Oracle® Application Server Containers for J2EE

JSP Tag Libraries and Utilities Reference

10*g* Release 2 (10.1.2)

Part No. B14016-01

November 2004

ORACLE®

Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference, 10*g* Release 2 (10.1.2)

Part No. B14016-01

Contents

2   JavaBeans for Extended Types

3   JSP Markup Language Tags

4   Data-Access JavaBeans and Tags

5   XML and XSL Tag Support

6   JESI Tags for Edge Side Includes

7   Web Object Cache Tags and API

8   File Access and Mail Beans and Tags

9   JSP Utilities and Utility Tags

10   Web Services Tags

A   JML Compile-Time Syntax and Tags

B   Third Party Licenses

Index

Send Us Your Comments

Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference, 10*g* Release 2 (10.1.2)

Part No. B14016-01

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

- Did you find any errors?

- Is the information clearly presented?

- Do you need more information? If so, where?

- Are the examples correct? Do you need more examples?

- What features did you like most about this manual?

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

- Electronic mail: appserverdocs_us@oracle.com

- FAX: (650) 506-7225   Attn: Java Platform Group, Information Development Manager

- Postal service:

  Oracle Corporation
  Java Platform Group, Information Development Manager
  500 Oracle Parkway, Mailstop 4op9
  Redwood Shores, CA   94065
  USA

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

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

x

Preface

JavaServer Pages (JSP) technology, as specified by an industry consortium led by Sun Microsystems, is a component of the standard Java 2 Enterprise Edition (J2EE). The J2EE component of the Oracle Application Server is known as the Oracle Application Server Containers for J2EE (OC4J).

This document provides reference information as well as some conceptual material for JSP tag libraries and utilities included with OC4J in Oracle Application Server 10*g* Release 2 (10.1.2). These libraries generally conform to the JSP specification.

For general information about the OC4J JSP implementation, including the JSP tag library framework, refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide*.

This preface contains the following sections:

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

Intended Audience

This document is intended for Web application developers using servlet and JavaServer Pages technology. It assumes that working Web, servlet, and JSP environments already exist, and that readers are already familiar with the following:

- General Web technology
- Java
- HTML
- Java servlets
- JavaServer Pages
- Configuration of their Web server and servlet environments
- Oracle JDBC (for JSP applications accessing Oracle Database)

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for background information about standard JavaServer Pages technology, the Oracle JSP implementation, and tag library support.

Documentation Accessibility

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

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

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

Structure

This document contains:

Chapter 1, "Overview of Tag Libraries and Utilities"

This chapter provides an overview of the tag libraries documented in the remainder of the manual, as well as overviews of tag libraries provided with other Oracle components, outside of OC4J.

Chapter 2, "JavaBeans for Extended Types"

This chapter discusses JavaBeans provided with the JSP Markup Language (JML) library that can be used as extended Java types.

Chapter 3, "JSP Markup Language Tags"

This provides JML syntax and tag descriptions, as well as an overview of the philosophy behind the JML tag library.

Chapter 4, "Data-Access JavaBeans and Tags"

This documents JavaBeans and tags for database access.

Chapter 5, "XML and XSL Tag Support"

This chapter describes tags to use in handling XML documents and outputting or transforming their data.

Chapter 6, "JESI Tags for Edge Side Includes"

This chapter describes the Oracle implementation of JESI tags to support Edge Side Includes technology for Web caching.

Chapter 7, "Web Object Cache Tags and API"

This describes concepts, custom tags, the Java API, and XML descriptor files for the Web Object Cache, an application-level Java caching interface provided with OC4J.

Chapter 8, "File Access and Mail Beans and Tags"

This chapter covers tags and JavaBeans for file access (uploading and downloading) and e-mail.

Chapter 9, "JSP Utilities and Utility Tags"

This chapter discusses miscellaneous utility features included with OC4J: `JspScopeListener` for event-handling, tags for using EJBs, and general utility tags.

Chapter 10, "Web Services Tags"

This chapter describes the Web services tag library, which allows developers to create JSP pages for use as client programs for Web services.

Appendix A, "JML Compile-Time Syntax and Tags"

This chapter provides an overview of the compile-time implementation of the Oracle JML sample tag library (the only way the library was supported in pre-JSP 1.1 releases), and documents tags not supported in the runtime implementation that is documented in Chapter 3.

Appendix B, "Third Party Licenses"

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

Related Documents

For more information, see these Oracle resources:

Additional OC4J documents available from the Oracle Java Platform Group:

- *Oracle Application Server Containers for J2EE User's Guide*

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

- *Oracle Application Server Containers for J2EE Stand Alone User's Guide*

  This version of the user's guide is specifically for the standalone version of OC4J, and is available when you download the standalone version from OTN. OC4J standalone is used in development environments, but not typically in production environments.

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

  This book provides information for servlet developers regarding use of servlets and the servlet container in OC4J, including basic servlet development, use of JDBC and EJBs, building and deploying applications, and servlet and Web site configuration.

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

  This book provides information for JSP developers who want to run their pages in OC4J. It includes a general overview of JSP standards and programming

considerations, as well as discussion of Oracle value-added features and steps for getting started in the OC4J environment.

- *Oracle Application Server Containers for J2EE Services Guide*

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

- *Oracle Application Server Containers for J2EE Security Guide*

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

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

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

Also available from the Oracle Java Platform group:

- *Oracle Database Java Developer's Guide*

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

- *Oracle Database JPublisher User's Guide*

Available from the Oracle Application Server group:

- *Oracle Application Server Administrator's Guide*

- *Oracle Application Server Security Guide*

- *Oracle Application Server Performance Guide*

- *Oracle Enterprise Manager Concepts*

- *Oracle HTTP Server Administrator's Guide*

- *Oracle Application Server Globalization Guide*

- *Oracle Application Server Web Cache Administrator's Guide*

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

- *Oracle Application Server Upgrading to 10g Release 2 (10.1.2)*

Available from the Oracle JDeveloper group:

- Oracle JDeveloper online help

- Oracle JDeveloper documentation on the Oracle Technology Network:

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

Available from the Oracle Server Technologies group:

- *Oracle XML Developer's Kit Programmer's Guide*

- *Oracle XML API Reference*

- *Oracle Database Application Developer's Guide - Fundamentals*

- *PL/SQL Packages and Types Reference*

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

- *Oracle Database SQL Reference*

- *Oracle Database Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle Database Reference*

Printed documentation is available for sale in the Oracle Store at

http://oraclestore.oracle.com/

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

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

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

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

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

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

The following resources are available from Sun Microsystems.

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

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

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

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

- `jsp-interest` discussion group for JavaServer Pages

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

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

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

    ```
    set jsp-interest digest
    ```

Conventions

The following conventions are also used in this manual:

Convention	Meaning
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a term defined in the text, the glossary, or in both locations.
Italics	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.

Convention	Meaning
`Monospace (fixed-width) font`	Monospace typeface within text indicates items such as executables, file names, directory names, Java class names, Java method names, variable names, other programmatic elements (such as JSP tags or attributes, or XML elements or attributes), or database SQL commands or elements (such as schema names, table names, or column names).
`Italic monospace (fixed-width) font`	Italic monospace font represents placeholders or variables.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
|	A vertical bar represents a choice of two or more options. Enter one of the options. Do not enter the vertical bar.

1

Overview of Tag Libraries and Utilities

This manual documents tag libraries, JavaBeans, and other utilities supplied with Oracle Application Server Containers for J2EE (OC4J) that are implemented according to JSP standards. There is also a discussion of support for the JavaServer Pages Standard Tag Library (JSTL), and a section summarizing tag libraries provided with Oracle components outside of OC4J.

Oracle-specific features, as well as an introduction to the OC4J JSP container, standard JSP technology, and standard JSP 1.2 tag library features, are covered in the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide.*

This chapter consists of the following sections:

- Overview of Tag Libraries and Utilities Provided with OC4J
- Summary of Oracle Caching Support for Web Applications
- Support for the JavaServer Pages Standard Tag Library
- Overview of Tag Libraries from Other Oracle Components

Tags and JavaBeans introduced in the first section provide functionality in several different areas, including type extensions, integration with XML/XSL, database access, and programming convenience.

> **Note:** The Sample Applications chapter available in previous releases has been removed. Applications that were listed there are available in the OC4J demos, available from the following location on the Oracle Technology Network (requiring an OTN membership, which is free of charge):
>
> `http://www.oracle.com/technology/tech/java/oc4j/demos/`

Overview of Tag Libraries and Utilities Provided with OC4J

The Oracle extensions that are introduced in the following sections are implemented through tag libraries or custom JavaBeans that comply with JSP and JavaBeans standards.

- Tag Syntax Symbology and Notes
- Overview of Extended Type JavaBeans
- Overview of JspScopeListener for Event-Handling
- Overview of Integration with XML and XSL
- Summary of Data-Access JavaBeans and Tag Library

- [Summary of JSP Markup Language (JML) Custom Tag Library](#)
- [Summary of Web Services Tags](#)
- [Summary of File Access and Mail Tags](#)
- [Summary of EJB Tags](#)
- [Summary of JSP Utility Tags](#)

Be aware that some custom tag libraries provided with OC4J—XML, data-access, and JML—pre-date the JavaServer Pages Standard Tag Library (JSTL) and have areas of duplicate functionality. For standards compliance, it is now generally advisable to use JSTL instead of these custom libraries. See "Support for the JavaServer Pages Standard Tag Library" on page 1-14.

Oracle is not desupporting the existing libraries, however. For features in the custom library that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try to have the features adopted into the JSTL standard as appropriate.

> **Notes:**
>
> - See "Summary of Oracle Caching Support for Web Applications" on page 1-9 for information about tag libraries provided with OC4J to support caching features.
>
> - See the OC4J demos for sample applications using the features introduced in this section. They can be downloaded from the following location on the Oracle Technology Network (requiring an OTN membership, which is free of charge):
>
> `http://www.oracle.com/technology/tech/java/oc4j/demos/`

Tag Syntax Symbology and Notes

For the syntax documentation in tag descriptions throughout this manual, note the following:

- *Italic* indicates 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 attribute values are separated by vertical bars: `|`
- Except where noted, you can use JSP runtime expressions to set tag attribute values: "`<%= jspExpression %>`"
- Tag descriptions in this manual use certain tag prefixes by convention; however, you can designate any desired prefix in your `taglib` directives.

Overview of Extended Type JavaBeans

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

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

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

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

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

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

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

For information, see Chapter 2, "JavaBeans for Extended Types".

Overview of JspScopeListener for Event-Handling

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

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

For information, see "JSP Event-Handling with JspScopeListener" on page 9-1.

Overview of Integration with XML and XSL

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

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

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

There is additional XML support as well:

- A utility tag converts data from an input stream to an XML DOM object.
- Several tags, for such features as caching and SQL operations, now can take XML objects as input or send them as output.

XML utility tags are summarized in Table 1–1. Note that there is also XML functionality in the `dbOpen` SQL tag and the `cacheXMLObj` Web Object Cache tag. For more information, see Chapter 5, "XML and XSL Tag Support".

You can find information about standard JSP 1.2 XML support in the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide*.

Table 1–1   Summary of XML Utility Tags

Tag	Description	Attributes
transform	Output XML data with an XSL transformation, either to an HTTP client or a specified XML DOM object.	href fromXMLObjName toXMLObjName toWriter
styleSheet	Same as transform tag.	href fromXMLObjName toXMLObjName toWriter
parsexml	Convert from an input stream to an XML DOM object.	resource toXMLObjName validateResource root

Summary of Data-Access JavaBeans and Tag Library

OC4J supplies a set of custom JavaBeans for use in accessing Oracle Database. The following beans are provided in the oracle.jsp.dbutil package:

- ConnBean opens a database connection. This bean also supports data sources and connection pooling.

- ConnCacheBean uses the Oracle connection caching implementation for database connections. (This requires JDBC 2.0.)

- DBBean executes a database query.

- CursorBean provides general DML support for queries; UPDATE, INSERT, and DELETE statements; and stored procedure calls.

For information, see "JavaBeans for Data Access" on page 4-1.

For JSP programmers, OC4J also provides a custom tag library for SQL functionality, wrapping the functionality of the JavaBeans. These tags are summarized in Table 1–2. For further information, see "SQL Tags for Data Access" on page 4-11.

Table 1–2   Summary of Data-Access Tag Library

Tag	Description	Attributes
dbOpen	Open a database connection. This tag also supports data sources and connection pooling.	connId scope dataSource user password URL commitOnClose
dbClose	Close a database connection.	connId scope
dbQuery	Execute a query.	queryId connId scope output maxRows skipRows bindParams toXMLObjName
dbCloseQuery	Close the cursor for a query.	queryId

Table 1–2   (Cont.) Summary of Data-Access Tag Library

Tag	Description	Attributes
dbNextRow	Process the rows of a result set.	queryId
dbExecute	Execute any SQL statement (DML or DDL).	connId scope output bindParams
dbSetParam	Set a parameter to bind into a `dbQuery` or `dbExecute` tag.	name value scope
dbSetCookie	Set a cookie.	name value domain comment maxAge version secure path

Summary of JSP Markup Language (JML) Custom Tag Library

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

OC4J provides custom tags as an alternative: the JSP Markup Language (JML). The Oracle JML tag library provides an additional set of JSP tags so that you can script your JSP pages without using Java statements. JML provides tags for variable declarations, 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` tag, repeatedly printing "Hello World" in progressively smaller headings (H1, H2, H3, H4, H5):

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

The JML tag library is summarized in Table 1–3. For more information, see Chapter 3, "JSP Markup Language Tags".

Table 1–3   Summary of JSP Markup Language Tag Library

Tag	Description	Attributes
useVariable	This tag offers a convenient alternative to the `jsp:useBean` tag for declaring simple variables.	id scope type value
useForm	This tag provides a convenient syntax for declaring variables and setting them to values passed in from the request.	id scope type param

Table 1–3   (Cont.) Summary of JSP Markup Language Tag Library

Tag	Description	Attributes
useCookie	This tag offers a convenient syntax for declaring variables and setting them to values contained in cookies.	id scope type cookie
remove	This tag removes an object from its scope.	id scope
if	This tag evaluates a single conditional statement. If the condition is true, then the body of the if tag is executed.	condition
choose	The choose tag, with associated when and otherwise tags, provides a multiple conditional statement.	(None)
when	This is used with the choose tag.	condition
otherwise	This is optionally used with the choose and when tags.	(None)
for	This tag provides the ability to iterate through a loop, as with a Java for loop.	id from to
foreach	This tag provides the ability to iterate over a homogeneous set of values in a Java array, Enumeration instance, or Vector instance.	id in limit type
return	When this tag is reached, execution returns from the page without further processing.	(None)
flush	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.	(None)

Note: Oracle JSP container versions preceding the JSP 1.1 specification use an Oracle-specific compile-time implementation of the JML tag library. Oracle still supports this implementation as an alternative to the standard runtime implementation, as documented in Appendix A, "JML Compile-Time Syntax and Tags".

Summary of Web Services Tags

The Web services tag library provided with OC4J enables developers to conveniently create JSP pages for Web service client applications. The implementation uses a SOAP-based mechanism, supporting RPC-style or document-style services. A client application would access a Web Services Description Language (WSDL) document, then use the WSDL information to access the operations of a Web service.

The tag library also uses the Oracle implementation of the dynamic invocation API, described in the *Oracle Application Server Web Services Developer's Guide*. When a client application acquires a WSDL document at runtime, the dynamic invocation API is the vehicle for invoking any SOAP operation described in the WSDL document.

The Web services tag library is summarized in Table 1–4. For more information, see Chapter 10, "Web Services Tags".

Table 1–4   Summary of Web Services Tag Library

Tag	Description	Attributes
webservice	Create a Web service proxy. The tag requires the URL of a WSDL document and uses either a binding and SOAP location or a service name and port in creating the proxy.	wsdlUrl id scope binding soapLocation service port
map	Use `map` tags nested within a `webservice` tag to have the Web service proxy add entries to the SOAP mapping registry for type mapping between SOAP/XML and Java. Use one `map` tag for each desired type mapping.	localName namespaceUri javaType encodingStyle java2xmlClassName xml2javaClassName
property	Optionally use this tag to specify a name/value pair that defines any of several supported custom properties for use by the Web service client application.	name value
invoke	Invoke an operation of a Web service. The `invoke` tag gains access to a Web service proxy either by being nested within a `webservice` tag or by accessing a Web service proxy scripting variable created in a `webservice` tag.	id operation webservice inputMsgName outputMsgName xmlToWriter toXMLObjName
part	Use this tag if the operation being performed requires input message part values, using one part tag for each input part.	name value

Summary of File Access and Mail Tags

OC4J provides tag libraries for file access (uploading and downloading) and for sending e-mail messages from an application.

For uploading files, you can use the `httpUpload` tag or the `oracle.jsp.webutil.fileaccess.HttpUploadBean` JavaBean. For downloading, there is the `httpDownload` tag or the `HttpDownloadBean` JavaBean. Table 1–5 summarizes the file access tags. For more information, see "File-Access JavaBeans and Tags" on page 8-1.

Table 1–5   Summary of File Access Tag Library

Tag	Description	Attributes
httpUploadForm	For convenience, you can use this tag to create a form in your application, using multipart encoded form data, that allows users to specify the files to upload.	formsAction maxFiles fileNameSize maxFileNameSize includeNumbers submitButtonText

Table 1–5   (Cont.) Summary of File Access Tag Library

Tag	Description	Attributes
httpUpload	Upload files from the client to a server. You can upload into either a file system or a database.	destination destinationType connId scope overwrite fileType table prefixColumn fileNameColumn dataColumn
httpDownload	Download files from a server to the client. You can download from either a file system or a database.	servletPath source sourceType connId scope recurse fileType table prefixColumn fileNameColumn dataColumn

For sending e-mail messages, optionally with server-side or client-side attachments, you can use the `oracle.jsp.webutil.email.SendMailBean` JavaBean or the `sendMail` tag. Table 1–6 summarizes the `sendMail` tag. See "Mail JavaBean and Tag" on page 8-18 for more information.

Table 1–6   Summary of sendMail Tag

Tag	Description	Attributes
sendMail	Send an e-mail message from a JSP page. Tag functionality includes globalization support.	host sender recipient cc bcc subject contentType contentEncoding serverAttachment clientAttachment

Summary of EJB Tags

OC4J provides a custom tag library to simplify the use of Enterprise JavaBeans in JSP pages. The functionality of the OC4J EJB tags follows the J2EE specification. The tags allow you to instantiate EJBs by name, using configuration information in the `web.xml` file.

There are tags to create a home instance, create an EJB instance, and iterate through a collection of EJBs. Table 1–7 summarizes the EJB tag library. See "EJB Tags" on page 9-10 for more information.

Table 1–7   Summary of EJB Tag Library

Tag	Description	Attributes
useHome	Look up the home interface for the EJB and create an instance of it.	id type location local
useBean	Instantiate and use the EJB. The functionality is similar to that of the standard `jsp:useBean` tag for a JavaBean.	id type value scope local
createBean	For first instantiating an EJB, if you do not use the `value` attribute of the EJB `useBean` tag, you must nest an EJB `createBean` tag within the `useBean` tag to do the work of creating the EJB instance.	instance
iterate	Iterate through a collection of EJB instances (more typical for entity beans).	id type collection max

Summary of JSP Utility Tags

OC4J has utility tags for displaying a date, displaying an amount of money in the appropriate currency, displaying a number, iterating through a collection, evaluating and including the tag body (depending on whether the user belongs to a specified role), and displaying the last modification date of the current file. Table 1–8 summarizes these tags. See "General Utility Tags" on page 9-16 for more information.

Table 1–8   Summary of General Utility Tag Library

Tag	Description	Attributes
displayCurrency	Display a specified amount of money, formatted as currency for the locale.	amount locale
displayDate	Display a specified date, formatted appropriately for the locale.	date locale
displayNumber	Display the specified number, for the locale and optionally in the specified format.	number locale format
iterate	Iterate through a collection.	id type collection max
ifInRole	Evaluate the tag body and include it in the body of the JSP page, depending on whether the user is in the specified application role.	role include
lastModified	Display the date of the last modification of the current file, in appropriate format for the locale.	locale

Summary of Oracle Caching Support for Web Applications

This section provides the following information:

- An introduction to caching features supported by the Oracle Application Server in general and the OC4J JSP container in particular

- A discussion of the role of the OC4J Web Object Cache in relation to other Oracle Application Server caching components

- A summary of tag libraries relating to caching features

The Oracle tag libraries introduced in this section comply with JSP standards.

> **Note:** See the OC4J demos for sample applications using the features introduced in this section. The can be downloaded from the following location on the Oracle Technology Network:
>
> http://www.oracle.com/technology/tech/java/oc4j/demos/

Oracle Application Server and JSP Caching Features

The Oracle Application Server and OC4J provide the following caching features:

- Oracle Application Server Web Cache

  This is an HTTP-level cache, maintained outside the application, providing very fast cache operations. It is a content-based cache, capable of caching static data (such as HTML, GIF, or JPEG files) or dynamic data (such as servlet or JSP results). Given that it exists as a content-based cache outside the application, it cannot cache objects (such as Java objects or XML DOM objects) in a Java object format. In addition, post-processing operations applicable to cached data cannot be coded in Java and are predefined by the cache itself.

  The OracleAS Web Cache provides an ESI processor to support Edge Side Includes, an XML-style markup language that allows dynamic content assembly away from the Web server. This technology enables you to separate cacheable pages into distinct cached objects, as desired. OC4J supports this technology through its JESI tag library.

  For an overview of Edge Side Includes and the OracleAS Web Cache, as well as detailed documentation of the JESI tag library, see Chapter 6, "JESI Tags for Edge Side Includes".

  For additional information about the OracleAS Web Cache, see the *Oracle Application Server Web Cache Administrator's Guide*.

- OC4J Web Object Cache

  This is an application-level cache that is embedded and maintained within a Java Web application. It is a hybrid cache, both Web-based and object-based. A custom tag library or API enables you to define page fragment boundaries and to capture, store, reuse, process, and manage the intermediate and partial execution results of JSP pages and servlets as cached objects. Each page fragment can produce a separate cache object. The produced objects can be HTML or XML text fragments, XML DOM objects, or Java serializable objects. These objects can be cached conveniently in association with HTTP request and session semantics. Alternatively, they can be reused outside HTTP, such as when an application outputs cached XML objects through Simple Mail Transfer Protocol (SMTP), Java Messaging Service (JMS), Advanced Queueing (AQ), or Simple Object Access Protocol (SOAP).

  For more information, see Chapter 7, "Web Object Cache Tags and API".

- Oracle Application Server Java Object Cache

  The Oracle Application Server Java Object Cache is a general-use Java cache to manage Java objects within a process, across processes, and on local disk. By

managing local copies of objects that are difficult or expensive to retrieve or create, the Java Object Cache can significantly improve application performance. By default, the OC4J Web Object Cache uses the Oracle Application Server Java Object Cache as its underlying cache repository.

For details about the Java Object Cache, see the *Oracle Application Server Containers for J2EE Services Guide*.

Role of the JSP Web Object Cache

It is important to understand the role of the OC4J Web Object Cache in the overall setup of a Web application. It works at the Java level and is closely integrated with the HTTP environment of servlet and JSP applications. By contrast, the Oracle Application Server Java Object Cache works at the Java object level, but is not integrated with HTTP. As for the OracleAS Web Cache, it is well integrated with HTTP and is orders of magnitude faster than the Web Object Cache, but it does not operate at the Java level and cannot be directly invoked by Web application code. For example, it cannot apply a style sheet to a cached DOM object within the J2EE container, reuse the cached result in other protocols, or allow direct DOM operations. OracleAS Web Cache can, however, apply a style sheet to text-based XML documents, as opposed to DOM objects, that were cached from the original Web server through HTTP.

The Web Object Cache is *not* intended for use as the main Web cache for an application. It is an auxiliary cache embedded within the same Java virtual machine that is running your servlets and JSP pages. Because the retrieval path for cached results in the Web Object Cache includes the JVM and the JSP and servlet engines, it generally takes much longer to serve a page from the Web Object Cache compared to the OracleAS Web Cache.

The Web Object Cache does not replace or eliminate the need for either the OracleAS Web Cache or the Oracle Application Server Java Object Cache. It is a complementary caching component in the overall framework of a Web application and should be used together with the other caching products, as appropriate. In fact, the Web Object Cache uses the Java Object Cache as its default repository. And through combined use of the OC4J JESI tags and Web Object Cache tags, you can use the Web Object Cache and OracleAS Web Cache together in the same page.

Web Object Cache Versus OracleAS Web Cache

Think of the OracleAS Web Cache as the primary caching component. It serves cached pages directly to HTTP clients and handles large volumes of HTTP traffic quickly, fitting the requirements of most Web sites. You can use the OracleAS Web Cache to store complete Web pages or partial pages (through use of the JESI tags). Cached pages can be customized, to a certain extent, before being sent to a client, including cookie-replacement and page-fragment concatenation, for example.

It is advisable to use the OracleAS Web Cache as much as possible to speed up response and reduce the load on the Web application server and back-end database. The caching needs of a large percentage of Web pages can be addressed by the OracleAS Web Cache alone.

As a complement to the OracleAS Web Cache, you can use the Web Object Cache to capture intermediate results of JSP and servlet execution and subsequently reuse these cached results in other parts of the Java application logic. It is not beneficial to use the Web Object Cache in your Web application unless you can repeatedly reuse objects after they are cached and you require post-processing on cached objects before they are served to a client.

Web Object Cache Versus Oracle Application Server Java Object Cache

In comparison to the Oracle Application Server Java Object Cache, the Web Object Cache makes it much easier to store and maintain partial execution results in dynamic Web pages. The Java Object Cache, being a pure object-based framework for any general Java application, is not aware of any HTTP environment in which it might be embedded. For example, its cache keys do not automatically depend on HTTP cookies or sessions. When you directly use the Java Object Cache within a Web application, you are responsible for creating any necessary interfacing with resulting Web pages. The Java Object Cache provides only a programmatic way to specify cache maintenance policies, whereas the Web Object Cache allows the alternative of specifying policies through configuration files.

Summary of Tag Libraries for Caching

OC4J supplies two tag libraries for use with Oracle Application Server caching features:

- JESI tag library
- Web Object Cache tag library

This section summarizes those libraries.

Summary of JESI Tag Library

OC4J provides the JESI tag library as a convenient interface to ESI tags and Edge Side Includes functionality for Web caching. Developers have the option of using ESI tags directly in any Web application, but JESI tags provide additional convenience in a JSP environment.

Table 1–9 summarizes the JESI tag library. See "Oracle JESI Tag Descriptions" on page 6-13 for more information.

Table 1–9   Summary of JESI Tag LIbrary

Tag	Description	Attributes
control	Control caching characteristics for JSP pages in the control/include usage model. You can use a JESI `control` tag in the top-level page or any included page.	expiration maxRemovalDelay cache control
include	This tag, like a standard `jsp:include` tag, enables you to dynamically insert output from the included page into output from the including page. However, it results in the included page being processed and assembled by the ESI processor, typically inside OracleAS Web Cache.	page alt ignoreError copyParam flush
param	This is a subtag of the JESI `include` tag. You can use one or more JESI `param` tags to pass additional query parameters to the included page.	name value
template	Use this tag (together with JESI `fragment` tags) when you are splitting a JSP page into separate cache fragments. The JESI `template` tag specifies caching behavior for the aggregate page, outside any fragments.	expiration maxRemovalDelay cache control
fragment	Use one or more JESI `fragment` tags within a JESI `template` tag, between the JESI `template` start-tag and end-tag, to denote separately cacheable fragments.	expiration maxRemovalDelay cache control

Table 1–9   (Cont.) Summary of JESI Tag LIbrary

Tag	Description	Attributes
codeblock	This is a subtag of the JESI `template` tag. You can use JESI `codeblock` tags to specify conditional execution of code blocks within the template code.	execute
invalidate	Use this tag, with its JESI `object` subtag, to explicitly invalidate one or more objects cached by OracleAS Web Cache.	url username password config output
object	Use this required subtag of the JESI `invalidate` tag to specify cached objects to invalidate, according to either the complete URI or a URI prefix.	uri prefix maxRemovalDelay
cookie	Optionally use this subtag of the JESI `object` tag to use cookie information as a further criterion for invalidation.	name value
header	Optionally use this subtag of the JESI `object` tag to use HTTP/1.1 header information as a further criterion for invalidation.	name value
personalize	Use this tag to allow page customization, by directing the ESI processor to perform cookie value substitution for every request for an object.	name default

Summary of Web Object Cache Tag Library

The OC4J Web Object Cache is a mechanism that allows Web applications written in Java to capture, store, reuse, post-process, and maintain the partial and intermediate results generated by a dynamic Web page, such as a JSP page or servlet. The programming interfaces it provides are a tag library (for use in JSP pages) and a Java API (for use in servlets).

Table 1–10 summarizes the Web Object Cache tag library. See "Web Object Cache Tag Descriptions" on page 7-14 for more information.

Table 1–10   Summary of Web Object Cache Tag LIbrary

Tag	Description	Attributes
cache	Use this tag to cache an object, for example a text fragment, in a JSP application. (Note, however, that there are separate tags for caching XML DOM objects or Java serializable objects.)	policy ignoreCache invalidateCache scope autoType selectedParam selectedCookies reusableTimeStamp reusableDeltaTime name expirationType TTL timeInaDay dayInaWeek dayInaMonth writeThrough printCacheBlockInfo printCachePolicy cacheRepositoryName reportException

Table 1–10   (Cont.) Summary of Web Object Cache Tag Library

Tag	Description	Attributes
cacheXMLObj	Generally speaking, use this tag instead of the `cache` tag if you are caching XML DOM objects. The `cacheXMLObj` tag supports all the `cache` tag attributes, as well as additional XML-specific parameters.	All attributes of the `cache` tag, plus: fromXMLObjName toXMLObjName toWriter
useCacheObj	Use this tag to cache any Java serializable object. The `useCacheObj` tag supports all the `cache` tag parameters, as well as additional attributes specific to its functionality.	All attributes of the `cache` tag, plus: type id cacheScope
cacheInclude	This tag combines functionality of the `cache` tag (but not the `cacheXMLObj` tag or `useCacheObj` tag) and the standard `jsp:include` tag.	policy page printCacheBlockInfo reportException
invalidateCache	Use this tag to programmatically invalidate a cache object. Most attributes of the `invalidateCache` tag behave the same way as attributes of the same names in the `cache` tag.	policy ignoreCache scope autoType selectedParam selectedCookies name invalidateNameLike page autoInvalidateLevel cacheRepositoryName reportException

Support for the JavaServer Pages Standard Tag Library

With Oracle Application Server 10*g* Release 2 (10.1.2), the OC4J JSP product includes an implementation of the JavaServer Pages Standard Tag Library (JSTL), as specified in the Sun Microsystems *JavaServer Pages Standard Tag Library, Version 1.0* specification. The following sections provide an overview of JSTL features and OC4J support:

- Overview and Philosophy of JSTL

- Summary of JSTL Expression Language

- Overview of JSTL Tags and Additional Features

- JSTL Usage Notes and Future Considerations

For complete information about JSTL, refer to the specification at the following location:

http://www.jcp.org/aboutJava/communityprocess/first/jsr052/index.html

---

Note: JSTL 1.0 requires a JSP 1.2 environment.

Overview and Philosophy of JSTL

JSTL is intended as a convenience for JSP page authors who are not familiar or not comfortable with scripting languages such as Java. Historically, scriptlets have been used in JSP pages to process dynamic data. With JSTL, the intent is for JSTL tag usage to replace the need for scriptlets.

Readers who have used previous versions of the OC4J JSP product will recognize this as similar to the goals of the Oracle JavaServer Pages Markup Language (JML) tag library. While the JML tag library is still supported, use of the standard JSTL is encouraged. Also see "JSTL Usage Notes and Future Considerations" on page 1-20.

Key JSTL features include the following:

- JSTL expression language (EL)

  The expression language further simplifies the code required to access and manipulate application data, making it possible to avoid request-time expressions as well as scriptlets. See the next section, "Summary of JSTL Expression Language".

- Core tags for expression language support, conditional logic and flow control, iterator actions, and access to URL-based resources

- Tags for XML processing, flow control, and XSLT transformations

- SQL tags for database access

- Tags for I18N-capable internationalization and formatting

  (The term "I18N" refers to an internationalization standard.)

Tag support is broken into four JSTL sublibraries according to the preceding functional areas. Table 1–11 shows the standard TLD URI and prefix for each sublibrary.

Table 1–11   JSTL Sublibraries

Functionality	URI	Prefix
Core	`http://java.sun.com/jstl/core`	c:
XML processing	`http://java.sun.com/jstl/xml`	x:
SQL database access	`http://java.sun.com/jstl/sql`	sql:
I18N internationalization and formatting	`http://java.sun.com/jstl/fmt`	fmt:

See "Overview of JSTL Tags and Additional Features" on page 1-17 for more information.

> **Note:** Given the constraints of having to work with JSP 1.2 containers, the JSTL 1.0 implementation was required to support both the expression language model and the request-time expression model. This dual support is accomplished through parallel JSTL sublibraries. For each sublibrary (core, XML, SQL, and I18N) there are separate TLDs, and hence separate TLD URIs, for the two versions.
>
> It is expected that most users will want to use the expression language model, corresponding to the URIs listed previously. To use the request-time expression model, add "_rt" to each URI in order to access the appropriate TLDs. By convention, add "_rt" to each prefix as well ("c_rt:", for example).

Summary of JSTL Expression Language

The JSTL expression language makes use of the fact that JSP scoped attributes and request parameters are the preferred vehicles for passing information to and from JSP

pages. By using the JSTL expression language, you can avoid having to use JSP scriptlets and request-time expressions.

In JSTL 1.0, the expression language can be used only in JSTL tag attribute values.

As an example, consider the following use of the JSTL `c:if` tag to pick out steel-making companies from a company list:

```
<c:if test="${company.industry == 'steel'}">
   ...
</c:if>
```

The rest of this section summarizes JSTL expression language syntax and documents how to enable JSTL expression language evaluation in your OC4J JSP applications.

JSTL Expression Language Syntax

This following list offers a brief summary of key syntax features of the JSTL expression language. This is followed by a few simple examples.

- Invocation

  The JSTL expression language is invoked through `${expression}` syntax. The most basic semantic is that invocation of a named variable `${foo}` yields the same result as the method call `PageContext.findAttribute(foo)`.

- Data structure access

  To access data within JavaBeans and within collections such as lists, maps, and arrays, the expression language supports the "`.`" and "`[]`" constructs. The "`.`" construct allows access to properties whose names are standard Java identifiers. The "`[]`" construct is for more generalized access, but for valid Java identifiers is equivalent to the "`.`" construct. The expressions `foo.bar` and `foo["bar"]` yield the same result, for example.

- Relational operators

  The expression language supports the relational operators `==` (or `eq`), `!=` (or `ne`), `<` (or `lt`), `>` (or `gt`), `<=` (or `le`), `>=` (or `ge`).

- Arithmetic operators

  The expression language supports the arithmetic operators `+`, `-`, `*`, `/` (or `div`), `%` (or `mod`, for remainder or modulo).

- Logical operators

  The expression language supports the logical operators `&&` (or `and`), `||` (or `or`), `!` (or `not`), `empty`.

Example: Basic The following example shows fairly basic invocations of the expression language, including the relational "`<=`" (less than or equal to) operator.

```
<c:if test="${auto.price <= customer.priceLimit}">
   The <c:out value="${auto.makemodel}"/> is in your price range.
</c:if>
```

Example: Accessing Collections The following example, from the Sun Microsystems *JavaServer Pages Standard Tag Library, Version 1.0* specification, shows use of the "`.`" and "`[]`" constructs:

```
<%-- "productDir" is a Map object containing the description of
     products, "preferences" is a Map object containing the
     preferences of a user --%>
```

```
product:
<c:out value="${productDir[product.custId]}"/>
shipping preference:
<c:out value="${user.preferences['shipping']}"/>
```

JSTL Expression Language Implicit Objects

JSTL offers the following implicit objects:

- `pageScope`: Allows access to page-scope variables.

- `requestScope`: Allows access to request-scope variables.

- `sessionScope`: Allows access to session-scope variables.

- `applicationScope`: Allows access to application-scope variables.

- `pageContext`: Allows access to all properties of the page context of a JSP page.

- `param`: This is a Java `Map` object where `param["foo"]` returns the first string value associated with the request parameter `foo`.

- `paramValues`: Use `paramValues["foo"]`, for example, to return an array of all string values associated with request parameter `foo`.

- `header`: Similarly to using `param`, you can use this to access the first string value associated with a request header.

- `headerValues`: Similarly to using `paramValues`, you can use this to access all string values associated with a request header.

- `initParam`: Allows access to context initialization parameters.

- `cookie`: Allows access to cookies received in the request.

JSTL Expression Language Additional Features

The expression language also offers the following features:

- It can provide default values where failure to evaluate an expression is considered to be recoverable.

- Where application data might not exactly match the type expected by a tag attribute or expression language operator, there are rules to convert the type of the resulting value to the expected type.

See the JSTL 1.0 specification for information.

