBean` action, you can use `CursorBean` methods to create a cursor in one of two ways. You can use the following methods to create the cursor (specifying a statement type as an integer and SQL statement as a string) and supply a connection:

- `void create(int, String)`
- `void setConnBean(ConnBean)`

Or you can combine the process into a single step:

- `void create(ConnBean, int, String)`

(Set up the `ConnBean` object as described in "ConnBean for a Database Connection" on page 5-14.)

The `int` input is to specify one of the following constants to specify the type of JDBC statement you want: `CursorBean.PLAIN_STMT` (for a `Statement` object), `CursorBean.PREP_STMT` (for a `PreparedStatement` object), or `CursorBean.CALL_STMT` (for a `CallableStatement` object).

The `String` input is to specify the SQL statement.

Then use the following method to execute the `INSERT`, `UPDATE`, or `DELETE` statement. (You can ignore the `boolean` return value.)

- `boolean execute()`

Or for update batching, use the following method, which returns the number of rows affected. (See below for how to enable update batching.)

- `int executeUpdate()`

> **Note:** The `execute()` and `executeUpdate()` methods can optionally take a `String` to specify a SQL operation. The corresponding `create()` call, as well as the `getCursorBean()` call in `ConnBean`, optionally does *not* take a `String` to specify the SQL operation. Specify an operation either on statement creation or execution, but not both.

Additionally, `CursorBean` supports Oracle JDBC statement and result set functionality such as the `registerOutParameter()` method, `setXXX()` methods, and `getXXX()` methods.

Use the following method to close the database cursor:

- `void close()`

## OracleJSP Tag Library for SQL

With release 8.1.7, OracleJSP supplies a custom tag library for SQL functionality (separate from the JML custom tag library).

The following tags are provided:

- `dbOpen`—Open a database connection.
- `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).

These tags are described in the following subsections. For examples, see "SQL Tag Examples" on page 5-28.

Note the following requirements for using SQL tags:

- Install the file `ojsputil.jar` and include it in your classpath. This file is provided with the OracleJSP installation.
- Make sure the tag library description file, `sqltaglib.tld`, is deployed with the application and is in the location specified in the `taglib` directives of your JSP pages, such as in the following example:

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
```

For general information about JSP 1.1 tag library usage, including tag library description files and `taglib` directives, see "Standard Tag Library Framework" on page 7-2.

### SQL dbOpen Tag

Use the `dbOpen` tag to open a database connection.

```
<sql:dbOpen
   [ connId="connection-id" ]
     user="username"
     password="password"
     URL="databaseURL" >
...
</sql:dbOpen>
```

Nested code that you want to execute through this connection can go into the tag body, between the `dbOpen` start and end tags. (See "SQL Tag Examples" on page 5-28.) If you use the optional `connId` parameter to set a connection identifier, then code to execute through this connection can reference the connection identifier and does *not* have to be between the `dbOpen` start and end tags. (The connection identifier can be any arbitrary string.)

Note that you do *not* have to hardcode a password into the JSP page (which would be a security concern). Instead, you can get it and other parameters from the `request` object, as follows:

```
<sql:dbOpen connId="conn1" user=<%=request.getParameter("user")%>
            password=<%=request.getParameter("password")%> URL="url" />
```

(In this example you do not need a tag body for code that will use this connection; statements using the connection can reference it through the `conn1` value of `connId`.)

If you set a connection identifier, then the connection is not closed until you close it explicitly with a `dbClose` tag. Without a connection identifier, the connection is closed automatically when the `</sql:dbOpen>` end tag is encountered.

This tag uses a `ConnBean` object for the connection. You can optionally set the additional `ConnBean` properties `stmtCacheSize`, `preFetch`, and `batchSize` to enable those Oracle JDBC features. See "ConnBean for a Database Connection" on page 5-14 for more information.

### SQL dbClose Tag

Use the `dbClose` tag to close a connection associated with the optional `connId` parameter specified in a `dbOpen` tag. If `connId` is not used in the `dbOpen` tag, then the connection is closed automatically when the `dbOpen` end tag is reached; no `dbClose` tag is required.

```
<sql:dbClose connId="connection-id" />
```

---

**Note:** In an OracleJSP environment, you can have the connection closed automatically with session-based event handling through the Oracle `JspScopeListener` mechanism. See "OracleJSP Event Handling—JspScopeListener" on page 5-32.

---

## SQL dbQuery Tag

Use the dbQuery tag to execute a query, outputting the result either as a JDBC result set, HTML table, or XML string. Place the SELECT statement (one only) in the tag body, between the dbQuery start and end tags.

```
<sql:dbQuery
   [ queryId="query-id" ]
   [ connId="connection-id" ]
   [ output="HTML|XML|JDBC"] >
    ...SELECT statement (one only)...
 </sql:dbQuery>
```

> **Important:** In release 8.1.7 (OracleJSP 1.1.0.0.0), do *not* terminate the SELECT statement with a semi-colon. This would result in a syntax error.

All parameters of this tag are optional, depending on your intended uses as described below.

You must use the queryId parameter to set a query identifier if you want to process the result set using a dbNextRow tag. The queryId can be any arbitrary string.

Additionally, if the queryId parameter is present, then the cursor is not closed until you close it explicitly with a dbCloseQuery tag. Without a query identifier, the cursor is closed automatically when the </sql:dbQuery> end tag is encountered.

If connId is not specified, then dbQuery must be nested within the body of a dbOpen tag and will use the connection opened in the dbOpen tag.

For the output type:

- HTML puts the result set into an HTML table (default).
- XML puts the result set into an XML string.
- JDBC puts the result set into a JDBC ResultSet object that can be processed using the dbNextRow tag to iterate through the rows.

This tag uses a CursorBean object for the cursor. See "CursorBean for DML and Stored Procedures" on page 5-20 for information about CursorBean functionality.

### SQL dbCloseQuery Tag

Use the `dbCloseQuery` tag to close a cursor associated with the optional `queryId` parameter specified in a `dbQuery` tag. If `queryId` is not used in the `dbQuery` tag, then the cursor is closed automatically when the `dbQuery` end tag is reached; no `dbCloseQuery` tag is required.

```
<sql:dbCloseQuery queryId="query-id" />
```

---

**Note:** In an OracleJSP environment, you can have the cursor closed automatically with session-based event handling through the Oracle `JspScopeListener` mechanism. See "OracleJSP Event Handling—JspScopeListener" on page 5-32.

---

### SQL dbNextRow Tag

Use the `dbNextRow` tag to process each row of a result set obtained in a `dbQuery` tag and associated with the specified `queryId`. Place the processing code in the tag body, between the `dbNextRow` start and end tags. The body is executed for each row of the result set.

For you to use the `dbNextRow` tag, the `dbQuery` tag must specify `output=JDBC`, and specify a `queryId` for the `dbNextRow` tag to reference.

```
<sql:dbNextRow queryId="query-id" >

...Row processing...

</sql:dbNextRow >
```

The result set object is created in an instance of the tag-extra-info class of the `dbQuery` tag (see "Tag Library Description Files" on page 7-10 for information about tag-extra-info classes).

### SQL dbExecute Tag

Use the `dbExecute` tag to execute any DML or DDL statement (one only). Place the statement in the tag body, between the `dbExecute` start and end tags.

```
<sql:dbExecute
   [connId="connection-id"]
   [output="yes|no"] >
   ...DML or DDL statement (one only)...
</sql:dbExecute >
```

---

**Important:** In release 8.1.7, do *not* terminate the DML or DDL statement with a semi-colon. This would result in a syntax error.

---

If you do not specify `connId`, then you must nest `dbExecute` within the body of a `dbOpen` tag and use the connection opened in the `dbOpen` tag.

If `output=yes`, then for DML statements, the HTML string "*number* row[s] affected" will be output to the browser to notify the user how many database rows were affected by the operation; for DDL statements, the statement execution status will be printed. The default setting is `no`.

This tag uses a `CursorBean` object for the cursor. See "CursorBean for DML and Stored Procedures" on page 5-20 for information about `CursorBean` functionality.

## SQL Tag Examples

The following examples show how to use the OracleJSP SQL tags. (To run them yourself, you will need to set the URL, user name, and password appropriately.)

### Example 1: Query with Connection ID

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
   <HTML>
       <HEAD>
           <TITLE>A simple example with open, query, and close tags</TITLE>
       </HEAD>
       <BODY BGCOLOR="#FFFFFF">
            <HR>
            <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
                        user="scott" password="tiger" connId="con1">
            </sql:dbOpen>
            <sql:dbQuery connId="con1">
               select * from EMP
            </sql:dbQuery>
            <sql:dbClose connId="con1" />
            <HR>
       </BODY>
  </HTML>
```

#### Example 2: Query Nested in dbOpen Tag

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
   <HTML>
       <HEAD>
           <TITLE>Nested Tag with Query inside Open  </TITLE>
       </HEAD>
       <BODY BGCOLOR="#FFFFFF">
            <HR>
            <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
                        user="scott" password="tiger">
               <sql:dbQuery>
                  select * from EMP
               </sql:dbQuery>
            </sql:dbOpen>
            <HR>
       </BODY>
  </HTML>
```

#### Example 3: Query with XML Output

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
   <HTML>
       <HEAD>
           <TITLE>A simple tagLib with XML output</TITLE>
       </HEAD>
       <BODY BGCOLOR="#FFFFFF">
            <HR>
            <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
                        user="scott" password="tiger">
               <sql:dbQuery output="xml">
                 select * from EMP
               </sql:dbQuery>
            </sql:dbOpen>
            <HR>
       </BODY>
  </HTML>
```

**Example 4: Result Set Iteration**

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
   <HTML>
       <HEAD>
           <TITLE>Result Set Iteration Sample </TITLE>
       </HEAD>
       <BODY BGCOLOR="#FFFFFF">
            <HR>
            <sql:dbOpen connId="con1" URL="jdbc:oracle:thin:@dlsun991:1521:816"
                       user="scott" password="tiger">
            </sql:dbOpen>
            <sql:dbQuery connId="con1" output="jdbc" queryId="myquery">
               select * from EMP
            </sql:dbQuery>
            <sql:dbNextRow queryId="myquery">
                <%= myquery.getString(1) %>
            </sql:dbNextRow>
            <sql:dbCloseQuery queryId="myquery" />
            <sql:dbClose connId="con1" />
            <HR>
       </BODY>
  </HTML>
```

**Example 5: DDL and DML Statements** This example uses an HTML form to let the user specify what kind of DML or DDL statement to execute.

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
<HEAD><TITLE>DML Sample</TITLE></HEAD>
<FORM METHOD=get>
<INPUT TYPE="submit" name="drop" VALUE="drop table test_table"><br>
<INPUT TYPE="submit" name="create"
                     VALUE="create table test_table (col1 NUMBER)"><br>
<INPUT TYPE="submit" name="insert"
                     VALUE="insert into test_table values (1234)"><br>
<INPUT TYPE="submit" name="select" VALUE="select * from test_table"><br>
</FORM>
<BODY BGCOLOR="#FFFFFF">
Result:
            <HR>
            <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
             user="scott" password="tiger">
              <% if (request.getParameter("drop")!=null) { %>
              <sql:dbExecute output="yes">
```

```
                drop table test_table
             </sql:dbExecute>
             <% } %>
             <% if (request.getParameter("create")!=null) { %>
             <sql:dbExecute output="yes">
                create table test_table (col1 NUMBER)
             </sql:dbExecute>
             <% } %>
             <% if (request.getParameter("insert")!=null) { %>
             <sql:dbExecute output="yes">
               insert into test_table values (1234)
             </sql:dbExecute>
             <% } %>
             <% if (request.getParameter("select")!=null) { %>
             <sql:dbQuery>
                select * from test_table
             </sql:dbQuery>
             <% } %>
           </sql:dbOpen>
           <HR>
</BODY>
</HTML>
```

# Oracle-Specific Programming Extensions

The OracleJSP extensions documented in this section are not portable to other JSP environments. This includes the following:

- event-handling through the Oracle `JspScopeListener` mechanism
- support for SQLJ, a standard syntax for embedding SQL statements directly into Java code
- use of JDBC performance enhancement features

---

**Notes:**

- For servlet 2.0 environments, OracleJSP provides non-portable extensions through a mechanism called `globals.jsa` to support a Web application framework. "OracleJSP Application and Session Support for Servlet 2.0" on page 5-37 describes this mechanism.
- OracleJSP also provides extended (and non-portable) NLS support, which is described in "OracleJSP Extended Support for Multibyte Parameter Encoding" on page 8-5.

---

## OracleJSP Event Handling—JspScopeListener

In standard servlet and JSP technology, only session-based events are supported. OracleJSP extends this support through the `JspScopeListener` interface and `JspScopeEvent` class in the `oracle.jsp.event` package. The OracleJSP mechanism supports the four standard JSP scopes for event-handling for any Java objects used in a JSP application:

- `page`
- `request`
- `session`
- `application`

For Java objects that are used in your application, implement the `JspScopeListener` interface in the appropriate class, then attach objects of that class to a JSP scope using tags such as `jsp:useBean`.

When the end of a scope is reached, objects that implement `JspScopeListener` and have been attached to the scope will be so notified. The OracleJSP container accomplishes this by sending a JspScopeEvent instance to such objects through the outOfScope() method specified in the JspScopeListener interface.

Properties of the JspScopeEvent object include the following:

- the scope that is ending (one of the constants PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE, or APPLICATION_SCOPE)
- the container object that is the repository for objects at this scope (one of the implicit objects page, request, session, or application)
- the name of the object that the notification pertains to (the name of the instance of the class that implements JspScopeListener)
- the JSP implicit application object

The OracleJSP event listener mechanism significantly benefits developers who want to always free object resources that are of page or request scope, regardless of error conditions. It frees these developers from having to surround their page implementations with Java try/catch/finally blocks.

For a complete sample, see "Page Using JspScopeListener—scope.jsp" on page 9-32.

## OracleJSP Support for Oracle SQLJ

SQLJ is a standard syntax for embedding static SQL instructions directly in Java code, greatly simplifying database access programming. OracleJSP and the OracleJSP translator support Oracle SQLJ, allowing you to use SQLJ syntax in JSP scriptlets. SQLJ statements are indicated by the #sql token.

### SQLJ JSP Code Example

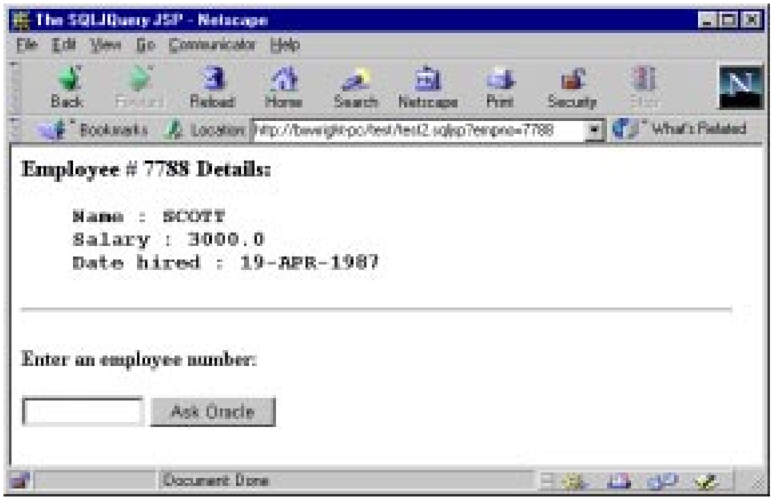
Following is a sample SQLJ JSP page. (The page directive imports classes that are typically required by SQLJ.)

```
<%@ 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 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.)
- OracleJSP requires Oracle SQLJ release 8.1.6.1 or higher.
- In the future, OracleJSP will support `language="sqlj"` in a `page` directive to trigger the Oracle SQLJ translator during JSP translation. For forward compatibility, it is recommended as a good programming practice that you begin using this directive immediately.

For further examples of using SQLJ in JSP pages, see "SQLJ Queries—SQLJSelectInto.sqljsp and SQLJIterator.sqljsp" on page 9-37.

For general information about Oracle SQLJ programming features and syntax, see the *Oracle8i SQLJ Developer's Guide and Reference*.

### Triggering the SQLJ Translator

You can trigger the OracleJSP translator to invoke the Oracle SQLJ translator by using the file name extension `.sqljsp` for the JSP source file.

This results in the OracleJSP 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-Demand Translation)" on page 6-6.

---

**Important:**

- To use Oracle SQLJ, you will have to install appropriate SQLJ ZIP files (depending on your environment) and add them to your classpath. See "Required and Optional Files for OracleJSP" on page A-3.
- Do not use the same base file name for a `.jsp` file and a `.sqljsp` file in the same application, because they would result in the same generated class name and `.java` file name.

---

### 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:

- For on-demand translation, use the OracleJSP `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. (The `sqljcmd` parameter is not available prior to OracleJSP release 1.1.0.0.0.)

  For information, see the `sqljcmd` description in "OracleJSP Configuration Parameters (Non-OSE)" on page A-15. For how to set configuration parameters, see "OracleJSP Configuration Parameter Settings" on page A-25.

- For pre-translation 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 6-27 and "Option Descriptions for ojspc" on page 6-27.

For general information about Oracle SQLJ options, see the *Oracle8i SQLJ Developer's Guide and Reference*.

# OracleJSP Application and Session Support for Servlet 2.0

OracleJSP defines a file, `globals.jsa`, as a mechanism for implementing the JSP specification in a servlet 2.0 environment. Web applications and servlet contexts were not fully defined in the servlet 2.0 specification.

This section discusses the `globals.jsa` mechanism and covers the following topics:

- Overview of globals.jsa Functionality
- Overview of globals.jsa Syntax and Semantics
- The globals.jsa Event Handlers
- Global Declarations and Directives

For sample applications, see "Samples Using globals.jsa for Servlet 2.0 Environments" on page 9-41.

---

**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 by OracleJSP 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 an OracleJSP 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 OracleJSP 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.

OracleJSP 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 OracleJSP 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. For servlets in a servlet 2.2 environment, you can include the necessary configuration in the standard `web.xml` deployment descriptor.

### 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 Apache/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 OracleJSP 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 OracleJSP 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, OracleJSP extends this functionality 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 OracleJSP container unloads an application.

For more information, see "The globals.jsa Event Handlers" on page 5-42.

## 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 5-46.

For details about each of the four event handlers, see "The globals.jsa Event Handlers" on page 5-42.

---

**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 OracleJSP 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) and white space—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 OracleJSP 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 "globals.jsa Example for Application Events—lotto.jsp" on page 9-41. 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 OracleJSP unloads the JSP application. Unloading occurs whenever a previously loaded page is reloaded after on-demand dynamic re-translation (unless the OracleJSP `unsafe_reload` configuration parameter is enabled), or when the OracleJSP 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) and white space—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 OracleJSP 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 "globals.jsa Example for Application Events—lotto.jsp" on page 9-41. 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 OracleJSP 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 OracleJSP 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 OracleJSP 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) and white space—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 OracleJSP 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
- globals.jsa Declarations
- Global JavaBeans
- globals.jsa Structure
- Global Declarations and Directives Example

### 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 declared in a `globals.jsa` file (such as by a `jsp:useBean` statement) does not need to 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 `jsp:useBean` or `jml:useVariable`. Globally declared objects must be of either `session` or `application` scope (not `page` or `request` scope).

Nested tags are supported. Thus, a `jsp:setProperty` command can be nested in a `jsp:useBean` declaration. (A translation error occurs if `jsp:setProperty` is used outside a `jsp:useBean` declaration.)

### 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 following sample `globals.jsa` file 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 "globals.jsa Example for Global Declarations—index2.jsp" on page 9-47.

```
<%-- 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 --%>
```

# 6

# JSP Translation and Deployment

This chapter primarily discusses considerations and procedures for deploying JSP applications into Oracle8*i* to run in the Oracle Servlet Engine. It also describes general OracleJSP translation features and briefly discusses deployment in other environments, particularly the Apache/JServ environment used by the Oracle Internet Application Server.

The following topics are covered:

# Functionality of the OracleJSP 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 OracleJSP translator, focusing on its behavior in on-demand translation environments, such as Apache/JServ or the Oracle Internet Application Server. The following topics are covered:

- Generated Code Features
- Generated Package and Class Names (On-Demand Translation)
- Generated Files and Locations (On-Demand Translation)
- Sample Page Implementation Class Source

> **Important:** Implementation details discussed in this section regarding package and class naming, output file locations, and generated code are for illustrative purposes only. The precise details apply to OracleJSP release 8.1.7 (1.1.0.0.0) only, in accordance with JSP 1.1 specifications, as applicable. These details are subject to change in future releases.

You must pre-translate JSP pages targeted for the Oracle Servlet Engine, either as a result of running the session shell `publishjsp` command (for deployment with server-side translation) or by running the `ojspc` pre-translation tool directly (for deployment with client-side translation). In either case, there are some differences in functionality compared with the discussion in this section, such as in placement of output files. See "Translating and Publishing JSP Pages in Oracle8i (Session Shell publishjsp)" on page 6-42 and "The ojspc Pre-Translation Tool" on page 6-23 for information.

## Generated Code Features

This section discusses general features of the page implementation class code that is produced by the OracleJSP translator in translating JSP source (`.jsp` and `.sqljsp` files).

### Features of Page Implementation Class Code

When the OracleJSP 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 (`oracle.jsp.runtime.HttpJsp`) provided by the OracleJSP 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 and expressions 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` (which can be done in a `page` directive).

For introductory information about key JSP and servlet classes and interfaces, see Appendix B, "Servlet and JSP Technical Background".

### Inner Class for Static Text

The service method, `_jspService()`, of the page implementation class includes print commands—`out.print()` calls on the implicit `out` object—to print any static text in the JSP page. The OracleJSP 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 (usually for deployment to Oracle8*i*), 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 OracleJSP 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 4-13.) You can request this feature through the OracleJSP `external_resource` configuration parameter (for on-demand translation) or the `ojspc -extres` option (for pre-translation). Enabling hotloading (for deployment to Oracle8*i*) also results in the static text going into a resource file.

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 only noteworthy if you are in a client-side pre-translation scenario.)

---

## Generated Package and Class Names (On-Demand Translation)

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 OracleJSP creates package and class names when it generates code during translation.

---

**Important:** In OracleJSP release 8.1.7 (1.1.0.0.0), the URL path directory and `.jsp` file name (which are used in generating package and class names) are restricted to valid Java package and class identifiers. For example, a path directory or `.jsp` name cannot begin with a number. It is also invalid to use Java reserved words, such as `for` or `class`, as a path directory or `.jsp` name (such as `class.jsp`). Such implementation details may change in future releases.

---

### 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 OracleJSP release 8.1.7 (1.1.0.0.0), 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 doc root or 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 OracleJSP release 8.1.7 (1.1.0.0.0), 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.

In OracleJSP release 8.1.7 (1.1.0.0.0), the translator determines the case of the class name according to the case of the file name. For example:

- `UserLogin.jsp` results in class `UserLogin`.
- `Userlogin.jsp` results in class `Userlogin`.
- `userlogin.jsp` results in 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 (On-Demand Translation)

This section describes files that are generated by the OracleJSP 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 6-34.

The following subsections mention several OracleJSP configuration parameters. For more information about them, see "OracleJSP Configuration Parameters (Non-OSE)" on page A-15 and "OracleJSP Configuration Parameter Settings" on page A-25.

### Files Generated by OracleJSP

This section considers both regular JSP pages (`.jsp` files) and SQLJ JSP pages (`.sqljsp` files) in listing files that are generated by the OracleJSP translator. For the file name examples, presume a file `Foo.jsp` or `Foo.sqljsp` is being translated.

Source files:

- A `.sqlj` file is produced by the OracleJSP translator if the page is a SQLJ JSP page (for example, `Foo.sqlj`).
- A `.java` file is produced for the page implementation class and inner class (for example, `Foo.java`). It is produced either directly by the OracleJSP 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 or an alternative release of the Oracle SQLJ translator by using the OracleJSP `sqljcmd` configuration parameter.)

Binary files:

- In the case of a SQLJ JSP page, 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 OracleJSP `sqljcmd` configuration parameter). The resource file or `.class` file has "Foo" as part of its name.
- A `.class` file is produced by the Java compiler for the page implementation class. (The Java compiler is `javac` by default, but you can specify an alternative compiler using the OracleJSP `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.

- A `.res` Java resource file is optionally produced for the static page content (for example, `Foo.res`) if the OracleJSP `external_resource` 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, such as `_Foo.java` or `_Foo.class`.

### OracleJSP Translator Output File Locations

OracleJSP uses the Web server document repository to generate or load translated JSP pages.

By default, the root directory is the Web server doc root directory (for Apache/JServ) or the servlet context root directory of the application the page belongs to.

You can specify an alternative root directory through the OracleJSP `page_repository_root` configuration parameter.

In OracleJSP release 8.1.7 (1.1.0.0.0), generated files are placed as follows (implementation details may change in future releases):

- If the `.jsp` (or `.sqljsp`) file is directly in the root directory, then OracleJSP will place generated files into a `_pages` subdirectory directly under the root directory.
- If the `.jsp` (or `.sqljsp`) file is in a subdirectory under the root directory, then an equivalent directory structure is created under the `_pages` subdirectory for the generated files.

  As an example, consider an Apache/JServ environment with an `htdocs` root directory. If a `.jsp` file is in the following directory:

  ```
  htdocs/subdir/test
  ```

  then generated files will be placed in the following directory:

  ```
  htdocs/_pages/subdir/test
  ```

## Sample Page Implementation Class Source

This section uses an example to illustrate the information in the preceding sections.

Consider the following scenario:

- JSP page code is in the file `hello.jsp`.
- The page is executed in an Apache/JServ environment.
- The `hello.jsp` file is located in the following directory:

  ```
  htdocs/test
  ```

---

**Important:** Code generation details discussed here are according to Oracle's implementation of the JSP 1.1 specification. Details may change in the future, as the result of either changes in the specification or changes in how Oracle implements aspects that are not specified.

---

### Sample Page Source: hello.jsp

Following is the JSP code in `hello.jsp`:

```
<HTML>
<HEAD><TITLE>The Hello 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>
```

### Sample: Generated Package and Class

Because `hello.jsp` is in the `test` subdirectory of the root directory (`htdocs`), OracleJSP release 8.1.7 (1.1.0.0.0) generates the following package name in the page implementation code:

```
package test;
```

The Java class name is identical to the base name of the `.jsp` file (case included), so the following class definition is generated in the page implementation code:

```
public class hello extends oracle.jsp.runtime.HttpJsp
{
   ...
}
```

(Because the page implementation class is invisible to the end-user, the fact that its name does not adhere to Java capitalization conventions is generally not a concern.)

### Sample: Generated Files

Because `hello.jsp` is located as follows:

```
htdocs/test/hello.jsp
```

OracleJSP release 8.1.7 (1.1.0.0.0) generates output files as follows (the page implementation class `.java` file and `.class` file, and the inner class `.class` file, respectively):

```
htdocs/_pages/test/hello.java
htdocs/_pages/test/hello.class
htdocs/_pages/test/hello$__jsp_StaticText.class
```

**Note:** These file names are based specifically on the OracleJSP 1.1.0.0.0 implementation; the exact details may change in future releases. All file names will always include the base "hello", however.

### Sample Page Implementation Code: hello.java

Following is the generated page implementation class Java code (`hello.java`), as generated by OracleJSP release 8.1.7 (1.1.0.0.0):

```
package test;

import oracle.jsp.runtime.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.io.*;
import java.util.*;
```

```
import java.lang.reflect.*;
import java.beans.*;


public class hello extends oracle.jsp.runtime.HttpJsp {

  public final String _globalsClassName = null;

  // ** Begin Declarations


  // ** End Declarations

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

    /* set up the intrinsic variables using the pageContext goober:
    ** session = HttpSession
    ** application = ServletContext
    ** out = JspWriter
    ** page = this
    ** config = ServletConfig
    ** all session/app beans declared in globals.jsa
    */
    JspFactory factory = JspFactory.getDefaultFactory();
    PageContext pageContext = factory.getPageContext( this, request, response,
null, true, JspWriter.DEFAULT_BUFFER, true);
    // Note: this is not emitted if the session directive == false
    HttpSession session = pageContext.getSession();
    if (pageContext.getAttribute(OracleJspRuntime.JSP_REQUEST_REDIRECTED,
PageContext.REQUEST_SCOPE) != null) {
      pageContext.setAttribute(OracleJspRuntime.JSP_PAGE_DONTNOTIFY, "true",
PageContext.PAGE_SCOPE);
      factory.releasePageContext(pageContext);
      return;
}
    ServletContext application = pageContext.getServletContext();
    JspWriter out = pageContext.getOut();
    hello page = this;
    ServletConfig config = pageContext.getServletConfig();

    try {
      // global beans
      // end global beans
```

```
      out.print(__jsp_StaticText.text[0]);
      String user=request.getParameter("user");
      out.print(__jsp_StaticText.text[1]);
      out.print(  (user==null) ? "" : user );
      out.print(__jsp_StaticText.text[2]);
      out.print(  new java.util.Date() );
      out.print(__jsp_StaticText.text[3]);

      out.flush();

    }
    catch( Exception e) {
      try {
        if (out != null) out.clear();
      }
      catch( Exception clearException) {
      }
      pageContext.handlePageException( e);
    }
    finally {
      if (out != null) out.close();
      factory.releasePageContext(pageContext);
    }