Overview of JSTL Tags and Additional Features

The following sections provide a summary of JSTL tags and discuss some additional JSTL features:

- Scoped Variables

- Configuration Data and the Config Class

- JSTL Tag Summary

Scoped Variables

JSTL tags make data available through JSP scoped attributes, referred to as *scoped variables*, which are used in place of scripting variables. JSTL tags that can make data available in this way have `var` and `scope` among their attributes, used as follows:

- `var`: the variable that is to be exposed

- scope: the scope of the variable, either page (default), request, session, or application

The scope attribute would not be relevant for NESTED variables (which would always have page scope), but variables in the JSTL are AT_END (available from the end-tag to the end of the page).

The following example uses the core library iterator action tag forEach and expression language support tag out to expose the current item of an employees collection:

```
<c:forEach var="employee" items="${customers}">
   The current employee is <c:out value="${customer}"/>
</c:forEach>
```

Configuration Data and the Config Class

JSTL includes functionality to dynamically override JSP configuration data for a particular scope, through a scoped variable. You can accomplish this using functionality of the javax.servlet.jsp.jstl.core.Config class.

According to the JSP specification, all scopes (page, request, session, and application) that exist within a JSP page context should together form a single namespace; that is, the name of a scoped variable should be unique across execution of a page.

The Config class has functionality to transparently manipulate configuration parameter names to produce the effect that each scope has its own namespace. Effectively, this enables you to set a configuration parameter for a particular scope only.

See the JSTL 1.0 specification for information.

JSTL Tag Summary

Table 1–12 summarizes the JSTL tags, organized into functional groups. The JSTL standard tag prefix is noted for each group.

Table 1–12   Summary of JavaServer Pages Standard Tag Library

Tag Group	Description of Group	Individual Tags
Core, EL support	Includes tags to evaluate an expression and output the result to the current JspWriter object, set the value of a scoped variable or of a property of a target object, remove a scoped variable, and catch a Throwable instance thrown by a nested action.	c:out c:set c:remove c:catch
Core, conditional	Includes tags to evaluate body content if a test attribute evaluates as true, and specify mutually exclusive conditional execution paths. The when and otherwise tags are used with the choose tag.	c:if c:choose c:when c:otherwise
Core, iterators	Includes tags to iterate body execution over a collection of objects, or a specified number of times, and iterate over a set of tokens separated by supplied delimiters.	c:forEach c:forTokens

Table 1–12   (Cont.) Summary of JavaServer Pages Standard Tag Library

Tag Group	Description of Group	Individual Tags
Core, URL-related	Includes tags to import the content of a URL-based resource, create a URL using appropriate rewriting rules, send an HTTP redirect to the client, and add a request parameter to a URL. The param tag is a subtag of the import, url, and redirect tags.	c:import c:url c:redirect c:param
XML, core	Includes tags to parse an XML document, evaluate an XPath expression and output the result to the current JspWriter object, and evaluate an XPath expression and store the result in a scoped variable. (See the note after this table regarding XPath.)	x:parse x:out x:set
XML, flow control	Includes tags to evaluate a specified XPath expression and render its content if the expression evaluates as true, specify mutually exclusive conditional execution paths, and evaluate a specified XPath expression and repeat body execution over the result. The when and otherwise tags are used with the choose tag.	x:if x:choose x:when x:otherwise x:forEach
XML, transforms	Includes tags to apply an XSLT style sheet transformation to a document, and set transformation parameters. The param tag is a subtag of the transform tag.	x:transform x:param
SQL	Includes tags to query a database, update a database (UPDATE/INSERT/DELETE), establish a transaction context for queries and updates, export a data source as a scoped variable or data source configuration variable, set the values for parameter placeholders ("?") in a SQL statement, and set the values for parameter placeholders where the type is java.util.Date. The param and dateParam tags are subtags of the query and update tags.	sql:query sql:update sql:transaction sql:setDataSource sql:driver sql:param sql:dateParam
I18N, internationalization	Includes tags to store a specified locale in the locale configuration variable, create an I18N localization context for use within the tag, create a localization context and store it for use outside the tag, look up a localized message in a resource bundle, and set the request character encoding. The param tag can be used with the message tag to replace a parameter in the message tag.	fmt:locale fmt:bundle fmt:message fmt:param fmt:requestEncoding

Table 1–12   (Cont.) Summary of JavaServer Pages Standard Tag Library

Tag Group	Description of Group	Individual Tags
I18N, formatting	Includes tags to specify a time zone for formatting or parsing, store a specified time zone in a scoped variable or time zone configuration variable, format a numeric value as appropriate for a locale or special customization, parse the string representation of a numeric value that had been formatted for a locale or special customization, format a date or time for a locale or special customization, and parse the string representation of a date or time that had been formatted for a locale or special customization.	fmt:timeZone fmt:setTimeZone fmt:formatNumber fmt:parseNumber fmt:formatDate fmt:parseDate

> **Note:**   JSTL tags for XML processing are based on XPath (XML Path), a W3C recommendation. XPath provides a concise notation for specifying and selecting parts of an XML document. Refer to the following Web site for information:
>
> http://www.w3.org/TR/xpath

JSTL Usage Notes and Future Considerations

Be aware off the following considerations:

- The Oracle Application Server 10*g* Release 2 (10.1.2) JSTL implementation is based on the Jakarta 1.0.3 JSTL version and is suitable for use with OC4J. See the following location for more information about Jakarta:

  http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html

- The custom JML, XML, and data-access (SQL) tag libraries provided with OC4J pre-date JSTL and have areas of duplicate functionality. For standards compliance, it is now generally advisable to use JSTL instead of the custom libraries, and Oracle plans to desupport the JML library in a future release. For features in the custom libraries that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try to have the features adopted into the JSTL standard as appropriate.

- For the filter functionality of JSTL XML tags to work in OC4J, you must set the SAX driver when starting OC4J. Otherwise, the JSTL demo `Filter.jsp` gives the following exception:

  ```
  javax.servlet.jsp.JspException: System property org.xml.sax.driver not
  specified
  ```

  When starting OC4J standalone, use the following option in the command line:

  ```
  -Dorg.xml.sax.driver=oracle.xml.parser.v2.SAXParser
  ```

  When starting Oracle Application Server, you can specify this setting through the system properties.

Overview of Tag Libraries from Other Oracle Components

A number of other Oracle components, outside OC4J, provide JSP tag libraries. The following sections summarize these libraries:

- Oracle ADF Business Components Tag Library

- Oracle JDeveloper User Interface Extension (UIX) Tag Library

- Oracle ADF Business Components UIX Tag Library

- Oracle Application Server Wireless Location Tag Library

- Oracle Application Server MapViewer Tag Library

- Oracle Ultra Search Tag Library

- Oracle Application Server Portal Tag Library

- Oracle Business Intelligence Beans Tag Library

- Oracle Application Server Multimedia Tag Library

The Oracle tag libraries introduced in this section comply with JSP standards.

The following discussion assumes some prior knowledge of the underlying components.

Oracle ADF Business Components Tag Library

Oracle Application Development Framework (Oracle ADF), offered as part of Oracle JDeveloper, includes the component technology Oracle ADF Business Components. This includes a library of custom Business Components data tags.

These data tags provide a simple tag-based approach for interaction with Business Components data sources, allowing complete access to business components with viewing, editing, and full DML control capabilities. The tag-based approach to building JSP applications with Business Components does not require extensive Java programming and is very much like coding an HTML page.

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

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

Oracle JDeveloper User Interface Extension (UIX) Tag Library

Oracle JDeveloper provides a set of custom tags known as User Interface Extension (UIX) tags. The tags invoke UIX controls, generating the HTML to render tabs, buttons, tables, headers, and other layout and navigational components that implement the Oracle browser look and feel.

The tags are included on several palette pages: UIX JSP Border Layout, UIX JSP Form, UIX JSP Layout, UIX JSP Message Components, UIX JSP Page Layout, UIX JSP Simple Components, and UIX JSP Table.

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

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

Oracle ADF Business Components UIX Tag Library

UIX JSP pages can include both Business Components data tags and Business Components UIX convenience tags that simplify the presentation of data.

The Business Components UIX convenience tags rely on an `ApplicationModule` data tag to get the data source from the Business Components application module. In addition to the Business Components UIX tags, you can use the (non-UIX) Business Components tags in UIX JSP pages.

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

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

Oracle Application Server Wireless Location Tag Library

Developers of location-based applications need specialized services for the following:

- Geocoding: associating geographical coordinates with addresses
- Mapping: providing a graphical map for a point, set of points, route, or driving maneuver
- Routing: providing driving directions
- Business directories ("yellow pages"): listing businesses by region and by either category or name
- Traffic: providing information about accidents, construction, and other incidents that affect traffic flow

The OracleAS Wireless location application components compose an API for performing geocoding, providing driving directions, and looking up business directories. Service proxies are included that map existing key providers to the API, and additional providers are expected to be accommodated in the future.

For JSP developers, a tag library is provided. For more information, refer to the *Oracle Application Server Wireless Developer's Guide*.

Oracle Application Server MapViewer Tag Library

The OracleAS MapViewer is a programmable tool for rendering maps using spatial data managed by Oracle Spatial or Oracle Locator (also referred to as Locator). OracleAS MapViewer provides tools that hide the complexity of spatial data queries and cartographic rendering, while providing customizable options for more advanced users. These tools can be deployed in a platform-independent manner and are designed to integrate with map-rendering applications.

For convenience, OracleAS MapViewer includes a JSP tag library that you can use to submit map requests.

For more information, see the *Oracle Application Server MapViewer User's Guide*.

Oracle Ultra Search Tag Library

Oracle Ultra Search provides a custom tag library for use by developers in incorporating content search functionality into JSP applications. The library includes the following functionality:

- The ability to retrieve search attributes, groups, languages, and lists of values (LOVs) for rendering the advance query form

- The ability to iterate through the resulting hit set and retrieve document attributes and properties for rendering the result page

- The ability to perform a search with "relevance boosting" and an estimation of the total hit count

For more information, see the *Oracle Ultra Search User's Guide.* Alternatively, refer to the Oracle Ultra Search online documentation, under Oracle Ultra Search JSP Tag Library.

Oracle Application Server Portal Tag Library

With OracleAS Portal, developers can accomplish the following:

- Build and deploy Internet portals to deliver relevant information and applications to customers, employees, and partners.

- Develop portals rapidly, without code, using productive online tools.

- Increase user productivity with single sign-on and self-service publishing.

- Add value quickly with over 250 prebuilt portlets based on open standards.

The OracleAS Portal tag library provides further convenience for developers building customizable Internet portals. A developer can create internal JSP pages, which are stored inside the Portal database and downloaded when the portal is executed, or external JSP pages, which are stored in the file system, or some combination.

For more information, refer to *Oracle9i Application Server Portal: Adding JSPs*, available through the Oracle Technology Network:

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

Oracle Business Intelligence Beans Tag Library

The Oracle Business Intelligence Beans (OracleBI Beans) product consists of Java components, utilities, and a JSP tag library that enable rapid development of analytical applications. OracleBI Beans applications leverage the capabilities of OLAP in Oracle Database. Using OracleBI Beans, you can develop both HTML-client and Java-client applications.

> **Note:**   The OracleBI Beans product is a component of the Oracle Developer Suite (OracleDS) and is for use with Oracle JDeveloper.

OracleBI Beans includes the following groups of Java components:

- Presentation beans: beans that let you view, manipulate, and print data

- OLAP beans: beans that interact with an Oracle OLAP data source

- Persistence service: a set of Java packages that support the storage and retrieval of object definitions in the OracleBI Beans Catalog

For further developer convenience, OracleBI Beans includes a JSP tag library. You can use JDeveloper to create OracleBI Beans JSP pages. A JDeveloper wizard prompts you for information related to the tag that you want to use and inserts the coded tag in the JSP page.

For more information, refer to the OracleBI Beans online help. Under "Building Web Modules", click "Using JSP Tags", then "List of BI Beans JSP Tags".

Oracle Application Server Multimedia Tag Library

Oracle Application Server provides the Multimedia Tag Library, a custom JSP tag library for use by developers and Web page authors when generating multimedia HTML tags in JSP pages and uploading multimedia data into *inter*Media objects.

Oracle *inter*Media enables Oracle Database to store, retrieve, manage, and manipulate images, audio, video, and other media data, while integrating it with other enterprise information. Specifically, Oracle *inter*Media supports media storage, media retrieval, media management, and manipulation of media data managed by Oracle and stored in binary large objects, file-based large objects, URLs that contain media data, and specialty servers. Oracle *inter*Media is accessible to applications through relational and object interfaces.

Oracle *inter*Media uses object types that are similar to Java classes to describe media data. These *inter*Media objects have a common media data storage model. Oracle *inter*Media also provides Java classes to enable users to write Java applications using *inter*Media objects. There are also Oracle *inter*Media Java classes for servlets and JavaServer Pages to facilitate retrieving and uploading media data from and to Oracle Database instances.

The Multimedia Tag Library includes a set of tags for retrieving media data and a set for uploading media data. The Multimedia JSP tags for media retrieval include a set of common attributes and tag-specific media-render-attributes. The common attributes are: custom-retrieval-attributes, database-connection-attributes, media-access-attributes, media-cache-control-attributes, and table-and-column-attributes. The media-render-attributes are described with each media retrieval tag.

See *Oracle Application Server Multimedia Tag Library for JSP User's Guide and Reference* for additional information about the Multimedia JSP tags.

2

JavaBeans for Extended Types

This chapter describes JavaBeans provided with OC4J for use as extended types. For JSP pages, these types offer advantages over Java primitive types or `java.lang` types.

The chapter consists of the following sections:

- Overview of JML Extended Types
- JML Extended Type Descriptions

Overview of JML Extended Types

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

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

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

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

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

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

To work around these limitations, OC4J provides the following JavaBean classes in the `oracle.jsp.jml` package to act as wrappers for the most common Java types:

- `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.JmlNumber" scope="request" >
    <jsp:setProperty name="maxSize" property="value" value="<%= 25 %>" />
</jsp:useBean>
```

JML Extended Type Descriptions

This section documents the public methods of the four extended types—`JmlBoolean`, `JmlNumber`, `JmlFPNumber`, and `JmlString`—followed by an example.

> **Note:** To use the JML extended types, verify that the `ojsputil.jar` file is installed and in your classpath. This file is supplied with OC4J.

Type JmlBoolean

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

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

- `boolean getValue()`

- `void setValue(boolean)`

The `setTypedValue()` method has several signatures and can set the `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 `valueOf()` method of the `java.lang.Boolean` class.

- `void setTypedValue(String)`

- `void setTypedValue(Boolean)`

- `void setTypedValue(boolean)`

- `void setTypedValue(JmlBoolean)`

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

- `boolean equals(boolean)`

The `typedEquals()` method has several signatures and tests whether the `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 `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 `value` property of the bean as a Java `int` value.

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

The `setTypedValue()` method has several signatures and can set the `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 `decode()` method of the `java.lang.Integer` class.

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

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

- `boolean equals(int)`

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

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

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

- `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 `value` property of the bean as a Java `double` value.

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

The `setTypedValue()` method has several signatures and can set the `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 `valueOf()` method of the `java.lang.Double` class.

- `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 `value` property is equal to the specified Java `double` value.

- `boolean equals(double)`

The `typedEquals()` method has several signatures and tests whether the `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 `value` property as a `java.lang.String` value (such as "3.57"). This method has the same functionality as the `toString()` method of the `java.lang.Double` class.

- `String toString()`

Type JmlString

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

The `getValue()` and `setValue()` methods get or set the `value` property of the bean 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 `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 `value` property is an empty (zero-length) string: `""`

- `boolean isEmpty()`

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

- `boolean equals(String)`

- `boolean equals(JmlString)`

JML Extended Types Example

This example illustrates the use of JML extended type JavaBeans for management of simple types at scope. The page declares four session objects, one for each JML type. The page presents a form that enables 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>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>
```

3

JSP Markup Language Tags

This chapter documents the Oracle JSP Markup Language (JML) tag library, which provides a set of JSP tags to allow developers to script JSP pages without using Java statements. The JML library provides tags for variable declarations, control flow, conditional branches, iterative loops, parameter settings, and calls to objects.

The chapter consists of the following sections:

- Overview of the JSP Markup Language (JML) Tag Library
- JSP Markup Language (JML) Tag Descriptions

> **Note:** The library described here, which uses a standard runtime implementation, is also supported through an Oracle-specific compile-time implementation. The compile-time syntax and tags are documented in Appendix A, "JML Compile-Time Syntax and Tags". General considerations in using compile-time tags instead of runtime tags are discussed in the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide.*

Overview of the JSP Markup Language (JML) Tag Library

OC4J supplies the JSP Markup Language (JML) tag library, developed according to JSP standards. JML tags are intended to simplify coding syntax for JSP developers who are not proficient with Java. There are two main categories of JML tags: logic/flow control and bean binding.

These topics are covered in the following sections:

- JML Tag Library Philosophy
- JML Tag Categories

Note the following requirements for using JML tags:

- Verify that the file `ojsputil.jar` is installed and in your classpath. This file is provided with the OC4J installation, in the "well-known" tag library directory.

- The tag library descriptor, `jml.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `jml.tld` is the following:

  ```
  http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jml.tld
  ```

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

> **Note:** The custom JML tag library provided with OC4J pre-dates the JavaServer Pages Standard Tag Library (JSTL) and has areas of duplicate functionality. For standards compliance, it is now generally advisable to use JSTL instead. See "Support for the JavaServer Pages Standard Tag Library" on page 1-14.
>
> Oracle is not desupporting the existing library, however. For features in the custom library that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try to have the features adopted into the JSTL standard as appropriate.

JML Tag Library Philosophy

JavaServer Pages technology is intended for two separate developer communities:

- Those whose primary skill is Java programming
- Those whose primary skill is in designing static content, particularly in HTML, and who 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 feature set split into two functional categories, as summarized in Table 3–1.

Table 3–1   JML Tag Functional Categories

Tag Categories	Functionality	Tags
Bean binding tags	The purpose of these tags is to declare or undeclare a JavaBean at a specified JSP scope. See "Bean Binding Tag Descriptions" on page 3-3.	useVariable useForm useCookie remove
Logic/flow control tags	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 3-5.	if choose..when..[otherwise] foreach return flush

JSP Markup Language (JML) Tag Descriptions

The following sections document the JML tags that are supported in the current JSP runtime implementation:

- Bean Binding Tag Descriptions
- Logic and Flow Control Tag Descriptions

> **Notes:**
>
> - The prefix "jml:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.
>
> - See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

Bean Binding Tag Descriptions

The following sections document JML tags 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" ] />
```

Attributes

- `id` (required): Specifies the variable being declared.

- `scope`: Defines the duration or scope of the variable (as with a `jsp:useBean` tag). The default scope is `page`.

- `type` (required): Specifies the type of the variable. Type specifications refer to `JmlString`, `JmlBoolean`, `JmlNumber`, or `JmlFPNumber`.

- `value`: Use this to set the variable directly in the declaration, as either a string literal or a JSP expression enclosed in `<%=... %>` syntax. If this attribute is not specified, then 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` (required): Specifies the variable being declared or referenced.

- `scope`: Defines the duration or scope of the variable (as with a `jsp:useBean` tag). The default is `"page"`.

- `type`: Specifies the type of the variable. Type specifications refer to `JmlString`, `JmlBoolean`, `JmlNumber`, or `JmlFPNumber`. The default is `"string"`.

- `param` (required): Specifies the name of the request parameter whose value is used in setting the variable. 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 `JmlString` 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` (required): Specifies the variable being declared or referenced.

- `scope`: Defines the duration or scope of the variable. This attribute is optional; the default is `"page"`.

- `type`: Identifies the type of the variable. Type specifications refer to `JmlString`, `JmlBoolean`, `JmlNumber`, or `JmlFPNumber`. The default is `"string"`.

- `cookie` (required): Specifies the name of the cookie whose value is used in setting this variable. 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 `JmlString` 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, typically a bean, from its scope.

Syntax

```
<jml:remove id = "beanInstanceName"
          [ scope = "page" | "request" | "session" | "application" ] />
```

Attributes

- `id` (required): Specifies the name of the bean being removed.

- `scope`: Specifies the scope of the bean being removed. 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

The following sections document JML tags that 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 jml:if tag (executed if the condition is true)...
</jml:if>
```

Attributes

- `condition` (required): Specifies the conditional expression to be evaluated.

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>&nbsp
</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 the `when` conditions are all false, 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 jml:when tag (executed if the condition is true)...
    </jml:when>
    ...
    [...optional additional when tags...]
    [ <jml:otherwise>
        ...body of jml:otherwise tag (executed if all when conditions false)...
    </jml:otherwise> ]
</jml:choose>
```

Attributes The `when` tag uses the following attribute:

- `condition` (required): Specifies the conditional expression to be evaluated.

The `choose` and `otherwise` tags have no attributes.

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` attribute 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 jml:for tag (executed once at each value of range, inclusive)...
</jml:for>
```

Attributes

- `id` (required): This is the name of 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.

- `from` (required): Specifies the start of the range. This is an expression that must evaluate to a Java `int` value.

- `to` (required): Specifies the end of the range. This is an expression that must evaluate to a Java `int` value.

Example The following example repeatedly prints "Hello World" in progressively smaller headings (H1, H2, H3, H4, H5).

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

JML foreach Tag

This tag provides the ability to iterate over a homogeneous set of values. The body of the tag is executed once for each 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 jml:foreach tag (executes once for each element in data structure)...
</jml:foreach>
```

Attributes

- `id` (required): This is the name of 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.

- `in` (required): Specifies a JSP expression that evaluates to a Java array, `Enumeration` object, or `Vector` object containing the set of values over which to iterate.

- `limit` (required): 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.

- `type` (required): Specifies the type of the loop variable. This should match the type of the set elements, as applicable.

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>
```

Alternatively, 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); %>
```

4

Data-Access JavaBeans and Tags

This chapter describes JavaBeans and tags provided with OC4J for use in accessing a database from servlets and JSP pages.

The chapter consists of the following sections:

- JavaBeans for Data Access
- SQL Tags for Data Access

JavaBeans for Data Access

The OC4J product includes a set of JavaBeans that you can use to access a database. The following sections describe the beans:

- Introduction to Data-Access JavaBeans
- Data-Access Support for Data Sources and Pooled Connections
- Data-Access JavaBean Descriptions

> **Note:** The JavaBeans described here are used by the tags discussed in "SQL Tags for Data Access" on page 4-11. Generally speaking, these beans and tags can be used with non-Oracle databases, assuming you have appropriate JDBC driver classes; however, numerous features described below, as noted, are Oracle-specific.

Introduction to Data-Access JavaBeans

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

- `ConnBean` opens a database connection. This bean also supports data sources and connection pooling. See "Data-Access Support for Data Sources and Pooled Connections" on page 4-2 for related information.

- `ConnCacheBean` uses the Oracle JDBC connection caching implementation for database connections. This requires JDBC 2.0.

- `DBBean` executes a database query. It also has its own connection mechanism, but does not support data sources.

- `CursorBean` provides general DML support for queries; `UPDATE`, `INSERT`, and `DELETE` statements; and stored procedure calls.

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

To use the data-access JavaBeans, verify that the file `ojsputil.jar` is installed and in your classpath. This file is provided with the OC4J installation. For XML-related methods and functionality, you will also need the file `xsu12.jar` (for JDK 1.2.x or higher), which is provided with Oracle Application Server.

You will also need appropriate JDBC driver classes installed and in your classpath, such as `ojdbc14.jar` for Oracle Database and JDK 1.4.

> **Note:** The Oracle data-access JavaBeans implement the Oracle `JspScopeListener` interface for event notification. Refer to "JSP Event-Handling with JspScopeListener" on page 9-1 for information about this interface.

Data-Access Support for Data Sources and Pooled Connections

The data-access JavaBeans, as well as the data-access tag library, support the use of data sources to specify connection properties. This is also how support for connection pooling is implemented. This mechanism supports both Oracle connection objects and OC4J connection objects.

To use a data source in a JSP page, you must define the data source, its JNDI name, and its connection and pooling properties. In OC4J, do this in a `<data-source>` element in the `data-sources.xml` file. Here is an example:

```
<data-source
    class="oracle.jdbc.pool.OracleDataSource"
    name="jdbc/ejbpool/OracleDS"
    location="jdbc/ConnectionDS"
    ejb-location="jdbc/ejbpool/OracleDS"
    url="jdbc:oracle:thin:@myhost:1521/myservice"
    username="scott"
    password="tiger"
    min-connections="3"
    max-connections="50"
    wait-timeout="10"
    inactivity-timeout="30" />
```

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

Data-Access JavaBean Descriptions

The following sections describe attributes and methods of the data-access JavaBeans—`ConnBean`, `ConnCacheBean`, `DBBean`, and `CursorBean`—and concludes with an example that uses a data source:

- ConnBean for a Database Connection
- ConnCacheBean for Connection Caching
- DBBean for Queries Only
- CursorBean for DML and Stored Procedures
- Example: Using ConnBean and CursorBean with a Data Source

ConnBean for a Database Connection

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

> **Note:** For queries only, if you do not require a data source, it is simpler to use `DBBean`, which has its own connection mechanism.

`ConnBean` has the following properties. The `user`, `password`, and `URL` properties are not required if you use a data source.

- `dataSource`: JNDI name for a data source location

  This is valid only for an environment that supports data sources. See "Data-Access Support for Data Sources and Pooled Connections" on page 4-2 for information about how to set up a data source in OC4J.

- `user`: user ID for database schema

- `password`: user password

- `URL`: database connection string

- `stmtCacheSize`: cache size for Oracle JDBC statement caching

  Setting `stmtCacheSize` enables Oracle JDBC statement caching.

- `executeBatch`: batch size for Oracle JDBC update batching

  Setting `executeBatch` enables Oracle JDBC update batching.

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

  Setting `preFetch` enables Oracle JDBC row prefetching.

- `commitOnClose`: "true" or "false" for whether to execute `commit` when the connection is closed

  The value of `commitOnClose` indicates whether an automatic `commit` should be executed when the connection is closed. A "true" setting results in a `commit`; a "false" setting results in a `rollback`. Prior to Oracle9*i*AS Release 2, an automatic `commit` was always executed, but in current releases the default is an automatic `rollback`. The `commitOnClose` property allows for backward compatibility to ease migration.

  Be aware that there can be an application-wide `commit-on-close` setting in the application `web.xml` file, but the setting of the `ConnBean` property is not automatically dependent on that setting. If a JSP pages uses `ConnBean` instead of a `dbOpen` tag, the value of the `commit-on-close` context parameter should be retrieved and then explicitly set as the `commitOnClose` value in the `ConnBean` instance. For reference, here is a sample `web.xml` entry that sets the `commit-on-close` context parameter:

```
<context-param>
   <param-name>commit-on-close</param-name>
   <param-value>true</param-value>
</context-param>
```

> **Note:** See the *Oracle Database JDBC Developer's Guide and Reference* for information about statement caching, update batching, and row prefetching.

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

- `void setDataSource(String)`

- `String getDataSource()`

- `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()`

- `void setCommitOnClose(String)`

- `String getCommitOnClose()`

> **Note:** 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.

Use the following methods to open and close a connection or to verify its status:

- `void connect()`

  Establish a database connection using ConnBean property settings.

- `void close()`

  Close the connection and any open cursors.

- `boolean isConnectionClosed()`

  Determine if the connection is closed.

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, you can specify the SQL operation in the `CursorBean` method call that executes the statement.

See "CursorBean for DML and Stored Procedures" on page 4-8 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. Refer to the *Oracle Database JDBC Developer's Guide and Reference* for information about connection caching.

> **Note:** To use data sources or simple connection objects, use `ConnBean` instead.

`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 use fewer than this number, 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. Refer to the *Oracle Database JDBC Developer's Guide and Reference* for information about Oracle JDBC statement caching features and limitations.

- `cacheScheme`: type of cache

  This is indicated by one of the following `int` constants.

  - `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, returning `null`, until connection objects have been released.

The `ConnCacheBean` class includes 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 *Oracle Database JDBC Developer's Guide and Reference*.

> **Note:** 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.

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.) Refer to the *Oracle Database JDBC Developer's Guide and Reference* for information about these features.

> **Note:** When you use `ConnCacheBean`, use normal `Connection` object functionality to create and execute statement objects (unlike the case with `ConnBean`).

DBBean for Queries Only

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

Notes:

- `DBBean` has its own connection mechanism but does not support data sources. If you require a data source, use `ConnBean` instead.

- Use `CursorBean` for any other DML operations (`UPDATE`, `INSERT`, `DELETE`, or stored procedure calls).

`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()`

Note: 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.

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 that contains 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 *Oracle Database JDBC Developer's Guide and Reference*.

> **Note:** To use connection caching, use `ConnCacheBean` and normal `Connection` object functionality. Do not use `CursorBean`.

`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 the result set

  This is indicated by one of the following `int` constants.

  - `TYPE_FORWARD_ONLY` (default): Use this for a result set that can scroll only forward (using the `next()` method) and cannot be positioned.

  - `TYPE_SCROLL_INSENSITIVE`: Use this for a result set that can scroll forward or backward and can be positioned, but is not sensitive to underlying data changes.

  - `TYPE_SCROLL_SENSITIVE`: Use this for a result set that can scroll forward or backward, can be positioned, and is sensitive to underlying data changes.

- `resultSetConcurrency`: updatability of the result set

  This is indicated by one of the following `int` constants.

  - `CONCUR_READ_ONLY` (default): Use this for a result set that is read-only (cannot be updated).

  - `CONCUR_UPDATABLE`: Use this for a result set that is updatable.

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`.

  > **Note:** 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.

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. Use the following methods to create the cursor and supply a connection in separate steps:

- `void create()`

- `void setConnBean(ConnBean)`

Alternatively, use the following method to combine the process into a single step:

- `void create(ConnBean)`

Set up the `ConnBean` object as described in

Use the following method to specify and execute a query (using a JDBC plain `Statement` object behind the scenes):

- `ResultSet executeQuery(String)`

  Input a string that contains 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. Use the following methods to create the cursor,

specifying a statement type as an integer and specifying a SQL statement as a string, and supply a connection:

- `void create(int, String)`

- `void setConnBean(ConnBean)`

Alternatively, use the following method to 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 4-3.

The `int` input takes 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.

Use the following method to execute the INSERT, UPDATE, or DELETE statement. You can ignore the `boolean` return value.

- `boolean execute()`

Alternatively, for update batching, use the following method, which returns the number of rows affected.

- `int executeUpdate()`

> **Note:** Specify the SQL operation either during statement creation or during statement execution, but not both. The `execute()` and `executeUpdate()` methods can optionally take a string to specify a SQL operation. This is also true of the `create()` method, as well as the `getCursorBean()` method in `ConnBean`.

Additionally, `CursorBean` supports Oracle JDBC functionality such as `registerOutParameter()` for callable statements, `setXXX()` methods for prepared statements and callable statements, and `getXXX()` methods for result sets and callable statements.

Use the following method to close the database cursor:

- `void close()`

Example: Using ConnBean and CursorBean with a Data Source

This following is a sample JSP page that uses `ConnBean` with a data source to open a connection, then uses `CursorBean` to execute a query.

```
<%@ page import="java.sql.*, oracle.jsp.dbutil.*" %>
<jsp:useBean id="cbean" class="oracle.jsp.dbutil.ConnBean" scope="session">
   <jsp:setProperty name="cbean" property="dataSource"
                    value="<%=request.getParameter("datasource")%>"/>
</jsp:useBean>
<% try {
     cbean.connect();
     String sql="SELECT ename, sal FROM scott.emp ORDER BY ename";
     CursorBean cb = cbean.getCursorBean (CursorBean.PREP_STMT, sql);
     out.println(cb.getResultAsHTMLTable());
     cb.close();
     cbean.close();
```

```
      } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println("<PRE>" + e + "</PRE>\n<P>"); }
%>
```

SQL Tags for Data Access

OC4J includes a set of tags you can use in JSP pages to execute SQL commands to access a database. The following sections describe the tags:

- Introduction to Data-Access Tags

- Data-Access Tag Descriptions

Note: The custom SQL tag library provided with OC4J pre-dates the JavaServer Pages Standard Tag Library (JSTL) and has areas of duplicate functionality. For standards compliance, it is now generally advisable to use JSTL instead. See "Support for the JavaServer Pages Standard Tag Library" on page 1-14.

Oracle is not desupporting the existing library, however. For features in the custom library that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try to have the features adopted into the JSTL standard as appropriate.

Introduction to Data-Access Tags

OC4J supplies a custom tag library for SQL functionality, consisting of the following tags:

- `dbOpen`: Open a database connection. This tag also supports data sources and connection pooling. See "Data-Access Support for Data Sources and Pooled Connections" on page 4-2 for related information.

- `dbClose`: Close a database connection.

- `dbQuery`: Execute a query.

- `dbCloseQuery`: Close the cursor for a query.

- `dbNextRow`: Process the rows of a result set.

- `dbExecute`: Execute any SQL statement (DML or DDL).

- `dbSetParam`: Set a parameter to bind into a `dbQuery` or `dbExecute` tag.

- `dbSetCookie`: Set a cookie.

Note the following requirements for using SQL tags:

- You will need the appropriate JDBC driver file, such as `ojdbc14.jar` for JDK 1.4, installed and in your classpath.

- Verify that the file `ojsputil.jar` is installed and in your classpath. This file is provided with the OC4J installation, in the "well-known" tag library directory.

- The tag library descriptor, `sqltaglib.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `sqltaglib.tld` is the following:

  ```
  http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/sqltaglib.tld
  ```

For general information about JSP tag library usage, including tag library descriptor files, taglib directives, the well-known tag library directory, and the meaning of uri values, refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide.*

Notes:

- The data-access tags use the beans described in "JavaBeans for Data Access" on page 4-1. Generally speaking, these beans and tags can be used with non-Oracle databases, assuming you have appropriate JDBC driver classes; however, numerous features described below, as noted, are Oracle-specific.

- For applications using the data-access tags, consider using the dbSetParam tag to supply only parameter values rather than textual completion of the SQL statement itself. This avoids the possibility of what is referred to as "SQL poisoning", where users might enter additional SQL in addition to the expected value.

Data-Access Tag Descriptions

The following sections provide detailed syntax for the data-access tags and an example using dbOpen and dbQuery tags with a data source:

- SQL dbOpen Tag
- SQL dbClose Tag
- SQL dbQuery Tag
- SQL dbCloseQuery Tag
- SQL dbNextRow Tag
- SQL dbExecute Tag
- SQL dbSetParam Tag
- SQL dbSetCookie Tag
- Example: Using dbOpen and dbQuery with a Data Source

Notes:

- The prefix "sql:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your taglib directive.

- See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

SQL dbOpen Tag

Use the dbOpen tag to open a database connection for subsequent SQL operations through such tags as dbQuery and dbExecute. Do this by specifying a data source location, in which case connection caches are supported, or by specifying the user, password, and URL individually. See "Data-Access Support for Data Sources and Pooled Connections" on page 4-2 for information about how to set up a data source in OC4J.

The implementation uses `oracle.jsp.dbutil.ConnBean` instances. For simple connections, but not connection caches, you can optionally set `ConnBean` properties such as `stmtCacheSize`, `preFetch`, and `batchSize` to enable those Oracle JDBC features. See "ConnBean for a Database Connection" on page 4-3 for more information.

The `ConnBean` object for the connection is created in an instance of the tag-extra-info class of the `dbOpen` tag. Refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about the standard JSP tag library framework and tag-extra-info classes.

Syntax

```
<sql:dbOpen
   [ connId = "connection_id" ]
   [ scope = "page" | "request" | "scope" | "application" ]
   [ dataSource = "JNDI_name" ]
   [ user = "username"
     password = "password"
     URL = "databaseURL" ]
   [ commitOnClose = "true" | "false" ] >

   ...

</sql:dbOpen>
```

Nested code that you want to execute through this connection can go into the tag body, between the `dbOpen` start-tag and end-tag.

> **Note:** You must set either the `dataSource` attribute or the `user`, `password`, and `URL` attributes. Optionally, you can use a data source to specify a URL, then use the `dbOpen` tag `user` and `password` attributes separately.
>
> When a data source is used, and is for a cache of connections, the first use of the cache initializes it. If you specify the user and password through the `dbOpen` tag `user` and `password` attributes, that will initialize the cache for that user and password. Subsequent uses of the cache are for the same user and password.

Attributes

- `connId`: Optionally use this to specify an ID name for the connection. You can then reference this ID in subsequent tags such as `dbQuery` or `dbExecute`. Alternatively, you can nest `dbQuery` and `dbExecute` tags inside the `dbOpen` tag. You can also reference the connection ID in a `dbClose` tag when you want to close the connection.

  You can still specify a connection ID if you nest `dbQuery` or `dbExecute` tags inside the `dbOpen` tag. In this case, the connection will be found through the connection ID. With the `scope` attribute, it is possible to have multiple connections using the same connection ID but different scopes.

  If you specify a connection ID, then the connection is not closed until you close it explicitly with a `dbClose` tag. Without a connection ID, the connection is closed automatically when the `dbOpen` end-tag is encountered.

- `scope` (used only with a `connId`): Use this to specify the desired scope of the connection instance. The default is `page` scope.

If you specify a scope setting in a `dbOpen` tag, then you must specify the same scope setting in any other tag—`dbQuery`, `dbExecute`, or `dbClose`—that uses the same connection ID.

- `dataSource` (required if you do not set the `user`, `password`, and `URL` attributes): Optionally use this to specify the JNDI name of a data source for database connections. First set up the data source in the OC4J `data-sources.xml` file. (See "Data-Access Support for Data Sources and Pooled Connections" on page 4-2.) The `dataSource` setting should correspond to the `location` name, `ejb-location` name, or `pooled-location` name in a `<data-source>` element in `data-sources.xml`.

  A data source must specify a URL setting, but does not have to specify a user/password pair. You can use the `dbOpen` tag `user` and `password` attributes instead.

  This attribute is supported only in OC4J environments.

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

- `user` (required if no user/password pair is specified through a data source): This is the user name for a database connection.

  If a user name is specified through both a data source and the `user` attribute, the `user` attribute takes precedence. It is advisable to avoid such duplication, because conflicts could arise if the data source is a pooled connection with existing logical connections using a different user name.

- `password` (required if no user/password pair is specified through a data source): This is the user password for a database connection.

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

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

  As with the `user` attribute, if a password is specified through both a data source and the `password` attribute, the `password` attribute takes precedence.

- `URL` (required if no data source is specified): This is the URL for a database connection. If a URL is supplied through a data source, the `dbOpen` tag `URL` attribute is ignored.

- `commitOnClose`: Set this to `"true"` for an automatic SQL commit when the connection is closed or goes out of scope. The default `"false"` setting results in an automatic SQL rollback.

  As a convenience, if you want to specify application-wide automatic `commit` or `rollback` behavior, set the parameter name `commit-on-close` in the application `web.xml` file, as in the following example:

```
<context-param>
   <param-name>commit-on-close</param-name>
   <param-value>true</param-value>
</context-param>
```

The `commitOnClose` setting in a `dbOpen` tag takes precedence over the `commit-on-close` setting in `web.xml`.

> **Note:** In previous releases, the behavior is always to commit automatically when the connection is closed. The `commitOnClose` attribute offers backward compatibility to simplify migration.

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; a `dbClose` tag is not required.

Note that by using the `JspScopeListener` utility provided with OC4J, you can have the connection closed automatically with session-based event-handling. Refer to "JSP Event-Handling with JspScopeListener" on page 9-1 for information.

Syntax

```
<sql:dbClose connId = "connection_id"
             [ scope = "page" | "request" | "scope" | "application" ] />
```

Attributes

- `connId` (required): This is the ID for the connection being closed, specified in the `dbOpen` tag that opened the connection.

- `scope`: This is the scope of the connection instance. The default is "`page`", but if the `dbOpen` tag specified a scope other than `page`, you must specify that same scope in the `dbClose` tag.

SQL dbQuery Tag

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

This tag uses an `oracle.jsp.dbutil.CursorBean` object for the cursor, so you can set properties such as the result set type, result set concurrency, batch size, and prefetch size, if desired. See "CursorBean for DML and Stored Procedures" on page 4-8 for information about `CursorBean` functionality.

For XML usage, this tag acts as an XML producer. See "XML Producers and XML Consumers" on page 5-1 for more information. Also see "Example Using the transform and dbQuery Tags" on page 5-7.

Syntax

```
<sql:dbQuery
             [ queryId = "query_id" ]
             [ connId = "connection_id" ]
             [ scope = "page" | "request" | "scope" | "application" ]
             [ output = "HTML" | "XML" | "JDBC" ]
             [ maxRows = "number" ]
             [ skipRows = "number" ]
             [ bindParams = "value" ]
             [ toXMLObjName = "objectname" ] >
```

```
...SELECT statement (one only)...

</sql:dbQuery>
```

> **Important:**
>
> - Do *not* terminate the SELECT statement with a semicolon. This currently results in a syntax error.
>
> - The dbQuery tag does not currently support LOB columns.

Attributes

- queryId: You can use this to specify an ID name for the cursor. This is required if you want to process the results using a dbNextRow tag.

  If the queryId parameter is present, then the cursor is not closed until you close it explicitly with a dbCloseQuery tag. Without a query ID, the cursor is closed automatically when the dbQuery end-tag is encountered. This is *not* a request-time attribute, meaning it cannot take a JSP expression value.

- connId: This is the ID for a database connection, according to the connId setting in the dbOpen tag that opened the connection. If you do not specify connId in a dbQuery tag, then the tag must be nested within the body of a dbOpen tag and will use the connection opened in the dbOpen tag. This is *not* a request-time attribute.

- scope: This is the scope of the connection instance. The default is "page", but if the associated dbOpen tag specified a scope other than page, you must specify that same scope in the dbQuery tag. This is *not* a request-time attribute.

- output: This is the desired output format, one of the following.

  – HTML specifies that the result set is to be output as an HTML table (default).

  – XML specifies that the result set is to be output as an XML string, or an XML DOM object if an object name is specified in the toXMLObjName attribute.

  – JDBC specifies that the result set is to be output as a JDBC ResultSet object that can be processed using the dbNextRow tag to iterate through the rows.

- maxRows: This is the maximum number of rows of data to display. The default is all rows.

- skipRows: This is the number of data rows to skip in the query results before displaying results. The default is 0.

- bindParams: Use this to bind a parameter into the query. The following example is from an application that prompts the user to enter an employee number, using bindParams to bind the specified value into the empno field of the query:

  ```
  <sql:dbQuery connId="con1" bindParams="empno">
      select * from EMP where empno=?
  </sql:dbQuery>
  ```

  Alternatively, you can set a parameter value with the dbSetParam tag to bind it in through the bindParams attribute. See "SQL dbSetParam Tag" on page 4-18.

- toXMLObjName: Specify an XML object name if you want to output the results as an XML DOM object. To use this, you must also set output to "XML".

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 specified in the `dbQuery` tag, then the cursor is closed automatically when the `dbQuery` end-tag is reached; a `dbCloseQuery` tag is not required.

Syntax

```
<sql:dbCloseQuery queryId = "query_id" />
```

Attributes

- `queryId` (required): The ID for the cursor to be closed, specified in the `dbQuery` tag that opened the cursor.

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-tag and end-tag. The body is executed for each row of the result set.

To use the `dbNextRow` tag, the `dbQuery` tag must set `output` to `"JDBC"` and specify a `queryId` for the `dbNextRow` tag to reference.

The result set object is created in an instance of the tag-extra-info class of the `dbQuery` tag. Refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about the standard JSP tag library framework and tag-extra-info classes.

Syntax

```
<sql:dbNextRow queryId = "query_id" >
...Row processing...
</sql:dbNextRow >
```

Attributes

- `queryId` (required): This is the ID of the cursor containing the results to be processed, specified in the `dbQuery` tag that opened the cursor.

Example The following example shows the combined use of a `dbOpen`, `dbQuery`, and `dbNextRow` tag.

```
<sql:dbOpen connId="con1" URL="jdbc:oracle:thin:@myhost:1521/myservice"
            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" />
```

SQL dbExecute Tag

Use the `dbExecute` tag to execute a single DML or DDL statement. Place the statement in the tag body, between the `dbExecute` start-tag and end-tag.

This tag uses an `oracle.jsp.dbutil.CursorBean` object for the cursor. See "CursorBean for DML and Stored Procedures" on page 4-8 for information about `CursorBean` functionality.

Syntax

```
<sql:dbExecute
          [ connId = "connection_id" ]
          [ scope = "page" | "request" | "scope" | "application" ]
          [ output = "yes" | "no" ]
          [ bindParams = "value" ] >

    ...DML or DDL statement (one only)...

</sql:dbExecute >
```

Important:

- Do *not* terminate the DML or DDL statement with a semicolon. This currently results in a syntax error.

- The `dbExecute` tag does not currently support LOB columns.

Attributes

- `connId`: This is the ID of a database connection, according to the `connId` setting in the `dbOpen` tag that opened the connection. If you do not specify `connId` in a `dbExecute` tag, then the tag must be nested within the body of a `dbOpen` tag and will use the connection opened in the `dbOpen` tag.

- `scope`: This is the scope of the connection instance. The default is "page", but if the `dbOpen` tag specified a scope other than `page`, you must specify that same scope in the `dbExecute` tag.

- `output`: 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 is "no".

- `bindParams`: Use this to bind a parameter into the SQL statement. The following example is from an application that prompts the user to enter an employee number, using `bindParams` to bind the specified value into the `empno` field of the `DELETE` statement:

```
<sql:dbExecute connId="con1" bindParams="empno">
     delete from EMP where empno=?
</sql:dbExecute>
```

  Alternatively, you can set a parameter value with the `dbSetParam` tag to bind it in through the `bindParams` attribute. See the next section, "SQL dbSetParam Tag".

SQL dbSetParam Tag

You can use this tag to set a parameter value to bind into a query, through the `dbQuery` tag, or to bind into any other SQL operation, through the `dbExecute` tag.

> **Note:** For applications using the data-access tags, consider using the dbSetParam tag to supply only parameter values rather than textual completion of the SQL statement itself. This avoids the possibility of what is referred to as "SQL poisoning", where users might enter more SQL code in addition to the expected value.

Syntax

```
<sql:dbSetParam name = "param_name"
                value = "param_value"
              [ scope = "page" | "request" | "scope" | "application" ] />
```

Attributes

- name (required): This is the name of the parameter to set.

- value (required): This is the desired value of the parameter.

- scope: This is the scope of the bind parameter. The default is page scope.

Example The following example uses a dbSetParam tag to set the value of a parameter named id2. This value is then bound into the SQL statement in the dbExecute tag.

```
<sql:dbSetParam name="id2" value='<%=request.getParameter("id")%>'
                scope="session" />
Result:
   <HR>
   <sql:dbOpen dataSource="<%= dataSrcStr %>" >
       <sql:dbExecute output="yes" bindParams="id2 name job sal">
            insert into emp(empno, ename, deptno, job, sal)
                   values (?, ?, 20, ?, ?)
       </sql:dbExecute>
   </sql:dbOpen>
   <HR>
```

SQL dbSetCookie Tag

You can use this tag to set a cookie. The dbSetCookie tag wraps functionality of the standard javax.servlet.http.Cookie class.

Syntax

```
<sql:dbSetCookie name = "cookie_name"
              [ value = "cookie_value" ]
              [ domain = "domain_name" ]
              [ comment = "comment" ]
              [ maxAge = "age" ]
              [ version = "protocol_version" ]
              [ secure = "true" | "false" ]
              [ path = "path" ] />
```

Attributes

- name (required): This is the name of the cookie.

- value: This is the desired value of the cookie. Because it is permissible to have a null-value cookie, this attribute is not required.

- domain: This is the domain name for the cookie. The form of the domain name is according to the RFC 2019 specification.

- comment: This is for a comment describing the purpose of the cookie.

- maxAge: This is the maximum allowable age of the cookie, in seconds. Use a setting of "-1" for the cookie to persist until the browser is shut down.

- version: This is the version of the HTTP protocol that the cookie complies with.

- secure: This informs the browser whether the cookie should be sent using a secure protocol, such as HTTPS.

- path: This specifies a file system path for the cookie, the location to which the client should return the cookie.

Example

```
<sql:dbSetCookie name="cId" value='<%=request.getParameter("id")%>'
                 maxAge='800000' />
```

Example: Using dbOpen and dbQuery with a Data Source

This section provides a sample JSP page that uses a dbOpen tag with a data source to open a connection, then uses a dbQuery tag to execute a query.

```
<%@ taglib uri="
http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/
sqltaglib.tld" prefix="sql" %>
<HTML>
<BODY>
   <sql:dbOpen dataSource='<%=request.getParameter("datasource") %>'
              connId="con1">
   </sql:dbOpen>
   <sql:dbQuery connId="con1">
      SELECT * FROM emp ORDER BY ename
   </sql:dbQuery>
   <sql:dbClose connId="con1" />
</BODY>
</HTML>
```

5

XML and XSL Tag Support

This chapter describes tags provided with OC4J that you can use for XML data and XSL transformation, and summarizes additional XML functionality in other OC4J tags. These tags are implemented according to the JSP specification.

The chapter consists of the following sections:

- Overview of Oracle Tags for XML Support
- XML Utility Tags

> **Note:** See the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for additional information about XML-related functionality for JSP pages.

Overview of Oracle Tags for XML Support

The following sections provide an overview of tags supplied with OC4J that have XML functionality. This includes tags that can take XML DOM objects as input, generate XML DOM objects as output, transform XML documents according to a specified stylesheet, and parse data from an input stream to an XML DOM object.

- XML Producers and XML Consumers
- Summary of OC4J Tags with XML Functionality

> **Note:** The custom XML tag library provided with OC4J pre-dates the JavaServer Pages Standard Tag Library (JSTL) and has areas of duplicate functionality. For standards compliance, it is now generally advisable to use JSTL instead. See "Support for the JavaServer Pages Standard Tag Library" on page 1-14.
>
> Oracle is not desupporting the existing library, however. For features in the custom library that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try to have the features adopted into the JSTL standard as appropriate.

XML Producers and XML Consumers

An XML-related operation can be classified as either of the following, or as both:

- *XML producer*, which outputs an XML object
- *XML consumer*, which takes an XML object as input

Similarly, an XML-related tag can be classified as an XML producer, an XML consumer, or both. XML producers can pass XML objects to XML consumers either explicitly or implicitly; the latter is also known as *anonymous passing*.

For explicit passing between XML-related tags, there is a `toXMLObjName` attribute in the producer tag and a `fromXMLObjName` attribute in the consumer tag. Behind the scenes, the passing is done through the `getAttribute()` and `setAttribute()` methods of the standard JSP `pageContext` object. The following example uses explicit passing:

```
<sql:dbQuery output="XML" toXMLObjName="foo" ... >
   ...SQL query...
</sql:dbQuery>
...
<ojsp:cacheXMLObj fromXMLObjName="foo" ... />
```

For implicit passing between XML-related tags, do not use the `toXMLObjName` and `fromXMLObjName` attributes. The passing is accomplished through direct interaction between the tag handlers, typically in a situation with a nested tag. The following example uses implicit passing:

```
<ojsp:cacheXMLObj ... >
   <sql:dbQuery output="XML" >
       ...SQL query...
   </sql:dbQuery>
</ojsp:cacheXMLObj>
```

Here, the XML produced in the `dbQuery` tag is passed to the `cacheXMLObj` tag directly, without being stored to the `pageContext` object.

For a tag to be able to function as a consumer with implicit passing, the tag handler implements the OC4J `ImplicitXMLObjConsumer` interface:

```
interface ImplicitXMLObjConsumer
{
   void setImplicitFromXMLObj();
}
```

Summary of OC4J Tags with XML Functionality

For the tag libraries supplied with OC4J, Table 5–1 summarizes the tags that can function as XML producers or consumers.

Table 5–1   OC4J Tags with XML Functionality

Tag	Library	Producer / Consumer	Related Attributes	Tag Information
transform / styleSheet	XML	Both	fromXMLObjName toXMLObjName	"XML transform and styleSheet Tags for XML/XSL Data Transformation" on page 5-4
parsexml	XML	Producer	toXMLObjName	"XML parsexml Tag to Convert from Input Stream" on page 5-5
cacheXMLObj	Web Object Cache	Both	fromXMLObjName toXMLObjName	"Web Object Cache cacheXMLObj Tag" on page 7-18
dbQuery	SQL	Producer	toXMLObjName	"SQL dbQuery Tag" on page 4-15

Table 5–1   (Cont.) OC4J Tags with XML Functionality

Tag	Library	Producer / Consumer	Related Attributes	Tag Information
invoke	Web Services	Producer	toXMLObjName	"Web Services invoke Tag" on page 10-13

> **Notes:**
>
> - The XML `transform` and `styleSheet` tags are equivalent and produce identical results.
>
> - For convenience, the `cacheXMLObj` tag is defined in the XML tag library descriptor file (`xml.tld`) as well as the Web Object Cache tag library descriptor file (`jwcache.tld`).

XML Utility Tags

The following sections describe XML utility tags supplied with OC4J:

- XML Utility Tag Descriptions

- XML Utility Tag Examples

Note the following requirements for the XML utility tag library:

- The XML tag library requires `ojsputil.jar`, `xmlparserv2.jar`, and `xsu12.jar` to be installed and in your classpath. These files are supplied with OC4J. The `ojsputil.jar` file is located in the "well-known" tag library directory.

- The tag library descriptor, `xml.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `xml.tld` is the following:

```
http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/xml.tld
```

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

> **Notes:**
>
> - The prefix "xml:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.
>
> - See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

XML Utility Tag Descriptions

The following sections describe XML utility tags:

- XML transform and styleSheet Tags for XML/XSL Data Transformation

- XML parsexml Tag to Convert from Input Stream

> **Important:** Tag attributes are request-time attributes, meaning they can take JSP expressions as input, unless otherwise noted.

XML transform and styleSheet Tags for XML/XSL Data Transformation

Many uses of XML and XSL for dynamic JSP pages require an XSL transformation to occur in the server before results are returned to the client. Oracle provides two synonymous tags in the XML library to simplify this process. You can output the result directly to the HTTP client or, alternatively, you can output to a specified XML DOM object. Use either the `transform` tag or the `styleSheet` tag, as described and shown in this section. The two tags have identical effects.

Each tag acts as both an XML producer and an XML consumer. They can take as input either of the following:

- An XML DOM object

- The tag body, containing JSP commands and static text that produce the XML code

The tags can output to either or both of the following, with the specified stylesheet being applied in either case:

- An XML DOM object

- The output writer to the browser, in which case the specified stylesheet is applied

When you use the tag body for input, the tag applies to what is between the start-tag and end-tag. You can have multiple XSL transformation blocks within a page, with each block bounded by its own `transform` or `styleSheet` tag, specifying its own `href` pointer to the appropriate style sheet.

Syntax

```
<xml:transform href="xslRef"
               [ fromXMLObjName = "objectname" ]
               [ toXMLObjName = "objectname" ]
               [ toWriter = "true" | "false" ] >

   [...body...]

</xml:transform >
```

or:

```
<xml:styleSheet href="xslRef"
               [ fromXMLObjName = "objectname" ]
               [ toXMLObjName = "objectname" ]
               [ toWriter = "true" | "false" ] >

   [...body...]

</xml:styleSheet >
```

Attributes

- `href` (required): Specify the XSL stylesheet to use for the XML data transformation. This is required whether you are outputting to an XML object (where you can have transformation without formatting) or to the browser.

  Note the following regarding the `href` attribute:

- 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/path`), an application-relative JSP reference (starting with "`/`"), or a page-relative JSP reference (not starting with "`/`"). Refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about application-relative and page-relative paths.

- Its value can be a static Java string constant literal, or it can be dynamically specified through a standard JSP request-time expression.

- `fromXMLObjName`: Use this to specify an input XML DOM object if input is from a DOM object instead of from the tag body. If there is both a tag body and a `fromXMLObjName` specification, `fromXMLObjName` takes precedence.

- `toXMLObjName`: Use this to specify the name of an output XML DOM object if output is to a DOM object, instead of or in addition to going to the JSP writer object for output to the HTTP client. This is not required if there is an implicit XML consumer, such as a tag within which the `transform` or `styleSheet` tag is nested.

- `toWriter`: This is "`true`" or "`false`" to indicate whether output goes to the JSP writer object for output to the HTTP client. This can be instead of or in addition to output to a DOM object. The default is "`true`", for backward compatibility. (Prior to Oracle9*i*AS Release 2, this was the only output choice; there was no `toXMLObjName` attribute.)

XML parsexml Tag to Convert from Input Stream

The XML tag library supplies an XML producer utility tag, `parsexml`, that converts from an input stream to an XML DOM object. This tag can take input from a specified resource or from the tag body.

Syntax

```
<xml:parsexml
            [ resource = "xmlresource" ]
            [ toXMLObjName = "objectname" ]
            [ validateResource = "dtd_path" ]
            [ root = "dtd_root_element" ] >

   [...body...]

</xml:parsexml >
```

Attributes

- `resource`: Use this to specify an XML resource if input is from a resource instead of from the tag body. For example:

  ```
  resource="/dir1/hello.xml"
  ```

  If there is both a tag body and a specified resource, the resource takes precedence.

- `toXMLObjName`: Specify the name of the XML DOM object where the output will go. This is not required if there is an implicit XML consumer, such as a tag within which the `parsexml` tag is nested.

- `validateResource`: For XML validation, you can specify the path to the appropriate DTD. Alternatively, the DTD can be embedded in the XML resource. This is *not* a request-time attribute.

- `root`: If validating, specify the root element in the DTD for validation. This is *not* a request-time attribute. If you specify `validateResource` without specifying `root`, the default root is the top-level of the DTD.

XML Utility Tag Examples

The following sections provide examples that use XML utility tags:

- Example Using the transform Tag
- Example Using the transform and dbQuery Tags
- Examples Using the transform and parsexml Tags

Example Using the transform Tag

This section provides a sample XSL stylesheet and a sample JSP page that uses the `transform` tag to filter its output through the stylesheet. This is a simplistic example, with the XML in the page being 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="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/xml.tld"
```

```
            prefix="xml" %>

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

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

</xml:transform>
```

This example results in the following output:

Example Using the transform and dbQuery Tags

This example returns a result set from a dbQuery tag, using a transform tag to filter the query results through the XSL style sheet rowset.xsl (code below). It uses a dbOpen tag to open a connection, with the connection string being obtained either from the request object or through the useDataSource.jsp page (code below). Data passing from the dbOpen tag to the transform tag is done implicitly. For related information, see "SQL dbQuery Tag" on page 4-15 and "SQL dbOpen Tag" on page 4-12.

JSP Page

```
<%@ page import="oracle.sql.*, oracle.jdbc.driver.*, oracle.jdbc.*, java.sql.*" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/xml.tld"
           prefix="xml" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/sqltaglib.tld"
           prefix="sql" %>

<%
   String dataSrcStr=request.getParameter("dataSrcStr");
   if (dataSrcStr==null) {
     dataSrcStr=(String)session.getValue("dataSrcStr");
   } else {
     session.putValue("dataSrcStr",dataSrcStr);
   }
   if (dataSrcStr==null) { %>
<jsp:forward page="../../sql/useDataSource.jsp" />
<%
   }
```

```
%>
<h3>Transform DBQuery Tag Example</h3>
<xml:transform href="style/rowset.xsl" >
    <sql:dbOpen connId="conn1" dataSource="<%= dataSrcStr %>" />
    <sql:dbQuery connId="conn1" output="xml" queryId="myquery" >
       select ENAME, EMPNO from EMP order by ename
    </sql:dbQuery>
    <sql:dbCloseQuery queryId="myquery" />
    <sql:dbClose connId="conn1" />
</xml:transform>
```

> **Note:** For the dbOpen tag in this example, assume that the data
> source specifies the user name and password as well as the URL.

rowset.xsl

```
<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:template match="ROWSET">
 <html><body>
 <h1>A Simple XML/XSL Transformation</h1>
 <table border="2">
<xsl:for-each select="ROW">
  <tr>
    <td><xsl:value-of select="@num"/></td>
    <td><xsl:value-of select="ENAME"/></td>
    <td><xsl:value-of select="EMPNO"/></td>
  </tr>
</xsl:for-each>
</table>
</body></html>
</xsl:template>
</xsl:stylesheet>
```

useDataSource.jsp

```
<body bgcolor="#FFFFFF">
<font size=+0>
<B>Please enter a suitable JDBC connection string, before you try the above
demo</B>
<pre>
    To use a data source that you have set up in data-sources.xml, enter the
    data source string below. Once you have set the data source string it
    will remain in effect until the session times out.
</pre>
<%
   String dataSrcStr;
   dataSrcStr=request.getParameter("dataSrcStr");
   if (dataSrcStr==null) {
     dataSrcStr=(String)session.getValue("dataSrcStr");
   }
   if (dataSrcStr==null) {
     dataSrcStr="jdbc/OracleCoreDS";  // default data source string
   }

   session.putValue("dataSrcStr",dataSrcStr);
%>
<FORM METHOD=get ACTION="<%= request.getParameter("nextaction") %>" >
```

```
<INPUT TYPE="text" NAME="dataSrcStr" SIZE=40 value="<%=dataSrcStr%>" >
<INPUT TYPE="submit" VALUE="Change Data Source String" >
</FORM>
</font>
```

Examples Using the transform and parsexml Tags

This section provides two examples that take output from a parsexml tag and filter it through a transform tag, using the XSL stylesheet email.xsl. In each case, data is collected by the parsexml tag handler from a specified resource XML file, then passed explicitly from the parsexml tag to the transform tag through the toxml1 XML object.

The first example uses the XML resource email.xml and the DTD email.dtd. No root attribute is specified, so validation is from the top-level element, <email>.

The second example uses the XML resource emailWithDtd.xml, which has the DTD embedded in the file. The root attribute explicitly specifies that validation is from the element <email>.

The files email.xml, email.dtd, emailWithDtd.xml, and email.xsl are also listed below.

Example 1 for transform and parsexml

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/xml.tld"
           prefix="xml" %>
<h3>XML Parsing Tag Email Example</h3>
<xml:transform fromXMLObjName="toxml1" href="style/email.xsl">
   <xml:parsexml resource="style/email.xml" validateResource="style/email.dtd"
                 toXMLObjName="toxml1">
   </xml:parsexml>
</xml:transform>
```

Example 2 for transform and parsexml

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/xml.tld"
           prefix="xml" %>
<h3>XML Parsing Tag Email Example</h3>
<xml:transform fromXMLObjName="toxml1" href="style/email.xsl">
   <xml:parsexml resource="style/emailWithDtd.xml" root="email"
                 toXMLObjName="toxml1">
   </xml:parsexml>
</xml:transform>
```

email.xml

```
<email>
<recipient>Manager</recipient>
<copyto>jsp_dev</copyto>
<subject>XML Bug fixed</subject>
<bugno>BUG 1109876!</bugno>
<body>for reuse tag and checked in the latest version!</body>
<sender>Developer</sender>
</email>
```

email.dtd

```
<!ELEMENT email (recipient,copyto,subject,bugno,body,sender)>
<!ELEMENT recipient (#PCDATA)>
<!ELEMENT copyto (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
```

```
<!ELEMENT bugno (#PCDATA)>
<!ELEMENT body (#PCDATA)>
<!ELEMENT sender (#PCDATA)>
```

emailWithDtd.xml

```
<!DOCTYPE email [
<!ELEMENT email (recipient,copyto,subject,bugno,body,sender)>
<!ELEMENT recipient (#PCDATA)>
<!ELEMENT copyto (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT bugno (#PCDATA)>
<!ELEMENT body (#PCDATA)>
<!ELEMENT sender (#PCDATA)>]>
<email>
<recipient>Manager</recipient>
<copyto>jsp_dev</copyto>
<subject>XML Bug fixed</subject>
<bugno>BUG 1109876!</bugno>
<body>for reuse tag and checked in the latest version!</body>
<sender>Developer</sender>
</email>
```

email.xsl

```
<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:template match="email">
 <html><body>
  To: <xsl:value-of select="recipient"/>
  CC: <xsl:value-of select="copyto"/>
  Subject:   <xsl:value-of select="subject"/> ...
  <xsl:value-of select="body"/> !!
  Thanks  <xsl:value-of select="sender"/>
</body></html>
</xsl:template>
</xsl:stylesheet>
```

6

JESI Tags for Edge Side Includes

This chapter describes the Edge Side Includes for Java (JESI) tag library that is supplied with OC4J. These tags operate on top of an Edge Side Includes (ESI) framework that is available in the Oracle Application Server Web Cache to provide ESI caching functionality in a JSP application.

The chapter consists of the following sections:

- Overview of Edge Side Includes Technology and Processing
- Overview of JESI Functionality
- Oracle JESI Tag Descriptions
- JESI Tag Handling and JESI-to-ESI Conversion

For an overview of Web caching, including a discussion of the OracleAS Web Cache, the Oracle Application Server Java Object Cache, and the OC4J Web Object Cache, see "Summary of Oracle Caching Support for Web Applications" on page 1-9.

> **Note:** The JESI specification is not yet finalized. Although every effort has been made to comply with the latest working version, it is not possible to assure that the OC4J 10.1.2 implementation will fully comply with the final version of the JESI specification.

Overview of Edge Side Includes Technology and Processing

JESI tags, which are used to break down dynamic content of JSP pages into cacheable components, are based upon the Edge Side Includes architecture and markup language.

Although the use of JESI tags does not depend on any particular ESI processor or caching system, a typical scenario among Oracle customers is to use the OracleAS Web Cache and its ESI processor.

The following sections provide background information about some of the underlying technology upon which the Oracle JESI tags are based.

- Edge Side Includes Technology
- Oracle Application Server Web Cache and ESI Processor

This discussion provides only a brief overview of the ESI architecture and language. For additional information about ESI technology, refer to the following Web site:

http://www.esi.org

Edge Side Includes Technology

This section introduces the features of ESI technology and the concept of ESI *surrogates*.

Introduction to ESI

Edge Side Includes is an XML-style markup language that allows dynamic content assembly at the "edge" of the network, away from the origin Web server, and is designed to take advantage of available tools such as Web caches and content delivery networks (CDNs) to improve performance for users.

ESI provides a way to reduce the load on Web and application servers by promoting processing on intermediaries, known as *surrogates* or *reverse proxies*, that understand the ESI language and act on behalf of the Web server. ESI content is intended for processing somewhere between the time it leaves the originating Web server and the time it is displayed in the user's browser. A surrogate is commanded through HTTP headers. Such a surrogate can be referred to as an *ESI processor* and can be included as part of the functionality of a Web cache.

ESI lends itself to a partial-page caching methodology, where each dynamic portion of a Web page can be cached individually and retrieved separately and appropriately.

Using the ESI markup tags, a developer can define aggregate Web pages and the cacheable components that are to be retrieved and assembled, as appropriate, by the ESI processor for viewing in the HTTP client. Think of an aggregate page, which is the resource associated with the URL that a user specifies, as a container for assembly. This includes retrieval and assembly instructions that are specified through the ESI tags.

> **Important:** Do not use ESI tags directly in a page where you use JESI tags.

More About Surrogates

Because surrogates act on behalf of Web servers, where page content is owned, they allow content owners to have sufficient control over their behavior. In this way, they offer greater potential for performance improvements than would otherwise be available.

The caching process in surrogates operates similarly to the caching process in HTTP, using similar freshness and validation mechanisms as the foundation. However, surrogates also possess additional control mechanisms.

Key ESI Features

Version 1.0 of the ESI language includes the following key areas of functionality:

- Inclusion

  An ESI processor assembles fragments of dynamic content, retrieved from the network, into aggregate pages to output to the user. Each fragment can have its own meta data to control its caching behavior. See Figure 6–1 below.

- Support of variables

  ESI supports the use of variables that are based on HTTP request attributes. ESI statements can use variables during processing or can output them directly into the processed markup.

- Conditional processing

ESI allows the use of boolean comparisons for conditional logic in determining how pages are processed.

■ Error handling and alternative processing

Some ESI tags support specification of a default resource or an alternative resource (or both), such as an alternate Web page, if the primary resource cannot be found.

Figure 6–1   ESI Include Model
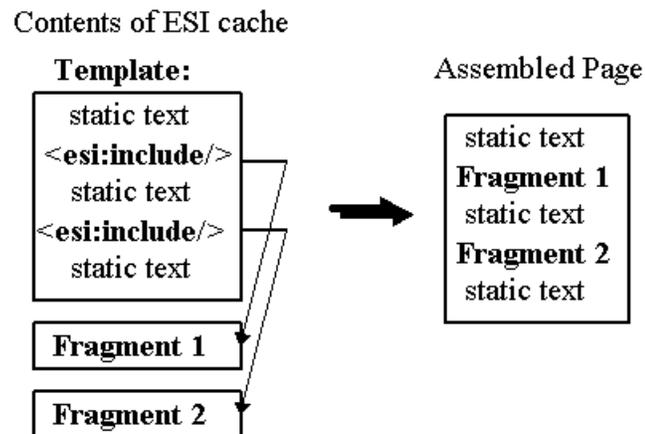
Oracle Application Server Web Cache and ESI Processor

This section introduces the OracleAS Web Cache and its ESI processor. See the *Oracle Application Server Web Cache Administrator's Guide* for more information.

Introduction to Oracle Application Server Web Cache

Oracle offers OracleAS Web Cache to help e-businesses manage Web site performance issues. It is a content-aware server accelerator, or *reverse proxy server*, that improves the performance, scalability, and availability of Web sites that run on the Oracle Application Server.

By storing pages from frequently accessed URLs in memory, OracleAS Web Cache eliminates the need to repeatedly process requests for those URLs on the application Web server. Unlike legacy proxy servers that handle only static documents, OracleAS Web Cache caches both static content and dynamically generated content from one or more application Web servers. As the result of more frequent cache hits, performance enhancement is greater than with legacy proxies and the load on application servers is less.

Conceptually, OracleAS Web Cache is positioned in front of application Web servers, caching their content and sending that content to Web browsers that request it. When Web browsers access the Web site, they send HTTP protocol or HTTPS protocol requests to OracleAS Web Cache, which, in turn, acts as a virtual server for the application Web servers. If the requested content has expired, has been invalidated, or is no longer accessible, then OracleAS Web Cache retrieves the new content from the application Web servers.

Steps in Oracle Application Server Web Cache Usage

Here are the steps for typical browser interaction with OracleAS Web Cache:

1. A browser sends a request to the Web site of a company.

2. The request, in turn, generates a request to the Domain Name System (DNS) for the IP address of the Web site.

3. DNS returns the IP address of OracleAS Web Cache.

4. The browser sends the request for the Web page to OracleAS Web Cache.

5. If the requested content is in its cache, then OracleAS Web Cache sends the content directly to the browser. This is known as a *cache hit*.

6. If OracleAS Web Cache does not have the requested content, or if the content is outdated or invalid, then the Web cache hands off the request to the application Web server. This is known as a *cache miss*.

7. The application Web server sends the content through OracleAS Web Cache.

8. OracleAS Web Cache sends the content to the client and makes a copy of the page in cache.

> **Note:** A page that is stored in the cache is removed at some point after it becomes invalid or outdated.

Oracle Application Server Web Cache ESI Processor

OracleAS Web Cache includes an ESI processor to support the use of the Edge Side Includes markup language in caching. (See "Edge Side Includes Technology" on page 6-2.)

Web developers in an OracleAS Web Cache environment can use the ESI language directly in their applications; however, for JSP developers, there are several reasons to use the JESI tag library that is provided as a convenient JSP interface to the ESI language. See "Advantages of JESI Tags" on page 6-4.

Overview of JESI Functionality

The following sections introduce JESI functionality and the Oracle implementation:

- Advantages of JESI Tags
- Overview of JESI Tags Implemented by Oracle
- JESI Usage Models
- Invalidation of Cached Objects
- Personalization of Cached Pages
- JESI Fallback Execution

You can access the proposed JESI specification at the following Web site:

http://www.esi.org

Advantages of JESI Tags

OC4J provides the JESI tag library as a convenient interface to ESI tags and Edge Side Includes functionality for Web caching. Developers have the option of using ESI tags directly in any Web application, but JESI tags provide additional convenience for JSP pages. Here are the main advantages in using JESI tags instead of using ESI tags directly:

- Standard JSP framework and convenient features

    – JESI tags allow use of the familiar and convenient features of JSP programming. For example, you can reference included pages according to page-relative or application-relative locations, instead of the complete URL or file path.

    – You can pass dynamic values to JESI tag attributes.

    – You can use JESI tags in combination with tags from other JSP tag libraries.

- JESI shortcut syntax

    JESI tags support convenient syntax and tag attributes for specifying meta data information (such as expiration for cached pages), explicitly invalidating pages as appropriate, and personalizing pages using cookie information.

- Application-level configuration files

    The JESI tag library can use application-level configuration files for convenient specification of deployment-time parameters and application default settings that are appropriate to a particular environment. In this way, you can deploy to different environments that have diverse needs and set appropriate defaults without changing application code. For example, you can use such a configuration file to preset the cache server URL, user name, and password for invalidation requests.

Overview of JESI Tags Implemented by Oracle

The Oracle implementation of JESI is layered on top of the standard ESI framework. It also conforms with the pending (as of the OC4J 10.1.2 implementation) JESI standard, JSR-128, which is sponsored by the Java Community Process (JCP) organization. For more information about the JCP organization and the status of JSR-128, go to the following location:

http://www.jcp.org

Because the JESI tag library is a standard implementation, note the following:

- An application that uses JESI tags does not depend on the OC4J JSP container. It is portable to any standard JSP container. (Outside of OC4J, you can use either the reference implementation to be provided with JSR-128, or the JESI implementation that is provided with the JSP container you are using, if applicable.)

- Even though this document discusses the OracleAS Web Cache and its ESI processor in particular, the JESI tag library does not depend on any particular caching environment and can work with any ESI processor that conforms to the ESI 1.0 specification.

The Oracle JESI tag library supports the following tags:

- JESI `control`, JESI `include`, JESI `param`, JESI `template`, JESI `fragment`, and JESI `codeblock` for dynamic caching of page content

- JESI `invalidate` (and subtags) for explicit invalidation of cached objects, when appropriate

- JESI `personalize` for page customization through cookies

JSP developers use these tags (such as JESI `include`) instead of corresponding ESI tags (such as `esi:include`). The usefulness and convenience of this is discussed in "Advantages of JESI Tags" on page 6-4.

> **Note:** The Oracle JESI tag library is implemented according to general standards for JSP custom tag libraries. For information about the standard JavaServer Pages tag library framework, refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide.*

JESI Usage Models

There are two models for how to use JESI tags to define aggregate pages and their cacheable components:

- Control/include model

- Template/fragment model

This section describes these models and concludes with some special notes about the JESI `include` tag.