  }
  private static class __jsp_StaticText {
    private static final char text[][]=new char[4][];
    static {
      text[0] =
      "<HTML>\r\n<HEAD><TITLE>The Welcome User
JSP</TITLE></HEAD>\r\n<BODY>\r\n".toCharArray();
      text[1] =
      "\r\n<H3>Welcome ".toCharArray();
      text[2] =
      "!</H3>\r\n<P><B> Today is ".toCharArray();
      text[3] =
      ". Have a nice day! :-)</B></P>\r\n<B>Enter name:</B>\r\n<FORM
METHOD=get>\r\n<INPUT TYPE=\"text\" NAME=\"user\" SIZE=15>\r\n<INPUT
TYPE=\"submit\" VALUE=\"Submit
name\">\r\n</FORM>\r\n</BODY>\r\n</HTML>".toCharArray();
    }
  }
}
```

# Overview of Features and Logistics in Deployment to Oracle8*i*

This section is an overview of considerations and logistics in deploying a JSP application into Oracle8*i* to run in the Oracle Servlet Engine. The following topics are covered:

- Database Schema Objects for Java
- Oracle HTTP Server as a Front-End Web Server
- URLs for the Oracle Servlet Engine
- Static Files for JSP Applications in the Oracle Servlet Engine
- Server-Side Versus Client-Side Translation
- Overview of Hotloaded Classes in Oracle8i

## Database Schema Objects for Java

Java code that executes in the Oracle Servlet Engine uses an Oracle8*i* JVM inside the database. The code must be loaded into a particular database schema as one or more *schema objects*.

The three kinds of schema objects for Java are:

- *source schema objects* (corresponding to Java source files)
- *class schema objects* (corresponding to Java class files)
- *resource schema objects* (corresponding to Java resource files)

Each schema object is an individual library unit in the database. When you query the `ALL_OBJECTS` table of the schema, Java schema objects are seen as type `JAVA SOURCE`, `JAVA CLASS`, or `JAVA RESOURCE`, respectively.

See the *Oracle8i Java Developer's Guide* for more information.

### Loading Java Files to Create Schema Objects

The JServer `loadjava` tool is used to load Java files into the database as schema objects. (See "Overview of the loadjava Tool" on page 6-36.)

When you compile on the client and load the `.class` file directly, `loadjava` stores the `.class` file as a class schema object in the database.

When you load a resource file (such as a `.res` file for static JSP content or `.ser` profile file for SQLJ), `loadjava` stores the resource file as a resource schema object in the database.

When you load a `.java` (or `.sqlj`) source file, `loadjava` stores the source file as a source schema object in the database and optionally compiles it inside the database to create one or more class schema objects.

When you load a `.jsp` or `.sqljsp` page source file (for server-side translation), `loadjava` stores the page source as a resource schema object. During server-side translation (through the JServer session shell `publishjsp` command), server-side `loadjava` is invoked automatically to create source schema objects, class schema objects, and resource schema objects during translation and compilation.

(See "Tools and Commands for Translation and Deployment to Oracle8i" on page 6-23 for an overview of the `loadjava` and session shell tools.)

### Schema Object Full Names and Short Names

The two forms of schema object names in Oracle8*i* are *full names* and *short names*.

Full names are fully qualified and are used as the schema object names wherever possible. If any full name contains more than 31 characters, however, or contains characters that are illegal or cannot be converted to characters in the database character set, then the Oracle8*i* server converts the full name to a short name to employ as the name of the schema object, keeping track of both names and how to convert between them. If the full name contains 31 characters or less and has no illegal or inconvertible characters, then the full name is used as the schema object name.

For more information about these and about other file naming considerations, including `DBMS_JAVA` procedures to retrieve a full name from a short name and a short name from a full name, see the *Oracle8i Java Developer's Guide.*

### Java Schema Object Package Determination During Loading

During loading of Java files into the database, the `loadjava` tool uses the following logic to determine the package for Java schema objects it creates:

- For source schema objects (created from `.java` and `.sqlj` files) and class schema objects (created from `.class` files or by compiling `.java` files), the schema package is determined by any package information in the Java code.

  For example, a class `Foo` that specifies the package `dir1.dir2` and is being loaded into the `SCOTT` schema will be stored in the schema as follows:

  ```
  SCOTT:dir1/dir2/Foo
  ```

---

**Note:** When pre-translating a JSP page with the `ojspc` tool (for deployment to Oracle8*i* with client-side translation), you can specify the package of the generated `.java` file through the `ojspc -packageName` option.

---

- For resource schema objects (created from `.res` and `.ser` Java resource files, for example), the schema package is determined by any path information in the `loadjava` command line (if the Java resource file is being loaded directly) or the JAR file (if the Java resource file is being loaded as part of a JAR file).

  For example, a `.res` file being loaded into the `SCOTT` schema as `dir3/dir4/abcd.res` will be stored in a schema object as follows:

  ```
  SCOTT:dir3/dir4/abcd.res
  ```

### Publishing Schema Objects

Any JSP page (or servlet) that will run in the Oracle Servlet Engine must be "published", a process that makes its executable Java code (the class schema objects) accessible through entries in the JServer JNDI namespace.

Publishing the JSP page links its page implementation class schema object to a servlet path (and optionally to a non-default servlet context path). The servlet path (and context path, if applicable) becomes part of the URL that an end-user would specify to access and execute the page. See "URLs for the Oracle Servlet Engine" on page 6-15 for more information.

To publish a JSP page, use the Oracle8*i* session shell `publishjsp` command for the "deployment with server-side translation" scenario, or the session shell `publishservlet` command for the "deployment with client-side translation" scenario. See "Translating and Publishing JSP Pages in Oracle8i (Session Shell publishjsp)" on page 6-42 or "Publishing Translated JSP Pages in Oracle8i (Session Shell publishservlet)" on page 6-63.

## Oracle HTTP Server as a Front-End Web Server

JSP pages and servlets running in the Oracle Servlet Engine are typically accessed through the Oracle HTTP Server (powered by Apache) and its `mod_ose` module, although it is possible to use OSE itself as the Web server.

For more information about the role of the Oracle HTTP Server and `mod_ose`, see "Role of the Oracle HTTP Server, Powered by Apache" on page 2-6.

## URLs for the Oracle Servlet Engine

As with servlet URLs in general, URLs to invoke JSP pages running in the Oracle Servlet Engine are formed by a combination of two components (in addition to the hostname and port):

- the *context path* of the servlet context in OSE, as determined when the servlet context was created
- the *servlet path* of the JSP page in OSE (often referred to as the "virtual path"), as determined when the JSP page was published

The context path for the OSE default context, `/webdomains/contexts/default`, is simply:

```
/
```

The context path for any other OSE servlet context you create, which you accomplish using the Oracle8*i* session shell `createcontext` command, is whatever you specify in the `createcontext -virtualpath` option. (It is conventional, but not required, to specify that the context path be the same as the context name.)

> **Note:** The `-virtualpath` option is required whenever you execute the `createcontext` command.

For general information about the session shell `createcontext` command, see the *Oracle8i Java Tools Reference*. For an overview of the Oracle8*i* session shell, see "Overview of the sess_sh Session Shell Tool" on page 6-38.

The servlet path (JSP page "virtual path") is determined by how you publish the JSP page, as follows:

- If you use the session shell `publishjsp` command (for server-side translation), then it is determined by the `publishjsp -virtualpath` option, or else is the same as the specified schema path by default.
- If you use the session shell `publishservlet` command (after client-side translation), then it is determined by the `publishservlet -virtualpath` option (which you must specify when you use `publishservlet` for a JSP page).

See "Translating and Publishing JSP Pages in Oracle8i (Session Shell publishjsp)" on page 6-42 or "Publishing Translated JSP Pages in Oracle8i (Session Shell publishservlet)" on page 6-63.

**Example 1** As an example, consider a JSP page that is published to the OSE default context with a servlet path (virtual path), as follows:

```
mydir/mypage.jsp
```

This page is accessed as follows:

```
http://host[:port]/mydir/mypage.jsp
```

You can access it from another page in the application, say `mydir/mypage2.jsp`, in either of the following ways (the first is a page-relative path; the second is an application-relative path):

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

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

**Example 2** Now consider a servlet context that is created as follows (`$` is the session shell prompt):

```
$ createcontext -virtualpath mycontext /webdomains mycontext
```

This does the following:

- It creates the servlet context `/webdomains/contexts/mycontext` (all servlet contexts in the `/webdomains` domain go under `/webdomains/contexts`).
- It specifies the context path to be the same as the context name (`mycontext`).

If `mydir/mypage.jsp` is published to the `mycontext` servlet context, it is accessed as follows:

```
http://host[:port]/mycontext/mydir/mypage.jsp
```

You can access it from another page in the application, say `mydir/mypage2.jsp`, in either of the following ways (the first is a page-relative path; the second is an application-relative path):

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

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

The syntax for the dynamic `jsp:include` statements is the same as in Example 1. Even though a different servlet context is used, the path of the pages relative to the context is unchanged.

**Example 3** Now consider a servlet context that is created as follows (`$` is the session shell prompt):

```
$ createcontext -virtualpath mywebapp /webdomains mycontext
```

This does the following:

- It creates the servlet context `/webdomains/contexts/mycontext` as in Example 2.
- However, it defines a context path, `mywebapp`, that is different from the context name. It is this context path, not the context name, that is used in the URL.

In this case, if `mydir/mypage.jsp` is published to the `mycontext` servlet context, it is accessed as follows:

```
http://host[:port]/mywebapp/mydir/mypage.jsp
```

You can access it from another page in the application, say `mydir/mypage2.jsp`, in either of the following ways (the first is a page-relative path; the second is an application-relative path):

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

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

## Static Files for JSP Applications in the Oracle Servlet Engine

This section describes the required placement of static files, such as HTML files, that are used in a JSP application that runs in the Oracle Servlet Engine.

The information in this section is independent of whether the Oracle HTTP Server (powered by Apache) is used as a front-end Web server for OSE, or OSE is used directly.

### Files for Dynamic Includes and Forwards

Static files that are dynamic `include` or `forward` targets (`jsp:include` or `jsp:forward`) in a JSP application running in the Oracle Servlet Engine must be manually moved or copied to the OSE doc root directory corresponding to the servlet context of the application. When you create an OSE servlet context (using the session shell `createcontext` command), you specify a doc root directory through the `createcontext -docroot` option. Each OSE doc root directory is linked to the JServer JNDI namespace.

OSE doc root directories are *outside* the database. The JNDI lookup mechanism for static files is a front-end for the file system of the server where the database resides.

The doc root for the OSE default servlet context, `/webdomains/contexts/default`, is the following:

```
$ORACLE_HOME/jis/public_html
```

Whenever you create an additional servlet context with the session shell `createcontext` command, you can use the `createcontext -docroot` option to specify a doc root directory. (For more information about the session shell `createcontext` command, see the *Oracle8i Java Tools Reference*.)

---

**Note:** If you are migrating your JSP application from Apache to OSE, it is advisable to copy static files from the Apache doc root to the OSE servlet context doc root, as opposed to mapping the OSE servlet context doc root to the Apache doc root. Mapping the doc roots may ultimately cause confusion.

---

### Files for Static Includes

Any file that is statically included (through an `include` directive) by a JSP page, whether it is another JSP page or a static file such as an HTML file, must be accessible by the OracleJSP translator during translation.

In the case of a JSP application targeted for OSE, there are two translation scenarios:

- server-side translation

  This is where you load a `.jsp` file into the database as a Java resource, then use `publishjsp` to invoke the OracleJSP translator in the server. (See "Deployment to Oracle8i with Server-Side Translation" on page 6-41.)

  In this case, static files must be loaded into the database beforehand, using `loadjava`, as resource schema objects.

- client-side translation

  This is where you translate a `.jsp` file on the client using `ojspc` and load the generated components into the database.

  In this case, static files do not have to be in the server at all. They only have to be accessible by `ojspc` on the client during translation. (For application-relative static `include` directives, see the discussion of the `ojspc -appRoot` option under "Option Descriptions for ojspc" on page 6-27.)

## Server-Side Versus Client-Side Translation

Developers who are deploying their JSP pages to Oracle8*i* to run in the Oracle Servlet Engine can translate either in the server or on the client.

Deployment with server-side translation requires two steps:

1. Run `loadjava` to load the JSP page source (`.jsp` or `.sqljsp` file) into Oracle8*i* as a resource schema object. (You must also load any required Java classes or other required JSP pages.)
2. Run the session shell `publishjsp` command. This will automatically accomplish the following:
   - The JSP page source is translated into Java code for the page implementation class (first producing a SQLJ source file and invoking the SQLJ translator in the case of a SQLJ JSP page).
   - The Java code is compiled into one or more class files.
   - The page implementation class is optionally hotloaded (if you specified the `publishjsp -hotload` option).
   - The page implementation class is published for execution in the database.

   This step also produces source schema objects, class schema objects, and resource schema objects for all generated `.java` files (and `.sqlj` files for `.sqljsp` pages), `.class` files, and resource files, respectively.

See "Deployment to Oracle8i with Server-Side Translation" on page 6-41 for more information.

Deployment with client-side translation requires three or, optionally, four steps:

1. Run the OracleJSP pre-translation tool, `ojspc`. This accomplishes the following:
   - The JSP page source is translated into Java code for the page implementation class. (In the case of a SQLJ JSP page, `ojspc` first produces a SQLJ source file then invokes the SQLJ translator to produce Java code.)
   - A Java resource file is optionally produced for static text, depending on the `ojspc -extres` and `-hotload` options.
   - The Java code is compiled into its class files.
2. Run the Oracle8*i* `loadjava` utility to load the class files and any resource files into Oracle8*i* as class schema objects and resource schema objects.

3. Optionally hotload the classes (if you enabled the `ojspc -hotload` option during translation) by using the Oracle8*i* session shell `java` command to execute the `main()` method of the page implementation class.

4. Run the session shell `publishservlet` command to publish the page implementation classes for execution in the database.

See "Deployment to Oracle8i with Client-Side Translation" on page 6-54 for more information.

If you are using Oracle JDeveloper, you may find it more convenient to translate on the client using the OracleJSP translator provided with JDeveloper and then deploy the resulting classes and resources, as in steps 2, 3, and 4.

If you are not using JDeveloper, however, translating in the server is likely to be more convenient, because the session shell `publishjsp` command combines translation, optional hotloading, and publishing into a single step.

In addition, either of the following situations may dictate the need to translate in the server:

- if required libraries are not available on the client
- if you want to compile against the exact set of classes that will be used at runtime

## Overview of Hotloaded Classes in Oracle8*i*

Oracle8*i* JServer offers a feature known as *hotloading* classes, for more efficient use of `static final` variables (constants). This becomes relevant whenever the hotloaded classes might be used by multiple concurrent users.

A separate JVM is invoked for each JServer database session. Normally each session gets its own copy of all `static final` variables in its session space or, in the case of literal strings, in a hashtable known as the *intern table* in shared memory. Use of literal strings in the intern table is synchronized across sessions.

The processing of literal strings is especially relevant to JSP pages. By default (without hotloading), the static text in a JSP page is ultimately represented as literal strings.

---

**Note:** This section refers to the OracleJSP pre-translation tool (`ojspc`), the Oracle session shell tool (`sess_sh`), and the session shell `publishjsp` command. For an overview of these tools, see "Tools and Commands for Translation and Deployment to Oracle8i" on page 6-23.

---

### Enabling and Accomplishing Hotloading

The ability to hotload a JSP page is enabled during translation, through the `ojspc -hotload` option (for client-side translation) or the `publishjsp -hotload` option (for server-side translation).

Enabling the `-hotload` option directs the OracleJSP translator to do the following:

- It generates code in the page implementation class to allow hotloading, by creating a hotloading method and a `main()` method that invokes the hotloading method.
- It writes static text to a Java resource file. (Otherwise, static text is written to an inner class of the page implementation class.)

The hotloading itself is accomplished as follows:

- For deployment with client-side translation, you must hotload as an extra deployment step. After translating with the `ojspc -hotload` option enabled and loading the page into the database, and before publishing the page, you must use the session shell `java` command to invoke the `main()` method of the page implementation class. Details of the process are discussed in "Deployment to Oracle8i with Client-Side Translation" on page 6-54.
- For deployment with server-side translation, hotloading is accomplished automatically as part of `publishjsp` functionality when you enable the `publishjsp -hotload` option.

The act of hotloading a page implementation class, either directly through the session shell `java` command or indirectly through the `publishjsp` command, actually just makes the inner class static text shareable among multiple JVMs in the database.

### Features and Advantages of Hotloading

Hotloading classes results in the following logistical features and advantages:

- The translator generates code to read the Java resource containing the static text in static initializers, to initialize the `char` arrays representing static text.
- During hotloading, each hotloaded inner class is initialized only once, and static JSP text is converted into static Java `char` arrays only once.

  These `char` arrays, instead of being stored in the synchronized intern table, are stored elsewhere in a global area that is shared across all sessions without synchronization (which is feasible because of the knowledge that none of the variables will change).

Hotloading, by avoiding synchronization and other costly overhead, can significantly improve the runtime performance and scalability of JSP pages executed in the Oracle Servlet Engine. Furthermore, when a hotloaded class is referenced, the class initializer is *not* rerun. The session has instant access to the literal strings and other `static final` variables.

In addition to allowing better performance of individual JSP pages, hotloading reduces overall CPU usage of the server.

---

**Note:** JSP pages that will not be used by multiple users concurrently, or small JSP pages with few literal strings, may have little or no performance improvement from hotloading.

---

# Tools and Commands for Translation and Deployment to Oracle8*i*

Oracle provides the following tools to use, as applicable, in translating JSP pages and deploying them into Oracle8*i*. How they are implemented depends on your operating system (such as shell scripts for Solaris or `.bat` files for Windows NT):

- `ojspc` (OracleJSP pre-translation tool)
- `loadjava` (tool for loading JSP pages or Java files into the database)
- `sess_sh` (Oracle8*i* session shell tool)

Deployment with client-side translation requires all three tools. Pre-translate JSP pages on the client using `ojspc`, load the translated pages into Oracle8*i* using `loadjava`, and publish them using the session shell `publishservlet` command.

Deployment with server-side translation does not require `ojspc`. Load the untranslated JSP pages into Oracle8*i* using `loadjava`, then translate and publish them using the session shell `publishjsp` command.

The `loadjava` and `sess_sh` tools are general-purpose tools for the Oracle8*i* JServer environment; `ojspc` is for JSP pages only.

**Notes:**

- Another tool, the JServer Accelerator, is relevant if you want to natively compile your application to run in Oracle8*i*. This tool, invoked as `ncomp`, is documented in the *Oracle8i Java Tools Reference*.
- The tools discussed in this section are located in the `[ORACLE_HOME]/bin` directory.

## The ojspc Pre-Translation Tool

The first step in deploying a JSP application to Oracle8*i* with client-side translation is to run the OracleJSP pre-translation tool, `ojspc`.

You will then use `loadjava`, introduced in the next section, to load the resulting `.class` files and resource files (if any) into the database as class schema objects and resource schema objects, respectively.

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


---

**Notes:** There are other possible scenarios, such as in a middle-tier environment, for using `ojspc` to pre-translate JSP page. See "Use of ojspc for Pre-Translation for Non-OSE Environments" on page 6-68.

---

### 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 OracleJSP 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).

And following is the default `ojspc` functionality for a SQLJ JSP page:

- It takes a `.sqljsp` file as an argument instead of a `.jsp` file.
- It invokes the OracleJSP translator to translate the `.sqljsp` file 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.

  For information about SQLJ profiles, see the *Oracle8i 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 `-hotload` and `-extres` option descriptions below), `ojspc` options direct the OracleJSP 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.

For general information about OracleJSP translator output (particularly in the on-demand translation scenario), see "Generated Files and Locations (On-Demand Translation)" on page 6-6.

---

**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 6-27.

The second column notes comparable or related OracleJSP configuration parameters for on-demand translation environments (such as Apache/JServ).

---

**Note:** A boolean `ojspc` option is enabled by typing only the option name, not by setting it to `true`. Setting it to `true` will cause an error.

---

***Table 6–1 Options for ojspc Pre-Translation Utility***

| Option | Related OracleJSP Configuration Parameters | Description | Default |
|---|---|---|---|
| -addclasspath | classpath (related, but with different functionality) | additional classpath entries for `javac` | empty (no additional path entries) |
| -appRoot | n/a | 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 | page_repository_root | location where `ojspc` should place generated binary files (`.class` and resource) | current directory |
| -extend | n/a | class for the generated page implementation class to extend | empty |

*Table 6–1 Options for ojspc Pre-Translation Utility (Cont.)*

| Option | Related OracleJSP Configuration Parameters | Description | Default |
|---|---|---|---|
| -extres | external_resource | boolean to direct `ojspc` to generate an external resource file for static text from the `.jsp` file | false |
| -hotload **(for OSE only)** | n/a | boolean to direct `ojspc` to implement code in the page implementation class to allow hotloading | false |
| -implement | n/a | 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 |
| -packageName | n/a | package name for the generated page implementation class | empty (generate package names per `.jsp` file location) |
| -S-<sqlj option> | sqljcmd | `-S` prefix followed by an Oracle SQLJ option (for `.sqljsp` files) | empty |
| -srcdir | page_repository_root | location where `ojspc` should place generated source files (`.java` and `.sqlj`) | current directory |
| -verbose | n/a | boolean to direct `ojspc` to print status information as it executes | false |
| -version | n/a | boolean to direct `ojspc` to display the OracleJSP version number | 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.
- 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 `-hotload`, *not* `-hotload true`.)

Following is an example:

```
% ojspc -d /myapp/mybindir -srcdir /myapp/mysrcdir -hotload 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.

(The `-addclasspath` setting is also used by the SQLJ translator for SQLJ JSP pages.)

---

**Notes:** In an on-demand translation scenario, the OracleJSP `classpath` configuration parameter provides related, although different, functionality. See "OracleJSP Configuration Parameters (Non-OSE)" on page A-15.

---

**-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:

- It is used 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.
- It is used 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 6-34.)

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 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 opening "`/`"), 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` and `-hotload` options are Java resource files, as are `.ser` profile files produced by the SQLJ translator for SQLJ JSP pages.)

The specified path is taken simply as a file system path (not an application-relative or page-relative path).

In environments such as Windows NT that allow spaces in directory names, enclose the directory name in quotes.

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 6-34 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 an on-demand translation scenario, the OracleJSP `page_repository_root` configuration parameter provides related functionality. See "OracleJSP Configuration Parameters (Non-OSE)" on page A-15.

---

**-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 is useful for source-level JSP debugging, such as when using Oracle JDeveloper.

---

**Note:** In an on-demand translation scenario, the OracleJSP `emit_debuginfo` configuration parameter provides the same functionality. See "OracleJSP Configuration Parameters (Non-OSE)" on page A-15.

---

**-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. For release 8.1.7 it will be the same name as the JSP name, but with the `.res` suffix (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 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 4-13.

---

**Notes:**

- The inner class is still created and must still be deployed.
- In an on-demand translation scenario, the OracleJSP `external_resource` configuration parameter provides the same functionality. See "OracleJSP Configuration Parameters (Non-OSE)" on page A-15.

---

**-hotload** (boolean; `ojspc` default: `false`) **(for OSE only)**

Enable this flag to allow hotloading. This is relevant only if you will be loading the translated pages into Oracle8*i* to run in the Oracle Servlet Engine.

The `-hotload` flag directs `ojspc` to do the following:

1. Perform `-extres` functionality, writing static output to a Java resource file (see the `-extres` description above).
2. Create a `main()` method and a hotloading method in the generated page implementation class to allow hotloading.

For an overview of hotloading, see "Overview of Hotloaded Classes in Oracle8i" on page 6-20. For how to accomplish the hotloading step (once hotloading has been enabled), see "Hotloading Page Implementation Classes in Oracle8i" on page 6-62.

---

**Note:** To write static content to a resource file without enabling hotloading (if the page will not be running in OSE, for example), use the `-extres` option.

---

**-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` *not* to compile the generated page implementation class source. This allows you to compile it later with an alternative Java compiler.

---

**Notes:**

- In an on-demand translation scenario, the OracleJSP `javaccmd` configuration parameter provides related functionality, allowing you to specify an alternative Java compiler directly. See "OracleJSP Configuration Parameters (Non-OSE)" on page A-15.
- For a SQLJ JSP page, enabling `-noCompile` does not prevent SQLJ translation, just Java compilation.

---

**-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, OracleJSP release 8.1.7 (1.1.0.0.0) would use `src.jspsrc` as the package name, by default. (Be aware that such implementation details are subject to change in future releases.)

**-S-<sqlj option> <value>** (`-S` followed by SQLJ option setting; `ojspc` default: empty)

For SQLJ JSP pages, use the `ojspc -S` option to pass Oracle SQLJ options to the SQLJ translator.

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 mypkg.MyCust -d /myapproot/mybindir MyPage.jsp
```

This invokes the Oracle SQLJ `-default-customizer` option to choose an alternative profile customizer, as well as setting the `ojspc -d` option.

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 *Oracle8i SQLJ Developer's Guide and Reference*.

**-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).

In environments such as Windows NT that allow spaces in directory names, enclose the directory name in quotes.

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 6-34 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 easily know what files have been produced.

---

**Notes:** In an on-demand translation scenario, the OracleJSP `page_repository_root` configuration parameter provides related functionality. See "OracleJSP Configuration Parameters (Non-OSE)" on page A-15.

---

**-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 OracleJSP version number and then exit.

### Summary of ojspc Output Files, Locations, and Related Options

By default, `ojspc` generates the same set of files that are generated by the OracleJSP 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
- a Java resource file or, optionally, a `.class` file for the SQLJ profile (SQLJ JSP pages only)
- optionally, a Java resource file for the static text of the page

For more information about files that are generated by the OracleJSP translator, see "Generated Files and Locations (On-Demand Translation)" on page 6-6.

To summarize some of the commonly used options described in "Option Descriptions for ojspc" on page 6-27, 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 alternative location
- `-d` to place binary files (`.class` files and Java resource files) in a specified alternative location
- `-noCompile` to *not* compile the generated page implementation class source (as a result of this, no `.class` files are produced)
- `-extres` to put static text into a Java resource file
- `-hotload` to put static text into a Java resource file and to enable hotloading (relevant only for pages targeting the Oracle Servlet Engine)
- `-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 determined by 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, presume 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`.

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 `ojspc` is run 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 `/home/mydir/def/ghi`, or in a `def/ghi` subdirectory under locations specified through the `-d` and `-srcdir` options.

> **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.

## Overview of the loadjava Tool

The `loadjava` command-line tool is supplied with Oracle8*i* to create schema objects from Java files and load them into a specified database schema.

For information beyond what is provided here, and for information about the associated `dropjava` tool (for removing Java source, class, and resource schema objects from the database), see the *Oracle8i Java Tools Reference*.

Generally speaking (not for JSP applications in particular), a Java developer can compile Java source on the client and then load the resulting class files, or can load Java source and have it compiled in Oracle8*i* automatically by the server-side compiler. In the first case, only class schema objects are created. In the second case, both source schema objects and class schema objects are created. In either case, the developer can also load Java resource files, creating resource schema objects.

The `loadjava` tool accepts source files, class files, resource files, JAR files, and ZIP files on the command line. Source files and class files cannot be loaded simultaneously, however. A JAR file, ZIP file, or `loadjava` command line can contain source files or class files, but not both. (In either case, resource files can be included.)

A JAR or ZIP file is opened and processed, with each file within the JAR or ZIP file resulting in one or more schema objects.

For OracleJSP, use `loadjava` as follows:

- For client-side translation, you will have already translated your JSP pages using `ojspc`, which, by default, also compiles the translated Java source. Then use `loadjava` to load the resulting `.class` files and any resource files (the `ojspc -hotload` option, for example, produces a resource file), typically all bundled into a JAR file.

  Alternatively, you can load the translated `.java` file instead of the compiled `.class` files. You can have the server-side compiler compile the `.java` file as it is being loaded.

- For server-side translation, use `loadjava` to load untranslated `.jsp` files, typically bundled into a JAR file, as resource schema objects. (They will be translated and published later, in the server, as a result of the session shell `publishjsp` command.)

Following is the complete `loadjava` option syntax. Brackets, `{...}`, are not part of the syntax. They are used to surround two possible option formats that are followed by option input.