Control/Include Model

The *control/include* approach to using JESI tags is modular, typically bringing most (or all) cacheable content into the aggregate page as included pages. This is particularly convenient when you are developing new pages. Use this model as follows:

- Use the JESI `control` tag in the top-level page to set caching parameters for content outside the included content, as applicable.

- Use JESI `include` tags to bring in dynamic content.

- Use a JESI `control` tag inside each included page to set caching parameters for those pages, as appropriate.

Each included file is a distinct cacheable object (although caching can be disabled according to tag settings), and any content in the top-level page is also a distinct object.

Both tags are optional, depending on the situation. A page can have a JESI `control` tag without any JESI `include` tags. In fact, this is a simple way to convert an existing page for JESI use. There is also no requirement for a JESI `control` tag in a page that uses JESI `include` tags. The ESI processor will be appropriately notified of the presence of the JESI `include` tags, regardless. And there is no requirement for an included page to have a JESI `control` tag.

The cacheability of a page, either top-level or included, is determined as follows.

- If there is a JESI `control` tag, cacheability depends on attribute settings or on the default attribute values, as applicable.

- If there is no JESI `control` tag, cacheability depends on configuration settings of the ESI processor.

- The JESI `control` tag in the top-level page has no effect on included pages.

See the following sections for tag syntax and examples:

Template/Fragment Model

In the *template/fragment* approach, content is contained in a single page and you split the page into separately cacheable fragments as desired. This model is particularly convenient when you are converting existing pages for JESI use and want certain portions to be separate cacheable components. Use this model as follows:

- Use the JESI `template` tag to enclose the aggregate of all visible content. This tag sets caching parameters for the content outside the fragments. There must be no visible content outside the `template` tag.

- Use JESI `fragment` tags as desired, between the `template` start-tag and end-tag, to define fragments within the aggregate, to be cached separately.

- Optionally use JESI `include` tags as well, either at the template level or the fragment level.

- Optionally use `codeblock` tags within the template tag, outside of any fragments, to mark conditional execution of blocks of code.

The JESI `template` tag and JESI `fragment` tag are always used together. If you do not need separate fragments in a page, use the JESI `control` tag instead of the JESI `template` tag.

Each fragment is a distinct, cacheable object. Any content at the template level, outside any fragments, is a distinct, cacheable object. Any page that is included through a JESI `include` tag is also a distinct, cacheable object. Cacheability is determined as follows.

- The cacheability of the template (content outside any fragments) depends on the JESI `template` tag attribute settings or on the default attribute values, as applicable.

- Similarly, the cacheability of a fragment depends on the attribute settings of the JESI `fragment` tag or on the default attribute values, as applicable.

- The cacheability of an included page is determined as follows.

  - If there is a JESI `control` tag, cacheability depends on attribute settings or on the default attribute values, as applicable.

  - If there is no JESI `control` tag, cacheability depends on configuration settings of the ESI processor.

Note: It is permissible for the template/fragment model to be used in a response that has already processed a JESI `control` tag. This might be required, for example, in a case where conditional generation of the aggregate response can include response from any of a set of alternative pages. In this case, the JSP container uses the attribute settings of the `control` tag, ignores the attribute settings of the `template` tag, but notes the fact that there is a `template` tag properly enclosing any `fragment` tags, as required. As is always the case in the template/fragment model, there must be no cacheable content outside the `template` tag.

Because the template and fragments are independent, cacheable objects, they can expire at different times in the ESI processor. When a cache miss occurs or an object that has expired is requested, the ESI processor makes a request to the origin server (OC4J in the case of Oracle Application Server) for a fresh copy.

If a requested object is a JESI template, the JSP container executes code in the page that is outside any fragments. In output that is generated by the JSP translator, the

translator also places ESI markup that designates where all the fragments should be included. The code that is contained in the JESI fragments will not be executed at that time. Figure 6–2, which follows, illustrates this.

Figure 6–2 JESI Request for Template
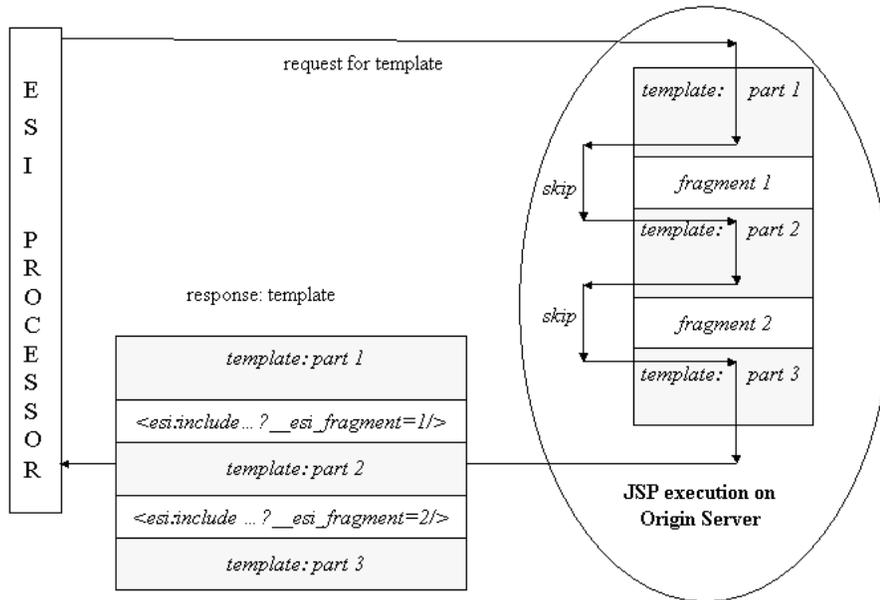
When a fragment expires, the ESI processor makes a request to the origin server for that particular fragment. To execute a fragment, the OC4J JSP container executes the template code (code outside of any fragments) plus the code of the fragment being requested. The template code is executed to allow a fragment to rely on certain side effects, such as declaration or initialization of variables.

The output of the fragment code is returned in the response; the output of the template code is discarded. Upon receiving the response, the ESI processor will cache the updated copy of the fragment. Figure 6–3, which follows, illustrates this.

Figure 6–3   JESI Request for Fragment
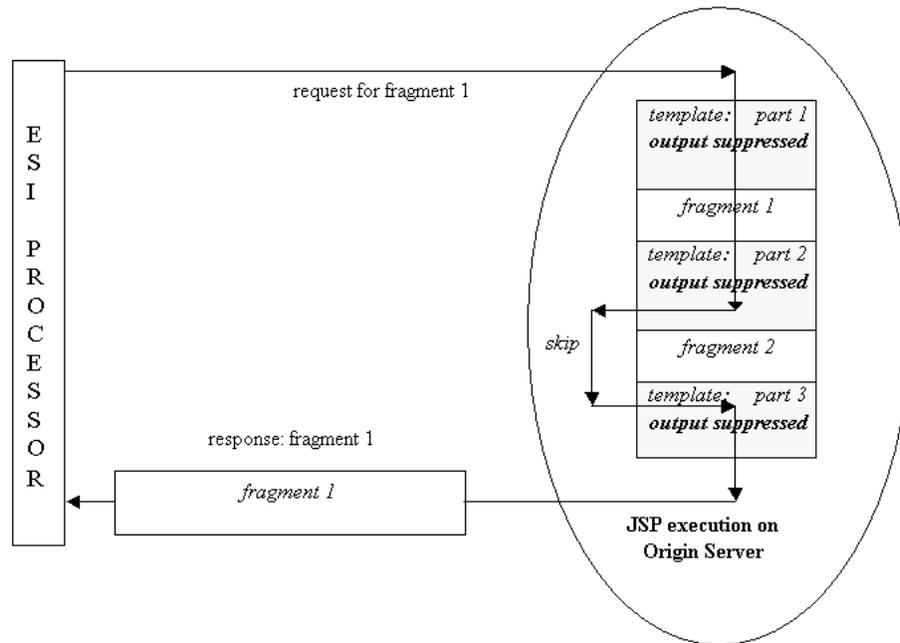
> **Note:**   To avoid needlessly repeating the execution of expensive template code, strategically place the code within JESI `codeblock` tags. Configure each `codeblock` tag according to when you want the code within it to be executed (whenever the template is requested, whenever a fragment is requested, or always).

Remember this behavior when choosing code placement and expiration policies for your templates and fragments. In particular, because template code is executed in every update request, be aware of where you place any expensive code. Do not place an expensive computation at the template level unless it must be executed every time or is appropriately placed within a `codeblock` tag. Otherwise, place expensive computation in a fragment that has as long an expiration time as possible.

Figure 6–4 shows one `codeblock` tag scenario, in which the code block is to be executed only when a fragment is requested. In this figure, the request is for the template, so the code block is not executed.

Figure 6–5 shows another `codeblock` tag scenario, in which the code block is still to be executed only when a fragment is requested. This time, however, the request is for the fragment, so the code block is executed.

Figure 6–4   JESI codeblock Fragment Execution with Request for Template
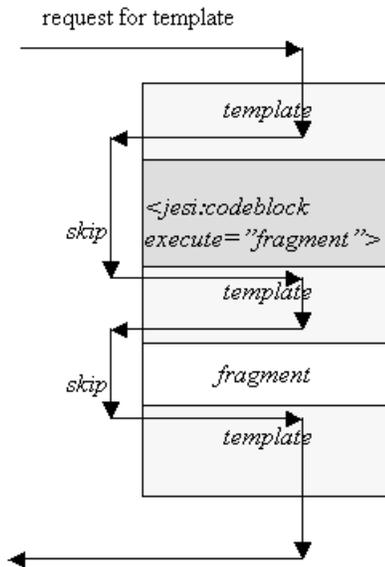
Figure 6–5   JESI codeblock Fragment Execution with Request for Fragment
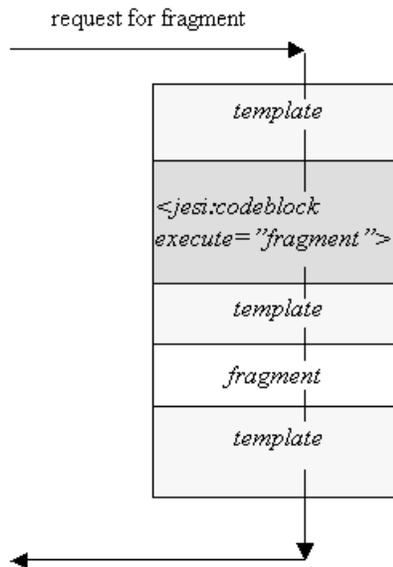
Additionally, remember that no two fragments are ever executed during the same request. For example, you should not declare or set the value of a scriptlet variable in one fragment and depend on that variable or the set value in another fragment. If a variable is needed in more than one fragment, then it should be declared and set in the template code (possibly inside a `codeblock` tag). Similarly, do not set a request or session attribute in one fragment and then try to read it in another fragment. Such "page-global logic" should also be placed at the template level.

Finally, remember that the different fragments of a page will be refreshed at different times, according to invalidation messages and expiration settings. Typically, in a

well-tuned application, most fragments would be served from the ESI cache, having to be regenerated only infrequently.

> **Note:** You can intersperse JESI `template`, JESI `fragment`, and JESI `include` tags with `jsp:include` tags if the JESI tags follow JESI rules and are properly nested with respect to each other. For example, you can have a page with a JESI `template` tag, have a `jsp:include` tag within the `template` tag, and have JESI `fragment` tags in the included page. You can also use the JESI `template` tag within an included page if there is no other `template` tag at a higher level, and if all output to the response buffer is within the `template` tag.

See the following sections for tag syntax and examples:

-
-
-
-
-

Notes About JESI and JSP Includes

In using either the control/include or template/fragment model, be aware of the following notes regarding the JESI `include` statement:

- A nested JESI inclusion is supported, either as a JESI `include` statement that includes a page that, in turn, has its own JESI `include` statement, or as a JESI `include` statement inside a fragment that is defined with a JESI `fragment` statement.

  In the second case, for example, the ESI processor executes the following steps:

  1. It requests the content of the aggregate page.

  2. It locates the content of the fragment in the cache (if applicable) or requests it.

  3. It locates the content of the included page in the cache (if applicable) or requests it.

- Despite conceptual similarities between the JESI `include` tag and the standard `jsp:include` tag, there are situations in which you should not substitute a JESI include tag for a `jsp:include` tag when you convert a JSP page for caching. Because the ESI processor uses separate HTTP requests, you are unable to pass an HTTP request or response object between one page and a page it includes through a JESI `include` tag. If the code in the included page requires access to the request or response object of the originating page, then you should consider using the JESI template/fragment model and putting the code in a JESI `fragment` tag (within the JESI `template` tag of the aggregate page) instead of using the JESI `include` tag.

Invalidation of Cached Objects

There might be situations where cached objects must be explicitly invalidated due to external circumstances, such as changes to relevant data in a database. There might

also be situations where execution of one page might invalidate the data of cached objects corresponding to another page.

For this reason, JESI provides the JESI `invalidate` tag and related subtags. These tags allow you to invalidate pages based on appropriate combinations of the following:

- Full URI or URI prefix

- Cookie name-value pair (optional)

- HTTP/1.1 request header name-value pair (optional)

Invalidation messages are in an XML-based format and specify the URLs to be invalidated. These messages are initiated by the JSP container when it executes the JESI `invalidate` tag, and transmitted to the cache server over HTTP using a `POST` method. The cache server then replies with an invalidation response, sent back over HTTP.

See "Descriptions of Tags and Subtags for Invalidation of Cached Objects" on page 6-25 for tag syntax and examples.

Personalization of Cached Pages

Dynamic Web pages frequently display customized information tailored to each individual user. For example, a welcome page might display the user's name and a special greeting, or current quotes for stocks the user owns.

For this kind of tailored output, the Web page depends on cookie information, which can be provided through the JESI `personalize` tag. Without this tag to inform the ESI processor of the need to perform cookie substitution, the Web page cannot be shared by multiple users at the ESI level.

See "Description of Tag for Page Personalization" on page 6-32 for tag syntax and examples.

JESI Fallback Execution

If no ESI processor is available for a page that uses JESI tags (such as on a system without OracleAS Web Cache, or in which Web Cache or its ESI processor is down), then the OC4J JSP container steps in to assemble the pages appropriately. Essentially, it takes over and provides the most crucial functionality to execute the pages properly. Caching does not take place, nor does error-checking of JESI tag attribute values.

In these circumstances, the JSP container processes the particular JESI tags as follows:

- It ignores JESI `control` tags.

- It executes JESI `include` tags as though they are `jsp:include` tags, and the associated JESI `param` tags as though they are `jsp:param` tags. Note that any scriptlet code that is nested within a JESI `include` tag will still be executed.

  > **Note:** In this circumstance, unlike with JESI `include` functionality, there is no longer a separate response object for an included page.

- It checks JESI `template` and `fragment` tags for proper nesting, but otherwise ignores them and executes all their tag bodies during a single request.

- It unconditionally executes any code in JESI `codeblock` tags.

- It ignores JESI `invalidation` tags and all subtags.

- For JESI `personalize` tags, it inserts the cookie value into the response body if the cookie previously existed. If the cookie did not previously exist and a default value is specified in the `personalize` tag, then the JSP container inserts the default value into the response body. If the cookie did not previously exist and no default value is specified, then the `personalize` tag has no effect.

Oracle JESI Tag Descriptions

The following sections describe the syntax and attributes of the JESI tags provided with OC4J, followed by usage examples:

- Descriptions of Tags for Dynamic Caching
- Descriptions of Tags and Subtags for Invalidation of Cached Objects
- Description of Tag for Page Personalization

Note the following requirements for the JESI tag library:

- The Oracle JESI tag library, a standard JavaServer Pages tag library implementation, is included in the `ojsputil.jar` file, which is provided with OC4J and is located in the "well-known" tag library directory. Verify that this file is installed and in your classpath.

- The tag library descriptor, `jesitaglib.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in the `ojsputil.jar` file. The `uri` value for `jesitaglib.tld` is the following:

  ```
  http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jesitaglib.tld
  ```

Refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

Notes:

- The prefix "jesi:" is used in the tag syntax here. This is by convention and is not required. You can specify any desired prefix in your `taglib` directive.

- See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

Descriptions of Tags for Dynamic Caching

The following sections cover the use of JESI tags for dynamic caching, document their syntax and attributes, and provide examples:

- JESI control Tag
- JESI include Tag
- JESI param Tag
- Examples: Control/Include Model
- JESI template Tag
- JESI fragment Tag

- JESI codeblock Tag

- Examples: Template/Fragment Model

See "JESI Usage Models" on page 6-6 for overviews of the control/include and template/fragment models.

JESI control Tag

The JESI `control` tag controls caching characteristics for JSP pages in the control/include usage model. You can use a JESI `control` tag in the top-level page or any included page, but it is not mandatory. For any page without a JESI `control` tag in the control/include model, cacheability is according to the configuration settings of the ESI processor. (See "JESI Usage Models" on page 6-6.)

Because action resulting from the JESI `control` tag sets the HTTP response header, this tag should appear as early as possible in the page, before any other JESI tags or any buffer flushes in the page.

Note the following:

- All attributes of the JESI `control` tag are optional. If you use a tag without any settings, then, by default, the cacheability of a response has an expiration setting of 24 hours, with immediate removal of expired objects.

- If you want caching behavior to be determined by the configuration of the ESI processor, then do not use a JESI `control` tag for the page in question.

- The JESI `control` tag of an originating page (a page with a JESI `include` tag) has no effect on included pages. Use a JESI `control` tag in each included page as well, as necessary.

- If the JSP container encounters multiple JESI `control` tags while generating a single response, then only the first one is processed. The rest are ignored. Note that a page that is included through a JESI `include` tag (where such a page might have its own JESI `control` tag) results in a separate response.

- If the JSP container has already encountered a JESI `template` tag when it encounters a JESI `control` tag while still generating the same response, then the `control` tag is ignored.

- If a page with a JESI `control` tag depends on request parameters, consider whether you must cache different versions of the page, depending on the request query string. Another alternative is to not cache the page at all (set `cache="no"`) if you anticipate that too many different request parameter values will result in too many cached versions of the page.

Syntax

```
<jesi:control
            [ expiration = "value" ]
            [ maxRemovalDelay = "value" ]
            [ cache = "yes" | "no" | "no-remote" ]
            [ control = "uninterpreted_string" ] />
```

Attributes

- `expiration`: Specifies the lifetime, in seconds, of the cached object. The default is 86400 (24 hours).

- `maxRemovalDelay`: Specifies the maximum time, in seconds, that the ESI processor continues to store the cached object after it has expired. The default is 0, for immediate removal.

- cache: Specifies whether the response corresponding to the tag is cacheable. A "yes" setting (the default) enables caching. Alternatively, you can set cache to "no" to disable caching, or to "no-remote" to enable caching only on the closest cache (instead of on a remote ESI processor or content delivery network).

  One reason to make a page noncacheable, for example, is if you are using a JESI include tag with copyParam="yes". See "JESI include Tag" below.

- control: The value of this attribute is appended without change to the Surrogate-Control response header that was created during processing of the JESI control tag. The OracleAS Web Cache ESI processor does not use this attribute; however, it would be useful if you are using another ESI processor for your application and want to pass it any additional proprietary information in the header.

  > **Notes:**
  >
  > - Do not confuse the control attribute name with the control tag name.
  >
  > - "JESI Tag Handling and JESI-to-ESI Conversion" on page 6-33 includes some discussion of the Surrogate-Control header.

JESI include Tag

The JESI include tag, as with a standard jsp:include tag, allows dynamic insertion of output from the included page into output from the originating page. It does so by directing the ESI processor to process and assemble the included pages. Each included page is a separate cacheable object (but might not be cached, depending on settings).

You can use this tag in either the control/include model or the template/fragment model, in any of the following scenarios:

- By itself, without a JESI control tag or JESI template and fragment tags

- In the control/include model, after a JESI control tag

- In the template/fragment model, within a JESI fragment tag, or within the JESI template tag but outside any fragments

(See "JESI Usage Models" on page 6-6.)

In addition, it is permissible to nest JESI includes, either by using a JESI include tag inside a page that is itself included through a JESI include tag, or by using a JESI include tag inside a page that is included through a standard jsp:include tag.

The cacheability of an included page is determined as follows:

- If there is a JESI control tag, cacheability depends on attribute settings or on the default attribute values, as applicable.

- If there is no JESI control tag, cacheability depends on configuration settings of the ESI processor.

Syntax

```
<jesi:include page = "uri"
             [ alt = "alternate_uri" ]
             [ ignoreError = "true" | "false" ]
             [ flush = "true" | "false" ]
             [ copyParam = "true" | "false" ] >
```

```
...optional jesi:param tags, related scriptlets...

</jesi:include>
```

Notes:

- As with standard `jsp:include` tags and their optional `jsp:param` subtags, you can use JESI `param` tags nested within a JESI `include` tag to specify new parameters that will be sent to the originating page (the page with the JESI `include` tag). See "JESI param Tag" on page 6-17 for tag syntax. In addition, the body of a JESI `include` tag can contain scriptlet code to be used in evaluating the added parameters. Output from the scriptlet code, however, and from the body of any JESI `include` tag in general, is discarded.

- In some cases, JESI `include` tags behave differently from `jsp:include` tags. This is because the JESI `include` tag results in separate request and response objects for the included page. A JESI `include` tag is not suitable, for example, when the originating page sets a request attribute and the included page reads this attribute from the request object.

- For backward compatibility, the deprecated "copyparam" form of the `copyParam` attribute is accepted. The change from `copyparam` to `copyParam` was made to comply with the proposed JESI specification. It is likely that `copyparam` will be desupported at some point.

Attributes

- `page` (required): Specifies the URI of the JSP page to be included, either a page-relative or application-relative location. (Refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* regarding syntax for page-relative and application-relative locations.) A full "`http://...`" or "`https://...`" URL is supported as well.

  The URI can optionally specify additional query parameters and values to pass to the included page, but using JESI `param` subtags is a preferred mechanism for this. See "JESI param Tag" on page 6-17.

- `alt`: Specifies a URI for an alternate page that is to be included if the page that is specified in the `page` attribute cannot be found. Syntax is the same as for the `page` attribute.

- `ignoreError`: Set this to "`true`" for continued processing of the originating page, even if no included page can be accessed (neither the `page` page nor the `alt` page). The default is "`false`".

- `flush`: This attribute is ignored, but is allowed in order to ease migration from `jsp:include` syntax.

- `copyParam`: If the included page makes use of request parameters, set this to "`true`" if you want to copy parameters and their values from the HTTP request string of the originating page to the included page. The default value is "`false`".

If request parameters are significant to the included page and
`copyParam="true"`, then either the originating page should not be cached
(`cache="no"` in the JESI `control`, JESI `template`, or JESI `fragment` tag), or
multiple versions of the originating page should be cached, according to
parameter settings.

As an example, avoid scenarios such as the following:

```
<jesi:control cache="yes"/>
...
<jesi:include page="arf.jsp" copyParam="true" />
```

The reason is that if a copy of this originating page is served from the cache, and if
parameters of this subsequent request are different than those of the original
request, then the page will not execute on the server or have a chance to properly
copy new parameters into `arf.jsp`. This would result in clients being served
`arf.jsp` generated from incorrect parameters.

However, this scenario would *not* be problematic in certain circumstances, such as
either of the following:

– The `arf.jsp` page does not use the request parameters.

– Appropriate versions of the originating page and `arf.jsp` are cached in the
  ESI processor, based on URL parameters. See the *Oracle Application Server Web
  Cache Administrator's Guide* for more information.

JESI param Tag

The JESI `param` tag is an optional subtag of the JESI `include` tag. These tags work
together in the same way that standard `jsp:include` and `jsp:param` tags work
together.

You can use one or more JESI `param` subtags to pass additional query parameters to
the target page of the JESI `include` tag. Doing this is more straightforward than the
alternative, which is to specify parameters in the `page` URI of the JESI `include` tag. If
you use both mechanisms, then parameters from `param` tags are appended after
parameters from the `include` tag `page` URI. Any parameters that are copied from the
original request, through an `include` tag `copyParam="yes"` setting, are appended
after parameters from JESI `param` tags.

See for a sample.

> **Note:** Be aware that the parameter name and value will be
> evaluated when the originating page (the page with the JESI
> `include` and `param` tags) is generated. If, afterward, the
> originating page is cached in an ESI processor, then the name and
> value of the parameter, passed down to the included page, remain
> unchanged until the originating page is regenerated. (This is similar
> to the treatment of request parameters that are copied from the
> request through a `copyParam="true"` setting.)

Syntax

```
<jesi:include page = "uri" ... >
    <jesi:param name="param_name"
          value="param_value" />
    ...
</jesi:include>
```

Attributes

- `name` (required): Specifies the name of the parameter.
- `value` (required): Specifies the value of the parameter.

Examples: Control/Include Model

This section provides examples of JESI tag usage in the control/include model.

Example 1: Control/Include The following example employs default cache settings; no JESI `control` tag is necessary. The JESI `include` tags specify no alternate files, and a "file not found" error will halt processing. The `flush` attribute is permissible, but ignored.

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jesitaglib.tld"
        prefix="jesi" %>
<html>
<body>
<jesi:include page="stocks.jsp" flush="true" />
<p>
<hr>
<jesi:include page="/weather.jsp" flush="true" />
<p>
<hr>
<jesi:include page="../sales.jsp" flush="true" />
</body>
</html>
```

Example 2: Control/Include This example uses the JESI `control` tag to specify nondefault cache settings for `maxRemovalDelay` and `expiration`. In addition, it explicitly enables caching of the page, though this is already enabled by default. The first JESI `include` tag specifies an alternate page in case `order.jsp` cannot be retrieved by the ESI processor, and specifies that processing should continue even if neither page can be retrieved. The second JESI `include` tag specifies no alternate page; processing will halt if the page cannot be retrieved.

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jesitaglib.tld"
        prefix="jesi" %>

<jesi:control maxRemovalDelay="1000" expiration="300" cache="yes"/>
<jesi:include page="order.jsp" alt="alt.jsp" ignoreError="true"/>
<jesi:include page="commit.jsp" />
```

Example 3: Control/Include This example is of an aggregate page with conditional output. A cookie represents the identity of a customer. If no cookie is found, the user will be shown a generic welcome page with general product information. If a cookie is found, the user will be shown a list of products according to the user profile. This list is brought into the page through a JESI `include` statement.

The JESI `control` tag also sets nondefault values for `maxRemovalDelay` and `expiration` and explicitly enables caching for the page.

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jesitaglib.tld"
        prefix="jesi" %>

<jesi:control maxRemovalDelay="1000" expiration="300" cache="yes"/>
<%
  String customerId=CookieUtil.getCookieValue(request,"customerid");
  if (customerId==null) {
    // some unknown customer
```

```
%>
    <jesi:include page="genericwelcome.jsp" />
<%
  }
  else {
    // a known customer; trying to retrieve recommended products from profiling
    String recommendedProductsDescPages[]=
                ProfileUtil.getRecommendedProductsDescURL(customerId);
    for (int i=0; i < recommendedProductsDescPages.length; i++) {
%>
    <jesi:include page="<%=recommendedProductsDescPages[i]%>" />
<%
    }
  }
%>
```

Example 4: Control/Include This example illustrates the use of JESI `include` statements with request parameters. Assume that the main page is accessed through the following URL:

`http://`*host*`:`*port*`/application1/main.jsp?p2=abc`

The main page takes the parameter setting `p2=abc`. Here is that page:

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jesitaglib.tld"
          prefix="jesi" %>
<html>
<jesi:control cache="no" />
<jesi:include page="a.jsp?p1=v1" />
<h3>hello ...</h3>
<jesi:include page="b.jsp" />
<h3>world ...</h3>
<jesi:include page="c.jsp?p1=v2" copyParam="true" />
</html>
```

The `a.jsp` page takes the parameter setting `p1=v1`. The `c.jsp` page takes the setting `p1=v2` as well as the setting `p2=abc`, as a result of the `copyParam` setting and the `p2` setting in the URL for the main page.

Additionally, the top-level page is noncacheable, according to the `cache="no"` setting. In fact, remember that you should use the `copyParam` setting in a JESI `include` tag only when the originating page is noncacheable, because the request attributes might change from one request to the next. Remember, too, that the `cache="no"` setting has no effect on the included pages. They are still cacheable by default. In other words, each is cacheable unless it has its own JESI `control` tag with `cache="no"` for some reason.

Example 5: Control/Include with param Tag This example illustrates use of a JESI `param` tag to add runtime values as new parameters to the included page request. Assume the main page is accessed through a URL such as the following, taking the parameter setting `p1=v1`:

`http://`*host*`:`*port*`/application/main.jsp?p1=v1`

Here is the page:

```
<jesi:control cache="yes" />
<jesi:include page="a.jsp" >
    <% String v2 = null;
       if(request.getParameter("p1").equals("v1")
           v2 = "v1 set";
```

```
        else
            v2 = "v2 unset";
    %>
    <jesi:param name="p2" value="<%=v2%>" />
</jesi:include>
```

JESI template Tag

Use the JESI `template` tag to specify caching behavior for the template content, outside any fragments, in the template/fragment usage model. (See "JESI Usage Models" on page 6-6.) The corresponding HTTP header will be set according to the ESI specification. The content outside the fragments is referred to here as the *template content* and is a separate cacheable object, and the content of each fragment set aside with a JESI `fragment` tag is a separate cacheable object.

> **Important:** All response output must be generated between the `template` start-tag and end-tag. Place the JESI `template` start-tag as early in the page as possible. It must appear before any JESI `fragment` tags or any buffer flushes in the page. It must also appear before any other visible output content such as text, HTML markup, new lines, or white space. Place the JESI `template` end-tag as late in the page as possible, after any JESI `fragment` tags and any other visible output content.

Always use the JESI `template` tag together with JESI `fragment` tags. If you have no need for separate fragments, then use a JESI `control` tag instead of a JESI `template` tag.

Note the following:

- All attributes of the JESI `template` tag are optional. If you use a tag without any settings, then, by default, the cacheability of a response has an expiration setting of 24 hours, with immediate removal of expired objects.

- In the template/fragment model, you cannot delegate decisions about cacheability to the ESI processor. You must have a JESI `template` tag, and cacheability of the template content is according to the `template` tag attribute settings or default values, as applicable. Similarly, each fragment must be set aside with a JESI `fragment` tag, and cacheability of each fragment is according to its `fragment` tag attribute settings or default values.

- Do not use multiple JESI `template` tags in a single JSP page. In addition, do not use additional JESI `template` tags in pages that are included, through `jsp:include` functionality, into the same response object. In either case, an exception will result.

- It is permissible to place a JESI `template` tag inside a page that is included through a standard `jsp:include` tag, as long as there is no `template` tag in any higher-level pages and you follow any other relevant restrictions that are mentioned in this section.

- If the JSP container has already encountered a JESI `control` tag when it encounters a JESI `template` tag while still generating the same response, then any attributes of the `template` tag are ignored and caching is according to the `control` tag.

> **Note:** In this situation, the `template` tag is not ignored completely. When the template/fragment model is used in a page that also has a JESI `control` tag (which can happen if a page with a `template` tag is included dynamically, and usually conditionally), the JSP container notes the fact that there is a `template` tag properly enclosing any `fragment` tags, as required.

- The JESI `template` tag cacheability settings have no effect on the enclosed fragments; fragments provide their own settings (or default values).

- If request parameters are significant to a fragment, then either the enclosing template content should not be cached (`cache="no"` in the JESI `template` tag), or separate versions of the template content should be cached, according to parameter values. Different versions of the fragment should also be cached, according to parameter values.

  In the background, a fragment involves an additional request, as with a page included through a JESI `include` tag. Request parameters (if any) are always passed from the template to the fragment, equivalent to JESI `include` tag functionality with a setting of `copyParam="true"`. (This kind of issue is also discussed in "JESI include Tag" on page 6-15.)

The JESI `template` tag has the same attributes, with the same usage, as the JESI `control` tag.

Syntax

```
<jesi:template
            [ expiration = "value" ]
            [ maxRemovalDelay = "value" ]
            [ cache = "yes" | "no" | "no-remote" ]
            [ control = "uninterpreted_string" ] >

...page content, jesi:fragment tags, optional jesi:include tags, optional
jesi:codeblock tags..

</jesi:template>
```

Attributes

For attribute descriptions, see "JESI control Tag" on page 6-14.

JESI fragment Tag

Use one or more JESI `fragment` tags within a JESI `template` tag, between the JESI `template` start-tag and end-tag, in the template/fragment model. (See "JESI Usage Models" on page 6-6.) Each JESI `fragment` tag defines a separate fragment of the JSP page, as desired, for caching behavior. Each fragment is a separate cacheable object.

When a particular fragment is requested for inclusion into the aggregate response through the ESI mechanism, the ESI processor retrieves only that fragment.

The JESI `fragment` tag has the same attributes, with the same usage, as the JESI `control` and JESI `template` tags.

Note the following:

- Each JESI `fragment` tag specifies its own caching instructions to the ESI processor. Cacheability is according to the specified attribute settings or the

default values, as applicable. The settings of the surrounding JESI `template` tag have no effect on the fragments.

- You cannot nest a JESI `fragment` tag within another JESI `fragment` tag.

- Unlike with the control/include model, it is not possible to delegate caching instructions to the ESI processor in the template/fragment model, given that a `template` tag, and `fragment` tags as applicable, are required. Caching is always according to `template` or `fragment` tag attribute settings or the default values.

- As long as you follow the restrictions mentioned in this section, it is permissible to place a JESI `fragment` tag inside a page that is included through a standard `jsp:include` tag. The JESI `template` tag that encloses the JESI `fragment` tag can appear in the same included page or in a higher level page such as the page containing the `jsp:include` statement.

Syntax

```
<jesi:fragment
            [ expiration = "value" ]
            [ maxRemovalDelay = "value" ]
            [ cache = "yes" | "no" | "no-remote" ]
            [ control = "uninterpreted_string" ] >

...JSP code fragment...

</jesi:fragment>
```

Attributes

For attribute descriptions, see "JESI control Tag" on page 6-14.

JESI codeblock Tag

In the template/fragment model, you can optionally use one or more JESI `codeblock` tags within template code, outside of any fragments, to mark conditional execution of particular blocks of code. Each `codeblock` tag surrounds a block of code and specifies when it should be executed:

- Only when the template is requested

or:

- Only when a fragment (any fragment) is requested

or:

- Always (whether a template or fragment is requested)

Without use of this tag, all template code is executed with every request—with each request for the template as well as with each request for any fragment, although template output is discarded in the case of a request for a fragment.

Although it is important to execute the template whenever a fragment is requested—to allow fragments to depend on template code side effects such as variable declaration or initialization—there might be blocks of code that are not critical to fragments. You can place any such code block into a `codeblock` tag with a specification to execute the block only when the template is requested.

Alternatively, there might be blocks of template code that are potentially vital to all fragments, but not to the template itself. You can place any such code block into a `codeblock` tag with a specification to execute the block only when any fragment is requested.

Note: It is advisable to *not* generate any visible output within a JESI `codeblock` tag. This is to avoid unexpected behavior due to differences in execution between requests for the template and requests for fragments. If `execute="template"` (or `"always"`) and the template is requested, then the code is executed and the content is output, as presumably intended. However, if `execute="fragment"` (or `"always"`) and the request is for a fragment, then the code is executed but the entire output of the template is suppressed, as is always the case when a fragment is requested. See Figure 6–3 in "Template/Fragment Model" on page 6-7.

Syntax

```
<jesi:template ... >
...
   <jesi:codeblock execute = "template" | "fragment" | "always" >
       ...request-dependent JSP content...
   </jesi:codeblock>
...
</jesi:template>
```

Attributes

- `execute` (required): Specify the value `"template"` to execute the code block only when the template is requested. Specify the value `"fragment"` to execute the code block only when any fragment is requested. A setting of `"always"` results in the code block being executed with every request for the page, and is equivalent to not using a `codeblock` tag at all.

Examples: Template/Fragment Model

This section contains examples of JESI tag usage in the template/fragment model.

Example 1: Template/Fragment This is a general example showing use of the JESI `template` and JESI `fragment` tags. Because only the `expiration` attribute is set in any of the tags, all other settings are according to defaults. The setting of the `cache` attribute defaults to `"yes"`, so the template and all three fragments are cached.

The template content (outside the fragments) uses an expiration of 3600 seconds, according to the JESI `template` tag. This applies to all the HTML blocks because they are outside the fragments. JSP code block #1 is cached with an expiration setting of 60; JSP code block #2 is cached with the default expiration setting; and JSP code block #3 is cached with an expiration setting of 600.

```
<%@ taglib uri="
http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/
jesitaglib.tld"
         prefix="jesi" %>
<jesi:template expiration="3600">
...HTML block #1...
   <jesi:fragment expiration="60">
   ...JSP code block #1...
   </jesi:fragment>
...HTML block #2...
   <jesi:fragment>
   ...JSP code block #2...
   </jesi:fragment>
```

```
...HTML block #3...
  <jesi:fragment expiration="600">
   ...JSP code block #3...
   </jesi:fragment>
...HTML block #4...
</jesi:template>
```

Example 2: Template/Fragment This example employs JESI `include` tags inside the fragments. The following are the cacheable objects for this page:

- Each included page

- Each fragment, outside of the page it includes

- The aggregate of the HTML blocks, which are all at the template level outside any fragments

```
<%@ taglib uri="
http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/
jesitaglib.tld"
          prefix="jesi" %>
<jesi:template expiration="3600">
...HTML block #1...
   <jesi:fragment expiration="60">
   ...JSP code block #1...
   <jesi:include page="stocks.jsp" />
   </jesi:fragment>
...HTML block #2...
   <jesi:fragment>
   ...JSP code block #2...
   <jesi:include page="/weather.jsp" />
   </jesi:fragment>
...HTML block #3...
   <jesi:fragment expiration="600">
   ...JSP code block #3...
   <jesi:include page="../sales.jsp" />
   </jesi:fragment>
...HTML block #4...
</jesi:template>
```

Example 3: Template/Fragment with Codeblock This is a conceptual example of how you can use the `codeblock` tag in the template/fragment model. In this case, to improve performance, the code that connects to the database would be placed in the code block so that it is not reexecuted needlessly.

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jesitaglib.tld"
          prefix="jesi" %>
<jesi:template>
Welcome to the Frequent Flyer Home page!
<jesi:codeblock execute="fragment" >
  /* Open a database connection and store it in the variable dbConn. */
</jesi:codeblock>
BEST DEALS
<jesi:fragment expiration="600" maxRemovalDelay="180">
...in Air Travel
/* Select the three cheapest USA domestic round-trip fares, using the database
connection stored in dbConn. */
</jesi:fragment>

<jesi:fragment expiration="600" maxRemovalDelay="180">
...in Accommodations
```

```
/* select the three best hotel deals, using the database connection stored in
dbConn. */
</jesi:fragment>

Click here to access your current Mileage account <...>
</jesi:template>
```

Descriptions of Tags and Subtags for Invalidation of Cached Objects

Use the JESI `invalidate` tag and the following subtags, as appropriate, to explicitly invalidate cached objects in the ESI processor:

- JESI `object`
- JESI `cookie` (subtag of JESI `object`)
- JESI `header` (subtag of JESI `object`)

The following sections cover the syntax of these tags, the JESI configuration file (which can be used to specify the user name, password, and URL to log in for invalidation), and some examples:

- JESI invalidate Tag
- JESI Configuration File
- JESI object Subtag
- JESI cookie Subtag
- JESI header Subtag
- Examples: Page Invalidation

See "Invalidation of Cached Objects" on page 6-11 for an overview.

JESI invalidate Tag

Use the JESI `invalidate` tag with its JESI `object` subtag to explicitly invalidate one or more cached objects.

Use the subtags as follows:

- Use the required JESI `object` subtag to specify what to invalidate, according to the URI or URI prefix.
- Optionally use JESI `cookie` subtags or JESI `header` subtags (or both) of the JESI `object` tag to specify further criteria for what to invalidate, according to cookie or HTTP header information.

Syntax

```
<jesi:invalidate
            [ url = "url"
              username = "user_name"
              password = "password" ]
            [ config = "configfilename" ]
            [ output = "browser" ] >
```

Required subtag (described in "JESI object Subtag" on page 6-28):

```
<jesi:object ... >
```

Optional subtag of JESI `object` (described in "JESI cookie Subtag" on page 6-29):

```
<jesi:cookie ... />
```

Optional subtag of JESI `object` (described in "JESI header Subtag" on page 6-30):

```
<jesi:header ... />

</jesi:object>
```

```
</jesi:invalidate>
```

Either specify the user, password, and URL all through their individual attributes, or all in the configuration file that is either referred to in the `config` attribute or is found in the default location. The default location is `/WEB-INF/jesi.xml` or, for backward compatibility, `/WEB-INF/config.xml`. See "JESI Configuration File" on page 6-27 for information about the file. If the user name, password, and URL are specified through the configuration file as well as through the attribute settings, then the attribute settings take precedence.

If you specify a `<user>` element for the OracleAS Web Cache "invalidator" account in the OC4J `jazn-data.xml` file, then you can use special syntax in the `password` attribute to refer to the information in `jazn-data.xml` instead of specifying the password in clear text. The password is specified in `jazn-data.xml` in an obfuscated form. See the `username` and `password` attribute descriptions below. See the *Oracle Application Server Containers for J2EE Security Guide* for information about the `jazn-data.xml` file.

Note: It is permissible to have multiple `object` tags within an `invalidate` tag.

Attributes

- `url`: Specifies the URL of the cache server. If this attribute is omitted, then you must specify the URL, as well as the user name and password, in the JESI configuration file.

- `username`: Specifies the user name for logging in to the cache server to perform invalidation. OracleAS Web Cache typically requires an "invalidator" user name. If this attribute is omitted, then you must specify the user name, as well as the password and URL, in the JESI configuration file.

  If the OC4J `jazn-data.xml` file contains a `<user>` element for the OracleAS Web Cache "invalidator" account, then you can use that account name for the `username` value, such as:

  `username="invalidator"`

- `password`: Specifies the password for logging in to the cache server to perform invalidation. If this attribute is omitted, then you must specify the password, as well as the user name and URL, in the JESI configuration file.

  If the OC4J `jazn-data.xml` file contains a `<user>` element for the OracleAS Web Cache "invalidator" account, then you can get the de-obfuscated password from that file by using special right-arrow syntax with a dash ("-") and right-carrot (">") followed by the invalidator account name, such as:

  `password="->invalidator"`

- `config`: Specifies a JESI configuration file, using either an application-relative or a page-relative location. You can use this file to provide the cache server URL, user name for invalidation, and password instead of using the corresponding tag attributes. Note the following:

- – You can use a configuration file in the default location instead of specifying one through the `config` attribute. See "JESI Configuration File" on page 6-27.

- – No configuration file is consulted if `username`, `password`, and `url` are all specified through tag attributes.

- `output`: Optionally sets an output device to receive the invalidation response from the cache server. Currently, the only supported setting is `"browser"`, which wraps the Web cache response with HTML formatting to show the message in the user's Web browser. If you do not set this parameter, then the invalidation response will not be displayed.

JESI Configuration File

The proposed JESI specification supports the use of a configuration file. Currently, you can use a configuration file only to specify the user name, password, and URL for invalidation. (Alternatively, you can specify the user name, password, and URL through attributes of each JESI `invalidate` tag. See "JESI invalidate Tag" on page 6-25.)

A JESI configuration file must have a `<jesi-config>` top-level element, an `<invalidation>` subelement under that, and `<username>`, `<password>`, and `<url>` subelements under the `<invalidation>` element.

> **Note:** For backward compatibility, the deprecated elements `<ojsp-config>` and `<web-cache>` are currently acceptable instead of `<jesi-config>` and `<invalidation>`, respectively. The new elements are to comply with the proposed JESI specification. It is likely that the `<ojsp-config>` and `<web-cache>` will be desupported in a future release.

In the current implementation there are two possible default files, or you can place the file anywhere within your application and specify its name and location through the `config` attribute of the `invalidate` tag, specifying either an application-relative or a page-relative location.

The preferred default file is `/WEB-INF/jesi.xml`, which conforms with the proposed JESI specification. For backward compatibility, the previous default file, `/WEB-INF/config.xml`, is also supported.

The following precedence is used to obtain the user name, password, and URL for invalidation:

1. If the JESI `invalidate` tag specifies the `username`, `password`, and `url` attribute settings (all three), then those values are used.

> **Note:** If the `invalidate` tag specifies one or two of these attributes but not all three, an exception is raised. An exception is also raised if one or more of the attribute values is an empty string or null.

2. If you do not specify `username`, `password`, and `url` in the `invalidate` tag, but the `config` attribute specifies a configuration file, then values from the specified configuration file are used.

3. If you do not specify `username`, `password`, `url`, and `config` in the `invalidate` tag, then the JSP container attempts to use a default configuration

file. First, the container searches for `/WEB-INF/jesi.xml` and uses the settings from that file if it is found. If that file is not found, then the container searches for `/WEB-INF/config.xml` and uses the settings from that file if it is found.

> **Notes:** If the `invalidate` tag does not specify the user name, password, and URL, an exception is thrown under either of the following circumstances:
>
> ■ If at any point a configuration file is found that does not specify all three attributes
>
> ■ If no configuration file is found

If the OC4J `jazn-data.xml` file contains a `<user>` element for the OracleAS Web Cache "invalidator" account, you can that account name in the JESI configuration file and get the password from `jazn-data.xml` by using special right-arrow syntax with a dash ("-") and right-carrot (">"), followed by the invalidator account name. See "Example 2: Configuration File Obtaining Password from jazn-data.xml" below.

Example 1: Configuration File with Clear Text for Password The following example shows a configuration file that is used instead of the `url`, `username`, and `password` attributes to set the URL and login information:

```
<?xml version="1.0" ?>
<jesi-config>
  <invalidation>
      <url>http://yourhost.yourcompany.com:4001</url>
      <username>invalidator</username>
      <password>invpwd</password>
  </invalidation>
</jesi-config>
```

Example 2: Configuration File Obtaining Password from jazn-data.xml The following example, instead of using clear text to specify the password, uses special "->" syntax to obtain the de-obfuscated password from the `jazn-data.xml` file. This example assumes `jazn-data.xml` contains a `<user>` element for the OracleAS Web Cache "invalidator" account:

```
<?xml version="1.0" ?>
<jesi-config>
  <invalidation>
      <url>http://yourhost.yourcompany.com:4001</url>
      <username>invalidator</username>
      <password>->invalidator</password>
  </invalidation>
</jesi-config>
```

JESI object Subtag

Use the required JESI `object` subtag of the JESI `invalidate` tag to specify cached objects to invalidate, according to either the complete URI or a URI prefix. Optionally, use JESI `cookie` subtags or JESI `header` subtags (or both) to specify further criteria for invalidation, based on cookie or HTTP header information.

Specify either the complete URI or the URI prefix in the `uri` attribute setting. Whether this field is interpreted as a full URI or as a prefix depends on the setting of the `prefix` attribute.

Syntax

```
<jesi:object uri = "uri_or_uriprefix"
          [ maxRemovalDelay = "value" ]
          [ prefix = "yes" | "no" ] >
```

Optional subtag (described in "JESI cookie Subtag" on page 6-29):

```
<jesi:cookie ... />
```

Optional subtag (described in "JESI header Subtag" on page 6-30):

```
<jesi:header ... />
```

```
</jesi:object>
```

Here is the syntax if you do not use either subtag:

```
<jesi:object uri = "uri_or_uriprefix"
          [ maxRemovalDelay = "value" ]
          [ prefix = "yes" | "no"] />
```

> **Notes:**
>
> ■ It is permissible to have multiple `object` tags within an `invalidate` tag.
>
> ■ It is permissible to have multiple `cookie` tags or `header` tags within an `object` tag.

Attributes

■ `uri` (required): Specifies either the complete URI of the page whose corresponding cached object is to be invalidated (if `prefix="no"`), or a URI prefix that specifies objects for multiple pages to be invalidated according to location (if `prefix="yes"`).

If a prefix is specified, then cached objects for all pages under that location are invalidated. For example, for a prefix of `"/abc/def"`, cached objects for all pages in the corresponding directory and any subdirectories are invalidated.

■ `prefix`: Set this to `"yes"` if the `uri` attribute is to be interpreted as a URI prefix only. Use the default `"no"` setting if the `uri` value is to be interpreted as a complete URI.

■ `maxRemovalDelay`: Specifies the maximum delay, in seconds, between the time when a cached object is invalidated and the time when it is removed and, therefore, can no longer be served by the ESI processor. This delay is 0 by default, for immediate removal.

JESI cookie Subtag

Use one or more JESI `cookie` subtags of the JESI `object` tag (which is a subtag of the JESI `invalidate` tag) if you want to use cookie information as further criteria for invalidation. This cookie information is in addition to the URI or URI prefix setting in the JESI `object` tag, and possibly in addition to JESI `header` tags as well. The `cookie` tag is useful for invalidating objects that have had multiple versions cached, based on cookie information.

The `cookie` tag has no body.

Syntax

```
<jesi:cookie name = "cookie_name"
            [ value = "cookie_value" ] />
```

> **Notes:**
> - It is permissible to have multiple `cookie` tags within an `object` tag.
> - Unlike most other JESI tag attributes, it is permissible for the `value` attribute to have a null or empty-string value.

Attributes

- `name` (required): This is the name of the cookie.
- `value`: This is the value of the cookie.

For each use of the `cookie` subtag, the request URL of the object to be invalidated must have a cookie that matches the `name` attribute setting and, if specified, the `value` attribute setting.

JESI header Subtag

Use one or more JESI `header` subtags of the JESI `object` tag (which is a subtag of the JESI `invalidate` tag) if you want to use HTTP/1.1 header information as further criteria for invalidation. This header information is in addition to the URI or URI prefix setting in the JESI `object` tag, and possibly in addition to JESI `cookie` tags as well. The `header` tag is useful for invalidating objects that have had multiple versions cached, based on header information.

The `header` tag has no body.

Syntax

```
<jesi:header name = "header_name"
            value = "header_value" />
```

> **Note:** It is permissible to have multiple `header` tags within an `object` tag.

Attributes

- `name` (required): This is the name of the HTTP/1.1 header.
- `value` (required): This is the value of the HTTP/1.1 header.

For each use of the `header` subtag, the request URL of the object to be invalidated must have a header that matches the `name` and `value` attribute settings.

Examples: Page Invalidation

This section provides examples of page invalidation using the JESI `invalidate` tag, its JESI `object` subtag, and the JESI `cookie` subtag of the JESI `object` tag.

Example 1: Page Invalidation This example invalidates a single object in the ESI processor, specified by its complete URI. (By default, the `uri` attribute of the `object` tag

specifies a full URI, not a URI prefix.) The JESI `invalidate` tag also specifies the URL for the cache server, and the user name and password for the invalidation account. In addition, it specifies that the invalidation response from the cache server should be displayed in the user's browser.

```
...
<jesi:invalidate url="http://yourhost.yourcompany.com:4001"
                 username="invalidator" password="invpwd"
                 output="browser">
     <jesi:object uri="/images/logo.gif"/>
</jesi:invalidate>
...
```

Example 2: Page Invalidation This example is equivalent to "Example 1: Page Invalidation" immediately above, but uses a configuration file to specify the cache server URL and login information.

```
...
<jesi:invalidate config="/myconfig.xml" output="browser">
     <jesi:object uri="/images/logo.gif"/>
</jesi:invalidate>
...
```

The JESI `invalidate` tag specifies an application-relative location for the configuration file. As an example, suppose `myconfig.xml` has the following content:

```
<?xml version="1.0" ?>
<jesi-config>
  <invalidation>
      <url>http://yourhost.yourcompany.com:4001</url>
      <username>invalidator</username>
      <password>invpwd</password>
  </invalidation>
</jesi-config>
```

Example 3: Page Invalidation This example invalidates all objects in the ESI processor, according to the URI prefix "/". It does not specify that the invalidation response should be displayed in the browser, so it will not be displayed at all.

```
...
<jesi:invalidate url="http://yourhost.yourcompany.com:4001"
                 username="invalidator" password="invpwd">
     <jesi:object uri="/" prefix="yes"/>
</jesi:invalidate>
...
```

Example 4: Page Invalidation This example invalidates a single object but allows it to be served stale for up to 30 minutes (1800 seconds).

```
...
<jesi:invalidate url="http://yourhost.yourcompany.com:4001"
                 username="invalidator" password="invpwd">
     <jesi:object uri="/images/logo.gif" maxRemovalDelay="1800"/>
</jesi:invalidate>
...
```

Example 5: Page Invalidation This example specifies the same object for invalidation as "Example 1: Page Invalidation" on page 6-30, but specifies that it should be invalidated only if its request URL has a cookie named `user_type` with the value `customer`.

```
...
```

```
<jesi:invalidate url="http://yourhost.yourcompany.com:4001"
                 username="invalidator" password="invpwd">
    <jesi:object uri="/images/logo.gif">
          <jesi:cookie name="user_type" value="customer"/>
    </jesi:object>
</jesi:invalidate>
...
```

Description of Tag for Page Personalization

To allow page customization when sharing the same cached page between multiple users, the ESI processor must be informed of dependencies by the page on cookie and session information. Cookie value replacement, for example, occurs in the ESI processor instead of in the Web server.

JESI personalize Tag

Use the JESI `personalize` tag to allow page customization, by directing the ESI processor to substitute cookie values from a current request before serving a cached page.

The effect of this tag is to insert an ESI placeholder with the cookie name and value into the response body. If the cookie that is specified in the `name` attribute is found in the request and has a non-null value, its value is used. If the cookie is not found in the request or has a null value, but a value is specified through the `default` attribute, then a new cookie is created and the `default` value is used. If the cookie did not previously exist and no `default` value is specified, the tag has no effect.

The `personalize` tag has no body.

Syntax

```
<jesi:personalize name = "cookie_name"
                  [ default = "default_value" ] />
```

> **Notes:**
>
> - For backward compatibility, the deprecated "`value`" form of the `default` attribute is accepted. The change from `value` to `default` was made to comply with the proposed JESI specification. It is likely that `value` will be desupported in a future release.
>
> - OC4J automatically places single quotes around the specified `default` (or `value`) setting to comply with the ESI specification. Prior to the OC4J 9.0.4 implementation, you had to include the single quotes as part of your setting.

Attributes

- `name` (required): Specifies the name of the cookie whose value is used as the basis for personalizing the page.

- `default`: This is an optional default value in case the cookie is not found or has a null value.

Example: Page Personalization

The following example shows usage of the JESI `personalize` tag:

```
<jesi:personalize name="user_id" default="guest" />
```

The corresponding ESI tag that is generated allows the ESI processor to find the necessary information. In this case, it looks for a cookie named `user_id` and retrieves its value. If it cannot find the cookie, it uses a default value of `"guest"`.

Handling this cookie-value replacement in the ESI processor allows the ESI processor to serve multiple customized pages from a single cached copy, without involving the application server.

JESI Tag Handling and JESI-to-ESI Conversion

JESI tag handler classes, supplied as part of the JESI tag library with OC4J, provide the bridge from JSP functionality to ESI functionality. Tag handlers generate ESI tags from JESI tags and, as appropriate, generate HTTP requests for invalidation, set HTTP response headers, and so on. Be aware, however, that there is not always a simple one-to-one mapping between JESI tags and ESI tags, or between JESI tag attributes and ESI tag attributes.

Example: JESI-to-ESI Conversion for Included Pages

As an example of JESI-to-ESI conversion, consider the following JSP code:

```
<p>BEGIN</p>
<jesi:control cache="no"/>
<jesi:include page="stocks.jsp" flush="true" />
<p>
<hr>
<jesi:include page="/weather.jsp" copyParam="true" flush="true" />
<p>
<hr>
<jesi:include page="../sales.jsp?tax=local" copyParam="true" flush="true" />
<p>END</p>
```

Assume that this JSP code is part of a page with the following URL:

```
http://host:port/application1/top.jsp
```

Further assume the following request:

```
http://host:port/application1/top.jsp?city=Washington_DC
```

In this case, the JESI `include` tag handler generates ESI markup such as in the following response.

In the response header:

```
Surrogate-Control: content="ESI/1.0",max-age=86400+0,no-store
```

In the response body:

```
<p>BEGIN</p>
<esi:include src="/application1/stocks.jsp"/>

<p>
<hr>
<esi:include src="/weather.jsp?city=Washington_DC"/>
```

```
<p>
<hr>
<esi:include src="/sales.jsp?tax=local&city=Washington_DC"/>

<p>END</p>
```

This response is read by the ESI processor before being delivered to the client. A `Surrogate-Control` header alerts the ESI processor that the response body contains ESI markup; therefore, the caching mechanism looks inside the response body for ESI tags. In addition, the `Surrogate-Control` header sets the cache directive to `no-store`, according to the `cache="no"` attribute setting. Expiration and maximum delay interval have no impact in this case.

In response to each of the three `esi:include` tags, the ESI processor makes an additional request to the URL that is specified. Each response is included into the top-level page, and only after that is the assembled page delivered to the client. Note that the client receives one response, but the cache initially makes four requests to obtain it. This might seem like a lot of overhead; however, the overall efficiency will be improved if many additional requests also use the same included pages, such as `weather.jsp`. No requests for these pages are required, because they are cached separately on the ESI processor.

Example: JESI-to-ESI Conversion for a Template and Fragment

Suppose that when employees connect to a corporate intranet site, the content of their pages is dynamic except for a few features that are present in every response. In particular, there is always a footer displaying the stock chart and latest business headlines for the company, and the business headlines are obtained from an external business news site. Because all returned pages will have to include the same information, and it is expensive to obtain, it is more efficient to cache the footer in the ESI processor.

The remainder of the page response is dynamic, incorporating the stock fragment in a slightly different way each time. To avoid having to rewrite the page, you can mark the footer as a JESI fragment and the enclosing page as a JESI template.

Also assume that a charity campaign is in progress and that the organizers want to display a bar chart showing their goal amount and the current donation amount as part of all corporate pages. This information is stored in a special database table and is updated twice a day. The chart is a good candidate to be an additional JESI fragment. Therefore, you would add a JESI `template` tag at the top of the page and use JESI `fragment` tags to enclose the fragments that are to be cached as separate entities.

Assume that the URL to the corporate page is as follows:

```
http://www.bigcorp.com/employee_page.jsp
```

Further assume that you have modified the page as follows:

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jesitaglib.tld"
          prefix="jesi" %>
<jesi:template cache="no" >

<p>BEGIN</p>
... some dynamic page content...
<jesi:fragment>
This_is_the_body_of_Charity_Chart
</jesi:fragment>
... some more dynamic content...
<jesi:fragment>
```

```
This_is_the_body_of_Business_Footer
</jesi:fragment>
</jesi:template>
<p>END</p>
```

When the page is requested, an HTTP response is generated as follows.

In the response header:

```
Surrogate-Control: content="ESI/1.0",max-age=86400+0,no-store
```

In the response body:

```
<p>BEGIN</p>
... some dynamic page content...
<esi:include src="/employee_page.jsp?__esi_fragment=1"/>
... some more dynamic content...
<esi:include src="/employee_page.jsp?__esi_fragment=2"/>
<p>END</p>
```

As with the JESI `include` example in "Example: JESI-to-ESI Conversion for Included Pages" on page 6-33, the ESI processor is alerted by the `Surrogate-Control` response header. Note the `no-store` directive, generated because of the `cache="no"` setting in the JESI `template` tag.

The ESI processor makes two additional requests, where it fetches and caches the two fragments. After that, the composite page is returned to the employee. When the employee works with the page again, the dynamic content will be newly generated but the chart and the footer will be served from the cache.

Note: `Surrogate-Control` headers are consumed by the ESI processor and are not seen in the final response to the client.

7

Web Object Cache Tags and API

This chapter describes the Web Object Cache, an application-level caching mechanism supplied with OC4J. For Web applications written in Java, you can use the Web Object Cache in conjunction with the Oracle Application Server Web Cache for increased speed and scalability.

The chapter consists of the following sections:

- Overview of the Web Object Cache
- Key Functionality of the Web Object Cache
- Attributes for Policy Specification and Use
- Web Object Cache Tag Descriptions
- Web Object Cache API Descriptions
- Cache Policy Descriptor
- Cache Repository Descriptor
- Configuration for Back-End Repository

For an overview of Web caching, including a discussion of the OracleAS Web Cache and Oracle Application Server Java Object Cache, see "Summary of Oracle Caching Support for Web Applications" on page 1-9.

Overview of the Web Object Cache

The OC4J Web Object Cache is a mechanism that allows Web applications written in Java to capture, store, reuse, post-process, and maintain the partial and intermediate results generated by a dynamic Web page, such as a JSP or servlet. For programming interfaces, it provides a tag library and a Java API.

The Web Object Cache works at the Java level and is closely integrated with the HTTP environment of JSP and servlet applications. Cached objects might consist of HTML or XML fragments, XML DOM objects, or Java serializable objects.

With the Web Object Cache programming interfaces, you can split Web pages into page blocks that define separate cache objects for finer control of caching. (The terms *block* and *object* are used somewhat interchangeably in this sense.) In this way, the application itself can control life span and other behavior of individual cache entities during runtime. Application developers have the best understanding of the life cycle patterns of their application Web pages, so are best suited to determine how to split pages into cache blocks. You can specify maintenance policies for cached objects either declaratively in an external file, the *cache policy descriptor*, or programmatically within the application itself.

The following sections provide an overview of the Web Object Cache:

- Benefits of the Web Object Cache
- Web Object Cache Components
- Cache Policy and Scope

Benefits of the Web Object Cache

> **Note:** The Web Object Cache is useful in particular scenarios and does not replace the need for other caching mechanisms, including the OracleAS Web Cache. For an overview of the Web Object Cache and how it relates to the OracleAS Web Cache and the Oracle Application Server Java Object Cache, including a discussion of when it is appropriate to use each one, see "Summary of Oracle Caching Support for Web Applications" on page 1-9.

Using the Web Object Cache can significantly reduce the amount of time spent in constructing page blocks or Java objects in dynamic applications, such as those with expensive intermediate operations like querying a database and formatting or transforming the results. Subsequent queries pull the information out of the cache, so the query and formatting do not have to be repeated.

Furthermore, developers can closely control the cache programmatically, through API calls or custom JSP tags. This can include controlling when cache entries are created, what they are named, when they expire, which users can see which cached data, and what operations can be applied to cached data before the results are served to the user.

Some kinds of Web applications benefit more than others by using the Web Object Cache, depending on the nature and use of their data. For example, applications such as catalog and directory browsing, delayed stock quotes, and personalized portals would particularly benefit. Applications such as real-time stock trading or real-time stock quotes, however, would not benefit, because the data has to be updated so frequently that the overhead of the caching operations would outweigh the benefits. (In these circumstances, however, the OracleAS Web Cache might still be useful because of its lighter overhead.)

In general, the Web Object Cache is most useful in the following situations:

- For special post-processing on cached data objects, such as XSLT or XML DOM operations
- For sharing data in a non-HTTP situation, such as reusing cached XML data or Java objects and sending the data to others through SMTP, JMS, AQ, or SOAP
- For special storage needs, such as storing cached data in a file system or database for persistent storage of data with a long lifetime
- For application-specific authorization, allowing different users to have different access rights to different data items, such as for a Web-based groupware application

  The application can have its own authorization scheme. The Web Object Cache is embedded within Java authorization logic.

Using the Web Object Cache in JSP pages is particularly convenient. JSP code generation can save much of the development effort.

Web Object Cache Components

The Web Object Cache consists of two main components:

- Cache repository

- Cache programming interfaces

This section also provides a brief introduction to the Oracle Application Server Java Object Cache, which is the default cache repository of the Web Object Cache.

Cache Repository

The cache repository is the component that is responsible for data storage, data distribution, and cache expiration. There can be multiple repository implementations for a programmable Web cache (such as the Web Object Cache), depending on the tier and platform. For example, the file system might be used for secondary storage in the middle tier, and database tables might be used for primary storage in the database tier.

The Web Object Cache uses the Oracle Application Server Java Object Cache as its default repository. The Java Object Cache is a general-purpose Java caching service and API designed for application use, with objects being accessible by name.

The Java Object Cache is a powerful and flexible programming facility. There are no restrictions on the types of objects that can be cached or the original source of the objects. The management of each object is easily customizable. Each object has a set of attributes such as the following:

- How the object is loaded into the cache

- Where the object is stored (in memory, on disk, or both)

- The lifetime, also known as the *time-to-live*, of the object

- Whom to notify when the object is invalidated

Objects can be invalidated as a group or individually.

For more information about the Java Object Cache, see the *Oracle Application Server Containers for J2EE Services Guide*.

> **Note:** See "Configuration for Back-End Repository" on page 7-41 for information about configuring the Java Object Cache or a file system as the back-end repository for the Web Object Cache.

Cache Programming Interfaces

The front-end caching interfaces are used through JSP pages and servlets to handle HTTP processing and to direct the semantics relating to the cache policy (rules and specifications determining how the cache works).

The OC4J Web Object Cache programming interfaces can be further divided as follows:

- Web Object Cache API

  This is the common layer across servlets and JSP pages, dealing with the HTTP semantics and cache policy. This layer communicates with the cache repository.

- Web Object Cache tag library

  This is a convenient wrapper, using JSP custom tag functionality, for the Web Object Cache API. Use custom tags in a JSP page to control the caching, with the API being called through the underlying tag handler classes.

This chapter describes these programming interfaces and their interaction with the cache repository. Cache tags are described in "Web Object Cache Tag Descriptions" on page 7-14. The underlying cache policy API is described in "Web Object Cache API Descriptions" on page 7-27. In servlets, you will use the underlying API; in JSP pages, you will typically use the more convenient tags.

Cache Policy and Scope

The *cache policy* is a set of specifications determining details of the cache and how it will behave. This includes the following:

- Cache scope
- Cache block naming rules
- Data expiration rules
- Cache repository name

You can set cache policy specifications (as described in "Attributes for Policy Specification and Use" on page 7-7) through any of the following:

- Cache tag attributes (for JSP pages)

    See "Web Object Cache Tag Descriptions" on page 7-14.

- Cache policy methods (for servlets)

    See "Web Object Cache API Descriptions" on page 7-27.

- External cache policy descriptor files (for JSP pages or servlets)

    See "Cache Policy Descriptor" on page 7-38.

A cache policy object—an instance of the `oracle.jsp.jwcache.CachePolicy` class—is created with policy settings based on these inputs. Because the expiration policy is part of the cache policy, each `CachePolicy` object includes an attribute that is an instance of the `oracle.jsp.jwcache.ExpirationPolicy` class.

Cache data can be of either *session scope*, where it is available to only the current HTTP session, or *application scope*, where it is available to all users of the application.

For example, consider an online banking application that caches the account balance. Only the current user is interested in this information, so `session` scope is appropriate.

By contrast, consider an online store with a welcome page that issues the same general product recommendations to all users. In this case, it is appropriate for the page to use a cache that has `application` scope.

Key Functionality of the Web Object Cache

The following sections discuss key areas of functionality of the Web Object Cache:

- Cache Block Naming: Implicit Versus Explicit
- Cache Block Runtime Functionality
- Data Invalidation and Expiration

Cache Block Naming: Implicit Versus Explicit

A cache block is associated with a cache block name, which can be determined either implicitly by the caching policy (generally advisable) or explicitly by your application

code. For retrieval, to avoid regenerating the page fragment in question, there is a lookup of the cache block name.

For implicit naming, there are two inputs:

- Cache policy

  A cache policy API layer performs naming logic.

- HTTP request object

  The caching logic borrows corresponding semantics from the standard Java servlet API.

For most situations, implicit naming will result in names that are sufficiently informative, because the HTTP request usually includes all the inputs to the Web application (inputs that determine what the application should generate).

Explicit naming might be desirable in some cases, however, such as when a group of users needs to share the same data. In this case, because relevant identification information might not be available directly from the user's HTTP request, an implicit cache name would not be useful. Instead, you can write code to explicitly generate a cache name that identifies the group. Preferably, the name-generation logic should still use only request parameters as input, not other states existing inside the application. This makes the semantics easier to follow and the code easier to debug.

Following is an example of explicit naming. In the `cache` tag, note the `name` attribute with a JSP expression that calls `someMethod()` to set the cache block name:

```
<ojsp:cache policy="/WEB-INF/policy1.cpd"
           name="<%= someObj.someMethod() %>" >
...static text...
<% // dynamic content ... %>
</ojsp:cache>
```

In the following example, because there is no `name` attribute in the `cache` tag, the cache block name will be determined implicitly according to the HTTP request and the cache policy:

```
<ojsp:cache policy="/WEB-INF/policy2.cpd" >
...static text...
<% // dynamic content ... %>
</ojsp:cache>
```

See "More About Cache Block Naming and the autoType Attribute" on page 7-10 for more information.

> **Note:** Cache blocks can be nested. In this case, the logic of the inner cache block will be executed only when the content of the outer block must be regenerated.

Cloneable Cache Objects

The OC4J Web Object Cache provides an interface, `oracle.jsp.jwcache.CloneableCacheObj,` that you can implement in serializable cache objects that you want to be cloneable. For mutable objects that are cached without being serialized, cloning is useful in providing a complete and hierarchical copy of the cache object. This section explains the usefulness of cloneability, first covering some necessary background information.

Memory-Oriented Repositories Versus Secondary Storage Repositories

There are two categories of repositories that can be used as the back-end of the Web Object Cache:

■ Secondary storage cache repository such as a file system repository

■ Memory-oriented cache repository such as the Oracle Application Server Java Object Cache, the default repository of the Web Object Cache

A secondary storage repository requires Java serialization during cache operations. During storage to the cache, objects are serialized into the repository; during retrieval from the cache, they are deserialized into memory. Therefore, as a result of the serialization/deserialization process, a complete and distinct copy of the cache object is automatically created during each cache operation.

This is not the case when you store or retrieve cache objects to or from a memory-oriented repository. With a memory-oriented repository, the identical object in the user application will be stored to the cache, or the identical object in the cache will be retrieved for the user. By default, no copy is made. If there are multiple retrievals, all retrievals share the same object.

Advantages in Cloning Copies of Cache Objects

In many cases in your applications, you will want to ensure that different retrievals use different copies of a cache object. There are two key reasons for this:

■ If the identical cache object is shared across multiple retrievals, changes made to the data in one place might unintentionally affect values retrieved and used elsewhere.

■ If the identical cache object is shared across multiple retrievals, then multiple Java threads might access the same object simultaneously. This would result in thread safety issues if the original object design was not thread-safe. Perhaps, for example, the object was originally intended for page-scope or request-scope usage only, where there could be only one thread for each object. This thread-behavior assumption would be violated.

To avoid these possible problems, use complete and hierarchical copies when you store and retrieve generic Java serializable data to or from a memory-oriented repository. "Complete and hierarchical" means copying not just the direct members referenced by the object, but also any indirect variables that are referenced. For example, assume an object `xyz` has a `java.util.Vector` instance as a member variable. Cloning a complete and hierarchical copy involves copying not just the `Vector` instance itself, but also all mutable objects or elements referenced by the `Vector` instance.

Use of the CloneableCacheObject Interface

If you implement the `CloneableCacheObject` interface and its `cloneCacheObj()` method in your cache objects, then the Web Object Cache will automatically call `cloneCacheObj()` to make a complete and hierarchical copy of each cache object whenever it is stored to or retrieved from a memory-oriented cache repository.

Cache Block Runtime Functionality

During runtime, when a Web Object Cache cache tag is encountered, the tag handler checks whether a corresponding cache object exists and was created recently enough to reuse. If so, the code in the body of the tag is not executed; instead, the cache object is reused. But if the cache object does not exist or is too old, the tag body code will be

executed to generate a new object (page fragment, XML DOM object, or Java serializable object). Then this freshly generated object will be captured, such as through special buffer writing or object passing, and stored into the cache.

If computations in content generation are costly, such as for a complicated database query, and the life span of the cache is appropriate, so that the cached data is reusable, then the Web Object Cache can save significant amounts of time and system resources. Application speed and throughput will be greatly improved.

Data Invalidation and Expiration

You can set up cache blocks to expire after a specified duration or at a specified time, or they can be invalidated explicitly by a method call or tag invocation.

Cache Block Expiration

Because cache blocks mainly consist of semi-static fragments of information, the Oracle implementation does not require a tightly coherent expiration model. A looser model typically provides acceptable results and requires less synchronization overhead.

There are two categories of expiration for data in Web Object Cache blocks:

- Duration (*time-to-live*): Expiration occurs after data has been in the cache for a specified amount of time.

- Fixed time/day: Expiration occurs regularly at a set time, such as at a specified time each day or on a specified day each week.

Expiration details are determined by the settings of attributes in an instance of the `oracle.jsp.jwcache.ExpirationPolicy` class. This `ExpirationPolicy` object is an attribute of the `CachePolicy` object associated with the cache block. See "Expiration Policy Attributes" on page 7-12.

In JSP pages, you can set `ExpirationPolicy` attributes through attributes of the Web Object Cache cache tags. In servlets, you can use methods of the `ExpirationPolicy` object directly. (See "ExpirationPolicy Methods" on page 7-33.) Alternatively, you can set `ExpirationPolicy` attributes through a cache policy descriptor. (See "Cache Policy Descriptor" on page 7-38.)

Cache Block Invalidation

Instead of depending on expiration to invalidate a cache, you can invalidate it explicitly in one of the following ways:

- Use the `invalidateCache` tag. See "Web Object Cache invalidateCache Tag" on page 7-23.

- Use the overloaded `invalidateCache()`, `invalidateCacheLike()`, or `invalidateCacheOtherPathLike()` method of a `CachePolicy` instance to explicitly invalidate one or more cache blocks. See "CachePolicy Methods" on page 7-28.

Attributes for Policy Specification and Use

This section describes cache policy attributes—specifically, attributes of the `CachePolicy` and `ExpirationPolicy` classes. You can set these attributes through custom tags in JSP pages, directly through the provided Java API in servlets, or through a cache policy descriptor file.

Cache Policy Attributes

Cache policies, introduced in "Cache Policy and Scope" on page 7-4, consist of the details that determine how cache blocks behave. You can set cache policy attributes in several ways, as described in subsequent sections:

- In JSP pages through custom tags

  See "Web Object Cache Tag Descriptions" on page 7-14.

- In servlets through method calls

  See "CachePolicy Methods" on page 7-28.

- Through a cache policy descriptor file

  See "Cache Policy Descriptor" on page 7-38.