```
loadjava {-user | -u} user/password[@database] [options]
file.java | file.class | file.jar | file.zip | file.sqlj | resourcefile
  [-debug]
  [-d | -definer]
  [{-e | -encoding} encoding_scheme]
  [-f | -force]
  [{-g | -grant} user [, user]...]
  [-o | -oci8]
  [ -order ]
  [-noverify]
  [-r | -resolve]
  [{-R | -resolver} "resolver_spec"]
  [{-S | -schema} schema]
  [ -stdout ]
  [-s | -synonym]
  [-t | -thin]
  [-v | -verbose]
```

Of particular significance are the `-user` and `-resolve` options (which can be abbreviated to `-u` and `-r`, respectively). Use the `-user` option to specify the schema name and password. Use the `-resolve` option to specify whether `loadjava` is to compile (if applicable) and resolve external references in the classes you are loading, after all classes on the command line have been loaded.

If you are loading a `.java` source file that you want compiled by the server-side compiler during loading, you must enable the `-resolve` option.

Following is an example for a client-side translation scenario where the JSP page has already been translated and compiled, producing file `HelloWorld.class` and another `.class` file for the page implementation inner class (with a name that starts with "HelloWorld"). Assume `%` is a UNIX prompt:

```
% loadjava -u scott/tiger -r HelloWorld*.class
```

Or you can bundle the files into a JAR file:

```
% loadjava -v -u scott/tiger -r HelloWorld.jar
```

The `loadjava -v` (`-verbose`) option, which provides detailed status reporting as loading progresses, is especially useful when you are loading a number of files or compiling in the server.

The following example is also for a client-side translation scenario (`HelloWorld.java` is the JSP translator output), but where you have elected to skip the compilation step on the client (using the `ojspc -noCompile` option) and instead have the server-side compiler handle the compilation:

```
% loadjava -v -u scott/tiger -r HelloWorld.java
```

The following example is for a server-side translation scenario:

```
% loadjava -u scott/tiger -r HelloWorld.jsp
```

## Overview of the sess_sh Session Shell Tool

The `sess_sh` (session shell) tool is provided with Oracle8*i* as an interactive interface to the session namespace of a database instance. You specify database connection arguments when you start `sess_sh`. It then presents you with its `$` prompt to indicate that it is ready for commands.

The session shell tool has many top-level commands you can run from the `$` prompt, each of which may have its own set of options. For OracleJSP developers, the `publishservlet` and `unpublishservlet` commands (for deployment with client-side translation), `publishjsp` and `unpublishjsp` commands (for deployment with server-side translation), and `createcontext` command (for creating OSE servlet contexts) are of primary interest.

Following are the key `sess_sh` syntax elements for starting the tool:

sess_sh -user *user* -password *password* -service *serviceURL*

- `-user` specifies the user name of the schema.
- `-password` specifies the password for the specified user name.
- `-service` specifies the URL of the database whose session namespace is to be "opened" by `sess_sh`. The `serviceURL` parameter should have one of the three following forms:

  sess_iiop://*host:port:sid*
  jdbc:oracle:*type:spec*
  http://*host*[:*port*]

Following are some general examples:

```
sess_iiop://localhost:2481:orcl
jdbc:oracle:thin:@myhost:1521:orcl
http://localhost:8000
```

Here is an example of a `sess_sh` command line:

```
% sess_sh -user SCOTT -password TIGER -service jdbc:oracle:thin:@myhost:5521:orcl
```

After starting `sess_sh`, you will see its command prompt:

```
$
```

In addition to *publish object commands*, such as `publishservlet` and `publishjsp`, the session shell tool offers *shell commands* that give the session namespace much of the "look and feel" of a UNIX file system as seen from one of the UNIX shells (such as the C shell). For example, the following `sess_sh` command displays the published objects and publishing contexts in the `/alpha/beta/gamma` publishing context (publishing contexts are nodes in the session namespace, analogous to directories in a file system):

```
$ ls /alpha/beta/gamma
```

As mentioned previously, key `sess_sh` commands for OracleJSP developers include the following:

```
$ publishjsp ...
$ unpublishjsp ...
$ publishservlet ...
$ unpublishservlet ...
$ createcontext ...
```

For information about the `publishservlet` and `unpublishservlet` commands, see "Publishing Translated JSP Pages in Oracle8i (Session Shell publishservlet)" on page 6-63. For information about the `publishjsp` and `unpublishjsp` commands, see "Translating and Publishing JSP Pages in Oracle8i (Session Shell publishjsp)" on page 6-42.

Each session shell command has a `-describe` option to describe its operation, a `-help` option to summarize its syntax, and a `-version` option to show its version number.

---

**Note:** This document provides only abbreviated discussion of `sess_sh` syntax and options. It presents only the simplest invocation and usage of the tool.

Beyond what is presented here, for example, commands can be specified within quotes on the `sess_sh` command line instead of at the `$` prompt.

There are also top-level options to connect with plain IIOP instead of the default session IIOP, to specify a role, to connect to the database with SSL server authentication, and to use a service name instead of an SID in the URL.

For complete information about the `sess_sh` tool, see the *Oracle8i Java Tools Reference*.

---

# Deployment to Oracle8*i* with Server-Side Translation

This section describes the steps for deployment to Oracle8*i* with server-side translation.

The steps are as follows:

1. Use `loadjava` to load untranslated JSP page or SQLJ JSP page source files into Oracle8*i*.
2. Use the session shell `publishjsp` command to translate and publish the pages.

The `publishjsp` step automatically handles translation, compilation, hotloading (if applicable), and publishing.

## Loading Untranslated JSP Pages into Oracle8*i* (loadjava)

As the first step for deployment with server-side translation, use the Oracle `loadjava` tool to load untranslated `.jsp` or `.sqljsp` files into Oracle8*i* as Java resource files.

If you are loading multiple files, it is recommended that you put the files into a JAR file for loading.

The `loadjava` tool is provided with Oracle8*i* as a general-purpose tool for loading Java files into the server. For an overview, see "Overview of the loadjava Tool" on page 6-36. For further information, see the *Oracle8i Java Tools Reference*.

Following is an example of loading an untranslated page:

```
% loadjava -u scott/tiger Foo.jsp
```

This loads `Foo.jsp` into the `SCOTT` schema (password `TIGER`) as a Java resource object. There is no need to specify the `loadjava -resolve` (`-r`) option.

This will result in the following resource schema object being created in the database:

- `SCOTT:Foo.jsp`

Note that any path information you specify for the `.jsp` file, either in a JAR file or on the `loadjava` command line, determines placement of the resource schema object. Consider the following modification of the previous example:

```
% loadjava -u scott/tiger xxx/yyy/Foo.jsp
```

This will result in the following resource schema object being created in the database:

- `SCOTT:xxx/yyy/Foo.jsp`

For an overview of how `loadjava` names the schema objects it produces, see "Database Schema Objects for Java" on page 6-12.

You can also load a `.sqljsp` file:

```
% loadjava -u scott/tiger Foo.sqljsp
```

This loads `Foo.sqljsp` into the `SCOTT` schema and will result in the following resource schema object being created in the database:

- `SCOTT:Foo.sqljsp`

If you want to load multiple `.jsp` (or `.sqljsp`) files, you can use a wildcard character (depending on your operating environment; assume `%` is a UNIX prompt):

```
% loadjava -u scott/tiger *.jsp
```

Or presume you had put the `.jsp` files into a JAR file:

```
% loadjava -u scott/tiger myjspapp.jar
```

## Translating and Publishing JSP Pages in Oracle8*i* (Session Shell publishjsp)

Translation, compilation, hotloading (if enabled), and publishing all occur automatically in the scenario of deployment with server-side translation, as the result of executing the JServer session shell `publishjsp` command. See "Overview of the sess_sh Session Shell Tool" on page 6-38 for how to start the session shell and connect to the database.

Run `publishjsp` after you have loaded a `.jsp` (or `.sqljsp`) file into Oracle8*i* as a resource schema object. (This section includes separate discussion for running `publishjsp` on a `.sqljsp` file because there are some logistical differences in the results.)

---

**Note:** JSP pages that are published with `publishjsp` can be "unpublished" (removed from the JServer JNDI namespace) with the session shell `unpublishjsp` command. See "Unpublishing JSP Pages with unpublishjsp" on page 6-53.

---

### Overview of publishjsp Syntax and Options

Starting `sess_sh` establishes a connection to the database. Once you start `sess_sh`, you can run the `publishjsp` command from the session shell `$` prompt.

The `publishjsp` command uses the following general syntax:

```
$ publishjsp [options] path/name.jsp
```

The options can be any of the following:

```
[-schema schemaname] [-virtualpath path] [-servletName name] [-packageName name]
[-context context] [-hotload] [-stateless] [-verbose] [-resolver resolver]
[-extend class] [-implement interface]
```

---

**Important:**

- Enable boolean options, such as `-hotload`, by typing only the option name in the command line (not setting it to `true`).
- For options where you specify a value, the value does *not* have to be in quotes.

---

The file *`name`*`.jsp` (or *`name`*`.sqljsp` for a SQLJ JSP page) is the JSP page resource schema object that you loaded with `loadjava` and is the only required parameter, along with any relevant schema *`path`* information.

By default, if no `-virtualpath` option is specified, *`path/name`*`.jsp` becomes the servlet path. For example, running `publishjsp` on `dir1/foo.jsp` (the path within the current schema or specified schema) results in `dir1/foo.jsp` as the servlet path.

By default, if no `-context` option is specified, the OSE default servlet context is used and "`/`" is the context path.

Together, the context path and servlet path (along with the host name and port) determine the URL to invoke the page, as described in "URLs for the Oracle Servlet Engine" on page 6-15.

The following informative options are also available:

- Use `-showVersion` by itself to display the OracleJSP version number and exit.
- Use `-usage` by itself to display a `publishjsp` option list and exit.

Following are the option descriptions:

- `-schema` *`schemaname`*

  Use this option to specify the schema where the JSP page resource schema object is located, if it is not in the same schema you logged in to through `sess_sh`.

  This schema must be accessible from your `sess_sh` login schema. The `publishjsp` command does not offer a way to specify passwords.

- `-virtualpath` *`path`*

  You can use this option to specify an alternative servlet path for the JSP page; otherwise, the servlet path is simply the specified `.jsp` file name itself along with any specified schema path.

  For example:

  ```
  -virtualpath altpath/Foo.jsp
  ```

  or perhaps simply:

  ```
  -virtualpath mypath.jsp
  ```

- `-servletName` *`name`*

  You can use this option to specify an alternative servlet name (in OSE `named_servlets`) for the JSP page; however, the servlet name has no bearing on how the page is invoked so is rarely needed.

  By default, the servlet name is the base name of the `.jsp` file along with any path you specified. For example, running `publishjsp` on `SCOTT:dir1/Foo.jsp` results in `dir1.Foo` as the servlet name in OracleJSP release 8.1.7 (1.1.0.0.0) (Be aware that implementation details are subject to change in future releases.)

- `-packageName` *`name`*

  You can use this option to specify a package name for the generated page implementation class; otherwise, it is determined by any path specification for the `.jsp` file when you run `publishjsp`. For example, running `publishjsp` on `SCOTT:dir1/Foo.jsp` results in package `dir1` for the page implementation class.

The `-packageName` option affects where schema objects are placed in the schema, but does not affect the servlet path of the JSP page.

- `-context` *`context`*

  You can use this option to specify a servlet context in the Oracle Servlet Engine. The context path of this servlet context becomes part of the URL used to invoke the page.

  If you do not use this option, the JSP page will be in the OSE default context, `/webdomains/contexts/default`, whose context path is simply "/".

  Any specified context should be under `/webdomains/contexts`, such as:

  /webdomains/contexts/mycontext

  > **Important:** Remember that it is the *context path* of the servlet context, not the context name itself, that is used in URLs to access the page.
  >
  > When a servlet context is created in OSE with the session shell `createcontext` command, both the context path (through the `createcontext -virtualpath` option) and the context name must be specified. It is convenient, and probably typical, to specify the context name and context path to be the same, but it is not required.

- `-hotload`

  Enable this flag to enable and perform hotloading. This results in the following steps being performed by the `publishjsp` command:

  1. Static output is written to a resource schema object instead of to the page implementation class schema object.
  2. A `main()` method and a hotloading method are implemented in the generated page implementation class to allow hotloading.
  3. The `main()` method is executed to perform hotloading.

To use `-hotload`, you must have permission for the JServer hotloader. This can be granted as follows (from SQL*Plus, for the `SCOTT` schema, for example):

```
dbms_java.grant_permission('SCOTT', 'SYS:oracle.aurora.security.JServerPermission', 'HotLoader', null);
```

For an overview of hotloading, see "Overview of Hotloaded Classes in Oracle8i" on page 6-20.

- `-stateless`

  This is a boolean option that tells the Oracle Servlet Engine that the JSP page is to be stateless—the JSP page should not have access to the `HttpSession` object during execution.

  This flag is used for `mod_ose` optimization. For information about the Apache `mod_ose` module, see the *Oracle8i Oracle Servlet Engine User's Guide*.

- `-verbose`

  Set this option to `true` to direct `publishjsp` to report the translation steps as it executes.

- `-resolver`

  Use this option to specify an alternative Java class resolver. The resolver is used in compiling and resolving Java source through `loadjava`, including locating classes used in JSP pages.

  The default resolver is `((* `*`user`*`) (* PUBLIC))`. For the `SCOTT` schema, for example, this is the following:

  ```
  ((* SCOTT) (* PUBLIC))
  ```

  For the `-resolver` option, you must specify the value in quotes as in the following example:

  ```
  $ publishjsp ... -resolver "((* BILL) (* SCOTT) (* PUBLIC))" ...
  ```

- `-extend`

  Use this option to specify a Java class that the generated page implementation class will extend.

- `-implement`

  Use this option to specify a Java interface that the generated page implementation class will implement.

### Examples: Publishing JSP Pages with publishjsp

This section provides examples of using `publishjsp` to translate and publish `.jsp` pages in Oracle8*i*. The pages will have already been loaded as resource schema objects in a particular schema, such as `SCOTT:Foo.jsp`.

(For information about running `publishjsp` on `.sqljsp` pages, see "Publishing SQLJ JSP Pages with publishjsp" on page 6-51.)

To review how the servlet path and context path combine in forming the URL to invoke the page, see "URLs for the Oracle Servlet Engine" on page 6-15.

**Notes:**

- The following examples use the `SCOTT` schema. `SCOTT` must either be the schema specified when starting `sess_sh`, or accessible from the schema specified.
- Each example lists the schema objects that are created, although this is secondary. All that matters in invoking the JSP page is the servlet path and context path. The page implementation class schema object is automatically mapped during the `publishjsp` publishing step.
- Application-relative and page-relative syntax for dynamic `jsp:include` and `jsp:forward` statements inside Oracle8*i* is the same as for any JSP environment. The relative paths are according to how the JSP pages were published (as shown in the examples below).
- The exact names of generated schema objects 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, such as `_Foo` instead of `Foo`.
- `$` is the `sess_sh` prompt.

#### Example 1

```
$ publishjsp -schema SCOTT dir1/Foo.jsp
```

This uses the default servlet context, which has a context path of "/".

The default servlet path is `dir1/Foo.jsp`.

After this command, `Foo.jsp` can be invoked as follows:

```
http://host[:port]/dir1/Foo.jsp
```

Access it dynamically from another JSP page in the application, suppose a page published as `dir1/Bar.jsp`, as follows (using page-relative syntax and then application-relative syntax):

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

or:

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

By default, `dir1` is the Java package for the page implementation class and inner class (because of the specified path in the `SCOTT` schema).

The following schema objects are created:

- `SCOTT:dir1/Foo` source schema object
- `SCOTT:dir1/Foo` class schema object
- a class schema object under `dir1` for the inner class for static text (with "Foo" in the name, such as `SCOTT:dir1/Foo$__jsp__StaticText`)

**Example 2**

```
$ publishjsp -schema SCOTT -context /webdomains/contexts/mycontext Foo.jsp
```

Presume `mycontext` had been created as follows:

```
$ createcontext -virtualpath mycontext /webdomains mycontext
```

The `publishjsp` command publishes the page to the `mycontext` servlet context, which was created with `mycontext` also specified as the context path.

The default servlet path is simply `Foo.jsp`.

After this command, `Foo.jsp` can be invoked as follows:

```
http://host[:port]/mycontext/Foo.jsp
```

Access it dynamically from another JSP page in the application, suppose a page published as `Bar.jsp`, as follows (using page-relative syntax and then application-relative syntax):

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

or:

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

Even though this example specifies a non-default servlet context, that is not relevant for dynamic `jsp:include` or `jsp:forward` commands. What is relevant is that the published path of the page relative to that context is simply `/Foo.jsp`.

By default, there is no Java package for the page implementation class and inner class (because no path is specified in the `SCOTT` schema).

The following schema objects are created:

- `SCOTT:Foo` source schema object
- `SCOTT:Foo` class schema object
- a class schema object for the inner class for static text (with "Foo" in the name, such as `SCOTT:Foo$__jsp__StaticText`)

**Example 3**

```
$ publishjsp -schema SCOTT -context /webdomains/contexts/mycontext dir1/Foo.jsp
```

Presume `mycontext` had been created as follows:

```
$ createcontext -virtualpath mywebapp /webdomains mycontext
```

The `publishjsp` command publishes the page to the `mycontext` servlet context, which was created with `mywebapp` specified as the context path.

The default servlet path is `dir1/Foo.jsp`.

After this command, `Foo.jsp` can be invoked as follows:

http://*host*[:*port*]/mywebapp/dir1/Foo.jsp

Access it dynamically from another JSP page in the application, suppose a page published as `dir1/Bar.jsp`, as follows (using page-relative syntax and then application-relative syntax):

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

or:

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

Example 1 and Example 3 use different servlet contexts, but in either case what is relevant for the application-relative `include` command is that the published path of the page relative to that context is `/dir1/Foo.jsp`.

By default, `dir1` is the Java package for the page implementation class and inner class.

The following schema objects are created:

- `SCOTT:dir1/Foo` source schema object
- `SCOTT:dir1/Foo` class schema object
- a class schema object for the inner class for static text (with "Foo" in the name, such as `SCOTT:dir1/Foo$__jsp__StaticText`)

#### Example 4

```
$ publishjsp -schema SCOTT -hotload -packageName mypkg dir1/Foo.jsp
```

This performs hotloading, uses the default servlet context, and overrides the default `dir1` package.

The context path is "/".

The `-packageName` option does not affect the servlet path, which, by default, remains `dir1/Foo.jsp`.

After this command, `Foo.jsp` can be invoked as follows:

http://*host*[:*port*]/dir1/Foo.jsp

Access it dynamically from another JSP page in the application, suppose a page published as `dir1/Bar.jsp`, as follows (using page-relative syntax and then application-relative syntax):

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

or:

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

The following schema objects are created:

- `SCOTT:mypkg/Foo` source schema object
- `SCOTT:mypkg/Foo` class schema object

- a class schema object under mypkg for the inner class (with "Foo" in the name, such as SCOTT:mypkg/Foo$__jsp__StaticText)
- SCOTT:mypkg/Foo.res resource schema object for the static text that is normally in the inner class (the resource is hotloaded as part of publishjsp functionality)

### Publishing SQLJ JSP Pages with publishjsp

This section provides an example of using publishjsp to translate and publish a .sqljsp page in Oracle8*i*. The page will have already been loaded as a resource schema object in a particular schema, such as SCOTT:Foo.sqljsp.

In addition, see "Examples: Publishing JSP Pages with publishjsp" on page 6-47.

To review how the servlet path and context path combine in forming the URL to invoke the page, see "URLs for the Oracle Servlet Engine" on page 6-15.

Be aware of the following for .sqljsp pages:

- Beyond what is created for a .jsp page, an additional schema objects is created—a resource schema object for the SQLJ profile. This is always a .ser resource schema object, as opposed to a class schema object, because there is no SQLJ -ser2class option when translating in the server.

  For information about SQLJ profiles, see the *Oracle8i SQLJ Developer's Guide and Reference*.
- The generated source schema object is SQLJ source instead of Java source.
- SQLJ has very limited option support in the server.

**Server-Side SQLJ Options** Client-side SQLJ options are not available for translation in the server (this is true in general, not just for JSP pages). Instead, there is a small set of options available through the standard Oracle8*i* JAVA$OPTIONS table as set by the dbms_java.set_compiler_option() stored procedure (using SQL*Plus, for example). Of these options, only the following is supported for JSP pages:

- online

This is a boolean option that enables online semantics-checking through the default oracle.sqlj.checker.OracleChecker front-end.

For more information about server-side SQLJ and semantics-checking, see the *Oracle8i SQLJ Developer's Guide and Reference*,

**Example of publishjsp for SQLJ JSP Page** Following is an example of `publishjsp` usage for a `.sqljsp` page (`$` is the `sess_sh` prompt).

**Notes:**

- This example uses the `SCOTT` schema. `SCOTT` must either be the schema specified when starting `sess_sh`, or accessible from the schema specified.
- This example documents the schema objects that are created, although this is secondary. All that matters in invoking the JSP page is the servlet path and context path. The page implementation class schema object is automatically mapped during the `publishjsp` publishing step.
- The exact names of generated schema objects 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, such as `_Foo` instead of `Foo`.

```
$ publishjsp -schema SCOTT dir1/Foo.sqljsp
```

This uses the default OSE servlet context, the context path of which is "/".

The servlet path, by default, is `dir1/Foo.sqljsp`.

After this command, `Foo.sqljsp` can be invoked as follows:

`http://`*`host`*`[:`*`port`*`]/dir1/Foo.sqljsp`

Access it dynamically from another JSP page in the application, suppose a page published as `dir1/Bar.jsp`, as follows (using page-relative syntax and then application-relative syntax):

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

or:

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

By default, `dir1` is the Java package for the page implementation class and inner class (because of the specified path in the `SCOTT` schema).

The following schema objects are created:

- `SCOTT:dir1/Foo` source schema object
- `SCOTT:dir1/Foo` class schema object
- class schema object under `dir1` for the inner class for static text (with "Foo" in the name, such as `SCOTT:dir1/Foo$__jsp__StaticText`)
- resource schema object under `dir1` for the SQLJ profile (with "Foo" in the name, such as `SCOTT:dir1/Foo_SJProfile0.ser`)

### Unpublishing JSP Pages with unpublishjsp

The `sess_sh` tool also has an `unpublishjsp` command that removes a JSP page from the JServer JNDI namespace. This does *not*, however, remove the page implementation class schema object from the database.

Unlike the `unpublishservlet` command, you do not need to specify a servlet name (unless you specified one when you ran `publishjsp`). Generally, the only required input is the servlet path (sometimes referred to as the "virtual path").

Following is the general syntax:

```
$ unpublishjsp [-servletName name] [-context context] [-showVersion] [-usage] [-verbose] servletpath
```

The `-servletName`, `-context`, `-showVersion`, `-usage`, and `-verbose` options are the same as for `publishjsp`, as described in "Overview of publishjsp Syntax and Options" on page 6-43.

In using `unpublishjsp`, specify the values for `-servletName` and `-context` that you specified in using `publishjsp`.

As an example, here is the command to unpublish the page that was published in Example 4 on page 6-50:

```
$ unpublishjsp dir1/Foo.jsp
```

(Remember that the `-packageName` option specified in Example 4 has no effect on the servlet path.)

# Deployment to Oracle8*i* with Client-Side Translation

This section describes the steps for deployment to Oracle8*i* with client-side translation.

The steps are as follows:

1. Use `ojspc` to pre-translate JSP pages or SQLJ JSP pages on the client.
2. Use `loadjava` to load files into Oracle8*i*—`.class` files (or, optionally, `.java` or `.sqlj` files instead) and any Java resource files resulting from the page translation.
3. (Optional) "Hotload" the pages into Oracle8*i* (if hotloading was enabled during translation). See "Overview of Hotloaded Classes in Oracle8i" on page 6-20 for background information about hotloading.
4. Use the session shell `publishservlet` command to publish the pages.

---

**Note:** For simplicity and convenience, deployment with server-side translation is generally recommended. See "Deployment to Oracle8i with Server-Side Translation" on page 6-41.

---

## Pre-Translating JSP Pages (ojspc)

To pre-translate JSP pages on a client (typically for pages that will run in the Oracle Servlet Engine), use the `ojspc` command-line tool to invoke the OracleJSP translator.

For general information about `ojspc` and description of its options, see "The ojspc Pre-Translation Tool" on page 6-23.

The rest of this section covers the following topics:

- Simplest ojspc Usage
- ojspc for SQLJ JSP Pages
- Enabling Hotloading with ojspc
- Other Key ojspc Features and Options for Deployment to Oracle8i
- ojspc Examples

---

**Note:** The exact names of generated files 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, such as `_Foo.java` or `_Foo.class`.

---

### Simplest ojspc Usage

The following example shows the simplest usage of `ojspc`:

```
% ojspc Foo.jsp
```

With this invocation, the following files are produced:

- `Foo.java`
- `Foo.class`
- a `.class` file for the inner class for static content (with "Foo" in the name)

By default, all output goes to the current directory, from which `ojspc` was run.

### ojspc for SQLJ JSP Pages

The `ojspc` tool also accepts `.sqljsp` files for JSP pages that use SQLJ code:

```
% ojspc Foo.sqljsp
```

For `.sqljsp` files, `ojspc` automatically invokes the SQLJ translator as well as the JSP translator.

With this invocation, the following files are produced:

- `Foo.sqlj` (produced from `Foo.sqljsp` by the JSP translator)
- `Foo.java` (produced from `Foo.sqlj` by the SQLJ translator)
- `Foo.class`
- a `.class` file for the inner class for static content (with "Foo" in the name)
- a Java resource file (`.ser`) or class file (`.class`), depending on the setting of the SQLJ `-ser2class` option, for the SQLJ "profile" (with "Foo" in the name)

  For information about SQLJ profiles, see the *Oracle8i SQLJ Developer's Guide and Reference*.

By default, all output goes to the current directory, from which `ojspc` was run.

### Enabling Hotloading with ojspc

Use the `ojspc -hotload` option to enable hotloading, which (among other things) results in static page content going into a Java resource file instead of into the inner class of the page implementation class.

The following example translates the page and directs the OracleJSP translator to enable hotloading:

```
% ojspc -hotload Foo.jsp
```

With this command, the translator will generate the following output:

- `Foo.java` (as usual)
- `Foo.class` (as usual)
- `Foo.res`, a Java resource file to contain the static page content
- a `.class` file for the inner class (as usual, with "Foo" in the name, although the static content goes into `Foo.res` instead of going into the inner class)

Be aware that the `ojspc -hotload` option merely enables hotloading; it does not actually hotload the page. Hotloading requires an additional deployment step, as described in "Hotloading Page Implementation Classes in Oracle8i" on page 6-62.

For an overview of hotloading, see "Overview of Hotloaded Classes in Oracle8i" on page 6-20.

### Other Key ojspc Features and Options for Deployment to Oracle8*i*

The following `ojspc` options, fully described in "Option Descriptions for ojspc" on page 6-27, are especially useful:

- `-appRoot`—Set an application root directory if you do not want the default (the current directory, from which `ojspc` was run).
- `-noCompile`—Enable this flag if you do not want to compile during translation. You would do this, for example, if you want to load the translated page into Oracle8*i* as a `.java` file and have compilation performed by the server-side compiler.
- `-d`—Specify the directory where `ojspc` will place the generated binary files (`.class` files and Java resource files). This makes it easier to know what was generated, and therefore what needs to be loaded into Oracle8*i*.
- `-srcdir`—Specify the directory where `ojspc` will place the generated source files. For example, this would be useful instead of `-d` if you are enabling `-noCompile` and will load your translated page into Oracle8*i* as `.java` source.

- `-extres`—Direct the OracleJSP translator to put static content into a Java resource file instead of into the inner class of the page implementation class.
- `-hotload`—Direct the OracleJSP translator to put static content into a Java resource file instead of into the inner class of the page implementation class, and generate code in the page implementation class to enable hotloading.
- `-S`—For SQLJ JSP pages, use the `-S` prefix to specify Oracle SQLJ options; `ojspc` will pass these option settings to the Oracle SQLJ translator.

### ojspc Examples

The following examples show the use of key `ojspc` options.

```
% ojspc -appRoot /myroot/pagesrc -d /myroot/bin -hotload /myroot/pagesrc/Foo.jsp
```

The preceding example accomplishes the following:

- Specifies an application root for application-relative static `include` directives in the translated page.
- Enables hotloading and produces the Java resource file `Foo.res` for static content.
- Places `Foo.java` into the current directory, by default. There is no package, because `Foo.jsp` is in the specified application root directory.
- Places `Foo.class`, `Foo.res`, and the `.class` file for the inner class into the `/myroot/bin` directory.

```
% ojspc -appRoot /myroot/pagesrc -srcdir /myroot/gensrc -noCompile -extres /myroot/pagesrc/Foo.jsp
```

The preceding example accomplishes the following:

- Specifies an application root for application-relative static `include` directives in the translated page.
- Produces the Java resource file `Foo.res` for static content (without enabling hotloading).
- Places `Foo.java` into the `/myroot/gensrc` directory. There is no package, because `Foo.jsp` is in the specified application root directory.
- Does *not* compile `Foo.java` (no `.class` files are produced).
- Places `Foo.res` into the current directory, by default.

```
% ojspc -appRoot /myroot/pagesrc -d /myroot/bin -extres -S-ser2class true /myroot/pagesrc/Foo.sqljsp
```

The preceding example accomplishes the following:

- Specifies an application root for application-relative static `include` directives in the translated page.
- Produces the Java resource file `Foo.res` for static content (without enabling hotloading).
- Places `Foo.sqlj` and `Foo.java` into the current directory, by default. There is no package, because `Foo.jsp` is in the specified application root directory.
- Places `Foo.class`, `Foo.res`, a `.class` file for the inner class, and a `.class` file for the SQLJ profile into the `/myroot/bin` directory. (Without the SQLJ `-ser2class` option setting, the profile would be generated in a `.ser` Java resource file instead of a `.class` file.)

## Loading Translated JSP Pages into Oracle8*i* (loadjava)

After client-side pre-translation, use the Oracle `loadjava` tool to load generated files into Oracle8*i*. You can use either of the following scenarios:

- Load `.class` files and Java resource files (if any).
- Use the `ojspc -noCompile` option during translation, then load the translated `.java` file and resource files (if any). The `.java` file can be compiled by the Oracle8*i* server-side compiler during loading.

In either case, whenever you have multiple files it is recommended that you put the files into a JAR file for loading.

The `loadjava` tool is provided with Oracle8*i* as a general-purpose tool for loading Java files into the database. For an overview, see "Overview of the loadjava Tool" on page 6-36. For further information, see the *Oracle8i Java Tools Reference.*

---

**Important:** In the next two subsections ("Loading Class Files with loadjava" and "Loading Source Files with loadjava"), be aware of the following important considerations.

- Even when you enable the `-extres` or `-hotload` option to place static text into a resource file, the page implementation inner class is still produced and must still be loaded.
- Like a Java compiler, `loadjava` resolves references to classes, but not to resources; be sure to correctly load the resource files your classes need—they must be in the same package as the `.java` file.

---

### Loading Class Files with loadjava

Assume you translated a JSP page `Foo.jsp` with the `ojspc -extres` or `-hotload` option enabled, producing the following files:

- `Foo.java`
- `Foo.class`
- `Foo$__jsp__StaticText.class`
- `Foo.res`

---

**Note:** Generated names used here are provided as examples only. Such implementation details are subject to change in future releases, although the base name (such as "Foo" here) will always be part of the generated names.

---

You can ignore `Foo.java`, but the binary files (`.class` and `.res`) must all be loaded into Oracle8*i*. Typically, you would put `Foo.class`, `Foo$__jsp__StaticText.class`, and `Foo.res` into a JAR file, suppose `Foo.jar`, and load it as follows (assume `%` is a UNIX prompt):

```
% loadjava -v -u scott/tiger -r Foo.jar
```

The `-u` (`-user`) option specifies the user name and password for the database schema; the `-r` (`-resolve`) option resolves the classes as they are loaded. Optionally use the `-v` (`-verbose`) option for detailed status output.

Alternatively, you can load the files individually, as follows. (The syntax depends on your operating environment. In these examples, assume % is a UNIX prompt.)