Specification of cache policy settings results in the creation of a cache policy object, which includes an expiration policy object as one of its attributes. Following is abbreviated code for the `CachePolicy` class (in package `oracle.jsp.jwcache`), for illustration purposes only, showing the names of the cache policy attributes:

```
class CachePolicy
{
   boolean ignoreCache;
   int scope;
   int autoType;
   String selectedParameters[];
   String selectedCookies[];
   Date reusableTimeStamp;
   long reusableDeltaTime;
   ExpirationPolicy expirationPolicy;
   String cacheRepositoryName;
   boolean reportException;
}
```

> **Note:** The names documented below for integer constants are for servlet usage. Different names can be used for the Web Object Cache tags. See "Web Object Cache cache Tag" on page 7-15.

Cache Policy Attribute Descriptions

Table 7–1 describes cache policy object attributes.

Table 7–1   Cache Policy Attribute Descriptions

Attribute	Type	Description
ignoreCache	boolean	This is for use during development only. When making frequent code changes, set this to `true` to disable the cache, typically so that results that were generated prior to your changes will not be returned. Default: `false`
scope	int	Specifies the scope of the cache. Use the integer constant `SCOPE_SESSION` for the cache block to be accessible only to the current HTTP session, or `SCOPE_APP` for the cache block to be accessible to all HTTP sessions of the application. Default: `SCOPE_APP`

Table 7–1   (Cont.) Cache Policy Attribute Descriptions

Attribute	Type	Description
autoType	int	Specifies whether the cache block is named explicitly or implicitly and how properties of the HTTP request are used in cache block naming (for implicit naming). The name is relevant in determining when the cache is reused for subsequent requests. See "More About Cache Block Naming and the autoType Attribute" on page 7-10.
		Default: implicitly, according to the URI plus all parameters plus selected cookies (`TYPE_URI_ALLPARAM`)
selectedParameters[]	String []	These are selected request parameter names used in cache block naming, used in conjunction with `autoType`. See "More About Cache Block Naming and the autoType Attribute" on page 7-10.
		Default: `null`
selectedCookies[]	String[]	These are selected cookie names used in cache block naming, used in conjunction with `autoType`. See "More About Cache Block Naming and the autoType Attribute" on page 7-10.
		Default: `null`
reusableTimeStamp	java.util.Date	This is an absolute time limit for cache usability, where any cache block created prior to that time will not be reused. Instead, data is regenerated but the cache block is unaltered. See "More About reusableTimeStamp and reusableDeltaTime" on page 7-12.
		Note the following regarding `reusableTimeStamp`:
		■ It can be expressed as milliseconds between midnight, January 1, 1970 and the desired absolute time limit, or as a `java.util.Date` instance. Additional convenient formats are available through the `cache` tag. (See "Web Object Cache Tag Descriptions" on page 7-14.)
		■ It takes precedence over `reusableDeltaTime`.
		■ If its value is set as the integer constant `REUSABLE_ALWAYS` or the string constant `REUSABLE_IGNORED`, then cache entries are always reusable, for as long as they remain in the cache.
		■ It is not available through the XML cache policy descriptor file.
		Default: always reusable

Table 7–1   (Cont.) Cache Policy Attribute Descriptions

Attribute	Type	Description
reusableDeltaTime	long	This is a relative time limit for cache usability, where a cache block is not reused if the difference between cache block creation time and current time is greater than `reusableDeltaTime`. Instead, data is regenerated but the cache block is unaltered. See "More About reusableTimeStamp and reusableDeltaTime" on page 7-12.
		Note the following regarding `reusableDeltaTime`:
		■ It is specified in seconds.
		■ The `reusableTimeStamp` attribute overrides it.
		■ If its value is set as the integer constant `REUSABLE_ALWAYS` or the string constant `REUSABLE_IGNORED`, then cache entries are always reusable, for as long as they remain in the cache.
		Default: always reusable
expirationPolicy	ExpirationPolicy	This is an expiration policy object (an instance of `oracle.jsp.jwcache.ExpirationPolicy`), which specifies circumstances under which the repository will remove cache blocks from storage.
		Default: the default expiration policy object
		For information about expiration policy objects, parameters, and defaults, see "Expiration Policy Attributes" on page 7-12.
cacheRepositoryName	String	This is the name of the cache repository. Each cache policy can use its own repository.
		The configurations of cache repositories are defined in the `/WEB-INF/wcache.xml` file.
		Default: "DefaultCacheRepository"
reportException	boolean	A `false` setting of this attribute results in most cache operation failures being silent, without any exception being reported to the browser.
		Default: `true`

More About Cache Block Naming and the autoType Attribute

As discussed in "Cache Block Naming: Implicit Versus Explicit" on page 7-4, cache blocks can be named either implicitly, sometimes called *auto-naming*, or explicitly, sometimes called *user-naming*.

More specifically, there are six ways for cache blocks to be named. Explicit naming is the first way. Specify this with an autoType setting of TYPE_USERSPECIFIED (an integer constant).

The other five ways are variations of implicit naming:

■ Implicit naming with only the request URI being used in the name

 Specify this with an autoType setting of TYPE_URI_ONLY.

■ Implicit naming according to the following:

 Request URI + query string + selected cookies

Specify this with an `autoType` setting of `TYPE_URI_QUERYSTR`. Specify the cookies in the `selectedCookies[]` attribute.

- Implicit naming according to the following:

  Request URI + all parameters + selected cookies (**default**)

  Specify this with an `autoType` setting of `TYPE_URI_ALLPARAM`. Specify the cookies in the `selectedCookies[]` attribute.

- Implicit naming according to the following:

  Request URI + selected parameters + selected cookies

  Specify this with an `autoType` setting of `TYPE_URI_SELECTEDPARAM`. Specify the parameters in the `selectedParameters[]` attribute and the cookies in the `selectedCookies[]` attribute.

- Implicit naming according to the following:

  Request URI + all but excluded parameters + selected cookies

  Specify this with an `autoType` setting of `TYPE_URI_EXCLUDEDPARAM`. Specify the cookies in the `selectedCookies[]` attribute and the excluded parameters in the `selectedParameters[]` attribute.

As an example, assume that you have developed a JSP page, `welcome.jsp`, with a personalized greeting for each user. The data with the personalized greeting is the only cache block in the page. Further assume that you have specified "request URI + selected parameters + selected cookies" naming, with `user` as the only selected parameter for cache block naming and no selected cookies for naming.

Now assume the page is requested as follows:

```
http://host:port/a.jsp?user=Amy
```

In this case, `a.jsp?user=Amy` becomes the cache block name.

Now assume that the page is later requested by another user, as follows:

```
http://host:port/a.jsp?user=Brian
```

This will not reuse the "Amy" cache, because the value of `user` is different. Instead, a new cache block is created with `a.jsp?user=Brian` as the name.

Now assume a later request by the first user, as follows:

```
http://host:port/a.jsp?mypar=3&user=Amy
```

Because the user is again Amy, this request will reuse the first cache, displaying Amy's customized information without having to regenerate it. The `mypar` parameter is irrelevant to the caching mechanism because it was not included in the `selectedParameters[]` list of the cache policy object, presumably because the value of `mypar` is not relevant in terms of cacheable page output.

Now assume the following subsequent request:

```
http://host:port/a.jsp?yourpar=4&user=Brian&hello=true&foo=barfly
```

Because the user is again Brian, this request will reuse the second cache, displaying Brian's customized information without having to regenerate it. The `yourpar`, `hello`, and `foo` parameters are irrelevant to the caching mechanism because they were not included in the `selectedParameters[]` list of the cache policy object.

More About reusableTimeStamp and reusableDeltaTime

Be aware that the concept of *reusable* is different than the concept of *time-to-live* (TTL) and is intended for more advanced use. Time-to-live, which controls the general lifetime of a cache, is described in "Expiration Policy Attributes" on page 7-12. Usually time-to-live is all that is required to appropriately limit the use of cached data.

The attributes for reusability—reusableTimeStamp and reusableDeltaTime—are intended for more specialized use and do not affect the expiration or invalidation of cached data. As an example, consider a situation where different users have different requirements for how up-to-date a Web report is. Assume that most users can accept a report produced anytime within the past day, and that they all want to be looking at the same version so they can compare figures. An appropriate TTL value, then, would be "one day".

Also presume, however, that there is a small group of privileged users for whom the data is much more time-sensitive. They want to have information that is no more than one hour old.

In this case, although TTL is set to "one day" for all users, there can be a reusableDeltaTime setting of "one hour" for the privileged users, which will result in the cache not being used for them if the data is more than one hour old. Remember, though, that reusableTimeStamp and reusableDeltaTime do *not* expire the cache or otherwise affect it. The cached data can still be used for non-privileged users, according to the time-to-live.

It is up to the application logic to set appropriate values of reusableTimeStamp and reusableDeltaTime for the privileged user group.

Expiration Policy Attributes

Expiration policies are introduced in "Data Invalidation and Expiration" on page 7-7. Expiration policies contain the details that determine when cache blocks expire, at which point their data should no longer be used and the data should be regenerated instead. (Note that for most discussion, you can think of the expiration policies as being part of the cache policies.) ExpirationPolicy attributes, as with CachePolicy attributes, can be set in any of the following ways:

- In JSP pages through custom tags

  See "Web Object Cache Tag Descriptions" on page 7-14.

- In servlets through method calls

  See "ExpirationPolicy Methods" on page 7-33.

- Through a cache policy descriptor file

  See "Cache Policy Descriptor" on page 7-38.

The following abbreviated code for the ExpirationPolicy class (in package oracle.jsp.jwcache), provided for illustration purposes only, shows the names of the expiration policy attributes:

```
class ExpirationPolicy
{
   int expirationType;
   long TTL;
   long timeInaDay;
   int dayInaWeek;
   int dayInaMonth;
   boolean writeThrough;
}
```

Table 7–2 describes the expiration policy object attributes.

Note: The names documented below for integer constants are for servlet usage. Different names can be used for the Web Object Cache tags. See "Web Object Cache cache Tag" on page 7-15.

Table 7–2   Expiration Policy Attribute Descriptions

Attribute	Type	Description
expirationType	int	This is the type of expiration policy and is one of the following, where TYPE_*XXX* values are integer constants: ■ Time-to-live, to expire after a certain amount of time according to the TTL attribute, specified with an expirationType setting of TYPE_TTL ■ Daily, to expire within a day at a certain time according to the timeInaDay attribute, specified with an expirationType setting of TYPE_DAILY ■ Weekly, to expire within a week on a certain day at a certain time according to the dayInaWeek and timeInaDay attributes, specified with an expirationType setting of TYPE_WEEKLY ■ Monthly, to expire within a month on a certain date at a certain time according to the dayInaMonth and timeInaDay attributes, specified with an expirationType setting of TYPE_MONTHLY Default: time-to-live
TTL	long	This is time-to-live, the amount of time the cache block is good for, expressed in seconds. The value must be a positive number. Default: 300 (5 minutes)
timeInaDay	long	This is the time of day used for daily, weekly, or monthly expiration, expressed in seconds from midnight, where 0 is 00:00:00 (midnight) and 86399 is 23:59:59. Default: 300 (00:05:00); ignored if expirationType=TYPE_TTL
dayInaWeek	int	This is the day of the week for weekly expiration, at the specified timeInaDay. Possible values are WEEKLY_SUNDAY, WEEKLY_MONDAY, WEEKLY_TUESDAY, WEEKLY_WEDNESDAY, WEEKLY_THURSDAY, WEEKLY_FRIDAY, or WEEKLY_SATURDAY (integer constants). Default: Wednesday; ignored unless expirationType=TYPE_WEEKLY

Table 7–2   (Cont.) Expiration Policy Attribute Descriptions

Attribute	Type	Description
dayInaMonth	int	This is the date of the month for monthly expiration, such as 10 for the 10th of each month, at the specified `timeInaDay`. The maximum setting is the number of days in the month when the cache block is created. For example, if a cache block is created in June and `dayInaMonth` has a setting of 31, then its effective value will be 30.
		Default: 10; ignored unless `expirationType=TYPE_MONTHLY`
writeThrough	boolean	This flag specifies whether the cache repository should treat the cache entry as a write-through cache, writing it immediately into secondary storage such as a file system or database. Set this to `true` for write-through mode. A write-through cache will survive a server restart or power failure.
		With a `false` setting, the cache entry is treated as a delayed-write cache, which is appropriate for caches that have a short life span, such as 5 or 10 minutes, and are not overly expensive to recompute.
		Note that some cache repositories might not support write-through mode; others might always use write-through mode.
		Default: `true`

Web Object Cache Tag Descriptions

From JSP pages, you can specify cache policy settings, expiration policy settings, and explicit invalidation through custom tags provided with OC4J. The following sections describe the tags:

- Cache Tag Descriptions

- Cache Invalidation Tag Description

Note the following requirements for the Web Object Cache tag library:

- The Web Object Cache classes are in the file `ojsputil.jar`, which is supplied with OC4J and is located in the "well-known" tag library directory. Verify that this file is installed and in your classpath.

- To use the Oracle Application Server Java Object Cache as the back-end repository, the file `cache.jar` must be installed and in your classpath. This file also comes with OC4J. In the OC4J 10.1.2 implementation, `cache.jar` is listed in the manifest classpath of `oc4j.jar`. If the Web Object Cache tag library is loaded by OC4J, then no action on your part is necessary.

- The tag library descriptor, `jwcache.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `jwcache.tld` is the following:

  ```
  http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jwcache.tld
  ```

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

> **Notes:**
>
> - The prefix "ojsp:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.
>
> - See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

Cache Tag Descriptions

This section describes the following tags:

- `cache`

  This tag is for general character-based caching (HTML or XML fragments).

- `cacheXMLObj`

  This tag is for caching XML objects; its parameters are a superset of the `cache` tag parameters. Because the Web Object Cache is particularly useful when post-processing XML documents, you will likely use the `cacheXMLObj` tag more often than the `cache` tag.

- `useCacheObj`

  This tag is for general caching of Java serializable objects. Some of the semantics and syntax are patterned after the standard `jsp:useBean` tag.

- `cacheInclude`

  This tag combines the functionality of the `cache` tag with that of the standard `jsp:include` tag.

This section also describes conditional execution of code within the cache tags, possible resulting problems, the workaround of dividing cache blocks into individual JSP pages, and, optionally, using the `cacheInclude` tag to combine the pages together appropriately.

Web Object Cache cache Tag

This section documents the syntax and attributes of the `cache` tag, which you can use to set up general caching in a JSP application, in contrast to the caching of XML objects or Java serializable object.

> **Note:** For caching XML objects, use the `cacheXMLObj` tag instead. For caching Java serializable objects, use the `useCacheObj` tag. These tags support all the `cache` tag attributes described here. See "Web Object Cache cacheXMLObj Tag" on page 7-18 and "Web Object Cache useCacheObj Tag" on page 7-20.

Syntax

```
<ojsp:cache
    [ policy = "filename" ]
    [ ignoreCache = "true" | "false" ]
    [ invalidateCache = "true" | "false" ]
    [ scope = "application" | "session" ]
    [ autoType = "user" | "URI" | "URI_query" | "URI_allParam" |
                 "URI_selectedParam" | "URI_excludedParam" ]
```

```
         [ selectedParam = "space-delimited_string_of_parameter_names" ]
         [ selectedCookies = "space-delimited_string_of_cookie_names" ]
         [ reusableTimeStamp = "yyyy.mm.dd hh:mm:ss z" |
                               "yyyy.mm.dd hh:mm:ss" | "yyyy.mm.dd"| "ignored" ]
         [ reusableDeltaTime = "number" | "ignored" ]
         [ name = "blockname" ]
         [ expirationType = "TTL" | "daily" | "weekly" | "monthly" ]
         [ TTL = "number" ]
         [ timeInaDay = "number" ]
         [ dayInaWeek = "Sunday" | "Monday" | "Tuesday" | "Wednesday" |
                        "Thursday" | "Friday" | "Saturday" ]
         [ dayInaMonth = "number" ]
         [ writeThrough = "true" | "false" ]
         [ printCacheBlockInfo = "true" | "false" ]
         [ printCachePolicy = "true" | "false" ]
         [ cacheRepositoryName = "name" ]
         [ reportException = "true" | "false" ] >

...Code for cache block...

</ojsp:cache>
```

> **Note:** Key default values are as follows: TTL 300 seconds,
> dayInaMonth 10 (10th of the month), cache repository name
> DefaultCacheRepository.

Attributes

Most of the parameters of the cache tag correspond to attributes in the CachePolicy
or ExpirationPolicy class, described earlier in this chapter (as referenced below).

- policy: Optionally use this to specify a cache policy descriptor, the settings of
  which would be used in defining the cache policy. You can use a cache policy
  descriptor instead of using the various individual cache tag attribute settings, or to
  establish default values that you can optionally override through tag attribute
  settings.

  Specify the descriptor file name according to JSP application-relative syntax. You
  can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer
  Pages Developer's Guide* for information about application-relative syntax.

  Here is a simple example of a cache policy descriptor:

  ```
  <!--
  test-policy.cpd
  -->

  <cachePolicy scope="application">
     <expirationPolicy expirationType="TTL" TTL="25" timeInaDay="00:10:00"
                       writeThrough="true" />
  </cachePolicy>
  ```

  See "Cache Policy Descriptor" on page 7-38 for more information.

- ignoreCache: See "Cache Policy Attributes" on page 7-8.

- invalidateCache: Enable this flag for the corresponding cache block (any
  pre-existing cache block with the same name) to first be invalidated. This is
  particularly useful where implicit cache block naming is used, but can also be used

for explicit names by specifying the cache block name in the `name` attribute of the `cache` tag. The default setting is `"false"`.

> **Note:** Do not confuse this attribute with the more general-purpose `invalidateCache` tag. See "Web Object Cache invalidateCache Tag" on page 7-23. The `invalidateCache` attribute is for more specialized or advanced use to invalidate individual cache blocks.

- `scope`: See "Cache Policy Attributes" on page 7-8.

- `autoType`: See "Cache Policy Attributes" on page 7-8. The correspondence between tag attribute settings and class attribute values (integer constants) is as follows:

  - The `user` setting is equivalent to `TYPE_USERSPECIFIED`.

  - `URI` is equivalent to `TYPE_URI_ONLY`.

  - `URI_query` is equivalent to `TYPE_URI_QUERYSTR`.

  - `URI_allParam` is equivalent to `TYPE_URI_ALLPARAM`.

  - `URI_selectedParam` is equivalent to `TYPE_URI_SELECTEDPARAM`.

  - `URI_excludedParam` is equivalent to `TYPE_URI_EXCLUDEDPARAM`.

- `selectedParam`: See "Cache Policy Attributes" on page 7-8.

- `selectedCookies`: See "Cache Policy Attributes" on page 7-8.

- `reusableTimeStamp`: See "Cache Policy Attributes" on page 7-8.

- `reusableDeltaTime`: See "Cache Policy Attributes" on page 7-8.

- `name`: Where you use explicit cache-block naming, use the `name` parameter to specify the block name.

- `expirationType`: See "Expiration Policy Attributes" on page 7-12.

- `TTL`: See "Expiration Policy Attributes" on page 7-12.

- `timeInaDay`: See "Expiration Policy Attributes" on page 7-12.

- `dayInaWeek`: See "Expiration Policy Attributes" on page 7-12.

- `dayInaMonth`: See "Expiration Policy Attributes" on page 7-12.

- `writeThrough`: See "Expiration Policy Attributes" on page 7-12.

- `printCacheBlockInfo` (for debugging): Enabling this parameter results in printing of the internal cache name, creation time, and expiration time of the cache block, within HTML or XML comment constructs. The default setting is `"false"`.

- `printCachePolicy` (for debugging): Enabling this parameter results in printing of the values of all cache policy attributes for this cache block, within HTML or XML comment constructs. The default setting is `"false"`.

- `cacheRepositoryName`: See "Cache Policy Attributes" on page 7-8.

- `reportException`: See "Cache Policy Attributes" on page 7-8.

Attribute Usage Notes

- The `name` attribute is relevant only when `autoType` is set to `user`.

- The selectedParam attribute is relevant only when autoType is set to URI_selectedParam or URI_excludedParam.

- The selectedCookies attribute is not relevant when autoType is set to user or URI.

- The timeInaDay attribute is not relevant when expirationType is set to TTL.

- The dayInaWeek attribute is relevant only when expirationType is set to weekly.

- The dayInaMonth attribute is relevant only when expirationType is set to monthly.

Example: cache Tag

This example lists and caches a set of items, using the cache tag.

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jwcache.tld"
           prefix="ojsp" %>
<title>listitem.jsp</title>
<%
   String itemid=request.getParameter("itemid");
   if (itemid==null) {
     out.println("Please select a category from the above drop down box.");
     return;
   }
%>
<% long l1=(new java.util.Date()).getTime(); %>
<ojsp:cache autoType="URI_selectedParam" selectedParam="itemid"
  printCacheBlockInfo="true" printCachePolicy="true"
  policy="/WEB-INF/test-policy.cpd"
>
   Item List: <b><%= itemid %></b><br>
   Time: <%= new java.util.Date() %>
   <br>
   <jsp:useBean class="java.util.Hashtable" id="table" scope="application" />
   <hr>
   <%
     Vector list=(Vector) table.get(itemid);
     if (list==null) {
       out.println("No such item!");
     }
     else {
       for (int i=0; i<list.size(); i++) {
   %>
   <%= list.elementAt(i) %><br>
   <%
       }
     }
   %>
   timestamp:<%= new java.util.Date() %>
   <br>
</ojsp:cache>
<% long l2=(new java.util.Date()).getTime(); %>
Time for general cache operation:<%= l2-l1 %>
<br>
```

Web Object Cache cacheXMLObj Tag

Generally speaking, use the cacheXMLObj tag instead of the cache tag if you are caching XML DOM objects.

The `cacheXMLObj` tag supports all the `cache` tag attributes described in "Web Object Cache cache Tag" on page 7-15, as well as the attributes described here.

Syntax (in addition to that of the cache tag)

```
<ojsp:cacheXMLObj
    ...
    [ fromXMLObjName = "objectname" ]
    [ toXMLObjName = "objectname" ]
    [ toWriter = "true" | "false" ] >

[...Code for cache block...]

</ojsp:cacheXMLObj>
```

Notes:

- This tag can optionally be in the form of a single tag with no body, in which case the `fromXMLObjName` attribute can be used for input instead:

  ```
  <ojsp:cacheXMLObj ... fromXMLObjName="..." ... />
  ```

- For convenience, this tag is duplicated in the XML tag library, being defined in the `xml.tld` tag library descriptor file.

- This tag can act as both an XML producer and an XML consumer. Do not use `fromXMLObjName` and `toXMLObjName` if the XML object is being passed implicitly. (See "XML Producers and XML Consumers" on page 5-1.)

Attributes (in addition to those of the cache tag)

- `fromXMLObjName`: For explicit passing, specify the name of the XML input object being passed to the cache (from the `pageContext` object).

- `toXMLObjName`: For explicit passing, specify the name of the XML output object being passed from the cache (to the `pageContext` object).

- `toWriter`: Set this to "`true`" to write the XML object to a JSP writer to output directly to the user's browser. The default value is "`false`".

Note: The `cacheXMLObj` tag is one of several custom tags supplied with OC4J that are XML-related, meaning these tags sometimes (or always) take an XML object as input or create one as output. Other such tags include the SQL library `dbQuery` tag, which can output query results as an XML DOM object, and the XML library `transform` and `styleSheet` tags, which can take an XML object as input and use XSLT transformation to create another XML object or a JSP writer as output. These tags are consistent in having a `fromXMLObjName` attribute and a `toXMLObjName` attribute for explicit passing of XML data. For general information, see "XML Producers and XML Consumers" on page 5-1.

Example: cacheXMLObj Tag

This example uses Web Object Cache tags, JESI tags, and tags from the XML and SQL tag libraries. (For JESI tag descriptions, see "Oracle JESI Tag Descriptions" on page 6-13. For a description of the XML transform tag, see "XML Utility Tags" on page 5-3. For SQL tag descriptions, see "SQL Tags for Data Access" on page 4-11.)

The SQL dbOpen and SQL dbQuery tags connect to the database and execute a query. The cacheXMLObj tag caches the XML DOM object produced by the query. In subsequent executions (for output through different stylesheets, for example), the query does not have to be reexecuted, because the DOM object can be retrieved from the Web Object Cache. The XML transform tag outputs the query results according to an XML stylesheet, specified through a variable. The JESI fragment tag encloses HTML output to be cached, which does not require application-level caching. The JESI template tag disables caching outside the fragment, through the cache="no" setting.

```
<jesi:template cache="no">
<% String userStyleLoc="style/rowset.xsl"; %>
<h3>Transform DBQuery Tag Example</h3>
<h4>Current Time=<%= new java.util.Date() %></h4>
<jesi:fragment expiration="60">
<!-- You can cache HTML in OracleAS Web Cache with JESI
  or you can cache it in Oracle Web Object Cache -->
<h4>Cached Time=<%= new java.util.Date() %></h4>
<sql:dbOpen connId="conn1" dataSource="<%= dataSrcStr %>" />
<xml:transform href="<%= userStyleLoc %>" >
<%-- The XML DOM object is produced by dbQuery
  And, the DOM object is cached in Oracle Web Object Cache.
  XSLT is performed on the cached object. --%>
  <ojsp:cacheXMLObj TTL="60" toWriter="false">
    <sql:dbQuery connId="conn1" output="xml" queryId="myquery" >
        select ENAME, EMPNO from EMP
    </sql:dbQuery>
  </ojsp:cacheXMLObj>
</xml:transform>
<sql:dbCloseQuery queryId="myquery" />
<sql:dbClose connId="con1" />
</jesi:fragment>
</jesi:template>
```

Web Object Cache useCacheObj Tag

Use the useCacheObj tag to cache any Java serializable object.

The useCacheObj tag supports all the cache tag attributes described in "Web Object Cache cache Tag" on page 7-15, as well as the attributes described here.

Syntax (in addition to that of the cache tag)

```
<ojsp:useCacheObj
   ...
     type="classname"
     id = "instancename"
  [ cacheScope = "application" | "session" ] >

...Code for cache block...

</ojsp:useCacheObj>
```

> **Note:** The `id` and `type` attributes are not request-time attributes, so cannot be set using JSP runtime expressions.

Attributes (in addition to those of the cache tag)

- `type` (required): Specify the class name of the Java object to cache.

- `id` (required): Specify the instance name of the Java object to cache.

- `cacheScope`: This attribute has the same usage as the `scope` attribute in the `cache` and `cacheXMLObj` tags. See "Cache Policy Attributes" on page 7-8.

The `type` and `id` attributes here are used similarly to the `type` (or `class`) and `id` attributes in a `jsp:useBean` tag.

Example: useCacheObj Tag

```
<ojsp:useCacheObj id="a2" policy="/WEB-INF/test-policy.cpd"
  type="examples.RStrArray" >
<%
  // create a temp writeable array
  WStrArray tmpa2=new WStrArray(3);
  tmpa2.setStr(2,request.getParameter("testing4"));
  tmpa2.setStr(1,"def");
  tmpa2.setStr(0, (new java.util.Date()).toString() );
  // create a readonly copy for the cache
  a2=new RStrArray(tmpa2);
  // storing the a2 into pagecontext
  // so useCacheObj tag can pick it up
  pageContext.setAttribute("a2",a2);
%>
</ojsp:useCacheObj>
```

Conditional Execution of Code Inside the Cache Tags

Be aware that code inside a cache tag (`cache`, `cacheXMLObj`, or `useCacheObj`) is executed conditionally. In particular:

- Any code inside a cache tag is executed only when the associated cache block is *not* reused.

  Consider the following example:

  ```
  <% String str=null; %>
  <% ojsp:useCacheObj ... >
     <% str = "abc"; //...more Java code...%>
  </ojsp:useCacheObj>
  <% out.print(str.length()); // May cause null pointer exception
  ```

  If the cache is available and reused, the code to properly initialize the string `str` is not executed.

- If you put a method-based variable declaration inside a cache tag, the variable is not available outside the tag.

  Consider the following example:

  ```
  <ojsp:useCacheObj ... >
     <% String str = "abc"; //...more Java code...%>
  </ojsp:useCacheObj>
  <% // String str will not be available here %>
  ```

If you are using the `cache` tag (not `cacheXMLObj` or `useCacheObj`), it might be helpful to break your cache blocks into separate JSP pages so that you would be less likely to fall into this type of situation. In this case, each cache block would be represented by its own URI and you could use dynamic-include functionality to combine the pages together as desired.

To make this more convenient, Oracle also provides the `cacheInclude` tag, described in the following section, "Web Object Cache cacheInclude Tag".

Web Object Cache cacheInclude Tag

The `cacheInclude` tag combines functionality of the `cache` tag (but not the `cacheXMLObj` tag or `useCacheObj` tag) and the standard `jsp:include` tag.

There are a number of advantages in putting cache blocks into separate pages and using `cacheInclude`, including general considerations of modularity and clarity as well as the issues discussed in the preceding section, "Conditional Execution of Code Inside the Cache Tags".

Be aware of the following limitations, however:

- You cannot use a runtime JSP expression in the `cacheInclude` tag.

- You must use implicit cache-block naming for the cache block.

- There is no `flush` parameter, unlike for the `jsp:include` tag.

If any of these limitations presents a problem, then use separate `cache` and `jsp:include` tags.

Also be aware of an important difference between the `cacheInclude` tag and the JESI `include` tag. (See "JESI include Tag" on page 6-15 for information about that tag.) Because the OracleAS Web Cache is in a different caching layer than the Web Object Cache, the including page and included page for a JESI `include` tag cannot share the same request object. There is no such limitation with the `cacheInclude` tag, however. The including page and included page share the same request object, so beans and attributes of `request` scope can be passed between the two pages.

Syntax

```
<ojsp:cacheInclude
     policy = "filename"
     page = "URI"
   [ printCacheBlockInfo = "true" | "false" ]
   [ reportException = "true" | "false" ] >

...Code for cache block...

</ojsp:cacheInclude>
```

> **Note:** For the `cacheInclude` tag, because `policy` and `page` are not request-time attributes, you do not have the option of determining their values through JSP expressions. (Be aware that `policy` *is* a request-time attribute for the `cache`, `cacheXMLObj`, and `useCacheObj` tags.)

Attributes

- `policy` (required): You must use a cache policy descriptor file to specify cache policy settings; individual parameter settings are not supported.

- `page` (required): Use the `page` attribute to specify the URI of the page to dynamically include, as with a standard `jsp:include` tag.

- `printCacheBlockInfo` (for debugging): See "Web Object Cache cache Tag" on page 7-15.

- `reportException`: See "Cache Policy Attributes" on page 7-8.

Attribute Usage Notes

Consider the following `cacheInclude` tag usage:

```
<ojsp:cacheInclude page="anotherPage.jsp" policy="foo.cpd" >
```

This is equivalent to the following:

```
<ojsp:cache policy="foo.cpd" >
   <% pageContext.include("anotherPage.jsp"); %>
</ojsp:cache>
```

It is also equivalent to the following:

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

Assume `anotherPage.jsp` consists of the following:

```
<ojsp:cache policy="foo.cpd" >
...anotherPage.jsp contents...
</ojsp:cache>
```

Cache Invalidation Tag Description

This section describes how to use the `invalidateCache` tag.

Web Object Cache invalidateCache Tag

To explicitly invalidate a cache block through program logic, you can use the `invalidateCache` tag. This section documents the syntax and attributes of this tag.

> **Notes:**
>
> - The `invalidateCache` tag does not accept new cookies; it can use only existing cookies of the current HTTP request. For information about inputting new cookies, see "CachePolicy Methods" on page 7-28.
>
> - Do not confuse the `invalidateCache` tag with the `invalidateCache` attribute of the cache tags. The attribute is of more limited use, to invalidate the pre-existing cache object.

Syntax

```
<ojsp:invalidateCache
   [ policy = "filename" ]
   [ ignoreCache = "true" | "false" ]
   [ scope = "application" | "session" ]
   [ autoType = "user" | "URI" | "URI_query" | "URI_allParam" |
                "URI_selectedParam" | "URI_excludedParam" ]
```

```
[ selectedParam = "space-delimited_string_of_parameter_names" ]
[ selectedCookies = "space-delimited_string_of_cookie_names" ]
[ name = "blockname" ]
[ invalidateNameLike = "true" | "false" ]
[ page = "URI" ]
[ autoInvalidateLevel = "application" | "page" | "param" | "cookie" ]
[ cacheRepositoryName = "name" ]
[ reportException = "true" | "false" ] />
```

> **Note:** The default value of `autoInvalidateLevel` depends on specifics of the page URI. See "Use of page and autoInvalidateLevel" on page 7-25.

Attributes

Most parameters of the `invalidateCache` tag also exist in the `cache` and `cacheXMLObj` tags and are used in the same way, as described earlier in this chapter (and as referenced below).

- `policy`: See "Web Object Cache cache Tag" on page 7-15.

- `ignoreCache`: See "Cache Policy Attributes" on page 7-8.

- `scope`: See "Cache Policy Attributes" on page 7-8.

- `autoType`: See "Cache Policy Attributes" on page 7-8. The correspondence between tag attribute settings and class attribute values (integer constants) is as follows:

  - The `user` setting is equivalent to `TYPE_USERSPECIFIED`.

  - `URI` is equivalent to `TYPE_URI_ONLY`.

  - `URI_query` is equivalent to `TYPE_URI_QUERYSTR`.

  - `URI_allParam` is equivalent to `TYPE_URI_ALLPARAM`.

  - `URI_selectedParam` is equivalent to `TYPE_URI_SELECTEDPARAM`.

  - `URI_excludedParam` is equivalent to `TYPE_URI_EXCLUDEDPARAM`.

- `selectedParam`: See "Cache Policy Attributes" on page 7-8.

- `selectedCookies`: See "Cache Policy Attributes" on page 7-8.

- `name`: Use this with `invalidateNameLike` to invalidate one or more cache blocks that were named through explicit cache-block naming, according to the instructions in "Use of name and invalidateNameLike" below.

- `invalidateNameLike`: Use this with `name` to invalidate one or more cache blocks that were named through explicit cache-block naming, according to the instructions in "Use of name and invalidateNameLike" below. The default setting is "false".

- `page`: Specify a page-relative or application-relative URI. Use this with `autoInvalidateLevel` to invalidate one or more cache blocks that were named through implicit cache-block naming, according to the instructions in "Use of page and autoInvalidateLevel" below.

- `autoInvalidateLevel`: Use this with `page` to invalidate one or more cache blocks that were named through implicit cache-block naming, according to the instructions in "Use of page and autoInvalidateLevel" below.

- `cacheRepositoryName`: See "Cache Policy Attributes" on page 7-8.

- `reportException`: See "Cache Policy Attributes" on page 7-8.

Use of name and invalidateNameLike To invalidate one or more cache blocks that were named through explicit cache-block naming, use the `name` and `invalidateNameLike` attributes together, as follows:

- If `invalidateNameLike="false"`, then use the `name` parameter to specify the name of a single cache block to invalidate.

- If `invalidateNameLike="true"`, and the underlying cache repository supports wild card characters, then you can use the wildcard "*" character in the `name` parameter to invalidate multiple cache blocks whose names fit the criteria. (The Oracle Application Server Java Object Cache currently does *not* support wild card characters.)

Use of page and autoInvalidateLevel To invalidate one or more cache blocks that were named through implicit cache-block naming, use the `page` and `autoInvalidateLevel` attributes together.

Use the `page` attribute to specify the appropriate URI of the Web page. With implicit naming, cache block names are based on Web page URIs.

Use `autoInvalidateLevel` to specify the scope of invalidation—`application` scope, `page` scope, `parameter` scope, or `cookie` scope—as follows:

- If `autoInvalidateLevel="application"`, then all cache blocks associated with the application that the page belongs to will be invalidated.

  For example, if there is an application under the `/mycontext` context path, and `autoInvalidateLevel="application"`, then all cache entries of all pages under `http://host:port/mycontext` will be invalidated.

  Here is a corresponding usage example:

  ```
  <ojsp:invalidateCache page="/" autoInvalidateLevel="application" />
  ```

- If `autoInvalidateLevel="page"`, then all cache block entries associated with the page will be invalidated. Consider the following example:

  ```
  http://host:port/mycontext/mypage01.jsp?foo=bar
  ```

  For this request, if `autoInvalidate="page"`, then all cache entries of `mypage01.jsp` will be invalidated, regardless of what request parameters and cookies they are associated with. This includes cache blocks associated with the following, for example:

  ```
  http://host:port/mycontext/mypage01.jsp?p1=v1
  ```

  Here is a corresponding usage example:

  ```
  <ojsp:invalidateCache page="/mypage01.jsp" autoInvalidateLevel="page" />
  ```

- If `autoInvalidateLevel="param"`, then all cache entries of the page that have the identical selected parameter names and values will be invalidated, regardless of what cookies they are associated with.

  For example, consider the following:

  ```
  <ojsp:invalidateCache policy="/WEB-INF/c1.cpd"
                        page="/mypage01.jsp?foo=bar"
                        autoInvalidateLevel="param" />
  ```

In this case, cache blocks associated with the following, for example, will *not* be invalidated:

```
http://host:port/mycontext/mypage01.jsp?foo=bar2
```

However, cache blocks associated with the following *will* be invalidated, regardless of what cookies they are associated with:

```
http://host:port/mycontext/mypage01.jsp?foo=bar
```

Continuing this example, consider the following:

```
http://host:port/mycontext/mypage01.jsp?foo=bar&p1=v1
```

Cache blocks associated with this request will be invalidated if `c1.cpd` selects the `foo` HTTP request parameter only, and the cache blocks are stored under the same cache policy, `c1.cpd`. However, the cache objects will *not* be invalidated if they were not stored under `c1.cpd`, or if `c1.cpd` also selects the `p1` parameter.

- If `autoInvalidateLevel="cookie"`, then the only cache entries invalidated are those associated with the same page, same selected parameters and values, and same cookies.

> **Note:** If the page URI includes a question mark, then the default `autoInvalidateLevel` is `param`. If there is no question mark, then the default is `page`.

Example: Use of Cache Invalidation Tag

This section provides a brief example of cache invalidation.

Example: invalidateCache Tag

The following page adds an item to a list of items previously cached, then invalidates the cache. The list will presumably be re-cached later with the new item.

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jwcache.tld"
          prefix="ojsp" %>
<title>added.jsp</title>
<jsp:useBean class="java.util.Hashtable" id="table" scope="application" />
<%
   String itemid=request.getParameter("itemid");
   String addItem=request.getParameter("addItem");
   Vector list=(Vector) table.get(itemid);
   if (list==null) {
     list=new Vector();
     table.put(itemid,list);
   }
   list.addElement(addItem);
%>
<b><%= addItem %></b> was added into category <b><%= itemid %></b>.<br>
<% String viewPage="listitem.jsp?itemid="+itemid; %>
<% long l1=(new java.util.Date()).getTime(); %>
<ojsp:invalidateCache page="<%= viewPage %>" autoInvalidateLevel="param"
  policy="/WEB-INF/test-policy.cpd"
    />
<% long l2=(new java.util.Date()).getTime(); %>
Existing cache entry has been invalidated. <br>
Invalidation took <%= l2-l1 %> milliseconds.
```

```
<br>
<jsp:include page="<%= viewPage %>" flush="true" />
<br>
<a href="seeitems.jsp" >Select items</a>
or
<a href="additem.html" >Add items</a>
<br>
```

Web Object Cache API Descriptions

From servlets, you can use `CachePolicy` methods to modify cache policy settings or to invalidate a cache block, and `ExpirationPolicy` methods to modify expiration settings. This requires creating a cache policy object and retrieving its expiration policy object attribute, which the JSP cache tag handlers do automatically.