```
% loadjava -v -u scott/tiger -r Foo*.class Foo.res
```

or:

```
% loadjava -v -u scott/tiger -r Foo*.*
```

All these examples result in the following schema objects being created in the database (you typically need to know only the name of the page implementation class schema object):

- `SCOTT:Foo` page implementation class schema object

  Or there may be an additional package designation, according either to the `ojspc -packageName` option or the relative location of the `.jsp` file to the current directory when you ran `ojspc`. For example, a `-packageName` setting of `"abc.def"` results in that being the package of the `Foo` class, so there would be a `SCOTT:abc/def/Foo` class schema object.

- `SCOTT:Foo$__jsp__StaticText` class schema object

  With the same package designation as the page implementation class.

- `SCOTT:Foo.res` resource schema object

  With a package designation according to any path specification, either in a JAR file or on the `loadjava` command line, when it was loaded.

For an overview of how `loadjava` names the schema objects it produces, see "Database Schema Objects for Java" on page 6-12.

---

**Note:** If you are loading a pre-translated SQLJ JSP page, you must also load the generated profile file—either a `.ser` Java resource file or a `.class` file, depending on the SQLJ `-ser2class` option. If it is a `.ser` file, schema object naming is comparable to that of a `.res` Java resource file; if it is a `.class` file, schema object naming is comparable to that of the other `.class` files.

---

### Loading Source Files with loadjava

Assume that you translated a JSP page, `Foo.jsp`, with the `ojspc -noCompile` and `-extres` options enabled, producing the following files:

- `Foo.java` (which you want to load into Oracle8*i* as source to be compiled by the server-side compiler)
- `Foo.res`

Typically, you would put `Foo.java` and `Foo.res` into a JAR file, suppose `Foo.jar`, and load it as follows:

```
% loadjava -v -u scott/tiger -r Foo.jar
```

When you enable the `loadjava -r` (`-resolve`) option, this results in the source file being compiled automatically by the server-side compiler, producing class schema objects. The `-u` (`-user`) option specifies the user name and password for the database schema. Optionally use the `-v` (`-verbose`) option for detailed status reporting.

Alternatively, you can load the files individually:

```
% loadjava -v -u scott/tiger -r Foo.java Foo.res
```

Or load them using a wildcard character:

```
% loadjava -v -u scott/tiger -r Foo.*
```

All these examples result in the following schema objects being created in the database (you typically need to know only the name of the page implementation class schema object):

- `SCOTT:Foo` source schema object

  When you load a source file into Oracle8*i* with `loadjava`, the source is stored separately as a source schema object, in addition to the class schema objects produced by the server-side compiler.

- `SCOTT:Foo` page implementation class schema object

  Or there may be an additional package designation for the `Foo` class and source schema objects, according either to the `ojspc -packageName` option or the relative location of the `.jsp` file to the current directory when you ran `ojspc`. For example, a `-packageName` setting of `"abc.def"` results in that being the package of the `Foo` class, so you would have a `SCOTT:abc/def/Foo` class schema object.

- `SCOTT:Foo$__jsp__StaticText` class schema object

  With the same package designation as the page implementation class.

- `SCOTT:Foo.res` resource schema object

  With a package designation according to any path specification, either in a JAR file or on the `loadjava` command line, when it was loaded.

For an overview of how `loadjava` names the schema objects it produces, see "Database Schema Objects for Java" on page 6-12.

---

**Notes:**

- Generated names used here are provided as examples only. Such implementation details are subject to change in future releases, although the base name (such as "Foo" here) will always be part of generated names.
- If you are loading translated source (`.java`) for a SQLJ JSP page, you must also load the generated profile file—either a `.ser` Java resource file or a `.class` file, depending on the SQLJ `-ser2class` option. If it is a `.ser` file, schema object naming is comparable to that of a `.res` Java resource file; if it is a `.class` file, schema object naming is comparable to that of other `.class` files. (Remember that the `ojspc -noCompile` option prevents Java compilation, but not SQLJ translation.)

---

## Hotloading Page Implementation Classes in Oracle8*i*

To optionally "hotload" translated JSP pages in Oracle8*i*, use the session shell `java` command to invoke the `main()` method of the page implementation class schema object. See "Overview of the sess_sh Session Shell Tool" on page 6-38 for how to start the tool and connect to the database.

You are required to have previously enabled hotloading through the `ojspc -hotload` option during translation. The `-hotload` option results in a `main()` method and hotloading method being implemented in the page implementation class. Invoking the `main()` method calls the hotloading method and hotloads the page implementation class.

Here is an example ($ is the `sess_sh` prompt):

```
$ java SCOTT:Foo
```

Assuming `Foo` is a class that was translated with the `-hotload` option enabled and was then loaded with `loadjava` into the `SCOTT` schema as in earlier examples, this session shell `java` command will hotload the `Foo` page implementation class.

For an overview of hotloading, see "Overview of Hotloaded Classes in Oracle8i" on page 6-20. For more information about the session shell `java` command, see the *Oracle8i Java Tools Reference*.

## Publishing Translated JSP Pages in Oracle8*i* (Session Shell publishservlet)

To publish translated pages as part of the "deployment with client-side translation" scenario, use the session shell `publishservlet` command. See "Overview of the sess_sh Session Shell Tool" on page 6-38 for how to start the tool and connect to the database.

The `publishservlet` command is for general use in publishing any servlet to run in OSE, but also applies to JSP page implementation classes (which are essentially servlets).

---

**Note:** Servlets and JSP pages that are published with `publishservlet` can be "unpublished" (removed from the JServer JNDI namespace) with the session shell `unpublishservlet` command. See "Unpublishing JSP Pages with unpublishservlet" on page 6-66.

---

### Overview of publishservlet Syntax and Options

Starting `sess_sh` establishes a connection to the Oracle8*i* database. Once you start `sess_sh`, you can run the `publishservlet` command from the session shell `$` prompt.

The `publishservlet` command uses the following general syntax:

```
$ publishservlet context servletName className -virtualpath path [-stateless] [-reuse] [-properties props]
```

When using `publishservlet`, you must specify the following:

1. a servlet context (*`context`* in the command line above)

   This is required by `publishservlet`. You can use the Oracle Servlet Engine's default servlet context:

   ```
   /webdomains/contexts/default
   ```

   This results in a context path of "/".

If you specify some other servlet context, then the context path of that servlet context will be used.

For example, if you specify a servlet context, `mycontext`, that was created as follows:

```
$ createcontext -virtualpath mywebapp /webdomains mycontext
```

then `mywebapp` will be the context path for the published JSP page.

2. a servlet name (*`servletName`* in the command line above)

   This is required by `publishservlet` to specify the name for the JSP page in the `named_servlets` directory, but has no practical use for the JSP developer or user other than for unpublishing. It can be an arbitrary string.

3. a class name (*`className`* in the command line above)

   This is the name of the page implementation class schema object being published.

4. a servlet path (referred to on the command line as the "virtual path")

   Use the `-virtualpath` option. This is required for a JSP page, although it is optional for publishing servlets in general.

Together, the context path and servlet path (along with the host name and port) determine the URL to invoke the page, as described in "URLs for the Oracle Servlet Engine" on page 6-15.

**Important:**

- The servlet context, servlet name, and class name are not preceded by any designating syntax so must appear on the command line in the above order relative to each other. (Any `publishservlet` options can be intermixed with these parameters, however.)
- Enable boolean options, such as `-stateless`, by typing only the option name in the command line (as opposed to setting it to `true`).

In addition to the required parameters, you can specify any of the following options:

- `-stateless`

  This is a boolean option that tells the Oracle Servlet Engine that the JSP page is to be stateless—it will not have access to the `HttpSession` object during execution.

- `-reuse`

  This is a boolean option to specify a new servlet path (referred to as the "virtual path") for a JSP page. If you enable it, then the specified servlet path will be linked to the specified servlet name in the JNDI namespace without `publishservlet` going through the complete publishing process.

  When you enable the `-reuse` option, specify a new servlet path, the servlet context, and a previously published servlet name.

- `-properties` *`props`*

  Use this option to specify properties to be passed to the JSP page as initialization parameters upon execution.

For more information about the `publishservlet` command, see the *Oracle8i Java Tools Reference*.

### Example: Publishing JSP Pages with publishservlet

The following example publishes a JSP page that has been loaded into Oracle8*i* (`$` is the `sess_sh` prompt):

```
$ publishservlet /webdomains/contexts/default -virtualpath Foo.jsp FooServlet SCOTT:Foo
```

For simplicity, the OSE default servlet context is specified, resulting in "`/`" as the context path.

`Foo.jsp` will be the servlet path. (You can specify any name you want for the servlet path, but naming it according to the original source file name is a good convention.)

`FooServlet` will be the servlet name in the OSE `named_servlets` directory, but this name generally will not be used, except for unpublishing.

`SCOTT:Foo` is the page implementation class schema object being published.

After the above `publishservlet` command, the end-user would invoke the JSP page with a URL as follows:

```
http://host[:port]/Foo.jsp
```

Access it dynamically from another JSP page in the application, suppose a page published as `Bar.jsp`, as follows (using page-relative syntax and then application-relative syntax):

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

or:

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

---

**Note:** Both the servlet path and the servlet name specified in the `publishservlet` command are entered into the JServer JNDI namespace, although only the servlet path is generally of interest to JSP users. OSE uses JNDI to look up any published JSP page or servlet.

---

### Unpublishing JSP Pages with unpublishservlet

The `sess_sh` tool also has an `unpublishservlet` command that removes a servlet or JSP page from the JServer JNDI namespace. This does *not*, however, remove the servlet class schema object or page implementation class schema object from the database.

Specify the context, servlet path (referred to on the command line as the "virtual path"), and servlet name. Following is the general syntax to unpublish a JSP page:

```
$ unpublishservlet -virtualpath path context servletName
```

For example, to unpublish the page that was published in the previous section:

```
$ unpublishservlet -virtualpath Foo.jsp /webdomains/contexts/default FooServlet
```

# Additional JSP Deployment Considerations

Most of this chapter focuses on translation and deployment when targeting the Oracle Servlet Engine, because running in the database is a special situation requiring special considerations and logistics.

This section covers a variety of additional deployment considerations and scenarios, mostly for situations where you are *not* targeting the Oracle Servlet Engine.

The following topics are covered:

- Doc Root for Oracle Internet Application Server Versus Oracle Servlet Engine
- Use of ojspc for Pre-Translation for Non-OSE Environments
- General JSP Pre-Translation Without Execution
- Deployment of Binary Files Only
- WAR Deployment
- Deployment of JSP Pages with JDeveloper

## Doc Root for Oracle Internet Application Server Versus Oracle Servlet Engine

Both the Oracle Servlet Engine and the Oracle Internet Application Server use the Oracle HTTP Server, essentially an Apache environment, as the Web server for HTTP requests. However, each environment uses its own doc root.

JSP pages and servlets running in the Oracle Servlet Engine, which are routed through the Apache `mod_ose` module provided by Oracle, use the OSE doc root of the relevant servlet context. OSE doc root directories are in the file system, but are linked to the Oracle8*i* JNDI mechanism.

Remember that for JSP pages running in OSE, only static files are located in or under the doc root. JSP pages are in the database.

The OSE doc root directory is either the default doc root—`$ORACLE_HOME/jis/public_html`—or a doc root specified using the session shell `createcontext` command `-docroot` option when the servlet context was created.

JSP pages and servlets running in the Apache/JServ environment of the Oracle Internet Application Server (release 1.0.x), which are routed through the Apache `mod_jserv` module provided with JServ, use the Apache doc root. This doc root (typically `htdocs`) is set in the `DocumentRoot` command of the Apache `httpd.conf` configuration file.

For JSP pages running in JServ, JSP pages as well as static files are located in or under the doc root.

If you are migrating between the Apache/JServ environment and the OSE environment, move or copy static files to the appropriate doc root.

---

**Note:** For an overview of the role of the Oracle HTTP Server and its `mod_ose` and `mod_jserv` modules, see "Role of the Oracle HTTP Server, Powered by Apache" on page 2-6.

---

## Use of ojspc for Pre-Translation for Non-OSE Environments

The Oracle `ojspc` tool, described in detail in "The ojspc Pre-Translation Tool" on page 6-23, is typically used for client-side JSP translation for deployment to Oracle8*i*. However, you can use `ojspc` to pre-translate JSP pages in any environment, which may be useful in saving end-users the translation overhead the first time a page is executed.

If you are pre-translating in some other target environment, specify the `ojspc -d` option to set an appropriate base directory for placement of generated binary files.

As an example, consider an Apache/JServ environment with the following JSP source file:

```
htdocs/test/foo.jsp
```

A user would invoke this with the following URL:

http://*host*[:*port*]/test/foo.jsp

During on-demand translation at execution time, the OracleJSP translator would use a base directory of `htdocs/_pages` for placement of generated binary files. Therefore, if you pre-translate, you should set `htdocs/_pages` as the base directory for binary output, such as in the following example (assume `%` is a UNIX prompt):

```
% cd htdocs
% ojspc -d _pages test/foo.jsp
```

The URL noted above specifies an application-relative path of `test/foo.jsp`, so at execution time the OracleJSP container looks for the binary files in a `test` subdirectory under the `htdocs/_pages` directory. This subdirectory would be created automatically by `ojspc` if it is run as in the above example. At execution time, the OracleJSP container would find the pre-translated binaries and would not have to perform translation, assuming that the source file was not altered after pre-translation. (By default, the page would be re-translated if the source file timestamp is later than the binary timestamp, assuming the source file is available and the `bypass_source` configuration parameter is not enabled.)

## General JSP Pre-Translation Without Execution

In an on-demand translation environment, it is possible to specify JSP pre-translation only, without execution, by enabling the `jsp_precompile` request parameter when invoking the JSP page from the end-user's browser.

Following is an example:

```
http://host[:port]/foo.jsp?jsp_precompile
```

Refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, for more information.

## Deployment of Binary Files Only

If your JSP source is proprietary, you can avoid exposing the source by pre-translating JSP 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 environment that supports the OracleJSP container. There are two aspects to this scenario:

- You must deploy the binary files appropriately.
- In the target environment, OracleJSP must be configured properly to run pages when the `.jsp` (or `.sqljsp`) source is not available.

### Deploying the Binary Files

After JSP pages have been translated, archive the directory structure and contents that are under the binary output directory, then copy the directory structure and contents to the target environment, as appropriate. For example:

- If you pre-translate with `ojspc`, you should specify a binary output directory with the `ojspc -d` option, then archive the directory structure under that specified directory.
- If you are archiving binary files produced during previous execution in an Apache/JServ (on-demand translation) environment, archive the output directory structure, typically under the `htdocs/_pages` directory.

In the target environment, restore the archived directory structure under the appropriate directory, such as under the `htdocs/_pages` directory in an Apache/JServ environment.

### Configuring OracleJSP for Execution with Binary Files Only

Set OracleJSP configuration parameters as follows to execute JSP pages when the `.jsp` or `.sqljsp` source is unavailable:

- `bypass_source=true`
- `developer_mode=false`

Without these settings, OracleJSP will always look for the `.jsp` or `.sqljsp` 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 `.jsp` or `.sqljsp` file.

With these parameters 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 an example, consider an Apache/JServ environment—if the binary files for `foo.jsp` are in the `htdocs/_pages/test` directory, then the page can be invoked with the following URL without `foo.jsp` being present:

```
http://host:[port]/test/foo.jsp
```

For how to set configuration parameters, see "OracleJSP Configuration Parameter Settings" on page A-25.

## 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*.

In typical JSP 1.1 implementations, JSP pages are deployed through the *WAR* (Web archive) mechanism. WAR files are created using 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 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 intended.

Deployment configurations in the `web.xml` deployment descriptor include mappings between servlet paths and the JSP pages and servlets that will be invoked. Many additional features can be specified in `web.xml` as well, such as timeout values for application modules, mappings of file name extensions to MIME types, and mappings of error codes to JSP error pages.

To summarize, the WAR file includes the following:

- `web.xml` deployment descriptor
- JSP pages
- required JavaBeans and other support classes
- required static HTML files

For more information, see the Sun Microsystems *Java Servlet Specification, Version 2.2*.

---

**Note:** The OracleJSP WAR file implementation and further documentation will be available through the Oracle Technology Network shortly after the Oracle8*i* 8.1.7 release.

In release 8.1.7, OracleJSP uses `web.xml` in only a limited way, for JSP tag library descriptors and servlet URL shortcuts.

---

## Deployment of JSP Pages with JDeveloper

Oracle JDeveloper release 3.1 includes a deployment option, "Web Application to Web Server", that was added specifically for JSP applications.

This option generates a deployment profile that specifies the following:

- a JAR file containing Business Components for Java (BC4J) classes required by the JSP application
- static HTML files required by the JSP application
- the path to the Web server

The developer can either deploy the application immediately upon creating the profile or save the profile for later use.

# 7

# JSP Tag Libraries and the Oracle JML Tags

This chapter discusses custom tag libraries, covering the basic framework that vendors can use to provide their own libraries and documenting the JML tag library that OracleJSP provides as a sample. This discussion includes the following topics:

- Standard Tag Library Framework
- Overview of the JSP Markup Language (JML) Sample Tag Library
- JSP Markup Language (JML) Tag Descriptions

# 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

---

**Note:** Do not use the `servlet.jar` file of the Tomcat 3.1 beta servlet/JSP implementation if you are using custom tags. The constructor signature was changed for the class `javax.servlet.jsp.tagext.TagAttributeInfo`, which will result in compilation errors. Instead, use the `servlet.jar` file that is provided with OracleJSP or the production version of Tomcat 3.1.

---

## 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:

- 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 referred to 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.

## 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 (TLD) 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 tag handler instance or different tag handler instances—this implementation detail is left to the discretion of JSP vendors. OracleJSP uses a separate tag handler instance for each use of a tag.

---

### Custom Tag Body Processing

Custom tags, like standard JSP tags, may or may not have a body. And in the case of a custom tag, even when there is a body, it may not need special handling by the tag handler.

There are three 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 have to 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 handling 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 integer 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 handling by the tag handler

- EVAL_BODY_TAG if there is a body that requires special handling 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 handling by the tag handler, the tag handler class implements 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. Action processing—whatever you want the action tag to accomplish—is implemented in the doStartTag() method. The doEndTag() method would implement 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" on page 7-5). 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 handling by the tag handler, the tag handler class implements 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.)

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 for use. This is a subclass of javax.servlet.jsp.JspWriter that can be used to process body evaluations so that they can re-extracted later. The BodyTag interface includes a setBodyContent() method that can be used by the JSP container 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 `foo` and `bar`. All that is required is knowledge of the name of the object (`myobj`).

---

**Important:** 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 integer 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
- a boolean indicating whether it is a newly declared variable
- its scope

> **Important:** The `getVariableInfo()` method must return a fully qualified class name, such as for a JML datatype, for the Java type of the scripting variable. (Note that primitive types are not supported.)

See "Sample Tag-Extra-Info Class: ExampleLoopTagTEI.java" on page 7-17 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 `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 document that contains information about a tag library and about 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 the attributes of the tag (the attributes that you specify whenever you use the custom tag)

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>
  <info>
      Perform a server-side action (one mandatory attr; one optional)
  </info>
  <attribute>
    <name>attr1</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>attr2</name>
    <required>false</required>
  </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.

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*.

---

**Note:** In the Tomcat 3.1 servlet/JSP implementation, the TLD file `bodycontent` parameter for a given tag is not read if the tag itself (in the JSP page) has no body. It is possible, therefore, to have an invalid `bodycontent` value in your TLD file (such as `none` instead of `empty`) without realizing it. Using the file in another JSP environment, such as OracleJSP, would then result in errors.

---

## Use of web.xml for Tag Libraries

The Sun Microsystems *Java Servlet Specification, Version 2.2* 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.)

---

**Important:** When a JSP application uses a `web.xml` file, you must deploy `web.xml` with the application. Treat it as a Java resource file.

---

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 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 7-14 for an example.

See the Sun Microsystems *Java Servlet Specification, Version 2.2* 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.

---

**Notes:**

- Do not use the sample `web.xml` file from the Tomcat 3.1 servlet/JSP implementation. It introduces new elements that will not pass the standard DTD XML validation.
- Do not use the term "urn" instead of "uri" in a `web.xml` file. Some JSP implementations allow this (such as Tomcat 3.1), but using "urn" will not pass the standard DTD XML validation.

---

## 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, taglib directives 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 1-8 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 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

### Sample JSP Page: exampletag.jsp

Following is a sample JSP page that uses the `loop` tag, specifying that the outer loop be executed five times and the inner loop three times:

```
examplestag.jsp
<%@ taglib prefix="foo" uri="/WEB-INF/exampletag.tld" %>
<% 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

Following is the 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.

```
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

Following is 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` (this must be specified 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

Following is 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:

- `examples.ExampleLoopTag` is the tag handler class.
- `examples.ExampleLoopTagTEI` is the tag-extra-info class.
- The `bodycontent` specification is `JSP`, meaning the JSP translator should process and translate the body code.
- There are two attributes, `index` and `count`, and both are mandatory. The `count` attribute can be a request-time JSP expression.

```
<?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 tab 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>
```

```
<!--
  there should be no <urn></urn> here
-->
  <info>
        A simple tab 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>
```

## Overview of the JSP Markup Language (JML) Sample Tag Library

OracleJSP supplies the JSP Markup Language (JML) sample tag library, which is portable to any standard JSP environment. JML tags, as with those of any standard tag library, are completely compatible with regular JSP script and can be used in any JSP page.

Many of the JML tags are intended to simplify coding syntax for JSP developers who are not proficient with Java. There are also tags for XML transformations (as described in Chapter 5), bean binding, and general utility.

The following topics are covered here:

- JML Tag Library Philosophy
- JML Tag Categories
- JML Tag Library Description File and taglib Directive

Note the following requirements for using JML tags:

- Install the file `ojsputil.jar` and include it in your classpath. This file is provided with the OracleJSP installation.
- Make sure that the tag library description file, `jml.tld`, is deployed with the application and is in the location specified in the `taglib` directives of your JSP pages. See "JML Tag Library Description File and taglib Directive" on page 7-22.

---

**Notes:**

- OracleJSP also provides a tag library for SQL functionality. This is described in "OracleJSP Tag Library for SQL" on page 5-24.
- Prior to OracleJSP release 1.1.0.0.0 and the release of the JSP 1.1 specification, OracleJSP supported JML tags only as Oracle extensions. (The tag library framework was not added to the JavaServer Pages specification until JSP 1.1.) For these releases, Oracle-specific JML tag processing was built into the OracleJSP translator. This is referred to as "compile-time JML support" and is described in Appendix C, "Compile-Time JML Tag Support".

---

## JML Tag Library Philosophy

JavaServer Pages technology is intended for two separate developer communities—those whose primary skill is Java programming and those whose primary skill is in designing static content, particularly in HTML, and may have limited scripting experience.

The JML tag library is designed to allow most Web developers, with little or no knowledge of Java, to assemble JSP applications with a full complement of program flow-control features.

This model presumes that the business logic is contained in JavaBeans that are developed separately by a Java developer.

## JML Tag Categories

The JML tag library covers a wide feature set. The major functional categories are summarized in Table 7–1.

***Table 7–1 JML Tag Functional Categories***

| Tag Categories | Tags | Functionality |
|---|---|---|
| bean binding tags | useVariable<br>useForm<br>useCookie<br>remove | These tags are to declare or undeclare a JavaBean at a specified JSP scope. See "Bean Binding Tag Descriptions" on page 7-30. |
| logic/flow control tags | if<br>choose..when..otherwise<br>foreach<br>return<br>flush | These tags offer simplified syntax to define code flow, such as for iterative loops or conditional branches. See "Logic and Flow Control Tag Descriptions" on page 7-34. |
| XML transformation tags | transform<br>styleSheet | These tags simplify the process of applying an XSL stylesheet to all or part of JSP page output. See "JML Tags for XSL Stylesheets" on page 5-10. |

## JML Tag Library Description File and taglib Directive

As with any tag library following the JSP 1.1 specification, the tags of the JML library are specified in an XML-style tag library description (TLD) file.

This TLD file is provided with the OracleJSP sample applications. It must be deployed with any JSP application that uses JML tags, and specified in a `taglib` directive for any page using JML tags.

### JML taglib Directive

A JSP page using JML tags must specify the TLD file in a `taglib` directive that supplies a standard universal resource indicator (URI) to locate the file. The URI syntax is typically application-relative, such as in the following example:

```
<%@ taglib uri="/WEB-INF/jml.tld" prefix="jml" %>
```

Alternatively, instead of using the full path to the TLD file, as in this example, you can specify a URI shortcut in the `web.xml` file then use the shortcut in your `taglib` directives. See "Use of web.xml for Tag Libraries" on page 7-12.

For general information about tag library description files, see "Tag Library Description Files" on page 7-10.

### JML TLD File Listing

This section lists the entire TLD file for the JML tag library, as supported in OracleJSP release 1.1.0.0.0.

```
<?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 tab library descriptor -->

<taglib>
  <!-- after this the default space is
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"
   -->

 <tlibversion>1.0</tlibversion>
 <jspversion>1.1</jspversion>
 <shortname>jml</shortname>
 <info>
  Oracle's jml tag library.  Not all of the jml
```

```
 tag's available in the Oracle JSP environment
 are provided in this library.  No jsp: tags are
 duplicated, some tags are unavailable, and some tags
 have stricter syntax.  No bean expressions are supported.