The following sections describe the API:

- Cache Policy Object Creation
- CachePolicy Methods
- Expiration Policy Object Retrieval
- ExpirationPolicy Methods
- CacheBlock Methods
- Tag Code Versus API Code

The Web Object Cache classes discussed here are in the `oracle.jsp.jwcache` package and are supplied in the file `ojsputil.jar`, which comes with OC4J. Verify that this file is installed and in your classpath. Also, to use the Oracle Application Server Java Object Cache as the back-end repository, the file `cache.jar` must be installed and in your classpath. This file also comes with OC4J.

For more information about the classes, interfaces, and methods described in this section, see the Javadoc that is supplied with OC4J.

Cache Policy Object Creation

There are two approaches to creating a `CachePolicy` object:

- Use the static `lookupPolicy()` method of the `CacheClientUtil` class.
- Use one of the public `CachePolicy` constructors.

> **Note:** Cache policy objects are not resource objects, such as database connections or cursors, so you can manipulate them without life-cycle or resource management concerns.

Using the lookupPolicy() Method

In most situations, the most convenient way to create a `CachePolicy` object is through the static `lookupPolicy()` method of the `CacheClientUtil` class, as in the following example:

```
CachePolicy cachePolicyObject = oracle.jsp.jwcache.CacheClientUtil.lookupPolicy
                                (servletConfig, request, "/WEB-INF/foo.cpd");
```

Input a servlet configuration object (a `javax.servlet.ServletConfig` instance), a request object (a `javax.servlet.http.HttpServletRequest` instance), and the URI path, relative to the application root, of an XML cache policy descriptor file.

Here is a simple example of a cache policy descriptor file:

```
<!--
test-policy.cpd
-->

<cachePolicy scope="application">
   <expirationPolicy expirationType="TTL" TTL="25" timeInaDay="00:10:00"
                     writeThrough="true" />
</cachePolicy>
```

See "Cache Policy Descriptor" on page 7-38 for more information.

Using a CachePolicy Constructor

The `CachePolicy` class has three public constructors: a simple constructor requiring only a servlet configuration object, a "copy" constructor that copies another `CachePolicy` object, and a "copy" constructor with a given servlet configuration object, as follows:

```
public CachePolicy(javax.servlet.ServletConfig config)

public CachePolicy(CachePolicy cPolicy)

public CachePolicy(javax.servlet.ServletConfig config,
                   CachePolicy cPolicy)
```

CachePolicy Methods

Several utility methods are available in `CachePolicy` objects, as well as getter and setter methods for key attributes.

CachePolicy Method Signatures and Common Parameters

The following abbreviated code, for illustration purposes only, contains signatures for key methods available in `CachePolicy` objects.

See "Cache Policy Attributes" on page 7-8 for a discussion of relevant attributes.

```
class CachePolicy
{
   boolean isRecent(CacheBlock block);
   void putCache(Object data, HttpServletRequest req, SectionId sectionId);
   void putCache(Object data, HttpServletRequest req, String specifiedName);
   void putAutoCacheForOtherPath(Object data, HttpServletRequest req,
                      String otherPath, StringSectionid sectionId);
   void putAutoCacheForOtherPath(Object data, HttpServletRequest req,
        String otherPath, Cookie[] newCookies, StringSectionid sectionId);
   CacheBlock getCache(HttpServletRequest req, SectionId sectionId);
   CacheBlock getCache(HttpServletRequest req, String specifiedName);
   CacheBlock getAutoCacheForOtherPath(HttpServletRequest req,
                         String otherPath, StringSectionId sectionId);
   CacheBlock getAutoCacheForOtherPath(HttpServletRequest req,
        String otherPath, Cookie[] newCookies, StringSectionId sectionId);
   void invalidateCache(HttpServletRequest req, SectionId sectionId);
   void invalidateCache(HttpServletRequest req, String specifiedName);
   void invalidateCacheLike(HttpServletRequest req, String specifiedName);
```

```
      void invalidateCacheLike(HttpServletRequest req, int autoInvalidateLevel);
      void invalidateCacheLike(HttpServletRequest req, String specifiedName,
                               int autoInvalidateLevel);
      void invalidateCacheOtherPathLike(HttpServletRequest req, String otherPath);
      void invalidateCacheOtherPathLike(HttpServletRequest req, String otherPath,
                               Cookie[] newCookies, int autoInvalidateLevel);
      Date getCurrentTime();
}
```

These methods use several common parameters:

- `req`, a `javax.servlet.http.HttpServletRequest` instance

  This is the current HTTP request object.

- `newCookies`, a `javax.servlet.http.Cookie[]` array

  This is an array of new cookies. If you pass in new cookies, they are used in cache operations that use the `otherPath` parameter (such as the `putAutoCacheForOtherPath()` method), assuming the cache policy selects some cookies and invalidation is at the cookie level. If you do not pass in new cookies, then cookies of the current HTTP request are used instead.

- `specifiedName`, a Java string

  For explicit cache-block naming, this is the name—either the desired cache block name if you are creating a new cache block, or the existing cache block name if you are retrieving an existing cache block.

- `sectionId`, an `oracle.jsp.jwcache.SectionId` instance, specifically a `StringSectionId` or `NumberSectionId` instance

  For implicit cache-block naming, this is a counter that is used in tracking cache blocks. In JSP pages, it is used, incremented, and maintained by JSP cache tag handlers. It is stored in the JSP `pageContext` object.

  `SectionId` is an interface that is implemented by two classes: `StringSectionId` and `NumberSectionId`. Where `StringSectionId` is specified in a method signature, you must use an instance of that class. Where `SectionId` is specified, you can use an instance of either class, but should typically use `StringSectionId`. The `NumberSectionId` class is primarily intended for use by JSP tag handlers.

  In a servlet, you must create a section ID instance manually. "Servlet Page: DemoCacheServlet.java" on page 7-36 demonstrates the use of a `StringSectionId` instance.

  > **Note:** When you construct a `StringSectionId` instance, the string must begin with an alphabetic (not numeric) character.

- `otherPath`, a Java string

  This is the URI of another JSP page that has an associated cache block that you want to store, retrieve, or invalidate.

- `autoInvalidateLevel`, an integer

  For implicit cache-block naming, you can use this to specify a level of invalidation, either `application`, `page`, `parameter`, or `cookie`. Use the `CachePolicy` integer constant `AUTO_INVALIDATE_APP_LEVEL`, `AUTO_INVALIDATE_PAGE_`

LEVEL, AUTO_INVALIDATE_PARAM_LEVEL, or AUTO_INVALIDATE_COOKIE_
LEVEL.

CachePolicy Method Descriptions

The CachePolicy methods function as follows:

- isRecent()

  This method checks the timestamp of the specified cache block and determines whether it is recent enough, given the current time and the values of the cache policy reusableTimeStamp and reusableDeltaTime attributes.

- putCache(...)

  Use this method to place an object into the cache repository. The data parameter is any serializable Java object you want to cache that will not require any further modification or mutation. In JSP pages, the JSP cache tag handler calls putCache() to cache a BodyContent instance. The cacheXMLObj tag handler calls it to cache an XML DOM object. In a servlet or useCacheObj tag, the cache target object can be any Java serializable object.

  You must also provide an HTTP request object, along with a cache block name (for explicit naming) or a section ID (for implicit naming).

  > **Note:** The putCache() method does nothing if the cache policy ignoreCache attribute is set to "true".

- putAutoCacheForOtherPath(...)

  Place the specified object into the cache repository according to a specified string-based section ID and a specified page path, optionally using specified cookies as well. You must also input an HttpServletRequest object. The cache policy must *not* use explicit naming (in other words, must not have autoType=TYPE_USERSPECIFIED).

- getCache(...)

  Use this method to retrieve a cached item from the repository, in the form of an CacheBlock instance. You can specify the cache block name (for explicit naming) or the section ID (for implicit naming). You must also provide an HTTP request object.

  > **Note:** The getCache() method does nothing if the cache policy ignoreCache attribute is true.

- getAutoCacheForOtherPath(...)

  Retrieve a cached item from the repository according to a specified string-based section ID and a specified page path, optionally using specified cookies as well. You must also input an HttpServletRequest object. The cache policy must *not* use explicit naming, otherwise an exception is thrown. (In other words, you cannot have autoType=TYPE_USERSPECIFIED.)

- invalidateCache(...)

  Use this method to invalidate a single cache block. Invalidation is according to the HTTP request object and also according to the specified cache block name (for explicit naming) or section ID (for implicit naming).

- `invalidateCacheLike(...)`

  Use this method to invalidate multiple cache blocks. If you use explicit cache-block naming and the cache repository supports wild-card naming, you can input the `specifiedName` parameter with "*" wild card characters. The Oracle Application Server Java Object Cache currently does *not* support wild card characters.

  If you use implicit cache-block naming, you must specify the `autoInvalidateLevel` parameter to determine, in combination with the `HttpServletRequest` object and optionally the `specifiedName` parameter, what cache blocks are invalidated. The `autoInvalidateLevel` parameter has the same functionality as in a JSP `invalidateCache` tag, as explained in "Web Object Cache invalidateCache Tag" on page 7-23 (using information from the request object, instead of using information from the `page` parameter of the `invalidateCache` tag).

- `invalidateCacheOtherPathLike(...)`

  Use this method to invalidate cache blocks associated with the URI you provide in the `otherPath` parameter. In the signature taking only a request object and the URI, the `autoInvalidateLevel` parameter is set automatically according to the URI. It is set to `param` level if there is a question mark ("?") in the URI or to `page` level otherwise.

  The detailed signature of this method enables you to specifically control the `autoInvalidateLevel` setting and the cookies used in invalidation.

- `getCurrentTime()`

  Retrieve the current time value, as a `java.util.Date` instance, of the underlying cache repository specified in this cache policy.

CachePolicy Getter and Setter Methods

You can use the following methods to retrieve or alter `CachePolicy` object attributes. See "Cache Policy Attributes" on page 7-8 for a discussion of these attributes.

- `boolean getIgnoreCache()`

- `void setIgnoreCache(boolean ignoreCache)`

- `void setIgnoreCache(String ignoreCacheStr)`

- `int getScope()`

- `void setScope(int scope)`

  For `scope` values, use the integer constants `SCOPE_APP` and `SCOPE_SESSION`.

- `int getAutoType()`

- `void setAutoType(int autoType)`

  For `autoType` values, use the integer constants `TYPE_USERSPECIFIED`, `TYPE_URI_ONLY`, `TYPE_URI_QUERYSTR`, `TYPE_URI_ALLPARAM`, `TYPE_URI_SELECTEDPARAM`, and `TYPE_URI_EXCLUDEDPARAM`.

- `String[] getSelectedParam()`

- `void setSelectedParam(String[] selectedParameters)`

- `void setSelectedParam(String selectedParamStr)`

- `String[] getSelectedCookies()`

- `void setSelectedCookies(String[] selectedCookies)`

- `void setSelectedCookies(String selectedCookiesStr)`

- `Date getReusableTimeStamp()`

- `void setReusableTimeStamp(Date reusableTimeStamp)`

- `void setReusableTimeStamp(long reusableTimeStamp)`

  For `reusableTimeStamp` values, the integer constant `REUSABLE_ALWAYS` indicates that the cache is always reusable.

- `long getReusableDeltaTime()`

- `void setReusableDeltaTime(long reusableDeltaTime)`

  For `reusableDeltaTime` values, the integer constant `REUSABLE_ALWAYS` indicates that the cache is always reusable.

- `ExpirationPolicy getExpirationPolicy()`

- `void setExpirationPolicy(ExpirationPolicy`
  `                         expirationPolicy)`

- `String getCacheRepositoryName()`

- `void setCacheRepositoryName(String repoName)`

- `boolean getReportException()`

- `void setReportException (boolean reportException)`

- `void setReportException (String reportExceptionStr)`

The following methods are also available, but are primarily intended for use by the Web Object Cache tag handlers:

- `void setScope(String scopeStr)`

  For `scope` values, there are the string constants `SCOPE_APP_STR` and `SCOPE_SESSION_STR`.

- `void setAutoType(String autoTypeStr)`

- `void setReusableTimeStamp(String reusableTimeStampStr)`

  For `reusableTimeStamp` values, the string constant `REUSABLE_IGNORED` indicates that the cache is always reusable.

- `void setReusableDeltaTime(String reusableDeltaTimeStr)`

  For `reusableDeltaTime` values, the string constant `REUSABLE_IGNORED` indicates that the cache is always reusable.

Expiration Policy Object Retrieval

Each `CachePolicy` object has an `ExpirationPolicy` attribute. If you want to set expiration policies for a cache block, you can use the `getExpirationPolicy()` method of its `CachePolicy` object, as in the following example:

```
CachePolicy cachePolicyObj = CacheClientUtil.lookupPolicy
                         (config, request, "/WEB-INF/mypolicy.cpd");
ExpirationPolicy expPolicyObj = cachePolicyObj.getExpirationPolicy();
```

ExpirationPolicy Methods

The `ExpirationPolicy` class has getter and setter methods for its attributes, as follows. For descriptions of these attributes, see "Expiration Policy Attributes" on page 7-12.

- `int getExpirationType()`

- `void setExpirationType(int expirationType)`

- `void setExpirationType(String expirationTypeStr)`

- `long getTTL()`

- `void setTTL(long ttl)`

- `long getTimeInaDay()`

- `void setTimeInaDay(long timeInaDay)`

- `void setTimeInaDay(String timeInaDayStr)`

- `int getDayInaWeek()`

- `void setDayInaWeek(int dayInaWeek)`

- `void setDayInaWeek(String dayInaWeekStr)`

- `int getDayInaMonth()`

- `void setDayInaMonth(int dayInaMonth)`

- `boolean getWriteThrough()`

- `void setWriteThrough(boolean writeThrough)`

- `void setWriteThrough(String writeThroughStr)`

Additionally, the `ExpirationPolicy` class has the following utility method:

- `long getExpirationTime(long createTime)`

  Given the creation time of a cache block expressed in milliseconds since midnight January 1, 1970, this method calculates and returns the expiration time, also in milliseconds since midnight January 1, 1970. That is, the timestamp when expiration should occur, according to the expiration policy.

The `ExpirationPolicy` class also defines the following integer constants for the `expirationType` attribute:

- `TYPE_TTL`

- `TYPE_DAILY`

- `TYPE_WEEKLY`

- `TYPE_MONTHLY`

And the following integer constants are defined for the `dayInaWeek` attribute:

- `WEEKLY_SUNDAY`

- `WEEKLY_MONDAY`

- `WEEKLY_TUESDAY`

- `WEEKLY_WEDNESDAY`

- `WEEKLY_THURSDAY`

- `WEEKLY_FRIDAY`

■ `WEEKLY_SATURDAY`

CacheBlock Methods

You can use the `getCache()` method of a `CachePolicy` object to retrieve the associated `CacheBlock` object, as documented in "CachePolicy Methods" on page 7-28 and shown in "Servlet Page: DemoCacheServlet.java" on page 7-36.

The following abbreviated code, for illustrative purposes only, shows the key methods of the `oracle.jsp.jwcache.CacheBlock` class:

```
class CacheBlock
{
   long getCreationTime();
   long getExpirationTime();
   Serializable getData();
}
```

Here are brief descriptions of these methods:

■ `getCreationTime()`: Returns the timestamp indicating when the cache block was created.

■ `getExpirationTime()`: Returns the timestamp indicating the expiration time of the cache block.

■ `getData()`: Returns the cache block data.

> **Note:** Creation time and expiration time are expressed in milliseconds since midnight, January 1, 1970.

Tag Code Versus API Code

This example presents code for three approaches to an application that caches and presents timestamp output from two cache fragments:

■ The first approach, `tagcode.jsp`, is a simple JSP page that uses the Oracle Web Object Cache tags.

■ The second approach, `servletcode.jsp`, is a more involved JSP page that uses the Web Object Cache API inside a Java scriptlet instead of using the Web Object Cache tags.

■ The third approach, `DemoCacheServlet.java`, uses the Web Object Cache API inside a servlet.

Following the three code samples is a listing of the cache policy descriptor, `test-policy.cpd`.

In each approach, the application will cache the two fragments it displays. You can reload repeatedly, but the times displayed in the fragments will not change until the cached fragments expire. The first fragment takes 25 seconds to expire, getting the 25-second time-to-live value from the `TTL` setting in the cache policy descriptor (`test-policy.cpd`). The second fragment takes 15 seconds to expire, overriding the cache policy descriptor time-to-live value with a value set directly in the page code.

Output for the sample applications looks something like the following:

```
fragment#1  (expires in 25 seconds based on TTL value test-policy)
Sun May 27 15:20:46 PDT 2001
```

```
fragment#2 (expires in 15 seconds because TTL overrides test-policy value)
Sun May 27 15:20:46 PDT 2001
```

Simple JSP Page: tagcode.jsp

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jwcache.tld"
           prefix="ojsp" %>
<title>tagcode.jsp</title>
<pre>
tagcode.jsp
<ojsp:cache policy="/WEB-INF/test-policy.cpd" >
  fragment#1  (expires in 25 seconds based on TTL value test-policy)
  <%= new java.util.Date() %>
</ojsp:cache>
<ojsp:cache policy="/WEB-INF/test-policy.cpd" TTL="15" >
  fragment#2 (expires in 15 seconds because TTL overrides test-policy value)
  <%= new java.util.Date() %>
</ojsp:cache>
</pre>
```

Scriptlet JSP Page: servletcode.jsp

Code notes are the same as for the servlet version in the next section, "Servlet Page: DemoCacheServlet.java".

```
<%@ page import="oracle.jsp.jwcache.*,java.io.*" %>
<title>servletcode.jsp</title>
<pre>
servletcode.jsp
<%
  CachePolicy cachePolicyObj = CacheClientUtil.lookupPolicy(config,request,
                            "/WEB-INF/test-policy.cpd" );  // Note A
  StringSectionId sectionId=new StringSectionId("s1");   // Note B
  CacheBlock cacheBlockObj=null;

  cacheBlockObj = cachePolicyObj.getCache(request,sectionId); // Note C
  if (!cachePolicyObj.isRecent(cacheBlockObj)) {  // Note D
    CharArrayWriter newOut=new CharArrayWriter();
    PrintWriter pw=new PrintWriter(newOut);

    // actual logic within a cache block
    pw.println
          ("fragment#1 (expires in 25 seconds based on TTL value test-policy)");
    pw.println(new java.util.Date());
    // which generates content into the "out" object

    if (cacheBlockObj == null) {    // Note E
      cachePolicyObj.putCache(newOut.toCharArray(),request,sectionId);
      // Note F
    }

    out.write(newOut.toCharArray());
    // writing out newly created data back to the original writer
  }
  else {
    out.write((char[])cacheBlockObj.getData());
    // writing the existing cached data to the writer
  }

  sectionId=new StringSectionId("s2");
  long timeToLive = 15;    // now set TTL to 15 on this block
```

```
                  ExpirationPolicy expirationPolicy = cachePolicyObj.getExpirationPolicy();
                  expirationPolicy.setTTL(timeToLive);
                  cachePolicyObj.setExpirationPolicy(expirationPolicy);
                  cacheBlockObj = cachePolicyObj.getCache(request,sectionId);
                  if (!cachePolicyObj.isRecent(cacheBlockObj)) {
                    CharArrayWriter newOut=new CharArrayWriter();
                    PrintWriter pw=new PrintWriter(newOut);

                    // actual logic within a cache block
                    pw.println
              ("fragment#2 (expires in 15 seconds because TTL overrides test-policy value)");
                    pw.println(new java.util.Date());
                    // which generates content into the "out" object

                    if (cacheBlockObj == null) {
                      cachePolicyObj.putCache(newOut.toCharArray(),request,sectionId);
                    }

                    out.write(newOut.toCharArray());
                    // writing out newly created data back to the original writer
                  }
                  else {
                    out.write((char[])cacheBlockObj.getData());
                    // writing the existing cached data to the writer
                  }

              %>
              </pre>
```

Servlet Page: DemoCacheServlet.java

Code notes are explained at the end of the code.

```
package demoPkg;

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

import java.io.PrintWriter;
import java.io.CharArrayWriter;

import oracle.jsp.jwcache.CachePolicy;
import oracle.jsp.jwcache.ExpirationPolicy;
import oracle.jsp.jwcache.StringSectionId;
import oracle.jsp.jwcache.CacheBlock;
import oracle.jsp.jwcache.CacheClientUtil;

public class DemoCacheServlet extends HttpServlet{

  public void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
  {
    // standard writer object from servlet engine
    PrintWriter out=response.getWriter();
    ServletConfig config=getServletConfig();

    try {
      CachePolicy cachePolicyObj = CacheClientUtil.lookupPolicy(config,request,
        "/WEB-INF/test-policy.cpd" );  // Note A
      StringSectionId sectionId=new StringSectionId("s1");   // Note B
```

```
                    CacheBlock cacheBlockObj=null;

                    cacheBlockObj = cachePolicyObj.getCache(request,sectionId); // Note C
                    if (!cachePolicyObj.isRecent(cacheBlockObj)) {  // Note D
                      CharArrayWriter newOut=new CharArrayWriter();
                      PrintWriter pw=new PrintWriter(newOut);

                      // actual logic within a cache block
                      pw.println("fragment#1");
                      pw.println(new java.util.Date());
                      // which generates content into the "out" object

                      if (cacheBlockObj == null) {    // Note E
                        cachePolicyObj.putCache(newOut.toCharArray(),request,sectionId);
                        // Note F
                      }

                      out.write(newOut.toCharArray());
                      // writing out newly created data back to the original writer
                    }
                    else {
                      out.write((char[])cacheBlockObj.getData());
                      // writing the existing cached data to the writer
                    }

                    sectionId=new StringSectionId("s2");
                    long timeToLive = 15;    // now set TTL to 15 on this block
                    ExpirationPolicy expirationPolicy = cachePolicyObj.getExpirationPolicy();
                    expirationPolicy.setTTL(timeToLive);
                    cachePolicyObj.setExpirationPolicy(expirationPolicy);
                    cacheBlockObj = cachePolicyObj.getCache(request,sectionId);
                    if (!cachePolicyObj.isRecent(cacheBlockObj)) {
                      CharArrayWriter newOut=new CharArrayWriter();
                      PrintWriter pw=new PrintWriter(newOut);

                      // actual logic within a cache block
                      pw.println("fragment#2");
                      pw.println(new java.util.Date());
                      // which generates content into the "out" object

                      if (cacheBlockObj == null) {
                        cachePolicyObj.putCache(newOut.toCharArray(),request,sectionId);
                      }

                      out.write(newOut.toCharArray());
                      // writing out newly created data back to the original writer
                    }
                    else {
                      out.write((char[])cacheBlockObj.getData());
                      // writing the existing cached data to the writer
                    }

                } catch (Throwable th) {
                  // your exception handling code here
                  th.printStackTrace(out);
                }
              }
            }
```

Code Notes The following notes describe some of the key functionality of the preceding example:

- The cache policy object is created in the `lookupPolicy()` call (Note A), with attribute settings according to the cache policy descriptor `test-policy.cpd`.

- The section ID is created for each cache block (Note B), as required for implicit cache-block naming. See "CachePolicy Methods" on page 7-28 for information about section IDs.

- The cache block is retrieved from the repository through the `getCache()` method of the cache policy object (Note C) and placed into the repository through the `putCache()` method, according to the section ID in each case.

- The `isRecent()` call determines if the cache block is recent enough to use (Note D). If so, the cached data is retrieved through the `getData()` method of the cache block. (See "CacheBlock Methods" on page 7-34.) If not, a special `PrintWriter` object is created to buffer the output and save it back to the cache repository. If the cache block object is not found (is null, Note E), then the `putCache()` method of the cache policy object is called to create a new cache block (Note F).

Cache Policy Descriptor: test-policy.cpd

This cache policy descriptor is used by all three approaches to the sample application: `tagcode.jsp`, `servletcode.jsp`, and `DemoCacheServlet.java`:

```
<!--
test-policy.cpd
-->

<cachePolicy scope="application">
   <expirationPolicy expirationType="TTL" TTL="25" timeInaDay="00:10:00"
                     writeThrough="true" />
</cachePolicy>
```

Cache Policy Descriptor

You can optionally use an XML-style cache policy descriptor to specify attribute settings for the `CachePolicy` and `ExpirationPolicy` objects. In any JSP pages or servlets that you use, you would then specify the cache policy descriptor through the `policy` attribute of a `cache`, `cacheXMLObj`, `useCacheObj`, `cacheInclude`, or `invalidateCache` tag.

The following sections provide the cache policy descriptor DTD, a sample cache policy descriptor, and information about loading and refreshing the cache policy descriptor:

- Cache Policy Descriptor DTD
- Sample Cache Policy Descriptor
- Cache Policy Descriptor Loading and Refreshing

Cache Policy Descriptor DTD

This section provides a listing of the Web Object Cache cache policy descriptor DTD, `cachepolicy.dtd`.

```
<!--
cachepolicy.dtd
-->
<!--
```

```
This DTD is used to validate any (Oracle programmable web)
cache policy descriptors (for example, "/WEB-INF/foo.cpd").
-->


<!--
The cachePolicy element is the root element of cache policy descriptors.
configuration descriptor.
-->


<!ELEMENT cachePolicy (
   selectedParam*, selectedCookie*,
   reusableTimeStamp?, reusableDeltaTime?,
   cacheRepositoryName?, expirationPolicy? ) >

<!ATTLIST cachePolicy ignoreCache (true | false) "false" >
<!ATTLIST cachePolicy scope (application | session) "application" >
<!ATTLIST cachePolicy autoType
   (user | URI | URI_query |
    URI_allParam | URI_selectedParam | URI_excludedParam )
    "URI_allParam" >
<!ATTLIST cachePolicy reportException (true | false) "true" >


<!ELEMENT selectedParam (#PCDATA) >
<!ELEMENT selectedCookie (#PCDATA) >
<!ELEMENT reusableTimeStamp (#PCDATA) >
<!ELEMENT reusableDeltaTime (#PCDATA) >
<!ELEMENT cacheRepositoryName (#PCDATA) >


<!ELEMENT expirationPolicy EMPTY >


<!ATTLIST expirationPolicy expirationType (TTL | daily | weekly | monthly)
         "TTL" >
<!ATTLIST expirationPolicy TTL CDATA "300" >
<!ATTLIST expirationPolicy timeInaDay CDATA #IMPLIED >
<!ATTLIST expirationPolicy dayInaWeek
   (Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday)
   "Wednesday" >
<!ATTLIST expirationPolicy dayInaMonth CDATA "10" >
<!ATTLIST expirationPolicy writeThrough (true | false) "true" >
```

Sample Cache Policy Descriptor

This section provides an example of a simple cache policy descriptor that sets the TTL and timeInaDay attributes.

```
<!--
test-policy.cpd
-->


<cachePolicy scope="application">
   <expirationPolicy expirationType="TTL" TTL="25" timeInaDay="00:10:00"
                   writeThrough="true" />
</cachePolicy>
```

Cache Policy Descriptor Loading and Refreshing

To create a CachePolicy object from an XML cache policy descriptor file, there must be a call to the static lookupPolicy() method of the CacheClientUtil class. For JSP pages, this is handled automatically. For servlets, you must include the

lookupPolicy() call in your code. See "Servlet Page: DemoCacheServlet.java" on page 7-36.

If the caching policy has not been previously loaded, the lookupPolicy() invocation results in the XML descriptor being parsed and used in constructing a new CachePolicy object and an ExpirationPolicy attribute of this object. See "Cache Policy Object Creation" on page 7-27 for information about the lookupPolicy() method.

The CachePolicy object is stored indirectly under the ServletContext object associated with your application. When the same caching policy is requested again, the stored policy object will be returned without the descriptor being reread or re-parsed. For performance reasons, because the cache policy descriptor files are seldom changed, as well as for security reasons, OC4J does not provide descriptor auto-reloading functionality. The resulting cache policy object is stored in the middle-tier JVM for faster access.

The CachePolicy object will be valid until the servlet context is destroyed or someone calls the static refreshPolicy() method of the CacheClientUtil class. This method has the same calling sequence as the lookupPolicy() method. For example:

```
oracle.jsp.jwcache.CacheClientUtil.refreshPolicy
                    (servletConfig, request, "/WEB-INF/foo.cpd");
```

When you alter and refresh the caching policy, active cache blocks are not affected.

Cache Repository Descriptor

Use an XML-style cache repository descriptor to specify what to use as the back-end cache repository for the Web Object Cache and how to configure it. The following sections list the DTD for cache repository descriptors, as well as a sample cache repository descriptor:

- Cache Repository Descriptor DTD
- Sample Cache Repository Descriptor

> **Note:** By default, the Web Object Cache uses the Oracle Application Server Java Object Cache as its cache repository.

Cache Repository Descriptor DTD

This section provides a listing of the Web Object Cache cache repository descriptor DTD, wcache.dtd.

```
<!--
Copyright 2000 Oracle Corporation
wcache.dtd
-->
<!--
This DTD is used to validate "/WEB-INF/wcache.xml", which is used to hold
web cache repositories configuration information for
Oracle programmable web caching components.
-->

<!--
The wcache-config element is the root element of web cache repositories
configuration descriptor.
```

```
-->

<!ELEMENT wcache-config (cache-repository*)>

<!ELEMENT cache-repository
(cache-repository-name,cache-repository-class,init-param*)>

<!ELEMENT cache-repository-name (#PCDATA)>
<!ELEMENT cache-repository-class (#PCDATA)>

<!ELEMENT init-param (param-name,param-value)>
<!ELEMENT param-name (#PCDATA)>
<!ELEMENT param-value (#PCDATA)>
```

Sample Cache Repository Descriptor

This section lists the cache repository descriptor provided with OC4J.

> **Note:** The DTD does not include reporoot, which is a
> specific-use parameter that only a file system cache implementation
> requires.

```
<wcache-config>

<cache-repository>
   <cache-repository-name>DefaultCacheRepository</cache-repository-name>
   <cache-repository-class>
      oracle.jsp.jwcache.repository.impl.OCSRepoImpl
   </cache-repository-class>
</cache-repository>

<cache-repository>
   <cache-repository-name>SimpleFSRepo</cache-repository-name>
   <cache-repository-class>
      oracle.jsp.jwcache.repository.impl.SimpleFSRepositoryImpl
   </cache-repository-class>
   <init-param>
      <param-name>reporoot</param-name>
      <param-value>/tmp/reporoot</param-value>
   </init-param>
</cache-repository>

</wcache-config>
```

Configuration for Back-End Repository

This section describes how to configure the Oracle Application Server Java Object
Cache or a file system as the back-end repository for the OC4J Web Object Cache.

Configuration Notes for Oracle Application Server Java Object Cache

The OC4J server.xml file must have a <javacache-config> element to specify
the Java Object Cache configuration file. This is a subelement of the
<application-server> element. By default, the entry is as follows:

```
<application-server ... >
...
```

```
    <javacache-config path="../../../javacache/admin/javacache.xml" />
...
</application-server>
```

As shown, and assuming the default configuration file directory location (where `server.xml` is located), the default is for OC4J instances to share the same Java Object Cache configuration file, `javacache.xml`, in the *ORACLE_HOME*/`javacache/admin` directory.

Here is a sample Java Object Cache configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration
 xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
   <logging>
     <location>javacache.log</location>
     <level>ERROR</level>
   </logging>
   <communication>
     <isDistributed>true</isDistributed>
     <coordinator discovery-port="7000"/>
   </communication>
   <persistence>
     <location>diskcache</location>
     <disksize>32</disksize>
   </persistence>
   <max-objects>1000</max-objects>
   <max-size>48</max-size>
   <clean-interval>30</clean-interval>
</cache-configuration>
```

For more information about the Java Object Cache, its configuration, and the `javacache.xml` file, see the *Oracle Application Server Containers for J2EE Services Guide*. For more information about the `server.xml` file, refer to the *Oracle Application Server Containers for J2EE User's Guide*.

Configuration Notes for File System Cache

To use a file system as the back-end repository, edit the cache repository descriptor, `wcache.xml`, to set `reporoot` to specify a root directory for the file system cache. This file is located in the `/WEB-INF` directory where the OC4J samples are installed. See "Cache Repository Descriptor" on page 7-40 for general information and for an example of a cache repository descriptor that sets a `reporoot` value.

For example, for a UNIX system:

```
<init-param>
   <param-name>reporoot</param-name>
   <param-value>/mydir/repositoryroot</param-value>
</init-param>
```

Alternatively, for a Windows system:

```
<init-param>
   <param-name>reporoot</param-name>
   <param-value>c:\mydir\repositoryroot</param-value>
</init-param>
```

8

File Access and Mail Beans and Tags

This chapter covers OC4J tags and JavaBeans for file access (uploading and downloading) and for e-mail. The e-mail tag and JavaBean make use of the file access functionality for attachments. The chapter is organized as follows:

- File-Access JavaBeans and Tags
- Mail JavaBean and Tag

File-Access JavaBeans and Tags

OC4J provides a standards-compliant tag library and JavaBeans that add convenient file upload and file download functionality for JSP pages and servlets. Files can be uploaded to or downloaded from a file system or database.

The following sections document the file-access tags and beans:

- Overview of OC4J File-Access Functionality
- File Upload and Download Tag Descriptions
- File Upload and Download JavaBean and Class Descriptions

Overview of OC4J File-Access Functionality

Developers have the option of using either custom tags or JavaBeans to program applications that allow users to upload or download files. In either case, the application is presumably programmed so that users specify through the browser where files come from on the client system for uploading or where they go to on the client system for downloading. For JSP pages for uploading, OC4J supplies a convenience tag, `httpUploadForm`, to create a form to use in specifying where the files come from.

For processing an upload, including specifying the destination file system or database location, use the `HttpUploadBean` JavaBean or the `httpUpload` tag. For processing a download, including specifying the source file system or database location, use `HttpDownloadBean` or the `httpDownload` tag. The beans extend `HttpFileAccessBean`, which is not intended for public use. All of the beans are in the `oracle.jsp.webutil.fileaccess` package.

Overview of File Uploading

For user specification in a JSP page of where uploaded files will come from, you can use the `httpUploadForm` tag to create a form. This tag lets users select the files for uploading and creates the necessary multipart HTTP request. You also have the option of using a standard HTML form to create the request.

Use the `HttpUploadBean` JavaBean or the `httpUpload` tag to receive and process the multipart form-encoded data stream and write the files to the appropriate location, either in the file system or a database. There is functionality to let you decide whether previous data will be overwritten if the target file or database row already exists.

> **Note:** The maximum file size for any upload is 2 GB.

File System Destination If the destination is in a file system, you must provide a properties file that designates a base directory. The properties file must be named `fileaccess.properties`, must be located in the `/WEB-INF` directory of your application, and must have a `fileaccess.basedir` entry that indicates an absolute directory path. Here is an example:

```
fileaccess.basedir=/tmp
```

> **Note:** On a Windows system, you must still use a forward-slash, not a back-slash, for the directory path:
>
> ```
> fileaccess.basedir=c:/tmp
> ```
>
> Furthermore, the specified drive (`C:` in this case) must be the same drive on which OC4J is installed.