 The differences are:
  *-jml:call - not available
  * jml:choose - works as documented
  * jml:flush - works as documented
  * jml:for - works as documented
  * jml:foreach - the type attribute is required, otherwise,
   as documented
  *!jml:forward - use jsp:forward
  *!jml:getProperty - use jsp:getProperty
  * jml:if - works as documented
  *!jml:include - use jsp:include
  *-jml:lock - not available
  *!jml:plugin - use jsp:plugin
  * jml:print - the expression to print must be supplied as
   an attribute.  i.e. the tag cannot have a body
  * jml:remove - works as documented
  * jml:return - works as documented
  *-jml:set - not available
  *!jml:setProperty - use jsp:setProperty
  * jml:styleSheet - works as documented
  * jml:transform - works as documented
  *!jml:useBean - use jsp:useBean
  * jml:useCookie - works as documented
  * jml:useForm - works as documented
  * jml:useVariable - works as documented
</info>

  <!-- The choose tag -->
<tag>
 <name>choose</name>
 <tagclass>oracle.jsp.jml.tagext.JmlChoose</tagclass>
 <bodycontent>JSP</bodycontent>
 <info>
  The outer tag of a multiple choice logic block,
  choose
   when condition1
   when condition2
   otherwise
  end choose
 </info>
```

```
</tag>

  <!-- The flush tag -->
<tag>
 <name>flush</name>
 <tagclass>oracle.jsp.jml.tagext.JmlFlush</tagclass>
 <bodycontent>empty</bodycontent>
 <info>
  Flush the current JspWriter
 </info>
</tag>

<!-- The for tag -->
<tag>
 <name>for</name>
 <tagclass>oracle.jsp.jml.tagext.JmlFor</tagclass>
 <teiclass>oracle.jsp.jml.tagext.JmlForTEI</teiclass>
 <bodycontent>JSP</bodycontent>
 <info>
  A simple for loop
 </info>

 <attribute>
  <name>id</name>
  <required>true</required>
 </attribute>
 <attribute>
  <name>from</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
  <name>to</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>

<!-- The foreach tag -->
<tag>
 <name>foreach</name>
 <tagclass>oracle.jsp.jml.tagext.JmlForeach</tagclass>
 <teiclass>oracle.jsp.jml.tagext.JmlForeachTEI</teiclass>
 <bodycontent>JSP</bodycontent>
 <info>
```

```
  A foreach loop for iterating arrays, enumerations,
  and vector’s.
 </info>

 <attribute>
  <name>id</name>
  <required>true</required>
 </attribute>
 <attribute>
  <name>in</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
  <name>type</name>
  <required>true</required>
 </attribute>
 <attribute>
  <name>limit</name>
  <required>false</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>


<!-- The if tag -->
<tag>
 <name>if</name>
 <tagclass>oracle.jsp.jml.tagext.JmlIf</tagclass>
 <bodycontent>JSP</bodycontent>
 <info>
  A classic if
 </info>

 <attribute>
  <name>condition</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>

  <!-- The otherwise tag -->
<tag>
 <name>otherwise</name>
 <tagclass>oracle.jsp.jml.tagext.JmlOtherwise</tagclass>
```

```
 <bodycontent>JSP</bodycontent>
 <info>
  (optional) final part of a choose block
 </info>
</tag>

  <!-- The print tag -->
<tag>
 <name>print</name>
 <tagclass>oracle.jsp.jml.tagext.JmlPrint</tagclass>
 <bodycontent>empty</bodycontent>
 <info>
  print the expression specified in the eval attribute
 </info>
 <attribute>
  <name>eval</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>

  <!-- The remove tag -->
<tag>
 <name>remove</name>
 <tagclass>oracle.jsp.jml.tagext.JmlRemove</tagclass>
 <bodycontent>empty</bodycontent>
 <info>
  remove the specified object from the pageContext
 </info>
 <attribute>
  <name>id</name>
  <required>true</required>
 </attribute>
 <attribute>
  <name>scope</name>
  <required>false</required>
 </attribute>
</tag>

  <!-- The return tag -->
<tag>
 <name>return</name>
 <tagclass>oracle.jsp.jml.tagext.JmlReturn</tagclass>
 <bodycontent>empty</bodycontent>
 <info>
```

```
  Skip the rest of the page
 </info>
</tag>

  <!-- The styleSheet tag -->
<tag>
 <name>styleSheet</name>
 <tagclass>oracle.jsp.jml.tagext.JmlStyleSheet</tagclass>
 <bodycontent>JSP</bodycontent>
 <info>
  Transform the body of the tag using a stylesheet
 </info>
 <attribute>
  <name>href</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>

  <!-- The transform tag -->
<tag>
 <name>transform</name>
 <tagclass>oracle.jsp.jml.tagext.JmlStyleSheet</tagclass>
 <bodycontent>JSP</bodycontent>
 <info>
  Transform the body of the tag using a stylesheet
 </info>
 <attribute>
  <name>href</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>

  <!-- The useCookie tag -->
<tag>
 <name>useCookie</name>
 <tagclass>oracle.jsp.jml.tagext.JmlUseCookie</tagclass>
 <teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
 <bodycontent>empty</bodycontent>
 <info>
  create a jml variable and initialize it to a cookie value
 </info>
 <attribute>
  <name>id</name>
```

```
    <required>true</required>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>type</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>cookie</name>
    <required>true</required>
  </attribute>
</tag>

    <!-- The useForm tag -->
<tag>
  <name>useForm</name>
  <tagclass>oracle.jsp.jml.tagext.JmlUseForm</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
  <bodycontent>empty</bodycontent>
  <info>
    create a jml variable and initialize it to a parameter value
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>type</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>param</name>
    <required>true</required>
  </attribute>
</tag>

    <!-- The useVariable tag -->
<tag>
```

```
  <name>useVariable</name>
  <tagclass>oracle.jsp.jml.tagext.JmlUseVariable</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
  <bodycontent>empty</bodycontent>
  <info>
   create a jml variable and initialize it to a parameter value
  </info>
  <attribute>
   <name>id</name>
   <required>true</required>
  </attribute>
  <attribute>
   <name>scope</name>
   <required>false</required>
  </attribute>
  <attribute>
   <name>type</name>
   <required>true</required>
  </attribute>
  <attribute>
   <name>value</name>
   <required>false</required>
   <rtexprvalue>true</rtexprvalue>
  </attribute>
 </tag>

   <!-- The when tag -->
 <tag>
  <name>when</name>
  <tagclass>oracle.jsp.jml.tagext.JmlWhen</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
   one part of a choose block, see choose
  </info>
  <attribute>
   <name>condition</name>
   <required>true</required>
   <rtexprvalue>true</rtexprvalue>
  </attribute>
 </tag>

</taglib>
```

# JSP Markup Language (JML) Tag Descriptions

This section documents the JML tags that are supported in the OracleJSP 1.1.0.0.0 runtime implementation, following the JSP 1.1 specification. They are categorized as follows:

- Bean Binding Tag Descriptions
- Logic and Flow Control Tag Descriptions

For an elementary sample using some of the tags described here, see "JML Tag Sample—hellouser_jml.jsp" on page 9-29.

Tags for XML transformations are documented separately, in "JML Tags for XSL Stylesheets" on page 5-10.

## Syntax Symbology and Notes

For the syntax documentation in the tag descriptions, note the following:

- *Italics* indicate that you must specify a value or string.
- Optional attributes are enclosed in square brackets: `[...]`
- Default values of optional attributes are indicated in **bold**.
- Choices in how to specify an attribute are separated by vertical bars: |
- The prefix "jml:" is used. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.

## Bean Binding Tag Descriptions

This section documents the following JML tags, which are used for bean-binding operations:

- JML useVariable Tag
- JML useForm Tag
- JML useCookie Tag
- JML remove Tag

### JML useVariable Tag

This tag offers a convenient alternative to the `jsp:useBean` tag for declaring simple variables.

#### Syntax

```
<jml:useVariable id = "beanInstanceName"
                [scope = "page | request | session | application"]
                 type = "string | boolean | number | fpnumber"
                [value = "stringLiteral | <%= jspExpression %>"] />
```

#### Attributes

- `id`—Names the variable being declared. This attribute is required.
- `scope`—Defines the duration or scope of the variable (as with a `jsp:useBean` tag). This attribute is optional; the default scope is `page`.
- `type`—Specifies the type of the variable (the type specifications refer to `JmlString`, `JmlBoolean`, `JmlNumber`, or `JmlFPNumber`). This attribute is required.
- `value`—Allows the variable to be set directly in the declaration, as either a string literal or a JSP expression enclosed in `<%=... %>` syntax. This attribute is optional. If it is not specified, the value remains the same as when it was last set (if it already exists) or is initialized with a default value. If it is specified, then the value is always set, regardless of whether this declaration instantiates the object or merely acquires it from the named scope.

**Example** Consider the following example:

```
<jml:useVariable id = "isValidUser" type = "boolean" value = "<%= dbConn.isValid() %>" scope = "session" />
```

This is equivalent to the following:

```
<jsp:useBean id = "isValidUser" class = "oracle.jsp.jml.JmlBoolean" scope = "session" />
<jsp:setProperty name="isValidUser" property="value" value = "<%= dbConn.isValid() %>" />
```

### JML useForm Tag

This tag provides a convenient syntax for declaring variables and setting them to values passed in from the request.

#### Syntax

```
<jml:useForm id = "beanInstanceName"
            [scope = "page | request | session | application"]
            [type = "string | boolean | number | fpnumber"]
             param = "requestParameterName" />
```

### Attributes

- `id`—Names the variable being declared or referenced. This attribute is required.
- `scope`—Defines the duration or scope of the variable (as with a `jsp:useBean` tag). This attribute is optional; the default scope is `page`.
- `type`—Specifies the type of the variable (the type specifications refer to `JmlString`, `JmlBoolean`, `JmlNumber`, or `JmlFPNumber`). This attribute is optional; the default setting is `string`.
- `param`—Specifies the name of the request parameter whose value is used in setting the variable. This attribute is required. If the request parameter exists, then the variable value is always updated, regardless of whether this declaration brings the variable into existence. If the request parameter does not exist, then the variable value remains unchanged.

**Example** The following example sets a session variable named `user` of the type `string` to the value of the request parameter named `user`.

```
<jml:useForm id = "user" type = "string" param = "user" scope = "session" />
```

This is equivalent to the following:

```
<jsp:useBean id = "user" class = "oracle.jsp.jml.JmlString" scope = "session" />
<jsp:setProperty name="user" property="value" param = "user" />
```

## JML useCookie Tag

This tag offers a convenient syntax for declaring variables and setting them to values contained in cookies.

### Syntax

```
<jml:useCookie id = "beanInstanceName"
              [scope = "page | request | session | application"]
              [type = "string | boolean | number | fpnumber"]
               cookie = "cookieName" />
```

### Attributes

- `id`—Names the variable being declared or referenced. This attribute is required.
- `scope`—Defines the duration or scope of the variable. This attribute is optional; the default scope is `page`.

- `type`—Identifies the type of the variable (the type specifications refer to `JmlString`, `JmlBoolean`, `JmlNumber`, or `JmlFPNumber`). This attribute is optional; the default setting is `string`.
- `cookie`—Specifies the name of the cookie whose value is used in setting this variable. This attribute is required. If the cookie exists, then the variable value is always updated, regardless of whether this declaration brings the variable into existence. If the cookie does not exist, then the variable value remains unchanged.

**Example** The following example sets a request variable named `user` of the type `string` to the value of the cookie named `user`.

```
<jml:useCookie id = "user" type = "string" cookie = "user" scope = "request" />
```

This is equivalent to the following:

```
<jsp:useBean id = "user" class = "oracle.jsp.jml.JmlString" scope = "request" />
<%
     Cookies [] cookies = request.getCookies();
     for (int i = 0; i < cookies.length; i++) {
             if (cookies[i].getName().equals("user")) {
                     user.setValue(cookies[i].getValue());
                     break;
             }
     }
%>
```

## JML remove Tag

This tag removes an object from its scope.

### Syntax

```
<jml:remove id = "beanInstanceName"
          [scope = "page | request | session | application" ] />
```

### Attributes

- `id`—Specifies the name of the bean being removed. This attribute is required.
- `scope`—This attribute is optional. If not specified, then scopes are searched in the following order: 1) page, 2) request, 3) session, 4) application. The first object whose name matches `id` is removed.

**Example** The following example removes the session `user` object:

```
<jml:remove id = "user" scope = "session" />
```

This is equivalent to the following:

```
<% session.removeValue("user"); %>
```

## Logic and Flow Control Tag Descriptions

This section documents the following JML tags, which are used for logic and flow control:

- JML if Tag
- JML choose...when...[otherwise] Tags
- JML for Tag
- JML foreach Tag
- JML return Tag
- JML flush Tag

These tags, which are intended for developers without extensive Java experience, can be used in place of Java logic and flow control syntax, such as iterative loops and conditional branches.

### JML if Tag

This tag evaluates a single conditional statement. If the condition is true, then the body of the `if` tag is executed.

#### Syntax

```
<jml:if condition = "<%= jspExpression %>" >
    ...body of if tag (executed if the condition is true)...
</jml:if>
```

#### Attributes

- `condition`—Specifies the conditional expression to be evaluated. This attribute is required.

**Example** The following e-commerce example displays information from a user's shopping cart. The code checks to see if the variable holding the current T-shirt order is empty. If not, then the size that the user has ordered is displayed. Assume `currTS` is of type `JmlString`.

```
<jml:if condition = "<%= !currTS.isEmpty() %>" >
     <S>(size: <%= currTS.getValue().toUpperCase() %>)</S> 
</jml:if>
```

## JML choose...when...[otherwise] Tags

The `choose` tag, with associated `when` and `otherwise` tags, provides a multiple conditional statement.

The body of the `choose` tag contains one or more `when` tags, where each `when` tag represents a condition. For the first `when` condition that is true, the body of that `when` tag is executed. (A maximum of one `when` body is executed.)

If none of the `when` conditions are true, and if the optional `otherwise` tag is specified, then the body of the `otherwise` tag is executed.

### Syntax

```
<jml:choose>
    <jml:when condition = "<%= jspExpression %>" >
        ...body of 1st when tag (executed if the condition is true)...
    </jml:when>
    ...
    [...optional additional when tags...]
    [ <jml:otherwise>
        ...body of otherwise tag (executed if all when conditions false)...
    </jml:otherwise> ]
</jml:choose>
```

**Attributes** The `when` tag uses the following attribute (the `choose` and `otherwise` tags have no attributes):

- `condition`—Specifies the conditional expression to be evaluated. This attribute is required.

**Example** The following e-commerce example displays information from a user's shopping cart. This code checks to see if anything has been ordered. If so, the current order is displayed; otherwise, the user is asked to shop again. (This example omits the code to display the current order.) Presume `orderedItem` is of the type `JmlBoolean`.

```
<jml:choose>
     <jml:when condition = "<%= orderedItem.getValue() %>"  >
          You have changed your order:
             -- output the current order --
     </jml:when>
     <jml:otherwise>
          Are you sure we can't interest you in something, cheapskate?
     </jml:otherwise>
</jml:choose>
```

## JML for Tag

This tag provides the ability to iterate through a loop, as with a Java `for` loop.

The `id` attribute is a local loop variable of the type `java.lang.Integer` that contains the value of the current range element. The range starts at the value expressed in the `from` attributed and is incremented by one after each execution of the body of the loop, until it exceeds the value expressed in the `to` attribute.

Once the range has been traversed, control goes to the first statement following the `for` end tag.

> **Note:** Descending ranges are not supported; the `from` value must be less than or equal to the `to` value.

### Syntax

```
<jml:for id = "loopVariable"
         from = "<%= jspExpression %>"
         to = "<%= jspExpression %>" >
      ...body of for tag (executed once at each value of range, inclusive)...
</jml:for>
```

### Attributes

- `id`—Names the loop variable, which holds the current value in the range. This is a `java.lang.Integer` value and can be used only within the body of the tag. This attribute is required.

- `from`—Specifies the start of the range. This is an expression that must evaluate to a Java `int` value. This is a required attribute.
- `to`—Specifies the end of the range. This is an expression that must evaluate to a Java `int` value. This is a required attribute.

**Example** The following example repeatedly prints "Hello World" as progressively smaller headings (H1, H2, H3, H4, H5).

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

## JML foreach Tag

This tag provides the ability to iterate over a homogeneous set of values.

The body of the tag is executed once per element in the set. (If the set is empty, then the body is not executed.)

The `id` attribute is a local loop variable containing the value of the current set element. Its type is specified in the `type` attribute. (The specified type should match the type of the set elements, as applicable.)

This tag currently supports iterations over the following types of data structures:

- Java array
- `java.util.Enumeration`
- `java.util.Vector`

### Syntax

```
<jml:foreach id = "loopVariable"
             in = "<%= jspExpression %>"
             limit = "<%= jspExpression %>"
             type = "package.class" >
    ...body of foreach tag (executes once for each element in data structure)...
</jml:foreach>
```

#### Attributes

- `id`—Names the loop variable, which holds the value of the current element at each step of the iteration. It can be used only within the body of the tag. Its type is the same as specified in the `type` attribute. The `id` attribute is required.
- `in`—Specifies a JSP expression that evaluates to a Java array, `Enumeration` object, or `Vector` object. This is a required attribute.
- `limit`—Specifies a JSP expression that evaluates to a Java `int` value defining the maximum number of iterations, regardless of the number of elements in the set. This is a required attribute.
- `type`—Specifies the type of the loop variable. This should match the type of the set elements, as applicable. This is a required attribute.

**Example** The following example iterates over the request parameters.

```
<jml:foreach id="name" in="<%= request.getParameterNames() %>" type="java.lang.String" >
    Parameter: <%= name %>
    Value: <%= request.getParameter(name) %> <br>
</jml:foreach>
```

Or, if you want to handle parameters with multiple values:

```
<jml:foreach id="name" in="<%= request.getParameterNames() %>" type="java.lang.String" >
    Parameter: <%= name %>
    Value: <jml:foreach id="val" in="<%=request.getParameterValues(name)%>"
                        type="java.lang.String" >
                <%= val %> :
           </jml:foreach>
    <br>
</jml:foreach>
```

### JML return Tag

When this tag is reached, execution returns from the page without further processing.

#### Syntax

```
<jml:return />
```

#### Attributes

None.

**Example** The following example returns without processing the page if the timer has expired.

```
<jml:if condition="<%= timer.isExpired() %>" >
     You did not complete in time!
     <jml:return />
</jml:if>
```

## JML flush Tag

This tag writes the current contents of the page buffer back to the client.

This applies only if the page is buffered; otherwise, there is no effect.

### Syntax

```
<jml:flush />
```

### Attributes

None.

**Example** The following example flushes the current page contents before performing an expensive operation.

```
<jml:flush />
<% myBean.expensiveOperation(out); %>
```

# 8

# OracleJSP NLS Support

OracleJSP provides standard National Language Support (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 how OracleJSP supports NLS. The following topics are covered (additional topics will be covered in a future release):

- Content Type Settings in the page Directive
- Dynamic Content Type Settings
- OracleJSP Extended Support for Multibyte Parameter Encoding

# Content Type Settings in the page Directive

You can use the `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 (Internet Assigned Numbers Authority) MIME type and `character_set` is an IANA character set. (When specifying a character set, the space after the semi-colon 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://venera.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):

`ftp://venera.isi.edu/in-notes/iana/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 browsers and Internet Explorer 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"));
  ```

# OracleJSP Extended Support for Multibyte Parameter Encoding

Character encoding of request parameters is not well defined in the HTTP specification. Most servlet containers must interpret them using the servlet default encoding, `ISO-8859-1`.

For such environments, where the servlet container cannot encode multibyte request parameters and bean property settings, OracleJSP offers extended support through the `translate_params` configuration parameter.

It is also possible to use equivalent code in the JSP page, which is necessary in the Oracle Servlet Environment.

---

**Important:** Do *not* enable the `translate_params` flag in the following circumstances:

- When the servlet container properly handles multibyte parameter encoding itself. Setting `translate_params` to `true` in this situation will cause incorrect results. As of this writing, however, it is known that Apache/JServ, JSWDK, and Tomcat all do *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 8-6.)

---

## Effect of translate_params in Overriding Non-Multibyte Servlet Containers

Set `translate_params` to `true` to override servlet containers that cannot encode multibyte request parameters and bean property settings. (For information about how to set OracleJSP configuration parameters, see "OracleJSP Configuration Parameter Settings" on page A-25.)

When this flag is enabled, OracleJSP encodes the request parameters and bean property settings based on the character set of the `response` object, as indicated by the `response.getCharacterEncoding()` method.

The `translate_params` flag 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

## Code Equivalent to the translate_params Configuration Parameter

The `translate_params` configuration parameter, being a runtime parameter, cannot be set in the Oracle Servlet Engine environment. (Translation-time configuration can be set for the OSE environment through `ojspc` command-line options. There is no equivalent for runtime parameters.)

For this reason, and possibly other reasons as well, it is useful to be aware of equivalent functionality that can be implemented 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:

- Sets `XXYYZZ` as the parameter name to search for. (Presume `XX`, `YY`, and `ZZ` are three Japanese characters.)
- 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.)
- 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.)
- Encodes the parameter value to `EUC-JP` for further processing or output to the browser.

See the next two sections for an NLS sample that depends on `translate_params` being enabled, and an NLS sample that contains the equivalent code so that it does not depend on the `translate_params` setting.

## NLS 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 OracleJSP 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, and so does not depend 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:

## NLS 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 would cause incorrect results. (This is not a concern in the OSE environment, where the `translate_params` flag is not supported.)

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 8-6.

```
<%@ 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>
```

# 9

# Sample Applications

This chapter provides a variety of code samples for JSP pages and the JavaBeans that they use (as applicable), in the following categories:

- Basic Samples
- JDBC Samples
- Database-Access JavaBean Samples
- Custom Tag Samples
- Samples for Oracle-Specific Programming Extensions
- Samples Using globals.jsa for Servlet 2.0 Environments

# Basic Samples

This section provides JSP samples that are fairly basic but also exemplify use of the Oracle JML datatypes. This includes an elementary "hello" sample, a sample of using a JavaBean, and a more intermediate shopping cart example. The following samples are provided:

- Hello Page—hellouser.jsp
- Usebean Page—usebean.jsp
- Shopping Cart Page—cart.jsp

These examples could use standard datatypes instead, but JML datatypes offer a number of advantages, as described in "JML Extended Datatypes" on page 5-2. JML datatypes are also portable to other JSP environments.

## Hello Page—hellouser.jsp

This sample is an elementary JSP "hello" page. Users are presented with a form to enter their name. After they submit the name, the JSP page redisplays the form with the name at the top.

```
<%-----------------------------------------------------------
   Copyright (c) 1999, Oracle Corporation. All rights reserved.
------------------------------------------------------------%>

<%@page session="false" %>

<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope="request" >
   <jsp:setProperty name="name" property="value" param="newName" />
</jsp:useBean>

<HTML>
<HEAD>
<TITLE>
Hello User
</TITLE>
</HEAD>

<BODY>

<% if (!name.isEmpty()) { %>
<H3>Welcome <%= name.getValue() %></H3>
<% } %>
```

```
<P>
Enter your Name:
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>
```

## Usebean Page—usebean.jsp

This page uses a simple JavaBean, `NameBean`, to illustrate usage of the `jsp:useBean` tag. Code for the bean is provided as well as code for the page.

### Code for usebean.jsp

```
<%-----------------------------------------------------------
   Copyright (c) 1999, Oracle Corporation. All rights reserved.
------------------------------------------------------------%>

<%@ page import="beans.NameBean"  %>

<jsp:useBean id="pageBean" class="beans.NameBean" scope="page" />
<jsp:setProperty name="pageBean" property="*" />

<jsp:useBean id="sessionBean" class="beans.NameBean" scope="session" />
<jsp:setProperty name="sessionBean" property="*" />

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

<H3> Welcome to the UseBean JSP </H3>
<P><B>Page bean: </B>
<% if (pageBean.getNewName().equals("")) { %>
  I don't know you.
<% } else { %>
  Hello <%= pageBean.getNewName() %> !
<%  } %>

<P><B>Session bean: </B>
<% if (sessionBean.getNewName().equals("")) { %>
  I don't know you either.
```

```
<% } else {
       if ((request.getParameter("newName") == null) ||
           (request.getParameter("newName").equals(""))) { %>
         Aha, I remember you.
<%     } %>
       You are <%= sessionBean.getNewName() %>.
<% } %>

<P>May we have your name?
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

### Code for NameBean.java

```
package beans;

public class NameBean {

  String newName="";

  public void NameBean() {  }

  public String getNewName() {
    return newName;
  }
  public void setNewName(String newName) {
    this.newName = newName;
  }
}
```

## Shopping Cart Page—cart.jsp

This sample shows how to use session state to maintain a shopping cart. The user chooses a T-shirt or sweatshirt to order and the order is then redisplayed. If shopping continues and the order is changed, the page redisplays the order, striking out the previous choices as appropriate.

The `cart.jsp` file is the primary source file; it references `index.jsp`. Code for both pages is provided.

### Code for cart.jsp

```
<%-----------------------------------------------------------
   Copyright (c) 1999-2000, Oracle Corporation. All rights reserved.
------------------------------------------------------------%>
<jsp:useBean id="currSS" scope ="session" class="oracle.jsp.jml.JmlString" />
<jsp:useBean id="currTS" scope ="session" class="oracle.jsp.jml.JmlString" />

<HTML>

<HEAD>
        <TITLE>Java Store</TITLE>
</HEAD>

<BODY BACKGROUND=images/bg.gif BGCOLOR=#FFFFFF>

<jsp:useBean id="sweatShirtSize" scope="page" class="oracle.jsp.jml.JmlString" >
   <jsp:setProperty name="sweatShirtSize" property="value" param="SS" />
</jsp:useBean>
<jsp:useBean id="tshirtSize" scope="page" class="oracle.jsp.jml.JmlString" >
   <jsp:setProperty name="tshirtSize" property="value" param="TS" />
</jsp:useBean>

<jsp:useBean id="orderedSweatshirt" scope="page"
class="oracle.jsp.jml.JmlBoolean" >
   <jsp:setProperty name="orderedSweatshirt" property="value"
      value= '<%= !(sweatShirtSize.isEmpty() ||
sweatShirtSize.getValue().equals("none")) %>'  />
</jsp:useBean>

<jsp:useBean id="orderedTShirt" scope="page" class="oracle.jsp.jml.JmlBoolean" >
   <jsp:setProperty name="orderedTShirt" property="value"
      value='<%= !(tshirtSize.isEmpty() || tshirtSize.getValue().equals("none"))
%>'  />
</jsp:useBean>

<P>
<TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0 WIDTH=100% HEIGHT=553>
        <TR>
                <TD WIDTH=33% HEIGHT=61> </TD>
                <TD WIDTH=67% HEIGHT=61> </TD>
        </TR>
        <TR>
                <TD WIDTH=33% HEIGHT=246> </TD>
                <TD WIDTH=67% HEIGHT=246 VALIGN=TOP BGCOLOR=#FFFFFF>
```

```
<% if (orderedSweatshirt.getValue() || orderedTShirt.getValue()) { %>
   Thank you for selecting our fine JSP Wearables!<P>

   <% if (!currSS.isEmpty() || !currTS.isEmpty()) { %>
   You have changed your order:
         <UL>
      <% if (orderedSweatshirt.getValue()) { %>
         <LI>1 Sweatshirt
         <% if (!currSS.isEmpty()) { %>
            <S>(size: <%= currSS.getValue().toUpperCase() %>)</S> 
         <% } %>
         (size: <%= sweatShirtSize.getValue().toUpperCase() %> )
      <% } else if (!currSS.isEmpty()) { %>
            <LI><S>1 Sweatshirt (size: <%= currSS.getValue().toUpperCase()
            %>)</S>
      <% } %>

         <% if (orderedTShirt.getValue()) { %>
         <LI>1 Tshirt
         <% if (!currTS.isEmpty()) { %>
            <S>(size: <%= currTS.getValue().toUpperCase() %>)</S> 
         <% } %>
         (size: <%= tshirtSize.getValue().toUpperCase() %>)
      <% } else if (!currTS.isEmpty()) { %>
            <LI><S>1 Tshirt (size: <%= currTS.getValue().toUpperCase()
            %>)</S>
         <% } %>
   </UL>
   <% } else { %>
   You have selected:
   <UL>
      <% if (orderedSweatshirt.getValue()) { %>
         <LI>1 Sweatshirt (size: <%= sweatShirtSize.getValue().toUpperCase()
                            %>)
      <% } %>

      <% if (orderedTShirt.getValue()) { %>
         <LI>1 Tshirt (size: <%= tshirtSize.getValue().toUpperCase() %>)
      <% } %>

   </UL>
   <% } %>
<% } else { %>
   Are you sure we can’t interest you in something?
```

```
   <% } %>

   <CENTER>
      <FORM ACTION="index.jsp" METHOD="GET"
            ENCTYPE="application/x-www-form-urlencoded">
      <INPUT TYPE="IMAGE" SRC="images/shop_again.gif" WIDTH="91" HEIGHT="30"
            BORDER="0">
      </FORM>
   </CENTER>
   </TD></TR>
</TABLE>

</BODY>

</HTML>

<%
if (orderedSweatshirt.getValue()) {
   currSS.setValue(sweatShirtSize.getValue());
} else {
   currSS.setValue("");
}

if (orderedTShirt.getValue()) {
   currTS.setValue(tshirtSize.getValue());
} else {
   currTS.setValue("");
}
%>
```

### Code for index.jsp

```
<%-----------------------------------------------------------
   Copyright (c) 1999-2000, Oracle Corporation. All rights reserved.
-------------------------------------------------------------%>
<jsp:useBean id="currSS" scope ="session" class="oracle.jsp.jml.JmlString" />
<jsp:useBean id="currTS" scope ="session" class="oracle.jsp.jml.JmlString" />

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>

<HEAD>
 <TITLE>untitled</TITLE>
</HEAD>
```

```
<BODY BACKGROUND="images/bg.gif" BGCOLOR="#FFFFFF">

<FORM ACTION="cart.jsp" METHOD="POST"
ENCTYPE="application/x-www-form-urlencoded">
<P>
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0" WIDTH="100%" HEIGHT="553">
 <TR>
  <TD WIDTH="33%" HEIGHT="61"> </TD>
  <TD WIDTH="67%" HEIGHT="61"> </TD>
 </TR>
 <TR>
  <TD WIDTH="33%" HEIGHT="246"> </TD>
  <TD WIDTH="67%" HEIGHT="246" VALIGN="TOP" BGCOLOR="#FFFFFF">
   <TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0" WIDTH="81%">
    <TR>
     <TD WIDTH="100%" BGCOLOR="#CCFFFF">
      <H4>JSP Wearables
     </TD>
    </TR>
    <TR>
     <TD WIDTH="100%" BGCOLOR="#FFFFFF">

      <BLOCKQUOTE>
      Sweatshirt
      <SPACER TYPE="HORIZONTAL" SIZE="10">($24.95)<BR>
      <SPACER TYPE="HORIZONTAL" SIZE="30">
       <INPUT TYPE="RADIO" NAME="SS" VALUE="xl"
          <%= currSS.getValue().equals("xl") ? "CHECKED" : "" %> >XL
      <SPACER TYPE="HORIZONTAL" SIZE="10">
       <INPUT TYPE="RADIO" NAME="SS" VALUE="l" <%= currSS.getValue().equals("l")
         ? "CHECKED" : "" %> >L
      <SPACER TYPE="HORIZONTAL" SIZE="10">
       <INPUT TYPE="RADIO" NAME="SS" VALUE="m" <%= currSS.getValue().equals("m")
         ? "CHECKED" : "" %> >M
      <SPACER TYPE="HORIZONTAL" SIZE="10">
       <INPUT TYPE="RADIO" NAME="SS" VALUE="s" <%= currSS.getValue().equals("s")
         ? "CHECKED" : "" %> >S
      <SPACER TYPE="HORIZONTAL" SIZE="10">
       <INPUT TYPE="RADIO" NAME="SS" VALUE="xs"
          <%= currSS.getValue().equals("xs") ? "CHECKED" : "" %> >XS
      <SPACER TYPE="HORIZONTAL" SIZE="10">
       <INPUT TYPE="RADIO" NAME="SS" VALUE="none"
          <%= currSS.getValue().equals("none") || currSS.isEmpty()  ?
          "CHECKED" : "" %> >NONE
```

```
        <BR>
        <BR>
        T-Shirt<SPACER TYPE="HORIZONTAL" SIZE="10"> (14.95)<BR>
        <SPACER TYPE="HORIZONTAL" SIZE="30">
         <INPUT TYPE="RADIO" NAME="TS" VALUE="xl"
            <%= currTS.getValue().equals("xl") ? "CHECKED" : "" %> >XL
        <SPACER TYPE="HORIZONTAL" SIZE="10">
         <INPUT TYPE="RADIO" NAME="TS" VALUE="l" <%= currTS.getValue().equals("l")
           ? "CHECKED" : "" %> >L
        <SPACER TYPE="HORIZONTAL" SIZE="10">
         <INPUT TYPE="RADIO" NAME="TS" VALUE="m" <%= currTS.getValue().equals("m")
           ? "CHECKED" : "" %> >M
        <SPACER TYPE="HORIZONTAL" SIZE="10">
         <INPUT TYPE="RADIO" NAME="TS" VALUE="s" <%= currTS.getValue().equals("s")
           ? "CHECKED" : "" %> >S
        <SPACER TYPE="HORIZONTAL" SIZE="10">
         <INPUT TYPE="RADIO" NAME="TS" VALUE="xs"
            <%= currTS.getValue().equals("xs") ? "CHECKED" : "" %> >XS
        <SPACER TYPE="HORIZONTAL" SIZE="10">
         <INPUT TYPE="RADIO" NAME="TS" VALUE="none"
            <%= currTS.getValue().equals("none") || currTS.isEmpty()  ?
            "CHECKED" : "" %> >NONE
        </BLOCKQUOTE>
       </TD>
      </TR>
      <TR>
       <TD WIDTH="100%">
        <DIV ALIGN="RIGHT">
        <P><INPUT TYPE="IMAGE" SRC="images/addtobkt.gif" WIDTH="103" HEIGHT="22"
           ALIGN="BOTTOM" BORDER="0">
        </DIV>
       </TD>
      </TR>
     </TABLE>
    </TD>
   </TR>
  </TABLE>

  </FORM>

  </BODY>

  </HTML>
```

# JDBC Samples

Examples in this section use JDBC to query a database. For the most part they use standard JDBC functionality, although the connection caching examples use Oracle's particular connection caching implementation. The following examples are provided:

- Simple Query—SimpleQuery.jsp
- User-Specified Query—JDBCQuery.jsp
- Query Using a Query Bean—UseHtmlQueryBean.jsp
- Connection Caching—ConnCache3.jsp and ConnCache1.jsp

See the *Oracle8i JDBC Developer's Guide and Reference* for information about Oracle JDBC in general and the Oracle JDBC connection caching implementation in particular.

## Simple Query—SimpleQuery.jsp

This page executes a simple query of the `scott.emp` table, listing employees and their salaries in an HTML table (ordered by employee name).

```
<%@ page import="java.sql.*" %>

<!------------------------------------------------------------------
 * This is a basic JavaServer Page that does a JDBC query on the
 * emp table in schema scott and outputs the result in an html table.
 *
--------------------------------------------------------------------!>
<HTML>
  <HEAD>
    <TITLE>
       SimpleQuery JSP
    </TITLE>
  </HEAD>
 <BODY BGCOLOR=EOFFFO>
 <H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
 !  I am SimpleQuery JSP.
 </H1>
 <HR>
 <B> I will do a basic JDBC query to get employee data
     from EMP table in schema SCOTT..
 </B>
```

```
 <P>
<%
    try {
      // Use the following 2 files whening running inside Oracle 8i
      // Connection conn = new oracle.jdbc.driver.OracleDriver().
      //                      defaultConnection ();
      Connection  conn =
          DriverManager.getConnection((String)session.getValue("connStr"),
                                             "scott", "tiger");
      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                             "FROM scott.emp ORDER BY ename");
      if (rset.next()) {
%>
     <TABLE BORDER=1 BGCOLOR="C0C0C0">
     <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
     <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
     <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
          <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
     </TR>

<%     while (rset.next()) {
%>

     <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
          <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
     </TR>

<% }
%>
      </TABLE>
<%  }
      else {
%>
     <P> Sorry, the query returned no rows! </P>

<%
      }
      rset.close();
      stmt.close();
    } catch (SQLException e) {
      out.println("<P>" + "There was an error doing the query:");
      out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
```

```
%>

 </BODY>
</HTML>
```

## User-Specified Query—JDBCQuery.jsp

This page queries the `scott.emp` table according to a user-specified condition and outputs the results.

```
<%@ page 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>
      <%= runQuery(searchCondition) %>
      <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>
<%!
  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((String)session.getValue("connStr"),
                                           "scott", "tiger");
        stmt = conn.createStatement();
        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><B>");
        do {  sb.append("<LI>" + rset.getString(1) +
                          " earns $ " + rset.getInt(2) + ".</LI>\n");
           } while (rset.next());
        sb.append("</B></UL>");
    }
    return sb.toString();
  }
%>
```

## Query Using a Query Bean—UseHtmlQueryBean.jsp

This page uses a JavaBean, `HtmlQueryBean`, to query the `scott.emp` table according to a user-specified condition. `HtmlQueryBean`, in turn, uses the class `HtmlTable` to format the output into an HTML table. This sample includes code for the JSP page, `HtmlQueryBean`, and `HtmlTable`.

### Code for UseHtmlQueryBean.jsp

```
<jsp:useBean id="htmlQueryBean" class="beans.HtmlQueryBean" scope="session" />
<jsp:setProperty name="htmlQueryBean" property="searchCondition" />

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

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

```
<% } %>

<P><B>Enter a search condition:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="searchCondition" SIZE=30>
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>
```

## Code for HtmlQueryBean.java

```
package beans;
import java.sql.*;

public class HtmlQueryBean {

  private String searchCondition = "";
  private String connStr = null;

  public String getResult() throws SQLException {
    return runQuery();
  }

  public void setSearchCondition(String searchCondition) {
    this.searchCondition = searchCondition;
  }

  public void setConnStr(String connStr) {
    this.connStr = connStr;
  }

  private String runQuery() {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rset = null;
    try {
      if (conn == null) {
      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
      conn = DriverManager.getConnection(connStr,
                                "scott","tiger");
      }
      stmt = conn.createStatement();
      rset = stmt.executeQuery ("SELECT ename as \"Name\", " +
```

```
                "empno as \"Employee Id\","+
                "sal as \"Salary\"," +
                "TO_CHAR(hiredate, 'DD-MON-YYYY') as \"Date Hired\"" +
                "FROM scott.emp " + (searchCondition.equals("") ? "" :
                "WHERE " + searchCondition ));
      return  format(rset);;
    } catch (SQLException e) {
      return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    }
    finally {
    try {
      if (rset!= null) rset.close();
      if (stmt!= null) stmt.close();
      if (conn!= null) conn.close();
      } catch (SQLException ignored) {}
    }
  }

  public static String format(ResultSet rs) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (rs == null || !rs.next())
      sb.append("<P> No matching rows.<P>\n");
    else {
      sb.append("<TABLE BORDER>\n");
      ResultSetMetaData md = rs.getMetaData();
      int numCols = md.getColumnCount();
      for (int i=1; i<= numCols; i++) {
        sb.append("<TH><I>" + md.getColumnLabel(i) + "</I></TH>");
      }
      do {
        sb.append("<TR>\n");
        for (int i = 1; i <= numCols; i++) {
          sb.append("<TD>");
          Object obj = rs.getObject(i);
          if (obj != null)  sb.append(obj.toString());
          sb.append("</TD>");
        }
        sb.append("</TR>");
      } while (rs.next());
      sb.append("</TABLE>");
    }
    return sb.toString();
  }
}
```

### Code for HtmlTable.java

```
import java.sql.*;

public class HtmlTable {

  public static String format(ResultSet rs) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (rs == null || !rs.next())
      sb.append("<P> No matching rows.<P>\n");
    else {
      sb.append("<TABLE BORDER>\n");
      ResultSetMetaData md = rs.getMetaData();
      int numCols = md.getColumnCount();
      for (int i=1; i<= numCols; i++) {
         sb.append("<TH><I>" + md.getColumnLabel(i) + "</I></TH>");
      }
      do {
        sb.append("<TR>\n");

        for (int i = 1; i <= numCols; i++) {
        sb.append("<TD>");
        Object obj = rs.getObject(i);
        if (obj != null)  sb.append(obj.toString());
        sb.append("</TD>");
      }
      sb.append("</TR>");
      } while (rs.next());
      sb.append("</TABLE>");
    }
    return sb.toString();
  }
}
```

## Connection Caching—ConnCache3.jsp and ConnCache1.jsp

This section provides two examples of connection caching using Oracle's caching implementation. This implementation uses the Oracle JDBC `OracleConnectionCacheImpl` class. For introductory information, see "Database Connection Caching" on page 4-6. For further information see, the *Oracle8i JDBC Developer's Guide and Reference*.

The first example, `ConnCache3.jsp`, performs its own cache setup.

The second example, `ConnCache1.jsp`, uses a separate page, `setupcache.jsp`, to do the setup.

Code is provided for all three pages.

---

**Note:** As a more convenient alternative, you can use the `ConnCacheBean` JavaBean provided with OracleJSP. See "Page Using ConnCacheBean—ConnCacheBeanDemo.jsp" on page 9-26.

---

### Code for ConnCache3.jsp (with cache setup)

This sample page handles its own connection cache setup.

```
<%@ page import="java.sql.*, javax.sql.*, oracle.jdbc.pool.*" %>

<!------------------------------------------------------------------
 * This is a JavaServer Page that uses Connection Caching at Session
 * scope.
--------------------------------------------------------------------!>

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

<HTML>
  <HEAD>
    <TITLE>
     ConnCache 3  JSP
    </TITLE>
  </HEAD>
 <BODY BGCOLOR=EOFFFO>
 <H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
 !  I am Connection Caching  JSP.
 </H1>
 <HR>
 <B> Session  Level Connection Caching.
 </B>

 <P>
<%
    try {
      ods.setURL((String)session.getValue("connStr"));
      ods.setUser("scott");
      ods.setPassword("tiger");
```

```
      Connection conn = ods.getConnection ();
      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                          "FROM scott.emp ORDER BY ename");

      if (rset.next()) {
%>
      <TABLE BORDER=1 BGCOLOR="C0C0C0">
      <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
      <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
           <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<%      while (rset.next()) {
%>

      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
           <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<% }
%>
      </TABLE>
<%  }
      else {
%>
      <P> Sorry, the query returned no rows! </P>

<%
      }
      rset.close();
      stmt.close();
      conn.close();  // Put the Connection Back into the Pool

    } catch (SQLException e) {
      out.println("<P>" + "There was an error doing the query:");
      out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

 </BODY>
</HTML>
```

## Code for ConnCache1.jsp and setupcache.jsp

This sample page statically includes another page, `setupcache.jsp`, for its connection cache setup. Code is provided for both pages.

### ConnCache1.jsp

```
<%@ include file="setupcache.jsp" %>
<%@ page import="java.sql.*, javax.sql.*, oracle.jdbc.pool.*" %>

<!------------------------------------------------------------------
 * This is a JavaServer Page that uses Connection Caching over application
 * scope. The Cache is created in an application scope in setupcache.jsp
 * Connection is obtained from the Cache and recycled back once done.
--------------------------------------------------------------------!>

<HTML>
  <HEAD>
    <TITLE>
      ConnCache1 JSP
    </TITLE>
  </HEAD>
 <BODY BGCOLOR=EOFFFO>
 <H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
 !  I am Connection Caching JSP.
 </H1>
 <HR>
 <B> I get the Connection from the Cache and recycle it back.
 </B>

 <P>
<%
    try {
      Connection conn = cods.getConnection();

      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                           "FROM scott.emp ORDER BY ename");
      if (rset.next()) {
%>
      <TABLE BORDER=1 BGCOLOR="C0C0C0">
      <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
      <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
```

```
          <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<%      while (rset.next()) {
%>

      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
          <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<% }
%>
      </TABLE>
<%  }
      else {
%>
      <P> Sorry, the query returned no rows! </P>

<%
      }
      rset.close();
      stmt.close();
      conn.close();  // Put the Connection Back into the Pool
    } catch (SQLException e) {
      out.println("<P>" + "There was an error doing the query:");
      out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

 </BODY>
</HTML>
```

### setupcache.jsp

```
<jsp:useBean id="cods" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
  scope="application">
<%
    cods.setURL((String)session.getValue("connStr"));
    cods.setUser("scott");
    cods.setPassword("tiger");
    cods.setStmtCache (5);
%>
</jsp:useBean>
```

# Database-Access JavaBean Samples

This section provides examples using the Oracle database-access JavaBeans. These beans are provided with OracleJSP but are generally portable to other JSP environments. Note, however, that the connection caching bean relies on the Oracle JDBC implementation of connection caching.

`DBBean` is the simplest of these JavaBeans, with its own connection functionality and supporting queries only. For more complicated operations, use appropriate combinations of `ConnBean` (for simple connections), `ConnCacheBean` (for connection caching), and `CursorBean` (for general SQL DML operations).

For more information, see "Oracle Database-Access JavaBeans" on page 5-13.

The following examples are included:

- Page Using DBBean—DBBeanDemo.jsp
- Page Using ConnBean—ConnBeanDemo.jsp
- Page Using CursorBean—CursorBeanDemo.jsp
- Page Using ConnCacheBean—ConnCacheBeanDemo.jsp

---

**Note:** Oracle also provides custom tags for SQL functionality that use these JavaBeans behind the scenes. For samples using these tags, see "SQL Tag Examples" on page 5-28.

---

## Page Using DBBean—DBBeanDemo.jsp

This page uses a `DBBean` object to connect to the database, execute a query, and output the results as an HTML table.

```
<%@ page import="java.sql.*" %>

<!------------------------------------------------------------------
 * This is a basic JavaServer Page that uses a DB Access Bean and queries
 * dept and emp tables in schema scott and outputs the result in an html table.
 *
--------------------------------------------------------------------!>

<jsp:useBean id="dbbean" class="oracle.jsp.dbutil.DBBean" scope="session">
  <jsp:setProperty name="dbbean" property="User" value="scott"/>
  <jsp:setProperty name="dbbean" property="Password" value="tiger"/>
  <jsp:setProperty name="dbbean" property="URL" value=
       "<%= (String)session.getValue(\"connStr\") %>" />
```

```
</jsp:useBean>

<HTML>
  <HEAD>
    <TITLE>
      DBBeanDemo JSP
    </TITLE>
  </HEAD>
 <BODY BGCOLOR=EOFFFO>
 <H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
 !  I am DBBeanDemo JSP.
 </H1>
 <HR>
 <B> I’m using DBBean and querying DEPT & EMP tables in schema SCOTT.....
     I get all employees who work in the Research department.
 </B>

 <P>
<%
    try {

      String sql_string = " select ENAME from EMP,DEPT " +
                          " where DEPT.DNAME = 'RESEARCH' " +
                          " and DEPT.DEPTNO = EMP.DEPTNO";

      // Make the Connection
      dbbean.connect();

      // Execute the SQL and get a HTML table
      out.println(dbbean.getResultAsHTMLTable(sql_string));

      // Close the Bean to close the connection
      dbbean.close();
    } catch (SQLException e) {
      out.println("<P>" + "There was an error doing the query:");
      out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

 </BODY>
</HTML>
```

## Page Using ConnBean—ConnBeanDemo.jsp

This page uses a `ConnBean` object (for a simple connection) to retrieve a `CursorBean` object, then uses the `CursorBean` object to output query results as an HTML table.

```
<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>

<!-----------------------------------------------------------------
 * This is a basic JavaServer Page that uses a Connection Bean and queries
 * emp table in schema scott and outputs the result in an html table.
 *
-------------------------------------------------------------------!>

<jsp:useBean id="cbean" class="oracle.jsp.dbutil.ConnBean" scope="session">
  <jsp:setProperty name="cbean" property="User" value="scott"/>
  <jsp:setProperty name="cbean" property="Password" value="tiger"/>
  <jsp:setProperty name="cbean" property="URL" value=
       "<%= (String)session.getValue(\"connStr\") %>"/>
  <jsp:setProperty name="cbean" property="PreFetch" value="5"/>
  <jsp:setProperty name="cbean" property="StmtCacheSize" value="2"/>
</jsp:useBean>

<HTML>
  <HEAD>
    <TITLE>
      Connection Bean Demo JSP
    </TITLE>
  </HEAD>
 <BODY BGCOLOR=EOFFFO>
 <H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
 !  I am Connection Bean Demo JSP.
 </H1>
 <HR>
 <B> I’m using connection and a query bean and querying employee names
     and salaries from EMP table in schema SCOTT..
 </B>

 <P>
<%
    try {

      // Make the Connection
      cbean.connect();
```

```
      String sql = "SELECT ename, sal FROM scott.emp ORDER BY ename";

      // get a Cursor Bean
      CursorBean cb = cbean.getCursorBean  (CursorBean.PREP_STMT, sql);

      out.println(cb.getResultAsHTMLTable());

      // Close the cursor bean
      cb.close();
      // Close the Bean to close the connection
      cbean.close();
    } catch (SQLException e) {
      out.println("<P>" + "There was an error doing the query:");
      out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

 </BODY>
</HTML>
```

## Page Using CursorBean—CursorBeanDemo.jsp

This page uses a `ConnBean` object (for a simple connection) and a `CursorBean` object to execute a PL/SQL statement, get a `REF CURSOR`, and translate the results into an HTML table.

```
<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>

<!-------------------------------------------------------------------
 * This is a basic JavaServer Page that uses a Cursor and Conn Beans and queries
 * dept table in schema scott and outputs the result in an html table.
 *
--------------------------------------------------------------------!>

<jsp:useBean id="connbean" class="oracle.jsp.dbutil.ConnBean" scope="session">
  <jsp:setProperty name="connbean" property="User" value="scott"/>
  <jsp:setProperty name="connbean" property="Password" value="tiger"/>
  <jsp:setProperty name="connbean" property="URL" value=
       "<%=    (String)session.getValue(\"connStr\") %>" />
</jsp:useBean>
<jsp:useBean id="cbean" class="oracle.jsp.dbutil.CursorBean" scope="session">
  <jsp:setProperty name="cbean" property="PreFetch" value="10"/>
  <jsp:setProperty name="cbean" property="ExecuteBatch" value="2"/>
```

```
</jsp:useBean>

<HTML>
  <HEAD>
    <TITLE>
      CursorBean Demo JSP
    </TITLE>
  </HEAD>
 <BODY BGCOLOR=EOFFFO>
 <H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
 !  I am Cursor Bean  JSP.
 </H1>
 <HR>
 <B> I'm using cbean and i'm quering department names from DEPT table
     in schema SCOTT..
 </B>

 <P>
<%

    try {

      // Make the Connection
      connbean.connect();

      String sql = "BEGIN OPEN ? FOR SELECT DNAME FROM DEPT; END;";

      // Create a Callable Statement
      cbean.create ( connbean, CursorBean.CALL_STMT, sql);
      cbean.registerOutParameter(1,oracle.jdbc.driver.OracleTypes.CURSOR);

      // Execute the PLSQL
      cbean.executeUpdate ();

      // Get the Ref Cursor
      ResultSet rset = cbean.getCursor(1);

      out.println(oracle.jsp.dbutil.BeanUtil.translateToHTMLTable (rset));

      // Close the RefCursor
      rset.close();

      // Close the Bean
      cbean.close();
```

```
      // Close the connection
      connbean.close();

    } catch (SQLException e) {
      out.println("<P>" + "There was an error doing the query:");
      out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

 </BODY>
</HTML>
```

## Page Using ConnCacheBean—ConnCacheBeanDemo.jsp

This page uses a ConnCacheBean object to obtain a connection from a connection cache. It then uses standard JDBC functionality to execute a query, formatting the results as an HTML table.

```
<%@ page import="java.sql.*, javax.sql.*, oracle.jsp.dbutil.ConnCacheBean" %>

<!------------------------------------------------------------------
 * This is a basic JavaServer Page that does a JDBC query on the
 * emp table in schema scott and outputs the result in an html table.
 * Uses Connection Cache Bean.
--------------------------------------------------------------------!>

<jsp:useBean id="ccbean" class="oracle.jsp.dbutil.ConnCacheBean"
             scope="session">
  <jsp:setProperty name="ccbean" property="user" value="scott"/>
  <jsp:setProperty name="ccbean" property="password" value="tiger"/>
  <jsp:setProperty name="ccbean" property="URL" value=
       "<%= (String)session.getValue(\"connStr\") %>"  />
  <jsp:setProperty name="ccbean" property="MaxLimit" value="5"  />
  <jsp:setProperty name="ccbean" property="CacheScheme" value=
       "<%= ConnCacheBean.FIXED_RETURN_NULL_SCHEME %>"  />
</jsp:useBean>
<HTML>
  <HEAD>
    <TITLE>
      SimpleQuery JSP
    </TITLE>
  </HEAD>
 <BODY BGCOLOR=EOFFFO>
```

```
 <H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
 !  I am Connection Cache Demo Bean
 </H1>
 <HR>
 <B> I will do a basic JDBC query to get employee data
     from EMP table in schema SCOTT. The connection is obtained from
     the Connection Cache.
 </B>

 <P>
<%
   try {
     Connection conn = ccbean.getConnection();

     Statement stmt = conn.createStatement ();
     ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                                "FROM scott.emp ORDER BY ename");
     if (rset.next()) {
%>
      <TABLE BORDER=1 BGCOLOR="C0C0C0">
      <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
      <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
           <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<%      while (rset.next()) {
%>

      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
           <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<% }
%>
      </TABLE>
<%  }
      else {
%>
      <P> Sorry, the query returned no rows! </P>

<%
      }
      rset.close();
```

```
        stmt.close();
        conn.close();
        ccbean.close();
      } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
      }
%>

 </BODY>
</HTML>
```

# Custom Tag Samples

This section includes the following:

- a sample using some of the Oracle JSP Markup Language (JML) custom tags
- referrals to additional custom tag samples elsewhere in this document

## JML Tag Sample—hellouser_jml.jsp

This section provides a basic sample using some of the Oracle JML custom tags.

This is a modified version of the `hellouser.jsp` sample provided earlier in this chapter. For contrast, both the JML code and the original code are provided here.

Note that the runtime implementation of the JML tag library is portable to other JSP environments. For an overview of the runtime implementation, see "Overview of the JSP Markup Language (JML) Sample Tag Library" on page 7-20. For information about the compile-time (non-portable) implementation, see Appendix C, "Compile-Time JML Tag Support".

### Code for hellouser_jml.jsp (using JML tags)

```
<%-----------------------------------------------------------
   Copyright (c) 1999, Oracle Corporation. All rights reserved.
------------------------------------------------------------%>

<%@page session="false" %>
<%@ taglib uri="WEB-INF/jml.tld" prefix="jml" %>

<jml:useForm id="name" param="newName" scope="request" />

<HTML>
<HEAD>
<TITLE>
Hello User
</TITLE>
</HEAD>

<BODY>

<jml:if condition="!name.isEmpty()" >
<H3>Welcome <jml:print eval="name.getValue()" /></H3>
</jml:if>

<P>
```

```
Enter your Name:
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>
```

### Code for hellouser.jsp (not using JML tags)

```
<%-----------------------------------------------------------
   Copyright (c) 1999, Oracle Corporation. All rights reserved.
------------------------------------------------------------%>

<%@page session="false" %>

<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope="request" >
      <jsp:setProperty name="name" property="value" param="newName" />
</jsp:useBean>

<HTML>
<HEAD>
<TITLE>
Hello User
</TITLE>
</HEAD>

<BODY>

<% if (!name.isEmpty()) { %>
<H3>Welcome <%= name.getValue() %></H3>
<% } %>

<P>
Enter your Name:
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>
```

## Pointers to Additional Custom Tag Samples

Additional custom tag samples are provided elsewhere in this document:

- For a complete example of defining and using a standard JSP 1.1-compliant custom tag, see "End-to-End Example: Defining and Using a Custom Tag" on page 7-15.
- For samples using the Oracle custom tag library for SQL functionality, see "SQL Tag Examples" on page 5-28.

# Samples for Oracle-Specific Programming Extensions

This section provides a variety of examples using Oracle-specific extensions. This includes the following:

- Page Using JspScopeListener—scope.jsp
- XML Query—XMLQuery.jsp
- SQLJ Queries—SQLJSelectInto.sqljsp and SQLJIterator.sqljsp

## Page Using JspScopeListener—scope.jsp

This sample illustrates the use of a `JspScopeListener` implementation to allow JSP objects attached to a scope to be notified when they are going "out of scope". The sample implements a generic listener that redispatches the out-of-scope notification to the registered object or method. In using this listener, `scope.jsp` is able to simulate page event handlers for request and page out-of-scope notification.

This sample creates and attaches a listener object to the `request` and `page` scopes. It registers local methods to handle out-of-scope notifications forwarded by the listener. To illustrate this, the sample keeps two counters—the first is a page count; the second is a count of the number of included files.

The current page count is logged when the page goes out of scope. The included page count is logged when the request goes out of scope. The sample then proceeds to include itself five times.

The sample outputs six messages indicating a page count of 1, followed by a single message indicating five `jsp:include` operations occurred.

For general information about the `JspScopeListener` mechanism, see "OracleJSP Event Handling—JspScopeListener" on page 5-32.

### Listener Implementation—PageEventDispatcher

`PageEventDispatcher` is a JavaBean that implements the `JspScopeListener` interface. The interface defines the `outOfScope()` event method, which takes a `JspScopeEvent` object as input. The `outOfScope()` method of a `PageEventDispatcher` object is called when the scope (`application`, `session`, `page`, or `request`) associated with the object is ending.

In this sample, a `PageEventDispatcher` object acts as a redispatcher for the JSP page, allowing the JSP page to host the equivalent of `globals.jsa` "on end" functionality for page and request events. The JSP page creates a `PageEventDispatcher` object for each scope for which it wants to provide an event handler. It then registers the event handler method with the `PageEventDispatcher` object. When the `PageEventDispatcher` object is notified that it is going out of scope, it calls the registered "on end" method of the page.

```
package oracle.jsp.sample.event;

import java.lang.reflect.*;
import oracle.jsp.event.*;

public class PageEventDispatcher extends Object implements JspScopeListener {

    private Object page;
    private String methodName;
    private Method method;

    public PageEventDispatcher() {
    }

    public Object getPage() {
        return page;
    }

    public void setPage(Object page) {
        this.page = page;
    }

    public String getMethodName() {
        return methodName;
    }

    public void setMethodName(String m)
            throws NoSuchMethodException, ClassNotFoundException  {
        method = verifyMethod(m);
        methodName = m;
    }

    public void outOfScope(JspScopeEvent ae) {
        int scope = ae.getScope();

        if ((scope == javax.servlet.jsp.PageContext.REQUEST_SCOPE  ||
             scope == javax.servlet.jsp.PageContext.PAGE_SCOPE) &&
             method != null) {
            try {
                Object args[] = {ae.getApplication(), ae.getContainer()};
```

```
                method.invoke(page, args);
            } catch (Exception e) {
                // catch all and continue
            }
        }
    }

    private Method verifyMethod(String m)
                throws NoSuchMethodException, ClassNotFoundException {
        if (page == null) throw new NoSuchMethodException
                                      ("A page hasn't been set yet.");

        /* Don't know whether this is a request or page handler so try one then
          the other
       */
        Class c = page.getClass();
        Class pTypes[] = {Class.forName("javax.servlet.ServletContext"),
                          Class.forName("javax.servlet.jsp.PageContext")};

        try {
            return c.getDeclaredMethod(m, pTypes);
        } catch (NoSuchMethodException nsme) {
            // fall through and try the request signature
        }

        pTypes[1] = Class.forName("javax.servlet.http.HttpServletRequest");
        return c.getDeclaredMethod(m, pTypes);
    }
}
```

### scope.jsp Source

This JSP page uses the preceding `PageEventDispatcher` class (which implements the `JspScopeListener` interface) to track events of `page` or `request` scope.

```
<%-- declare request and page scoped beans here --%>

<jsp:useBean id = "includeCount" class = "oracle.jsp.jml.JmlNumber" scope = "request" />
<jsp:useBean id = "pageCount" class = "oracle.jsp.jml.JmlNumber" scope = "page" >
   <jsp:setProperty name = "pageCount"
                    property = "value" value = "<%= pageCount.getValue() + 1 %>" />
</jsp:useBean>
```

```
<%-- declare the event dispatchers --%>
<jsp:useBean id = "requestDispatcher" class = "oracle.jsp.sample.event.PageEventDispatcher"
            scope = "request" >
   <jsp:setProperty name = "requestDispatcher" property = "page" value = "<%= this %>" />
   <jsp:setProperty name = "requestDispatcher" property = "methodName"
                    value = "request_OnEnd" />
</jsp:useBean>

<jsp:useBean id = "pageDispatcher" class = "oracle.jsp.sample.event.PageEventDispatcher"
             scope = "page" >
    <jsp:setProperty name = "pageDispatcher" property = "page" value = "<%= this %>" />
    <jsp:setProperty name = "pageDispatcher" property = "methodName" value = "page_OnEnd" />
</jsp:useBean>

<%!
        // request_OnEnd Event Handler
        public void request_OnEnd(ServletContext application, HttpServletRequest request) {
                // acquire beans
                oracle.jsp.jml.JmlNumber includeCount =
                    (oracle.jsp.jml.JmlNumber) request.getAttribute("includeCount");

                // now cleanup the bean
                if (includeCount != null) application.log
                   ("request_OnEnd: Include count = " + includeCount.getValue());
        }

        // page_OnEnd Event Handler
        public void page_OnEnd(ServletContext application, PageContext page) {
                // acquire beans
                oracle.jsp.jml.JmlNumber pageCount =
                    (oracle.jsp.jml.JmlNumber) page.getAttribute("pageCount");

                // now cleanup the bean -- uncomment code for real bean
                if (pageCount != null) application.log
                   ("page_OnEnd: Page count = " + pageCount.getValue());
        }
%>

<%-- Page implementation goes here --%>

<jsp:setProperty name = "includeCount" property = "value"
                 value = '<%= (request.getAttribute("javax.servlet.include.request_uri")
                         != null) ? includeCount.getValue() + 1 : 0 %>' />

<h2> Hello World </h2>
```

```
Included: <%= request.getAttribute("javax.servlet.include.request_uri") %>
              Count: <%= includeCount.getValue() %> <br>

<% if (includeCount.getValue() < 5) { %>
        <jsp:include page="scope.jsp" flush = "true" />
<% } %>
```

## XML Query—XMLQuery.jsp

This example connects to a database, executes a query, and uses functionality of the `oracle.xml.sql.query.OracleXMLQuery` class to output the results as an XML string.

This is Oracle-specific functionality. The `OracleXMLQuery` class is provided with Oracle8*i* as part of the XML-SQL utility.

For general information about XML and XSL usage with JSP pages, see "OracleJSP Support for XML and XSL" on page 5-9.

```
<%-----------------------------------------------------------
   Copyright (c) 1999, Oracle Corporation. All rights reserved.
------------------------------------------------------------%>

<%@ page import = "java.sql.*,oracle.xml.sql.query.OracleXMLQuery" %>
<html>
  <head><title> The XMLQuery Demo </title></head>
<body>
<h1> XMLQuery Demo  </h1>
<h2> Employee List in XML </h2>
<b>(View Page Source in your browser to see XML output)</b>
<% Connection conn = null;
   Statement stmt = null;
   ResultSet rset = null;
   try {

     // determine JDBC driver name from session value
     // if null, use JDBC kprb driver if in JServer, JDBC oci otherwise
     String dbURL = (String)session.getValue("connStr");
     if (dbURL == null)
       dbURL = (System.getProperty("oracle.jserver.version") == null?
               "jdbc:oracle:oci8:@" : "jdbc:oracle:kprb:@");

     DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

```
     conn = DriverManager.getConnection(dbURL, "scott", "tiger");
     stmt =  conn.createStatement ();
     rset = stmt.executeQuery ("SELECT ename, sal " +
                              "FROM scott.emp ORDER BY ename");
     OracleXMLQuery xq = new OracleXMLQuery(conn, rset);  %>
       <PRE> <%=   xq.getXMLString()  %>  </PRE>
<%   } catch (java.sql.SQLException e) {  %>
         <P> SQL error: <PRE> <%= e %> </PRE> </P>
<%   } finally {
         if (stmt != null) stmt.close();
         if (rset != null) rset.close();
         if (conn != null) conn.close();
     }  %>
</body>
</html>
```

## SQLJ Queries—SQLJSelectInto.sqljsp and SQLJIterator.sqljsp

This section provides examples of using SQLJ in JSP pages to query a database.

The first example, `SQLJSelectInto.sqljsp`, selects a single row using SQLJ `SELECT INTO` syntax.

The second example, `SQLJIterator.sqljsp`, selects multiple rows into a SQLJ iterator, which is similar to a JDBC result set.

For information about using SQLJ in JSP pages, see "OracleJSP Support for Oracle SQLJ" on page 5-33.

For general information about Oracle SQLJ programming features and syntax, see the *Oracle8i SQLJ Developer's Guide and Reference*.

### Code for SQLJSelectInto.sqljsp (select single row)

This example selects a single row from the database, using SQLJ `SELECT INTO` syntax.

```
<%@ page import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>

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

<%
   String connStr=request.getParameter("connStr");
```

```
   if (connStr==null) {
     connStr=(String)session.getValue("connStr");
   } else {
     session.putValue("connStr",connStr);
   }
   if (connStr==null) { %>
<jsp:forward page="../setconn.jsp" />
<%
   }
%>

<%
   String empno = request.getParameter("empno");
   if (empno != null) { %>
      <H3> Employee # <%=empno %> Details: </H3>
      <%= runQuery(connStr,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 connStr, 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(connStr, "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();
   }
%>
```

### Code for SQLJIterator.sqljsp (select multiple rows)

This example selects multiple rows from the database, using a SQLJ iterator.

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

<!--------------------------------------------------------------------
 * This is a SQLJ JavaServer Page that does a SQLJ query on the
 * emp table in schema scott and outputs the result in an html table.
 *
--------------------------------------------------------------------!>

<%! #sql iterator Empiter(String ename, double sal, java.sql.Date hiredate) %>

<%
   String connStr=request.getParameter("connStr");
   if (connStr==null) {
     connStr=(String)session.getValue("connStr");
   } else {
     session.putValue("connStr",connStr);
   }
   if (connStr==null) { %>
<jsp:forward page="../setconn.jsp" />
<%
   }
%>

<%
   DefaultContext dctx = null;
   dctx = Oracle.getConnection(connStr, "scott", "tiger");
%>

<HTML>
<HEAD> <TITLE> The SqljIterator SQLJSP </TITLE>  </HEAD>
<BODY BGCOLOR="E0FFF0">
```

```
 <% String user;
     #sql [dctx] {SELECT user INTO :user FROM dual};
 %>

 <H1> Hello, <%= user %>!   </H1>
 <HR>
 <B> I will use a SQLJ iterator to get employee data
     from EMP table in schema SCOTT..
 </B>
 <P>
<%
    Empiter emps;
    try {
      #sql [dctx] emps = { SELECT ename, sal, hiredate
                 FROM scott.emp ORDER BY ename};
      if (emps.next()) {
%>
      <TABLE BORDER=1 BGCOLOR="C0C0C0">
      <TH WIDTH=200 BGCOLOR=white> Employee Name </TH>
      <TH WIDTH=100 BGCOLOR=white> Salary </TH>
      <TR> <TD> <%= emps.ename() %> </TD>
           <TD> <%= emps.sal() %> </TD>
      </TR>

<%      while (emps.next()) {
%>
      <TR> <TD> <%= emps.ename() %> </TD>
           <TD> <%= emps.sal() %> </TD>
      </TR>
<% } %>
      </TABLE>
<%  } else { %>
      <P> Sorry, the query returned no rows! </P>
<%    }
      emps.close();
    } catch (SQLException e) {  %>
       <P>There was an error doing the query:<PRE> <%= e %> </PRE> <P>
<%    } %>
</BODY>
</HTML>
```

# 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:

- globals.jsa Example for Application Events—lotto.jsp
- globals.jsa Example for Application and Session Events—index1.jsp
- globals.jsa Example for Global Declarations—index2.jsp

For information about `globals.jsa` usage, see "OracleJSP Application and Session Support for Servlet 2.0" on page 5-37.

---

**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=false`) by updating either the `lotto.jsp` source or the `globals.jsa` file. (The OracleJSP container always terminates a running application before retranslating and reloading an active page.)

---

## globals.jsa Example for Application Events—lotto.jsp

This sample illustrates OracleJSP `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, a user is identified by their IP address.

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();
        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>

<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;
                application.setAttribute("today", today);
        }

        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>
```

## 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, as these values are maintained at `application` scope.
- The count values use the OracleJSP `oracle.jsp.jml.JmlNumber` extended datatype. This is a built-in bean that simplifies the use of data values at application scope. (For information about the JML extended datatypes, see "JML Extended Datatypes" on page 5-2.)

### 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>
```

## 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 "globals.jsa Example for Application and Session Events—index1.jsp" on page 9-44, 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 had to provide `jsp:useBean` statements to locally declare the beans they were accessing.)

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>
```

# A

# General Installation and Configuration

This appendix provides general information about installing OracleJSP, configuring the Web server to run OracleJSP, and configuring OracleJSP. The technical information focuses on common Web servers and servlet environments: Apache/JServ, the Sun Microsystems JavaServer Web Developer's Kit (JSWDK), and Tomcat (from Apache, in cooperation with Sun Microsystems). For Oracle environments that support OracleJSP, reference is made to documentation for those products for installation and configuration instructions.

For the Oracle Servlet Engine, translation-time configuration is handled through options of the OracleJSP pre-translation utility, as described in "The ojspc Pre-Translation Tool" on page 6-23.

This appendix includes the following topics:

## System Requirements

OracleJSP is a pure Java environment. The system on which you install it must meet the following minimum requirements.

| | |
|---|---|
| operating system: | any OS that supports JDK 1.1.x or higher |
| Java version: | JDK 1.1.x or 1.2.x<br>(Oracle recommends the most current version available for your platform, preferably JDK 1.1.8 or higher.) |
| Java compiler: | the standard `javac` provided with your JDK<br>(You can, however, use alternative compilers instead.) |
| Web server: | any Web server that supports servlets |
| servlet environment: | any servlet container implementing the servlet 2.0 specification or higher |

---

**Note:** The servlet library for your servlet environment must be installed on your system and included in the classpath in your Web server configuration file. This library contains the `javax.servlet.*` packages.

For example, in an Apache/JServ environment (including Oracle Internet Application Server), you will need `jsdk.jar`, which is provided with the Sun Microsystems JSDK 2.0. In a Sun Microsystems JSWDK environment, you will need `servlet.jar` (servlet 2.1 version), which is provided with JSWDK 1.0. In a Tomcat environment, you will need `servlet.jar` (servlet 2.2 version), which is provided with Tomcat 3.1. Do not confuse JSDK (Java Servlet Developer's Kit) with JSWDK (JavaServer Web Developer's Kit).

See "Configuration of Web Server and Servlet Environment to Run OracleJSP" on page A-7 for further discussion of classpath settings in a Web server configuration file.

---

# OracleJSP Installation and Web Server Configuration

This section discusses OracleJSP installation and related Web server configuration for various JSP environments. The following environments are considered:

- Apache/JServ
- Sun Microsystems JSWDK
- Tomcat
- Oracle Servlet Engine (OSE)
- other Oracle environments

This discussion assumes that your target system, which might be either a development environment or a deployment environment, meets the requirements specified in "System Requirements" on page A-2. It also assumes that you have verified you can do the following:

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

---

**Note:**

- Examples here are for a UNIX environment, but the basic information (such as directory names and file names) applies to other environments as well.
- Web server configuration information focuses on prevalent non-Oracle environments. For Oracle environments, refer to documentation for the particular product (such as Oracle Internet Application Server or Web-to-go).

---

## Required and Optional Files for OracleJSP

This section summarizes JAR and ZIP files required for OracleJSP, as well as optional JAR and ZIP files to use Oracle JDBC and SQLJ functionality, JML or SQL custom tags, or custom data-access JavaBeans. The summary of files is followed by a discussion of how to install OracleJSP files on non-Oracle environments, and a list of Oracle environments that already provide OracleJSP.

Required files must also be added to your classpath. (See "Add OracleJSP-Related JAR and ZIP Files to Web Server Classpath" on page A-8.)

### Summary of Files

**Note:** Refer to the *Oracle8i Java Developer's Guide* for the locations of these files on the Oracle8*i* product CD.

The following files are provided with OracleJSP and must be installed on your system:

- `ojsp.jar` (OracleJSP)
- `xmlparserv2.jar` (for XML parsing—required for the `web.xml` deployment descriptor and any tag library descriptors)
- `servlet.jar` (standard servlet library, servlet 2.2 version)

In addition, if your JSP pages will use Oracle JSP Markup Language (JML) tags, SQL tags, or database-access JavaBeans, you will need the following files:

- `ojsputil.jar`
- `xsu12.jar` (for JDK 1.2.x) or `xsu111.jar` (for JDK 1.1.x) (in OSE, or for XML functionality on the client)

To run in the Oracle Servlet Engine, `xsu12.jar` or `xsu111.jar` must be installed prior to or simultaneously with `ojsputil.jar`. (This should be handled automatically in a normal Oracle8*i* installation.) To run in a client environment, however, `xsu12.jar` or `xsu111.jar` is required only if you will use XML functionality in the database-access JavaBeans (such as getting a result set as an XML string). The `xsu12.jar` and `xsu111.jar` files are included with Oracle8*i* release 8.1.7.

**Servlet Library Notes** Note that OracleJSP 8.1.7 requires and supplies the 2.2 version of the servlet library, which is where the standard `javax.servlet.*` packages are located. Your Web server environment likely requires and supplies a different servlet library version. You must be careful in your classpath to have the version for your Web server precede the version for OracleJSP. "Add OracleJSP-Related JAR and ZIP Files to Web Server Classpath" on page A-8 further discusses this.

Table A–1 summarizes the servlet library versions. Do not confuse the Sun Microsystems JSWDK (JavaServer Web Developer's Kit) with the Sun Microsystems JSDK (Java Servlet Developer's Kit).

***Table A–1 Servlet Library Versions***

| Servlet Library Version | Library File Name | Provided with: |
|---|---|---|
| servlet 2.2 | `servlet.jar` | OracleJSP, Tomcat 3.1 |
| servlet 2.1 | `servlet.jar` | Sun JSWDK 1.0 |
| servlet 2.0 | `jsdk.jar` | Sun JSDK 2.0; also used with Apache/JServ |

(For Apache/JServ, `jsdk.jar` must be downloaded separately.)

The remainder of this section discusses files that are necessary only if you use certain optional or extended features.

**Files for JDBC (optional)** The following files are required if you will use Oracle JDBC. (Be aware that Oracle SQLJ uses Oracle JDBC.)

- `classes12.zip` (for JDK 1.2.x environments)

or:

- `classes111.zip` (for JDK 1.1.x environments)

**Files for SQLJ (optional)** The following files are necessary if your JSP pages use Oracle SQLJ for their Oracle8*i* access:

- `translator.zip` (for the SQLJ translator, for JDK 1.2.x or 1.1.x)

as well as the appropriate SQLJ runtime:

- `runtime12.zip` (for JDK 1.2.x with Oracle JDBC 8.1.7)

or:

- `runtime12ee.zip` (for JDK 1.2.x enterprise edition with Oracle JDBC 8.1.7)

or:

- `runtime11.zip` (for JDK 1.1.x with Oracle JDBC 8.1.7)

or:

- `runtime.zip` (generic: for JDK 1.2.x or 1.1.x with any Oracle JDBC version)

(The JDK 1.2.x enterprise edition allows datasource support, in compliance with the SQLJ ISO specification.)

## File Installation for Non-Oracle Environments

To run OracleJSP in a non-Oracle environment—typically Apache/JServ, the Sun Microsystems JSWDK, or Tomcat—download the OracleJSP files from the Oracle Technology Network (OTN) at the following URL:

```
http://technet.oracle.com/tech/java/servlets/index.htm
```

Click on "Software" in the button bar near the top of this page.

You will need an OTN account, but membership is free of charge. Click on "Membership" in the bar at the top of the page if you do not already have an account.

For the OTN download, OracleJSP files are contained within `ojsp.zip`, which includes files mentioned in this section as well as configuration files discussed later in this appendix, release notes, documentation files, and sample applications.

Installation and configuration instructions are included in `ojsp.zip`—see `install.htm` for top-level information and links. However, you can use this appendix for detailed information about configuring the predominant non-Oracle Web server environments—Apache/JServ, the Sun Microsystems JSWDK, and Tomcat—to use OracleJSP.

You can choose any desired root directory for OracleJSP, as long as the location you choose is reflected in your Web server classpath settings (discussed in "Add OracleJSP-Related JAR and ZIP Files to Web Server Classpath" on page A-8).

Oracle JDBC and SQLJ files are also available from OTN separately at the following URL:

```
http://technet.oracle.com/tech/java/sqlj_jdbc/index.htm
```

Click on "Software" in the button bar near the top of this page.

> **Note:** Oracle Internet Application Server uses an Apache/JServ environment, but you should use application server installation and configuration instructions instead of the Apache/JServ instructions in this appendix.

### Oracle Environments Providing OracleJSP

The following Oracle environments provide OracleJSP and a Web server or Web listener, starting with the version numbers noted:

- Oracle Servlet Engine (OSE) 8.1.7
- Oracle Internet Application Server 1.0.0
- Oracle Application Server 4.0.8.2
- Oracle Web-to-go 1.3 (can be used with Oracle8*i* Lite)
- Oracle JDeveloper 3.0

In any of these environments, OracleJSP components are included with the product installation.

If you are targeting OSE you will need a client-side development and testing environment—probably Oracle JDeveloper or perhaps a non-Oracle development tool. When you have completed preliminary testing in your development environment, you can deploy JSP pages to the Oracle8*i* database, as described in Chapter 6, "JSP Translation and Deployment".

## Configuration of Web Server and Servlet Environment to Run OracleJSP

Configuring your Web server to run OracleJSP pages requires the following general steps:

1. Add OracleJSP-related JAR and ZIP files to the Web server classpath.
2. Configure the Web server to map JSP file name extensions (`.jsp` and `.JSP` and, optionally, `.sqljsp` and `.SQLJSP`) to the Oracle `JspServlet`, which is the front-end of the OracleJSP container.

These steps apply to any Web server environment, but the information in this section focuses on the most prevalent non-Oracle environments— Apache/JServ, the Sun Microsystems JSWDK, and Tomcat.

The Oracle Servlet Engine, provided with the Oracle8*i* JServer environment, is automatically configured upon installation to run OracleJSP. For other Oracle environments, refer to the documentation for those products, because procedures vary. (Much of the installation and configuration may be automatic.)

### Add OracleJSP-Related JAR and ZIP Files to Web Server Classpath

You must update the Web server classpath to add JAR and ZIP files that are required by OracleJSP, and in the proper order. (In particular, you must be careful as to where you place the servlet 2.2 version of `servlet.jar` in the classpath, as described below.) This includes the following:

- `ojsp.jar`
- `xmlparserv2.jar`
- `servlet.jar` (servlet 2.2 version)

  (Note that the `servlet.jar` supplied with OracleJSP is identical to the `servlet.jar` provided with Tomcat 3.1.)
- `ojsputil.jar` (optional, for JML tags, SQL tags, and database-access JavaBeans)
- `xsu12.jar` (for JDK 1.2.x) or `xsu111.jar` (for JDK 1.1.x) (optional, for JML tags, SQL tags, and database-access JavaBeans; see "Summary of Files" on page A-4)
- additional optional ZIP and JAR files, as necessary, for JDBC and SQLJ (see "Summary of Files" on page A-4)

---

**Important:** You must also ensure that OracleJSP can find `javac` (or an alternative Java compiler, according to your `javaccmd` configuration parameter setting). For `javac` in a JDK 1.1.x environment, the JDK `classes.zip` file must be in the Web server classpath. For `javac` in a JDK 1.2.x environment, the JDK `tools.jar` file must be in the Web server classpath.

---

**Apache/JServ Environment** In an Apache/JServ environment, add appropriate `wrapper.classpath` commands to 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 Apache/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 OracleJSP, JDBC, and SQLJ. (Replace `[Oracle_Home]` with your Oracle Home path.)

```
# servlet 2.0 APIs (required by Apache/JServ, from Sun JSDK 2.0):
wrapper.classpath=jsdk2.0/lib/jsdk.jar
#
# servlet 2.2 APIs (required and provided by OracleJSP):
wrapper.classpath=[Oracle_Home]/ojsp/lib/servlet.jar
# OracleJSP 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
```

**Important:** In an Apache/JServ environment, `jsdk.jar` must precede `servlet.jar` in the classpath.

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
```

**JSWDK Environment** Update the `startserver` script in the `jswdk-1.0` root directory to add OracleJSP files to the `jspJars` environment variable. Append them to the last `.jar` file listed, using the appropriate directory syntax and separator character for your operating system, such as a colon (`:`) for UNIX or a semi-colon (`;`) for Windows NT. Here is an example:

```
jspJars=./lib/jspengine.jar:./lib/ojsp.jar:./lib/ojsputil.jar
```

This example (with UNIX syntax) assumes that the JAR files are in the `lib` subdirectory under the `jswdk-1.0` root directory.

Similarly, update the `startserver` script to specify any additional required files in the `miscJars` environment variable, such as in the following example:

```
miscJars=./lib/xml.jar:./lib/xmlparserv2.jar:./lib/servlet.jar
```

This example (with UNIX syntax) also assumes that the files are in the `lib` subdirectory under the `jswdk-1.0` root directory.

---

**Important:** In a JSWDK environment, the servlet 2.1 version of `servlet.jar` (provided with Sun JSWDK 1.0) must precede the servlet 2.2 version of `servlet.jar` (provided with OracleJSP) in your classpath.

The servlet 2.1 version is typically in the `jsdkJars` environment variable. The overall classpath is formed through a combination of various `xxxJars` environment variables, including `jsdkJars`, `jspJars`, and `miscJars`. Examine the `startserver` script to verify that `miscJars` is added to the classpath *after* `jsdkJars`.

---

**Tomcat Environment** For Tomcat, the procedure for adding files to the classpath is more operating-system dependent than for the other servlet environments discussed here.

For a UNIX operating system, copy the OracleJSP JAR and ZIP files to your `[TOMCAT_HOME]/lib` directory. This directory is automatically included in the Tomcat classpath.

For a Windows NT operating system, update the `tomcat.bat` file in the `[TOMCAT_HOME]\bin` directory to individually add each OracleJSP file to the `CLASSPATH` environment variable. The following example presumes that you have copied the files to the `[TOMCAT_HOME]\lib` directory:

```
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\ojsp.jar;%TOMCAT_HOME%\lib\ojsputil.jar
```

The servlet 2.2 version of `servlet.jar` (the same version that is provided with OracleJSP) is already included with Tomcat, so it needs no consideration.

### Map JSP File Name Extensions to Oracle JspServlet

You must configure the Web server for the following:

- to recognize appropriate file name extensions as JSP pages

  Map `.jsp` and `.JSP`. Also map `.sqljsp` and `.SQLJSP` if your JSP pages use Oracle SQLJ.

- to find and execute the servlet that begins processing JSP pages

  In OracleJSP, this is `oracle.jsp.JspServlet`, which you can think of as the front-end of the Oracle JSP container.

> **Important:** With the above configurations, OracleJSP will support page references that use either a `.jsp` file name extension or a `.JSP` file name extension, but the case in the reference must match the actual file name in a case-sensitive environment. If the file name is `file.jsp`, you can reference it that way, but not as `file.JSP`. If the file name is `file.JSP`, you can reference it that way, but not as `file.jsp`. (The same holds true for `.sqljsp` versus `.SQLJSP`.)

**Apache/JServ Environment** In an Apache/JServ environment, mapping each JSP file name extension to the Oracle `JspServlet` requires just a single step. In the JServ `conf` directory, update the configuration file—`jserv.conf` or `mod_jserv.conf`—to add `ApJServAction` commands to perform the mappings.

(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 Apache/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 an Apache/JServ term that is similar conceptually to "servlet context".) Consult your Apache/JServ documentation for more information.

**JSWDK Environment** In a JSWDK environment, mapping each JSP file name extension to the Oracle `JspServlet` requires two steps.

The first step is to update the `mappings.properties` file in the `WEB-INF` directory *of each servlet context* to define JSP file name extensions. Do this with the following commands:

```
# Map JSP file name extensions (.sqljsp and .SQLJSP are optional).
.jsp=jsp
.JSP=jsp
.sqljsp=jsp
.SQLJSP=jsp
```

The second step is to update the `servlet.properties` file in the `WEB-INF` directory *of each servlet context* to define the Oracle `JspServlet` as the servlet that begins JSP processing. In addition, be sure to comment out the previously defined mapping for the JSP reference implementation. Do this as follows:

```
#jsp.code=com.sun.jsp.runtime.JspServlet (replacing this with Oracle)
jsp.code=oracle.jsp.JspServlet
```

**Tomcat Environment** In a Tomcat environment, mapping each JSP file name extension to the Oracle `JspServlet` requires a single step. Update the servlet mapping section of the `web.xml` file as shown below.

> **Note:** There is a global `web.xml` file in the `[TOMCAT_HOME]/conf` directory. To override any settings in this file for a particular application, update the `web.xml` file in the `WEB-INF` directory under the particular application root.

```
# Map file name extensions (.sqljsp and .SQLJSP are optional).

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

```
</servlet-mapping>

<servlet-mapping>
   <servlet-name>
      oracle.jsp.JspServlet
   </servlet-name>
   <url-pattern>
      *.JSP
   </url-pattern>
</servlet-mapping>

<servlet-mapping>
   <servlet-name>
      oracle.jsp.JspServlet
   </servlet-name>
   <url-pattern>
      *.sqljsp
   </url-pattern>
</servlet-mapping>

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

You can optionally set an alias for the `oracle.jsp.JspServlet` class name, as follows:

```
<servlet>
   <servlet-name>
      ojsp
   </servlet-name>
   <servlet-class>
      oracle.jsp.JspServlet
   </servlet-class>
   ...
</servlet>
```

Setting this alias allows you to use "ojsp" instead of the class name for your other settings, as follows:

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

# OracleJSP Configuration

The OracleJSP front-end servlet, `JspServlet`, supports a number of configuration parameters to control OracleJSP operation. They are set as servlet initialization parameters for `JspServlet`. How you accomplish this depends on the Web server and servlet environment you are using.

This section describes the OracleJSP configuration parameters and how to set them in the most prevalent Web server and servlet environments.

Only a limited number of these parameters are of interest in the Oracle products that supply OracleJSP, and how to set them may vary from product to product. Consult the product documentation for more information.

Configuration settings that apply to the Oracle Servlet Engine are typically supported as equivalent options in the OracleJSP pre-translation tool (`ojspc`). OSE does not employ the Oracle `JspServlet` in translating or running JSP pages.

## OracleJSP Configuration Parameters (Non-OSE)

This section describes the configuration parameters supported by the Oracle `JspServlet` for environments such as Apache/JServ, the Sun Microsystems JSWDK, or Tomcat. (Note that the Oracle Internet Application Server uses an Apache/JServ environment.)

For the Oracle Servlet Engine environment, some of the equivalent configuration functionality is supported through equivalent `ojspc` options.

### Configuration Parameters Summary Table

Table A–2 summarizes the configuration parameters supported by Oracle `JspServlet` (the front-end of the OracleJSP container). For each parameter, the table notes the following:

- whether it is used during page translation or page execution
- whether it is typically of interest in a development environment, deployment environment, or both
- any equivalent `ojspc` translation options for pages that are targeted for the Oracle Servlet Engine (which does not use `JspServlet`)

  OSE does not support execution-time configuration parameters.

Be aware of the following:

- The parameters `bypass_source`, `emit_debuginfo`, `external_resource`, `javaccmd`, and `sqljcmd` were not supported prior to OracleJSP release 1.1.0.0.0.
- The parameter `alias_translation` is for use in the Apache/JServ environment only.
- The parameter `session_sharing` is for use with `globals.jsa` only (presumably in a servlet 2.0 environment such as Apache/JServ).

---

**Notes:**

- See "The ojspc Pre-Translation Tool" on page 6-23 for a description of the `ojspc` options.
- The distinction between execution-time and translation-time is not particularly significant in a real-time translation environment, but may be of interest with respect to OSE.

---

***Table A–2 OracleJSP Configuration Parameters***

| Parameter | Related ojspc Options | Description | Default | Used in JSP Translation or Execution? | Used in Development or Deployment Environment? |
|---|---|---|---|---|---|
| alias_translation **(Apache-specific)** | n/a | boolean; `true` to work around Apache/JServ limitations in directory aliasing for JSP page references | false | execution | development and deployment |
| bypass_source | n/a | boolean; `true` for OracleJSP to ignore `FileNotFound` exceptions on `.jsp` source; uses pre-translated and compiled code when source is not available | false | execution | deployment (also used by JDeveloper) |
| classpath | -addclasspath (related, but different functionality) | additional classpath entries for OracleJSP class loading | null (no addl. path) | translation or execution | development and deployment |

***Table A–2 OracleJSP Configuration Parameters (Cont.)***

| Parameter | Related ojspc Options | Description | Default | Used in JSP Translation or Execution? | Used in Development or Deployment Environment? |
|---|---|---|---|---|---|
| developer_mode | n/a | boolean; `false` to *not* check timestamps to see if retranslation is necessary when a page is requested | true | execution | development and deployment |
| emit_debuginfo | -debug | boolean; `true` to generate a line map to the original `.jsp` file for debugging | false | translation | development |
| external_resource | -extres | boolean; `true` for OracleJSP to place all static content of the page into a separate Java resource file during translation | false | translation | development and deployment |
| javaccmd | -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 | translation | development and deployment |
| page_repository_root | -srcdir<br>-d | alternative root directory (fully qualified path) for OracleJSP to use in loading and generating JSP pages | null (use default root) | translation or execution | development and deployment |
| session_sharing **(for use with globals.jsa)** | n/a | boolean; `true` for JSP session data to be propagated to the underlying servlet session for applications using `globals.jsa` | true | execution | development and deployment |

***Table A–2 OracleJSP Configuration Parameters (Cont.)***

| Parameter | Related ojspc Options | Description | Default | Used in JSP Translation or Execution? | Used in Development or Deployment Environment? |
|---|---|---|---|---|---|
| sqljcmd | n/a | SQLJ command line—`sqlj` options, or alternative SQLJ translator run in a separate JVM (`null` means use the Oracle SQLJ version provided with OracleJSP, with default option settings) | null | translation | development and deployment |
| translate_params | n/a | boolean; `true` to override servlet containers that do not perform multibyte encoding | false | execution | development and deployment |
| unsafe_reload | n/a | boolean; `true` to *not* restart the application and sessions whenever a JSP page is retranslated and reloaded | false | execution | development |

## Configuration Parameter Descriptions

This section describes the OracleJSP configuration parameters in more detail.

**alias_translation** (boolean; OracleJSP default: `false`) **(Apache-specific)**

This parameter allows OracleJSP to work around limitations in the way Apache/JServ handles directory aliasing. For information about the current limitations, see "Directory Alias Translation" on page 4-38.

You must set `alias_translation` to `true` for `httpd.conf` directory aliasing commands (such as the following example) to work properly in the Apache/JServ servlet environment:

```
Alias /icons/ "/apache/apache139/icons/"
```

**bypass_source** (boolean; OracleJSP default: `false`)

Normally, when a JSP page is requested, OracleJSP will throw 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, OracleJSP checks the page source to see if it has been modified since the page implementation class was generated.)

Set this parameter to `true` for OracleJSP to proceed and execute the page implementation class even if it cannot find the page source.

If `bypass_source` is enabled, OracleJSP will still check 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 6-69.)
- Oracle 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; OracleJSP default: `null`)

Use this parameter to add classpath entries to the OracleJSP default classpath for use during translation, compilation, or execution of JSP pages. For information about the OracleJSP classpath and class loader, see "Classpath and Class Loader Issues (Non-OSE Only)" on page 4-21.

The exact syntax depends on your Web server environment and operating system. See "OracleJSP Configuration Parameter Settings" on page A-25 for some examples.

Overall, OracleJSP 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 loaded through the `classpath`-specified path are loaded by the JSP class loader, not the system class loader. During JSP execution, classes loaded by the JSP class loader cannot access (or be accessed by) classes loaded by the system class loader or any other class loader.

**Notes:**

- OracleJSP runtime automatic class reloading applies only to classes in the OracleJSP classpath. This includes paths specified through this `classpath` parameter. (See "Dynamic Class Reloading" on page 4-26 for information about this feature.)
- When you are pre-translating pages to run in the Oracle Servlet Engine, the `ojspc -addclasspath` option offers some related, though different, functionality. See "Option Descriptions for ojspc" on page 6-27.

**developer_mode** (boolean; OracleJSP default: `true`)

Set this flag to `false` to instruct OracleJSP to *not* routinely compare the timestamp of the page implementation class to the timestamp of the `.jsp` source file when a page is requested. Ordinarily, OracleJSP checks every time to see if the source has been modified since the page implementation class was generated. If that is the case, OracleJSP retranslates the page. With `developer_mode=false`, OracleJSP 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.

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.

**emit_debuginfo** (boolean; OracleJSP default: `false`) **(for developer only)**

Set this flag to true to instruct OracleJSP to generate a line map to the original `.jsp` file for debugging. Otherwise, lines will be mapped to the generated page implementation class.

**Notes:**

- Oracle JDeveloper enables `emit_debuginfo`.
- When you are pre-translating pages to run in the Oracle Servlet Engine, the `ojspc -debug` option is equivalent. See "Option Descriptions for ojspc" on page 6-27.

**external_resource** (boolean; OracleJSP default: `false`)

Set this flag to true to instruct the OracleJSP 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. For release 8.1.7, it will be the same name as the JSP name, but with the `.res` suffix. (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 resource file is placed in 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. In extreme cases, it may even prevent the service method from exceeding the 64K method size limit imposed by the Java VM. For more information, see "Workarounds for Large Static Content in JSP Pages" on page 4-13.

> **Note:** When you are pre-translating pages to run in the Oracle Servlet Engine, the `ojspc -extres` option is equivalent.
>
> The `ojspc -hotload` option is also relevant, performing the `-extres` functionality along with additional steps to allow hotloading into Oracle8*i*. See "Option Descriptions for ojspc" on page 6-27.

**javaccmd** (compiler executable; OracleJSP 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)

Specifying an alternative compiler results in OracleJSP spawning that executable as a separate process in a separate JVM, instead of spawning the JDK default compiler within OracleJSP's JVM. You can fully specify the path for the executable, or specify only the executable and let OracleJSP look for it in the system path.

The following `javaccmd` setting enables the `javac -verbose` option:

```
javaccmd=javac -verbose
```

The exact syntax depends on your servlet environment. See "OracleJSP Configuration Parameter Settings" on page A-25.

---

**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.
- When you are pre-translating pages to run in the Oracle Servlet Engine, the `ojspc -noCompile` option allows similar functionality. It results in no compilation by `javac`, so you can compile the translated classes manually using your desired compiler. See "Option Descriptions for ojspc" on page 6-27.

---

**page_repository_root** (fully qualified directory path; OracleJSP default: `null`)

OracleJSP uses the Web server document repository to generate or load translated JSP pages. By default, in an on-demand translation scenario, the root directory is the Web server doc root directory (for Apache/JServ) or the servlet context root directory of the application the page belongs to. JSP page source is in the root directory or some subdirectory. Generated files are written to a `_pages` subdirectory or some corresponding subdirectory.

Set the `page_repository_root` option to instruct OracleJSP to use a different root directory.

For information about file locations relative to the root directory and `_pages` subdirectory, see "OracleJSP Translator Output File Locations" on page 6-7.

---

**Notes:**

- The specified directory, `_pages` subdirectory, and any appropriate subdirectories under these are created automatically if they do not already exist.
- When you are pre-translating pages to run in the Oracle Servlet Engine, the `ojspc` options `-srcdir` and `-d` provide related functionality. See "Option Descriptions for ojspc" on page 6-27.

---

**session_sharing** (boolean; OracleJSP default: `true`) **(for use with globals.jsa)**

When a `globals.jsa` file is used 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 default `true` 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, an application's 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 "Overview of globals.jsa Functionality" on page 5-37.

**sqljcmd** (SQLJ translator executable; OracleJSP default: `null`)

This parameter is useful in any of the following circumstances:

- if you want to set SQLJ command-line options
- if you want to use a different SQLJ translator (or at least a different version) than the one provided with OracleJSP
- if you want to run SQLJ in a separate process from OracleJSP

Specifying a SQLJ translator executable results in OracleJSP spawning that executable as a separate process in a separate JVM, instead of spawning the default SQLJ translator within OracleJSP's JVM.

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

Following is an example of a `sqljcmd` setting:

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

(The exact syntax depends on your servlet environment. See "OracleJSP Configuration Parameter Settings" on page A-25.)

**Notes:**

- Appropriate SQLJ files 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, `translator.zip` and the appropriate SQLJ runtime ZIP file must be in the classpath. See "Summary of Files" on page A-4.)
- Presumably the great majority of OracleJSP developers will use Oracle SQLJ (as opposed to some other SQLJ product) if they use SQLJ code in their JSP pages; however, this option is useful if you want to use a different Oracle SQLJ version (for example, one intended for use with Oracle JDBC 8.0.x / 7.3.x drivers instead of Oracle8*i* drivers) or if you want to set SQLJ options.

**translate_params** (boolean; OracleJSP default: `false`)

Set this flag to `true` to override servlet containers that do not encode multibyte (NLS) request parameters or bean property settings. With this setting, OracleJSP encodes request parameters and bean property settings. Otherwise, OracleJSP returns the parameters from the servlet container unchanged.

Because the Oracle Servlet Engine does not support execution-time configuration parameters, `translate_params` cannot be set for the OSE environment. See "Code Equivalent to the translate_params Configuration Parameter" on page 8-6 for a workaround.

For more information about the functionality and use of `translate_params`, including situations where it should not be used, see "OracleJSP Extended Support for Multibyte Parameter Encoding" on page 8-5.

**unsafe_reload** (boolean; OracleJSP default: `false`) **(for developer only)**

By default, OracleJSP restarts the application and sessions whenever a JSP page is dynamically retranslated and reloaded (which occurs when the JSP translator finds a `.jsp` source file with a more recent timestamp than the corresponding page implementation class).

Set this parameter to `true` to instruct OracleJSP *not* to restart the application after dynamic retranslations and reloads. This avoids having existing sessions become invalid.

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 OracleJSP never retranslates after the initial request).

**Important:** This parameter is intended for developers only and is *not* recommended for deployment environments.

## OracleJSP Configuration Parameter Settings

How to set the JSP configuration parameters discussed in the preceding section ("OracleJSP Configuration Parameters (Non-OSE)" on page A-15) depends on your Web server and servlet environment.

Non-Oracle environments support configuration parameter settings through properties files or similar functionality.

The Oracle Servlet Engine, provided with the Oracle8*i* JServer environment, does not directly support OracleJSP configuration parameters (because it does not use `JspServlet`). However, some of the translation parameter settings have equivalent OracleJSP translator options. These options are noted in the "Configuration Parameters Summary Table" on page A-15.

Other Oracle products that support OracleJSP have their own mechanisms for configuration settings—consult the product documentation.

The remainder of this section describes how to set configuration parameters in the Apache/JServ, Sun Microsystems JSWDK, and Tomcat environments.

### Setting OracleJSP Parameters in Apache/JServ

Each Web application in an Apache/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 OracleJSP configuration parameters in an Apache/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
```

(The preceding is a single wrap-around line.)

The servlet path, `servlet.oracle.jsp.JspServlet`, also depends on how you mount the zone. It does not represent a literal directory path.

---

**Note:** 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.

---

### Setting OracleJSP Parameters in JSWDK

To set OracleJSP configuration parameters in a JSWDK environment, set the `jsp.initparams` property in the `servlet.properties` file in the `WEB-INF` directory of the application's servlet context, as in the following example (which happens to use UNIX syntax):

```
jsp.initparams=developer_mode=false,classpath=/mydir/myapp.jar
```

---

**Note:** Because `initparams` parameters are comma-separated, there can be no commas within a parameter setting. Spaces and other special characters do not cause a problem, however.

---

### Setting OracleJSP Parameters in Tomcat

To set OracleJSP configuration parameters in a Tomcat environment, add `init-param` entries in the `web.xml` file as shown below.

---

**Note:** There is a global `web.xml` file in the `[TOMCAT_HOME]/conf` directory. To override any settings in this file for a particular application, update the `web.xml` file in the `WEB-INF` directory under the particular application root.

---

```
<servlet>
   <init-param>
      <param-name>
         developer_mode
      </param-name>
      <param-value>
         true
```

```
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         external_resource
      </param-name>
      <param-value>
         true
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         javaccmd
      </param-name>
      <param-value>
         javac -verbose
      </param-value>
   </init-param>
</servlet>
```

### Oracle Servlet Engine JSP Configuration

Because the Oracle Servlet Engine does not use the OracleJSP `JspServlet` front-end, it requires other mechanisms for OracleJSP configuration settings.

Appropriate translation-time configuration parameters have equivalent support through command-line options of `ojspc`, which is the utility to pre-translate JSP pages for the OSE environment. The correlation between OracleJSP configuration parameters and `ojspc` options is noted in the table in "Configuration Parameters Summary Table" on page A-15.

There is no such equivalent support for runtime configuration parameters, however. The most significant of these is `translate_params`, required for NLS use in servlet environments that do not support multibyte encoding of request parameters. The Oracle Servlet Engine requires this functionality, but it is left to the developer to write equivalent code in the JSP page. For details, see "Code Equivalent to the translate_params Configuration Parameter" on page 8-6.

# B

# 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.

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

# 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* for more information about the concepts discussed here.

For more information about the methods this section discusses, refer to Sun Microsystems Javadoc at the following location:

`http://java.sun.com/products/servlet/2.2/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 is implemented indirectly by subclassing 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(...)`, which the servlet uses 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:

- creates an instance of the servlet and calls its `init()` method to initialize it
- calls the `service()` method of the servlet
- 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 object 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 binds the specified object to the session, under the specified name.

- `public Object getAttribute(String name)`

  This 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 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 Java virtual machine (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 Java virtual machine. 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 earlier 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 2.1* (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.

## 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 the application in any given JVM, providing information from the server to the servlets and JSP pages that comprise 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 B-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 used in initializing and terminating instances of the generated class:

- `jspInit()`
- `jspDestroy()`

Any code for these methods must be included in scriptlets in your JSP page, as in the following example:

```
<%!
   void jspInit()
   {
      ...your implementation code...
   }
%>
```

(JSP syntax is described later in this chapter. See "Scripting Elements" on page 1-12.)

`HttpJspPage` adds specification for the following method:

- `_jspService()`

Code for this method is typically generated automatically by the translator and includes code from scriptlets in the JSP page, code resulting from any JSP directives, and any static content of the page. (JSP directives are used to provide information for the page, such as specifying the Java language for scriptlets and providing package imports. See "Directives" on page 1-10.)

As with the `Servlet` methods discussed in "The Servlet Interface" on page B-3, 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 B-3 for a discussion of the `Servlet` interface and its key methods.

# C

# Compile-Time JML Tag Support

OracleJSP 1.0.0.6.x releases, because they were JSP 1.0 implementations, could support JML tags only as Oracle-specific extensions. (The tag library framework was not added to the JavaServer Pages specification until JSP 1.1.) For those releases, JML tag processing was built into the OracleJSP translator. This is referred to as "compile-time JML support".

Release 1.1.0.0.0 continues to support the compile-time JML implementation; however, it is generally advisable to use the runtime implementation, documented in Chapter 7, "JSP Tag Libraries and the Oracle JML Tags", wherever possible.

This appendix discusses features of the compile-time implementation that are not in common with the runtime implementation. This includes the following topics:

- JML Compile-Time Versus Runtime Considerations and Logistics
- JML Compile-Time / 1.0.0.6.x Syntax Support
- JML Compile-Time / 1.0.0.6.x Tag Support

# JML Compile-Time Versus Runtime Considerations and Logistics

This section discusses two aspects of compile-time tag libraries compared to runtime tag libraries:

- general considerations in when it may be advantageous to use a compile-time tag library implementation (for any library, not just JML)
- the `taglib` directive required for the compile-time JML implementation in particular

## General Compile-Time Versus Runtime Considerations

The Sun Microsystems *JavaServer Pages Specification, Version 1.1*, 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. Such implementations would depend on the OracleJSP translator, so would not be portable to other JSP environments.

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 as first introduced in older versions of OracleJSP. There are also a few additional tags in that implementation, and some additional expression syntax that is supported. (See "JML Compile-Time / 1.0.0.6.x Syntax Support" on page C-4 and "JML Compile-Time / 1.0.0.6.x Tag Support" on page C-7.)

It is generally advisable, however, to use the JML runtime implementation that adheres to the JSP 1.1 specification.

## The taglib Directive for Compile-Time JML Support

The OracleJSP 1.0.0.6.x/compile-time JML support implementation uses a custom class supplied by Oracle, `OpenJspRegisterLib`, to implement JML tag support.

In a JSP page using JML tags with the compile-time implementation, the `taglib` directive must specify the fully qualified name of this class (as opposed to specifying a TLD file as in standard JSP 1.1 tag library usage).

Following is an example:

```
<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>
```

For information about usage of the taglib directive for the JML runtime implementation, see "The taglib Directive" on page 7-13.

# JML Compile-Time/1.0.0.6.x Syntax Support

This section describes Oracle-specific bean reference syntax and expression syntax supported by the compile-time JML implementation, for specifying tag attribute values. The following topics are covered:

- JML Bean References and Expressions, Compile-Time Implementation
- Attribute Settings with JML Expressions

This functionality is not portable to other JSP environments.

## JML Bean References and Expressions, Compile-Time Implementation

Generally speaking, a *bean reference* is any reference to a JavaBean instance (bean) that results in accessing either a property or a method of the bean. This includes a reference to a property or method of a bean where the bean itself is a property of another bean.

This becomes cumbersome, because standard JavaBeans syntax requires that properties be accessed by calling their accessor methods rather than by direct reference. For example, consider the following direct reference:

```
a.b.c.d.doIt()
```

This must be expressed as follows in standard JavaBeans syntax:

```
a.getB().getC().getD().doIt()
```

Oracle's compile-time JML implementation, however, offers abbreviated syntax.

### JML Bean References

Oracle-specific syntax supported by the compile-time JML implementation allows bean references to be expressed using direct "." (dot) notation. Note that standard bean property accessor method syntax is also still valid.

Consider the following standard JavaBean reference:

```
customer.getName()
```

In JML bean reference syntax, you can express this in either of the following ways:

```
customer.getName()
```

or:

```
customer.name
```

JavaBeans can optionally have a default property, whose reference is assumed if no reference is explicitly stated. Default property names can be omitted in JML bean references. In the example above, if `name` is the default property, then the following are all valid JML bean references:

```
customer.getName()
```

or:

```
customer.name
```

or simply:

```
customer
```

Most JavaBeans do not define a default property. Of those that do, the most significant are the JML datatype JavaBeans described in "JML Extended Datatypes" on page 5-2.

### JML Expressions

JML expression syntax supported by the compile-time JML implementation is a superset of standard JSP expression syntax, adding support for the JML bean reference syntax documented in the preceding section.

A JML bean reference appearing in a JML expression must be enclosed in the following syntax:

`$[`*`JML_bean_reference`*`]`

## Attribute Settings with JML Expressions

Tag attribute documentation under "JSP Markup Language (JML) Tag Descriptions" on page 7-30 notes standard syntax that is portable. You can set attributes, as documented there, for either the runtime or the compile-time JML implementation and even for non-Oracle JSP environments.

If you intend to use only the Oracle-specific compile-time implementation, however, you can set attributes using JML bean references and JML expression syntax, as documented in "JML Bean References and Expressions, Compile-Time Implementation" on page C-4.

Note the following:

- Wherever Chapter 7 documents an attribute that accepts either a string literal or an expression, you can use a JML expression in its `$[...]` syntax inside standard JSP `<%=...%>` syntax.

  Consider an example using the JML `useVariable` tag. You would use syntax such as the following for the runtime implementation:

```
<jml:useVariable id = "isValidUser" type = "boolean" value = "<%= dbConn.isValid() %>" scope = "session" />
```

  You can alternatively use syntax such as the following for the compile-time implementation (the `value` attribute can be either a string literal or an expression):

```
<jml:useVariable id = "isValidUser" type = "boolean" value = "<%= $[dbConn.valid] %>" scope = "session" />
```

- Wherever Chapter 7 documents an attribute that accepts an expression only, you can use a JML expression in its `$[...]` syntax without being nested in `<%=...%>` syntax.

  Consider an example using JML `choose...when` tags. You would use something such as the following syntax for the runtime implementation (presume `orderedItem` is a `JmlBoolean` instance):

```
<jml:choose>
     <jml:when condition = "<%= orderedItem.getValue() %>"  >
               You have changed your order:
             -- outputs the current order --
     </jml:when>
     <jml:otherwise>
               Are you sure we can't interest you in something?
     </jml:otherwise>
</jml:choose>
```

  You can alternatively use syntax such as the following for the compile-time implementation (the `condition` attribute can be an expression only):

```
<jml:choose>
     <jml:when condition = "$[orderedItem]"  >
               You have changed your order:
             -- outputs the current order --
     </jml:when>
     <jml:otherwise>
               Are you sure we can't interest you in something?
     </jml:otherwise>
</jml:choose>
```

# JML Compile-Time/1.0.0.6.x Tag Support

This section presents the following:

- a summary of all compile-time tags, noting which are desupported in the runtime implementation
- a description of tags supported by the compile-time implementation that are desupported in the runtime implementation (because such tags are not documented in "JSP Markup Language (JML) Tag Descriptions" on page 7-30)

> **Note:** In most cases, JML tags that are desupported in the runtime implementation have standard JSP equivalents. Some of the compile-time tags, however, were desupported because they have functionality that is difficult to implement when adhering to the JSP 1.1 specification.

## JML Tag Summary, 1.0.0.6.x/Compile-Time Versus 1.1.0.0.0/Runtime

Most JML tags are available in both the runtime model and the compile-time model; however, there are exceptions, as summarized in Table C–1.

*Table C–1 JML Tags Supported: Compile-Time Model Versus Runtime Model*

| Tag | Supported in OracleJSP Compile-Time Implementation? | Supported in OracleJSP Runtime Implementation? |
|---|---|---|
| **Bean Binding Tags**: | | |
| useBean | yes | no; use `jsp:useBean` |
| useVariable | yes | yes |
| useForm | yes | yes |
| useCookie | yes | yes |
| remove | yes | yes |
| **Bean Manipulation Tags** | | |
| getProperty | yes | no; use `jsp:getProperty` |
| setProperty | yes | no; use `jsp:setProperty` |
| set | yes | no |
| call | yes | no |

*Table C–1 JML Tags Supported: Compile-Time Model Versus Runtime Model (Cont.)*

| Tag | Supported in OracleJSP Compile-Time Implementation? | Supported in OracleJSP Runtime Implementation? |
|---|---|---|
| lock | yes | no |
| **Control Flow Tags** | | |
| if | yes | yes |
| choose | yes | yes |
| for | yes | yes |
| foreach | yes; `type` attribute is optional | yes; `type` attribute is required |
| return | yes | yes |
| flush | yes | yes |
| include | yes | no; use `jsp:include` |
| forward | yes | no; use `jsp:forward` |
| **XML Tags** | | |
| transform | yes | yes |
| styleSheet | yes | yes |
| **Utility Tags** | | |
| print | yes; use double-quotes to specify a string literal | no; use JSP expressions |
| plugin | yes | no; use `jsp:plugin` |

## Descriptions of Additional JML Tags, Compile-Time Implementation

This section provides detailed descriptions of JML tags that are still supported by the JML compile-time implementation, but are not supported by the JML runtime implementation. These tags are not documented under "JSP Markup Language (JML) Tag Descriptions" on page 7-30.

In summary, this consists of the following JML tags.

- JML useBean Tag
- JML getProperty Tag
- JML setProperty Tag
- JML set Tag

- JML call Tag
- JML lock Tag
- JML include Tag
- JML forward Tag
- JML print Tag
- JML plugin Tag

For the syntax documentation in the tag descriptions, note the following:

- *Italics* indicate you must specify a value or string.
- Optional attributes are enclosed in square brackets: `[...]`
- Default values of optional attributes are indicated in **bold**.
- Choices in how to specify an attribute are separated by vertical bars: |
- The prefix "jml:" is used. This is by convention, but is not required. You can specify any desired prefix in your `taglib` directive.

### JML useBean Tag

This tag declares an object to be used in the page, locating the previously instantiated object at the specified scope by name if it exists. If it does not exist, the tag will create a new instance of the appropriate class and attach it to the specified scope by name.

The syntax and semantics are the same as for the standard `jsp:useBean` tag, except that wherever a JSP expression is valid in `jsp:useBean` usage, either a JML expression or a JSP expression is valid in JML `useBean` usage.

#### Syntax

```
<jml:useBean id = "beanInstanceName"
             scope ="page | request | session | application"
             class ="package.class" | type = "package.class" |
             class ="package.class" type = "package.class" |
             beanName = "package.class | <%= jmlExpression %>"
             type = "package.class" />
```

Alternatively, you can have additional nested tags, such as `setProperty` tags, and use a `</jml:useBean>` end tag.

#### Attributes

Refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1* for information about attributes and their syntax.

#### Example

```
<jml:useBean id = "isValidUser" class = "oracle.jsp.jml.JmlBoolean" scope = "session" />
```

### JML getProperty Tag

This tag is functionally identical to the standard `jsp:getProperty` tag. It prints the value of the bean property into the response.

For general information about `getProperty` usage, refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

#### Syntax

```
<jml:getProperty name = "beanInstanceName" property = "propertyName" />
```

#### Attributes

- `name`—This is the name of the bean whose property is being retrieved. This attribute is required.
- `property`—This is the name of the property being retrieved. This attribute is required.

**Example** The following example outputs the current value of the `salary` property. (Assume `salary` is of type `JmlNumber`.)

```
<jml:getProperty name="salary" property="value" />
```

This is equivalent to the following:

```
<%= salary.getValue() %>
```

### JML setProperty Tag

This tag covers the functionality supported by the standard `jsp:setProperty` tag, but also adds functionality to support JML expressions. In particular, you can use JML bean references.

For general information about `setProperty` usage, refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

#### Syntax

```
<jml:setProperty name = "beanInstanceName"
        property = " * " |
        property = "propertyName" [ param = "parameterName" ] |
        property = "propertyName"
        value = "stringLiteral | <%= jmlExpression %>" />
```

#### Attributes

- `name`—This is the name of the bean whose property is being set. This attribute is required.
- `property`—This is the name of the property being set. This attribute is required.
- `value`—This is an optional parameter that lets you set the value directly instead of from a request parameter. The JML `setProperty` tag supports JML expressions in addition to standard JSP expressions to specify the value.

**Example** The following example updates `salary` with a six percent raise. (Assume `salary` is of type `JmlNumber`.)

```
<jml:setProperty name="salary" property="value" value="<%= $[salary] * 1.06 %>" />
```

This is equivalent to the following:

```
<% salary.setValue(salary.getValue() * 1.06); %>
```

### JML set Tag

This tag provides an alternative for setting a bean property, using syntax that is more convenient than that of the `setProperty` tag.

#### Syntax

```
<jml:set name = "beanInstanceName.propertyName"
         value = "stringLiteral | <%= jmlExpression %>" />
```

#### Attributes

- `name`—This is a direct reference (JML bean reference) to the bean property to be set. This attribute is required.
- `value`—This is the new property value. It is expressed either as a string literal, a JML expression, or a standard JSP expression. This attribute is required.

**Example** Each of the following examples updates `salary` with a six percent raise. (Assume `salary` is of type `JmlNumber`.)

```
<jml:set name="salary.value" value="<%= salary.getValue() * 1.06 %>" />
```

or:

```
<jml:set name="salary.value" value="<%= $[salary.value] * 1.06 %>" />
```

or:

```
<jml:set name="salary" value="<%= $[salary] * 1.06 %>" />
```

These are equivalent to the following:

```
<% salary.setValue(salary.getValue() * 1.06); %>
```

### JML call Tag

This tag provides a mechanism to invoke bean methods that return nothing.

#### Syntax

```
<jml:call method = "beanInstanceName.methodName(parameters)" />
```

#### Attributes

- `method`—This is the method call as you would write it in a scriptlet, except that the *`beanInstancename.methodName`* portion of the statement can be written as a JML bean reference if enclosed in JML expression `$[... ]` syntax. This attribute is required.

**Example** The following example redirects the client to a different page:

```
<jml:call name='response.sendRedirect("http://www.oracle.com/")' />
```

This is equivalent to the following:

```
<% response.sendRedirect("http://www.oracle.com/"); %>
```

## JML lock Tag

This tag allows controlled, synchronous access to the named object for any code that uses it within the tag body.

Generally, JSP developers need not be concerned with concurrency issues. However, because application-scoped objects are shared across all users running the application, access to critical data must be controlled and coordinated.

You can use the JML `lock` tag to prevent concurrent updates by different users.

### Syntax

```
<jml:lock name = "beanInstanceName" >
   ...body...
</jml:lock>
```

### Attributes

- `name`—This is the name of the object that should be locked during execution of code in the `lock` tag body. This is a required attribute.

**Example** In the following example, `pageCount` is an application-scoped `JmlNumber` value. The variable is locked to prevent the value from being updated by another user between the time this code gets the current value and the time it sets the new value.

```
<jml:lock name="pageCount" >
    <jml:set name="pageCount.value" value="<%= pageCount.getValue() + 1 %>" />
</jml:lock>
```

This is equivalent to the following:

```
<% synchronized(pageCount)
   {
      pageCount.setValue(pageCount.getValue() + 1);
   }
%>
```

## JML include Tag

This tag includes the output of another JSP page, a servlet, or an HTML page in the response of this page (the page invoking the `include`). It provides the same functionality as the standard `jsp:include` tag except that the `page` attribute can also be expressed as a JML expression.

#### Syntax

```
<jml:include page = "relativeURL | <%= jmlExpression %>" flush = "true" />
```

#### Attributes

For general information about `include` attributes and usage, refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

**Example** The following example includes the output of `table.jsp`, a presentation component that renders an HTML table, based on data in the query string and request attributes.

```
<jml:include page="table.jsp?maxRows=10" flush="true" />
```

## JML forward Tag

This tag forwards the request to another JSP page, a servlet, or an HTML page. It provides the same functionality as the standard `jsp:forward` tag except that the `page` attribute can also be expressed as a JML expression.

#### Syntax

```
<jml:forward page = "relativeURL | <%= jmlExpression %>" />
```

#### Attributes

For general information about `forward` attributes and usage, refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

#### Example

```
<jml:forward page="altpage.jsp" />
```

## JML print Tag

This tag provides essentially the same functionality as a standard JSP expression: `<%= expr %>`. A specified JML expression or string literal is evaluated, and the result is output into the response. With this tag, the JML expression does not have to be enclosed in `<%= ... %>` syntax; however, a string literal must be enclosed in double-quotes.

### Syntax

```
<jml:print eval = '"stringLiteral"' | "jmlExpression" />
```

### Attributes

`eval`—Specifies the string or expression to be evaluated and output. This attribute is required.

**Examples** Either of the following examples outputs the current value of `salary`, which is of type `JmlNumber`:

```
<jml:print eval="$[salary]"/>
```

or:

```
<jml:print eval="salary.getValue()" />
```

The following example prints a string literal:

```
<jml:print eval='"Your string here"' />
```

## JML plugin Tag

This tag has functionality identical to that of the standard `jsp:plugin` tag.

For information about `plugin` attributes, usage, and examples, refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

# Index

## Symbols



## A



## B



## C

## D

## E

## F



## G



## H

## I



## J

## L



## M



## N



## O

## P

## R



## S

# T

## U



## V



## W



## X