There should be subdirectories as appropriate under the base directory, such as a subdirectory for each authorized user. Destination subdirectories under the base directory must be specified through an attribute of the upload bean or tag. All directories and subdirectories must already exist and be writable; they cannot be created or made writable through OC4J functionality.

Database Destination If the destination is in a database, you can optionally use a default table, `fileaccess`, that you create through the supplied `fileaccess.sql` script, or you can use any other previously existing table containing the required column types. In either case, you must provide a connection to the database, as an instance of either `oracle.jsp.dbutil.ConnBean` or the standard `java.sql.Connection`. You can provide a `ConnBean` instance explicitly, or in a JSP page you also have the option of providing it implicitly as a result of nesting the `httpUpload` tag inside a `dbOpen` tag. (For information about the `ConnBean` JavaBean and `dbOpen` tag, see Chapter 4, "Data-Access JavaBeans and Tags".)

> **Note:** The `java.sql.Connection` type is currently supported for the file-access beans only, not the tags.

You are also required to specify a destination through an attribute of the upload bean or tag. The destination is simply a Java string value that will be placed in the *prefix* column of the database table. The prefix is equivalent to a file system path.

File data is written to a database as either a BLOB or a CLOB. You can specify which through an upload bean or tag attribute.

If you do not use the default `fileaccess` table, you must use attributes of the upload bean or tag to specify the database table name and the names of the columns that will contain the file data, the file prefix, and the file name. Any other table you use must adhere to the pattern of `fileaccess`, as follows:

- It must have a concatenated unique key consisting of the column that holds the file name and the column that holds the prefix.

- It must have a BLOB or CLOB column for the file data.

- Any column other than the file data column must allow null data.

> **Notes:**
>
> - When you use a `ConnBean` instance, the connection will be closed automatically at the end of the scope designated in the `jsp:useBean` tag that invokes it. There is no such functionality for a `Connection` instance.
>
> - `ConnBean` uses and requires the `JspScopeListener` interface. See "JSP Event-Handling with JspScopeListener" on page 9-1 for information about that utility.

Security Considerations for Uploading For uploading to a database, the database table does not have a column to indicate a particular authorized user for any given file. Therefore, without precaution, each user can see files that were uploaded by other users, without having to know the file prefixes. To prevent this, you can prepend an appropriate user name to each prefix.

Overview of File Downloading

Use the `HttpDownloadBean` JavaBean or the `httpDownload` tag as follows:

- To allow users to specify the file system source directory or the database prefix to match for file retrieval

  Note the following:

  – Matching the prefix for downloads from a database is case-sensitive.

  – Matching the source directory for downloads from a file system is case-sensitive in a case-sensitive operating system, such as UNIX.

  – There is currently no support for specifying file names, either partial or complete.

- To obtain and display a list of the files that are available for download

  Once presented with a list of available files, the user can download them one at a time from the list.

There is also functionality to specify whether you want *recursive* downloading, where files in subdirectories or with additional database prefix information will also be available for download. For database downloading, a prefix is equivalent to a file system path and can be used to group files into a hierarchy. As an example of recursive downloading from a database, assume you have specified `/user` as the prefix. Recursive downloading would find matches for files with any prefixes starting with "`/user`", such as "`/user/bill`" and "`/user/mary`", and also such as "`/user1`", "`/user2`", "`/user1/tom`", and "`/user2/susan`".

For downloading files from a file system, utilize the mechanism described in "Overview of File Uploading" on page 8-1. Use the `fileaccess.properties` file to specify a base directory and use attributes in the download bean or tag to specify the rest of the file path.

For downloading files from a database, as with uploading files to a database, you must provide an instance of `oracle.jsp.dbutil.ConnBean` or `java.sql.Connection`. In addition, if you are not using the default `fileaccess` table (that you can create using the supplied `fileaccess.sql` script), you must provide all the necessary information about the database table and columns. Specify this information through attributes of the download bean or tag.

The actual downloading of the files is accomplished by `DownloadServlet`, supplied with OC4J. In using the download tag, specify the path of this servlet through a tag attribute. For a file system source, hyperlinks are automatically created to the servlet so that the user can select a link for each file in order to download the file. For a database source, the servlet will fetch the selected CLOB or BLOB data that forms the file contents. (See "The Download Servlet" on page 8-12.)

Security Considerations for Downloading For downloading, consider limiting the users' ability to see what is in the source (server-side) file system or database. Without precaution, the following scenarios are possible:

- For file system downloading, a source value of "*" (perhaps specified through user input) would mean that all directories under the base directory would be available for downloading, with the names of all the files presumably being displayed for the user to choose from.

- For recursive downloading from a database, all files having a prefix beginning with the `source` string (perhaps specified through user input) would be available for downloading, with the names of all these files presumably being displayed. A source of "*" matches all prefixes.

If this is of concern, you can consider protective measures such as the following:

- Not accepting `source` values of "*" when downloading from file systems

- Not allowing recursive downloading from databases

- Automatically prepending the `source` value with a partial directory path or prefix string, such as a user name, to restrict the areas to which users have access

File Upload and Download JavaBean and Class Descriptions

This section describes attributes and methods of the file upload and download JavaBeans provided with OC4J: `HttpUploadBean` and `HttpDownloadBean`, respectively.

There is also brief discussion of `DownloadServlet`, provided with OC4J to perform the actual file downloading, and the class `FileAccessException` that is used by the file-access JavaBeans for exceptions relating to file uploads and downloads.

To comply with the JavaBean specification, the file upload and download JavaBeans provide no-argument constructors.

> **Note:** To use the file upload and download JavaBeans, verify that the file `ojsputil.jar` is installed and in your classpath. This file is provided with OC4J.

The HttpUploadBean

The `oracle.jsp.webutil.fileaccess.HttpUploadBean` JavaBean provides numerous setter methods for specifying information used for the uploading. It also includes most corresponding getter methods. Once you have set all the required and

appropriate attributes, use the `upload()` method to perform the upload. There is also a method to display the names of the files that were uploaded, typically so you can provide an informative message to the browser.

`HttpUploadBean`, as with `HttpDownloadBean`, extends `HttpFileAccessBean`, which itself is not intended for public use.

See "Overview of File Uploading" on page 8-1 for related information.

Summary of Required Attributes

The following list summarizes required attributes for `HttpUploadBean`:

- Always required: `destination`

- Also required for uploads to a database: `destinationType`, `connection`

- Also required for uploads to a database table other than the default `fileaccess` table: `table`, `prefixColumn`, `fileNameColumn`, `dataColumn`

- Also required for uploads to a database table using a CLOB column for file data: `fileType`

In addition, for an upload to a file system, you must call the `setBaseDir()` method to provide a servlet context and HTTP request object so that the bean can find the `fileaccess.properties` file that specifies the base directory.

Methods

Here are descriptions of the public methods of `HttpUploadBean`.

> **Note:** Many of the attributes and setter methods for `HttpUploadBean` are the same as for `HttpDownloadBean`.

- ```
  void upload(javax.servlet.http.HttpServletRequest req)
                                 throws FileAccessException
  ```

  Once all required and appropriate bean attributes have been set, use this method for the upload. The `req` parameter is the HTTP request instance containing the multipart form-encoded files. For a JSP page, use the implicit `request` object.

- ```
  void setBaseDir(javax.servlet.ServletContext sc,
                  javax.servlet.http.HttpServletRequest req)
                                 throws FileAccessException
  ```

  For an upload to a file system, use this method to determine what to use as a base directory. It gets this information from the `fileaccess.properties` file in your application `/WEB-INF` directory, which it finds through the servlet context input parameter. The `baseDir` setting, together with the `destination` setting, specifies the absolute path to the upload directory.

  The `req` parameter is the servlet request instance to use in requesting the base directory information. For JSP pages, use the implicit `request` object.

  This method is not relevant for database uploads.

- ```
  void setDestination(String destination)
  ```

  This method is always required.

  For an upload to a file system, `destination` and the base directory together specify the absolute path to the upload directory.

For an upload to a database, `destination` is used as the file prefix. (There is no "base directory".) The prefix is equivalent to a file system path and can be used to group files into a hierarchy. It is permissible to include separator characters such as "`.`" and "`/`" in the destination string.

> **Note:** Typically, the `destination` value will be based at least partially on user input.

- `void setDestinationType(String destinationType)`
                                        `throws FileAccessException`

- `void setDestinationType(int destinationType)`
                                        `throws FileAccessException`

Use the overloaded `setDestinationType()` method to specify whether the upload is to a file system or a database.

To upload to a database, set `destinationType` to one of the following: the string "`database`", the defined `String` constant `FileAccessUtil.DATABASE`, the `int` value 1, or the defined `int` constant `FileAccessUtil.LOCATION_TYPE_ DATABASE`.

Uploading to a file system is the default, but if you want to specify this explicitly, set `destinationType` to one of the following: the string "`filesystem`", the defined `String` constant `FileAccessUtil.FILESYSTEM`, the `int` value 0, or the defined `int` constant `FileAccessUtil.LOCATION_TYPE_FILESYSTEM`.

`FileAccessUtil` is in the `oracle.jsp.webutil.fileaccess` package.

- `String getDestinationType()`

Retrieve the destination information. Note there is a getter method for the string version only.

- `void setOverwrite(String overwrite)`
                                    `throws FileAccessException`

- `void setOverwrite(boolean overwrite)`

Use the overloaded `setOverwrite()` method to overwrite existing files or update rows with the same file name and prefix. This is relevant for both file system and database uploads.

Overwriting is enabled by default, but you can enable it explicitly with an `overwrite` setting of the string "`true`" or the boolean value `true`. Disable overwriting with a setting of the string "`false`" or the boolean value `false`. String settings are case-insensitive. No settings are accepted other than those listed here.

- `void setFileType(String fileType)`
                                    `throws FileAccessException`

- `void setFileType(int fileType) throws FileAccessException`

For an upload to a database, use the overloaded `setFileType()` method to specify whether the data is to be stored in a BLOB for binary data (the default) or a CLOB for character data. For a CLOB, set `fileType` to one of the following: the string "`character`", the defined `String` constant `FileAccessUtil.CHARACTER_FILE`, or the `int` value 1. To explicitly specify a BLOB, set `fileType` to one of the following: the string "`binary`", the defined

String constant `FileAccessUtil.BINARY_FILE`, or the int value 0. String settings are case-insensitive. No settings are accepted other than those listed here.

`FileAccessUtil` is in the `oracle.jsp.webutil.fileaccess` package.

- `String getFileType()`

Retrieve the file type information. Note there is a getter method for the string version only.

- `void setTable(String tableName)`

For an upload to a database table other than the default `fileaccess` table, use this method to specify the table name.

- `String getTable()`

Retrieve the table name.

- `void setPrefixColumn(String prefixColumnName)`

For an upload to a database table other than the default `fileaccess` table, use this method to specify the name of the column containing the file prefix. (In `fileaccess`, this column name is `fileprefix`.) The `destination` value will be written into this column.

- `String getPrefixColumn()`

Retrieve the name of the column containing the file prefix.

- `void setFileNameColumn(String fileNameColumnName)`

For an upload to a database table other than the default `fileaccess` table, use this method to specify the name of the column containing the file name. (In `fileaccess`, this column name is `filename`.) File names will include any file name extensions.

- `String getFileNameColumn()`

Retrieve the name of the column containing the file name.

- `void setDataColumn(String dataColumnName)`

For an upload to a database table other than the default `fileaccess` table, use this method to specify the name of the BLOB or CLOB column containing the file contents. (In `fileaccess`, this column name is `data`.)

- `String getDataColumn()`

Retrieve the name of the column containing the file contents.

- `void setConnection(ConnBean conn)`
- `void setConnection(java.sql.Connection conn)`

For an upload to a database table (default table or otherwise), use this overloaded method to provide a database connection. You can provide an instance of either `oracle.jsp.dbutil.ConnBean` or the standard `java.sql.Connection` type. For information about the `ConnBean` JavaBean, see "ConnBean for a Database Connection" on page 4-3.

If you use a `Connection` instance, you must explicitly open and close it. For a `ConnBean` instance, this is handled automatically.

- `java.util.Enumeration getFileNames()`

This method returns an `Enumeration` instance containing the names of the files that were uploaded. (This functionality is not available through the `httpUpload` tag.)

Example: This example uses a plain HTML form to specify a file to upload to a file system, then uses a JSP page that employs `HttpUploadBean` for the upload.

Here is the HTML form, which specifies `beanUploadExample.jsp` for its action and will generate the multipart upload stream.

> **Note:** Remember to set the base directory appropriately for uploads to a file system. See "File System Destination" on page 8-2.

```
<html><body>
<form action="beanUploadExample.jsp" ENCTYPE="multipart/form-data" method=POST>
<br> File to upload: <INPUT TYPE="FILE" NAME="File" SIZE="50" MAXLENGTH="120" >
<br><INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Send"> </form>
</body></html>
```

And here is the `beanUploadExample.jsp` page.

```
<%@ page language="java"
          import="java.util.*, oracle.jsp.webutil.fileaccess.*" %>
<html><body>
<% String userdir = "fileaccess"; %>   // user's part of the upload directory
<jsp:useBean id="upbean"
   class="oracle.jsp.webutil.fileaccess.HttpUploadBean" >
   <jsp:setProperty name="upbean" property="destination"
                     value="<%= userdir %>" />
</jsp:useBean>
<%   upbean.setBaseDir(application, request);
        upbean.upload(request);
        Enumeration fileNames = upbean.getFileNames();
        while (fileNames.hasMoreElements()) { %>
              <br><%= (String)fileNames.nextElement() %>
  <% } %>
<br>Done!
</body></html>
```

The HttpDownloadBean

The `oracle.jsp.webutil.fileaccess.HttpDownloadBean` JavaBean provides numerous setter methods for specifying information used for downloading. It also includes most corresponding getter methods. Once you have set all the required and appropriate attributes, use the `listFiles()` method to list the files available for download. The actual downloading is accomplished through `DownloadServlet`, supplied with OC4J, one file at a time. See "The Download Servlet" on page 8-12.

> **Note:** You must construct the URL for `DownloadServlet` in your application code.

`HttpDownloadBean`, as with `HttpUploadBean`, extends `HttpFileAccessBean`, which itself is not intended for public use.

See "Overview of File Uploading" on page 8-1 for related information.

Summary of Required Attributes

The following list summarizes required attributes for `HttpDownloadBean`:

- Always required: `source`

- Also required for uploads to a database: `sourceType`, `connection`

- Also required for downloads from a database table other than the default `fileaccess` table: `table`, `prefixColumn`, `fileNameColumn`, `dataColumn`

- Also required for downloads from a database table using a CLOB column for file data: `fileType`

In addition, for a download from a file system, you must call the `setBaseDir()` method to provide a servlet context and HTTP request object so that the bean can find the `fileaccess.properties` file that specifies the base directory.

Methods

Here are descriptions of the public methods of `HttpDownloadBean`.

Note: Many of the attributes and setter methods for `HttpDownloadBean` are the same as for `HttpUploadBean`.

- `void listFiles(javax.servlet.http.HttpServletRequest req)`
                                       `throws FileAccessException`

  Once all required and appropriate bean attributes have been set, use this method to list the files available for download. These are files in the source directory or matching the source database prefix. The `req` parameter is the HTTP response instance. For a JSP page, use the implicit `request` object.

  For use from the file list, you can create `HREF` links to `DownloadServlet`, passing it each file and file prefix, allowing users to select the link for each file they want to download.

  Note: The `listFiles()` method writes the file names to memory and to the JSP page or servlet. If you later want to access the file names again, use the `getFileNames()` method to read them from memory.

- `java.util.Enumeration getFileNames()`

  This method returns an `Enumeration` instance containing the names of the files that are available for download. It requires that the `listFiles()` method was already called. The `listFiles()` method writes the file names to memory and to the JSP page or servlet. The `getFileNames()` method reads them from memory.

- `void setBaseDir(javax.servlet.ServletContext sc,`
                   `javax.servlet.http.HttpServletRequest req)`
                                       `throws FileAccessException`

  For a download from a file system, use this method to determine what to use as the base directory. It gets this information from the `fileaccess.properties` file in your application `/WEB-INF` directory, which it finds through the servlet context input parameter. The `baseDir` setting, together with the `source` setting, specifies the absolute path to the directory from which files will be downloaded.

  The `sc` parameter is the servlet context instance for the application. For JSP pages, use the implicit `application` object.

The `req` parameter is for the HTTP request instance to use in requesting the base directory information. For JSP pages, use the implicit `request` object.

A base directory is not relevant for downloads from a database.

- `void setSource(String source)`

  This is always required.

  For a download from a file system, `source` and the base directory together specify the absolute path to the directory from which files will be downloaded. If `source` is set to "`*`", then all directories under the base directory will be available for downloading.

  For a download from a database, `source` is used as the file prefix. (The base directory is not relevant.) The prefix is equivalent to a file system path and can be used to group files into a hierarchy. If recursive downloading is enabled (through the `setRecurse()` method), "`%`" will be appended onto the `source` value, and the `WHERE` clause for the query will contain an appropriate `LIKE` clause. Therefore, all files with prefixes that are partially matched by the `source` value will be available for download. If you want to match all rows in the database table, set `source` to "`*`".

  > **Note:** Typically, the `source` value will be based at least partially on user input.

- `void setSourceType(String sourceType)`
  `throws FileAccessException`
- `void setSourceType(int sourceType)`
  `throws FileAccessException`

  Use the overloaded `setSourceType()` method to specify whether the download is from a file system or a database.

  To download from a database, set `sourceType` to one of the following: the string "`database`", the defined `String` constant `FileAccessUtil.DATABASE`, the `int` value 1, or the defined `int` constant `FileAccessUtil.LOCATION_TYPE_DATABASE`.

  Downloading from a file system is the default, but if you want to specify this explicitly, set `sourceType` to one of the following: the string "`filesystem`", the defined `String` constant `FileAccessUtil.FILESYSTEM`, the `int` value 0, or the defined `int` constant `FileAccessUtil.LOCATION_TYPE_FILESYSTEM`.

  `FileAccessUtil` is in the `oracle.jsp.webutil.fileaccess` package.

- `String getSourceType()`

  Retrieve the source type information. Note there is a getter method for the string version only.

- `void setRecurse(String recurse) throws FileAccessException`
- `void setRecurse(boolean recurse)`

  Use the overloaded `setRecurse()` method to enable or disable recursive downloading functionality, where files in file system subdirectories or with additional database prefix information will also be listed as available for downloading. As an example of this functionality from a database, assume `source` is set to "`/user`". Recursiveness would also find matches for files with

prefixes such as "`/user/bill`" and "`/user/mary`", and also such as "`/user1`", "`/user2`", "`/user1/tom`", and "`/user2/susan`".

Recursiveness is enabled by default, but you can enable it explicitly with a `recurse` setting of the string "`true`" or the boolean `true`. Disable the recursive functionality with a setting of the string "`false`" or the boolean `false`. String settings are case-insensitive. No settings are accepted other than those listed here.

- ```
  void setFileType(String fileType)
                            throws FileAccessException
  ```
- `void setFileType(int fileType) throws FileAccessException`

  For a download from a database, use the overloaded `setFileType()` method to specify whether the data is stored in a BLOB for binary data (the default) or a CLOB for character data. For a CLOB, set `fileType` to one of the following: the string "`character`", the defined `String` constant `FileAccessUtil.CHARACTER_FILE`, or the `int` value 1. To explicitly specify a BLOB, set `fileType` to one of the following: the string "`binary`", the defined `String` constant `FileAccessUtil.BINARY_FILE`, or the `int` value 0. String settings are case-insensitive. No settings are accepted other than those listed here.

  `FileAccessUtil` is in the `oracle.jsp.webutil.fileaccess` package.

- `String getFileType()`

  Retrieve the file type information. Note there is a getter method for the string version only.

- `void setTable(String tableName)`

  For a download from a database table other than the default `fileaccess` table, use this method to specify the table name.

- `String getTable()`

  Retrieve the table name.

- `void setPrefixColumn(String prefixColumnName)`

  For a download from a database table other than the default `fileaccess` table, use this method to specify the name of the column containing the file prefix. (In `fileaccess`, this column name is `fileprefix`.)

- `String getPrefixColumn()`

  Retrieve the name of the column containing the file prefix.

- `void setFileNameColumn(String fileNameColumnName)`

  For a download from a database table other than the default `fileaccess` table, use this method to specify the name of the column containing the file name. (In `fileaccess`, this column name is `filename`.) The file name includes any file name extension.

- `String getFileNameColumn()`

  Retrieve the name of the column containing the file name.

- `void setDataColumn(String dataColumnName)`

  For a download from a database table other than the default `fileaccess` table, use this method to specify the name of the BLOB or CLOB column that holds the file contents. (In `fileaccess`, this column name is `data`.)

- `String getDataColumn()`

Retrieve the name of the column contraining the file contents.

- `void setConnection(ConnBean conn)`

- `void setConnection(java.sql.Connection conn)`

  For a download from a database table (default table or otherwise), use this method to provide a database connection. You can provide an instance of either `oracle.jsp.dbutil.ConnBean` or the standard `java.sql.Connection` type. For information about the `ConnBean` JavaBean, see "ConnBean for a Database Connection" on page 4-3.

  If you use a `Connection` instance, you must explicitly open and close it. For a `ConnBean` instance, this is handled automatically.

Example This example is a JSP page that uses `HttpDownloadBean` for a download from a file system. Note that the page must construct the URL for the download servlet.

```
<%@ page language="java" import="java.util.*, oracle.jsp.webutil.fileaccess.*" %>
<html><body>
<% String servletPath = "/servlet/download/";  // path to the download servlet
   String userDir = "fileaccess/";   // user part of  download directory
%>
<jsp:useBean id="dbean"
   class="oracle.jsp.webutil.access.HttpDownloadBean" >
   <jsp:setProperty name="dbean" property="source" value='<%=userDir %>' />
</jsp:useBean>
<%   dbean.setBaseDir(application, request);
     dbean.listFiles(request); %>
The following files were found:
<%  Enumeration fileNames = dbean.getFileNames();
       while (fileNames.hasMoreElements()) {
               String name = (String)fileNames.nextElement();  %>
             <br><a href="<%= servletPath + name %>" > <%= name %></a>
<%   } %>
<br>Done!
</body></html>
```

The Download Servlet

To use download functionality, through either `HttpDownloadBean` or the `httpDownload` tag, you must have the class `oracle.jsp.webutil.fileaccess.DownloadServlet` available in your Web server. Its mapping in your Web server must be reflected in your servlet path settings, either through the `servletPath` attribute if you use the `httpDownload` tag, or in your application code if you use `HttpDownloadBean`.

FileAccessException Class

The `oracle.jsp.webutil.fileaccess.FileAccessException` class is a convenience class supplied with OC4J for file-access exception-handling. It wraps the functionality of the standard `java.sql.SQLException` and `java.io.IOException` classes. It handles exceptions from either of the file-access beans in addition to handling SQL and I/O exceptions.

File Upload and Download Tag Descriptions

For file uploading, OC4J supplies the `httpUpload` tag. This tag, in turn, uses `HttpUploadBean`. For convenience, you can also use the `httpUploadForm` tag in

programming the form through which users specify the files to upload, or you can code the form manually.

For file downloading, OC4J provides the custom `httpDownload` tag. This tag uses `HttpDownloadBean`. This section describes these tags and their attributes.

Note the following requirements for the file upload and download tags:

- Verify that the file `ojsputil.jar` is installed and in your classpath. This file is provided with OC4J, in the "well-known" tag library directory.

- The tag library descriptor, `fileaccess.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `fileaccess.tld` is the following:

  ```
  http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/fileaccess.tld
  ```

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

Notes:

- The prefix "fileaccess:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.

- See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

The **httpUploadForm** Tag

For convenience, you can use the `httpUploadForm` tag to create a form in your application, using multipart encoded form data, that allows users to specify the files to upload.

Syntax

```
<fileaccess:httpUploadForm formsAction = "action"
                    [ maxFiles = "max_number" ]
                    [ fileNameSize = "file_input_box_num_chars" ]
                    [ maxFileNameSize = "max_file_name_num_chars" ]
                    [ includeNumbers = "true" | "false" ]
                    [ submitButtonText = "button_label_text" ] />
```

Note: The `httpUploadForm` tag can optionally use a body. For example, the body might consist of a user prompt.

Attributes

- `formsAction` (required): This is to indicate the action that will be performed after the form is submitted. For example, `formsAction` could be the name of a JSP page that uses `HttpUploadBean` or the `httpUpload` tag.

- `maxFiles`: Use this if you want to specify the number of input lines you want to appear in the form. The default is 1.

- `fileNameSize`: Use this if you want to specify the character-width of the file name input box (or boxes). The default is 20 characters.

- `maxFileNameSize`: Use this if you want to specify the maximum number of characters allowed in a file name. The default is 80 characters.

- `includeNumbers`: Set this to `"true"` if you want the file name input boxes to be numbered. The default setting is `"false"`.

- `submitButtonText`: Use this if you want to specify the text that appears on the "submit" button of the form. The default is `"Send"`.

The httpUpload Tag

This tag wraps the functionality of the `HttpUploadBean` JavaBean, paralleling its attributes. See "Overview of File Uploading" on page 8-1 and "The HttpUploadBean" on page 8-4 for related information.

Syntax

```
<fileaccess:httpUpload destination = "dir_path_or_prefix"
                  [ destinationType = "filesystem" | "database" ]
                  [ connId = "id" ]
                  [ scope = "request" | "page" | "session" | "applicaton" ]
                  [ overwrite = "true" | "false" ]
                  [ fileType = "character" | "binary" ]
                  [ table = "table_name" ]
                  [ prefixColumn = "column_name" ]
                  [ fileNameColumn = "column_name" ]
                  [ dataColumn = "column_name" ] />
```

> **Note:** For uploads to a file system, the base directory is automatically retrievable by the tag handler from the JSP page context.

Attributes

- `destination` (required): For uploading to a file system, this indicates the path, beneath the base directory supplied in the `/WEB-INF/fileaccess.properties` file, of the directory into which files will be uploaded. For uploading to a database, `destination` indicates the file prefix, conceptually equivalent to a file system path.

  > **Note:** Typically, the `destination` value will be based at least partially on user input.

- `destinationType`: Set this to `"database"` for uploading to a database. The default is to upload to a file system, but you can also explicitly set it to `"filesystem"`. These values are case-insensitive.

- `connId`: For uploading to a database, use this attribute to provide a `ConnBean` connection ID for the database connection to be used. Or, alternatively, use the `httpUpload` tag inside a `dbOpen` tag to implicitly use the `dbOpen` connection. For information about the `ConnBean` JavaBean and `dbOpen` tag provided with OC4J, see Chapter 4, "Data-Access JavaBeans and Tags".

- `scope`: For uploading to a database, use this attribute to specify the scope of the `ConnBean` instance for the connection. The scope setting here must match the scope setting when the `ConnBean` instance was created, such as in a `dbOpen` tag. If the `httpUpload` tag is nested inside a `dbOpen` tag, then there is no need to specify `connId` or `scope`. In this case, information will be taken from the `dbOpen` tag. Otherwise, the default scope setting is "`page`".

- `overwrite`: Set this to "`false`" if you do not want to overwrite existing files that have the same paths and names as the files you are uploading, or if you do not want to update rows with the same file name and prefix for database uploading. In this case, an error will be generated if a file already exists. By default, `overwrite` is set to "`true`" and `httpUpload` overwrites files.

- `fileType`: For uploading to a database, set this attribute to "`character`" for character data, which will be written into a CLOB. The default setting is "`binary`" for binary data, which will be written into a BLOB.

- `table`: For uploading to a database table other than the default `fileaccess` table, use this attribute to specify the table name.

- `prefixColumn`: For uploading to a database table other than the default `fileaccess` table, use this attribute to specify the name of the column containing file prefixes. This column is where the `destination` values will be written.

- `fileNameColumn`: For uploading to a database table other than the default `fileaccess` table, use this attribute to specify the name of the column containing file names.

- `dataColumn`: For uploading to a database table other than the default `fileaccess` table, use this attribute to specify the name of the column containing file contents.

Example This example has a page that uses the `httpUploadForm` tag to create the HTML form for specifying files to upload. The `httpUploadForm` tag specifies `httpUploadExample.jsp` as its forms action. The `httpUploadExample.jsp` page uses the `httpUpload` tag to upload to the default `fileaccess` table in a database.

Here is the page for the HTML form:

```
<%@ page language="java" import="java.io.*"  %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/fileaccess.tld"
          prefix="upload" %>
<html> <body>
<fileaccess:httpUploadForm
     formsAction="httpUploadExample.jsp"
     maxFiles='<%= request.getParameter("MaxFiles") %>'
     includeNumbers="true"  fileNameSize="50"  maxFileNameSize="120" >
         <br> File:
</fileaccess:httpUploadForm>
</body> </html>
```

And following is the `httpUploadExample.jsp` page. Note that the `httpUpload` tag gets its database connection as a result of being inside a `dbOpen` tag. Also note that `useDataSource.jsp` is used to obtain the connection, if necessary. See "useDataSource.jsp" on page 5-8.

```
<%@ page language="java" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/fileaccess.tld"
          prefix="upload" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/sqltaglib.tld"
          prefix="sql" %>
```

```
<% String dataSrcStr=request.getParameter("dataSrcStr"); // get conn string
    if (dataSrcStr==null) {  dataSrcStr=(String)session.getValue("dataSrcStr"); }
    else {  session.putValue("dataSrcStr",dataSrcStr); }
    if (dataSrcStr==null) { %>
        <jsp:forward page="useDataSource.jsp" />
<% } %>
<html><body>
<sql:dbOpen dataSource="<%= dataSrcStr %>" >
    <fileaccess:httpUpload  destinationType = "database"
                            destination="tagexample" />
</sql:dbOpen>
Done! </body></html>
```

> **Note:**   For the `dbOpen` tag in this example, assume that the data
> source specifies the user name and password as well as the URL.

The httpDownload Tag

This tag wraps the functionality of the `HttpDownloadBean` JavaBean, paralleling its
attributes. See "Overview of File Downloading" on page 8-3 and "The
HttpDownloadBean" on page 8-8 for related information.

Syntax

```
<fileaccess:httpDownload servletPath = "path"
                  source = "dir_path_or_prefix"
               [ sourceType = "filesystem" | "database" ]
               [ connId = "id" ]
               [ scope = "request" | "page" | "session" | "applicaton" ]
               [ recurse = "true" | "false" ]
               [ fileType = "character" | "binary" ]
               [ table = "table_name" ]
               [ prefixColumn = "column_name" ]
               [ fileNameColumn = "column_name" ]
               [ dataColumn = "column_name" ] />
```

> **Notes:**
>
> - The `httpDownload` tag can optionally use a body. For
>   example, the body might consist of a user prompt.
>
> - For downloads from a file system, the base directory is
>   automatically retrievable by the tag handler from the JSP page
>   context.

Attributes

- `servletPath` (required): This is the path to the Oracle `DownloadServlet`,
  which executes the actual download of each file. For example, if
  `DownloadServlet` has been installed in the application `app` and mapped to the
  name `download`, then use `"/app/download/"`, with leading and trailing slashes,
  as the `servletPath` setting. The `httpDownload` tag handler uses this path in
  constructing the URL to `DownloadServlet`.

  See "The Download Servlet" on page 8-12 for more information about this servlet.

- source (required): For downloading from a file system, this attribute indicates the path, beneath the base directory supplied in the file /WEB-INF/fileaccess.properties, of the directory from which files are retrieved. A value of "*" results in all directories under the base directory being available.

  For downloading from a database, this attribute indicates the file prefix, conceptually equivalent to a file system path. If recursive downloading is enabled (through the recurse attribute), "%" will be appended onto the source value, and the WHERE clause for the query will contain an appropriate LIKE clause. Therefore, all files with prefixes that are partially matched by the source value will be available for download. If you want to match all rows in the database table, set source to "*".

  > **Note:** Typically, the source value is based at least partially on user input.

- sourceType: Set this to "database" for downloading from a database. The default is to download from a file system, or you can explicitly set this to "filesystem".

- connId: For downloading from a database, use this attribute to provide a ConnBean connection ID for the database connection to be used. Or, alternatively, you can use the httpDownload tag inside a dbOpen tag to implicitly use the dbOpen connection. For information about the ConnBean JavaBean and dbOpen tag provided with OC4J, see Chapter 4, "Data-Access JavaBeans and Tags".

- scope: For downloading from a database, use this attribute to specify the scope of the ConnBean instance for the connection. The scope setting here must match the scope setting when the ConnBean instance was created, such as in a dbOpen tag. If the httpDownload tag is nested inside a dbOpen tag, then there is no need to specify connId or scope. In this case, information will be taken from the dbOpen tag. Otherwise, the default scope setting is "page".

- recurse: Set this to "false" if you do not want recursive downloading functionality, where files in file system subdirectories or with additional database prefix information will also be listed as available for download. As an example of this functionality from a database, assume you have set source to "/user". Recursiveness would also find matches for files with prefixes such as "/user/bill" and "/user/mary", and also such as "/user1", "/user2", "/user1/tom", and "/user2/susan". The default mode is recursiveness, or you can enable it explicitly with a setting of "true".

- fileType: For downloading from a database, set this attribute to "character" for character data, which will be retrieved from a CLOB. The default setting is "binary" for binary data, which will be retrieved from a BLOB.

- table: For downloading from a database table other than the default fileaccess table, use this attribute to specify the table name.

- prefixColumn: For downloading from a database table other than the default fileaccess table, use this attribute to specify the name of the column containing file prefixes, which is where source values are stored.

- fileNameColumn: For downloading from a database table other than the default fileaccess table, use this attribute to specify the name of the column containing file names. File names include any file name extensions.

- dataColumn: For downloading from a database table other than the default fileaccess table, use this attribute to specify the name of the column that stores the file contents.

Example This example is a JSP page that uses the httpDownload tag to download from the default fileaccess table of a database. The tag body content ("
:") will be output before each file name in the list of files available for download. Note that you must specify the DownloadServlet servlet path in the httpDownload tag. The tag handler will use it in constructing the URL to DownloadServlet, which performs the actual downloading.

```
<%@ page language="java" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/fileaccess.tld"
          prefix="download" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/sqltaglib.tld"
          prefix="sql" %>
<% String dataSrcStr=request.getParameter("dataSrcStr");
   if (dataSrcStr==null) { dataSrcStr=(String)session.getValue("dataSrcStr");}
   else { session.putValue("dataSrcStr",dataSrcStr);}
   if (dataSrcStr==null) { %>
      <jsp:forward page="useDataSource.jsp" />
<% } %>
<html> <body>
<% String servletPath = "/servlet/download/"; %>
<sql:dbOpen dataSource="<%= dataSrcStr %>" >
<fileaccess:httpDownload sourceType = "database"
            source="tagexample" servletPath = '<%= servletPath %>' >
   <br>:
</fileaccess:httpDownload>
</sql:dbOpen>
<br>Done!
</body> </html>
```

> **Note:** For the dbOpen tag in this example, assume that the data source specifies the user name and password as well as the URI.

Mail JavaBean and Tag

It is often useful to send e-mail messages from a Web application, based on Web site status or user actions, for example. Sun Microsystems has specified a platform-independent and protocol-independent framework for this through its javax.mail package and subpackages, known as the JavaMail API.

For further convenience, Oracle supplies a JavaBean and JSP custom tag based on the JavaMail API to use in providing e-mail functionality through your servlets or JSP pages. The bean and tag, as with other JavaBeans and custom tags supplied with OC4J, are implemented according to JSP and servlet standards.

The following sections describe the mail JavaBean and tag:

- General Considerations for the Mail JavaBean and Tag
- Mail Attachments
- SendMailBean Description
- The sendMail Tag Description

For more information about the JavaMail API, refer to the following Sun Microsystems Web site:

`http://java.sun.com/products/javamail/1.2/docs/javadocs/index.html`

General Considerations for the Mail JavaBean and Tag

Be aware of the following points, which apply to use of either the mail JavaBean (`SendMailBean`) or the mail tag (`sendMail`):

- The files `mail.jar`, containing the JavaMail packages, and `jaf.jar`, for the JavaBeans Activation Framework, must be in your classpath for mail functionality. These files are provided with OC4J.

- To enable support for attachments, the file `sendmail.properties` must exist, with an appropriate setting, in the application `/WEB-INF` directory. See "Enabling Attachments" on page 8-19.

- There is no particular limit to the size of an e-mail message, other than limits of the JVM, system memory, or mail server.

- Setting up default mail sessions is specific to the particular Web server. The current implementations of the mail bean and tag do not support automatic use of the default mail session. As an alternative, you can write your own code to obtain the default mail session if one exists for your platform, and to make the session available to the mail bean or tag.

Mail Attachments

The mail bean and tag support the sending of attachments with e-mail messages. (This support was introduced in the OC4J 9.0.3 implementation.) There are three modes of operation:

- No support for attachments

- Support for attaching one or more files that are on the OC4J server machine, known as *server-side attachments*

- Support for attaching one file that is on the client machine, known as a *client-side attachment*

For a client-side attachment, the file is automatically uploaded to the server machine as part of the process. Multiple client-side attachments are not supported.

Enabling Attachments

Whether attachments are enabled, and which kind of attachments, is determined by a `sendmail.properties` file in the application `/WEB-INF` directory. A file with the following content disables attachments:

```
## email attachment permissions
sendmail.attachment=none
```

You must create this file in `/WEB-INF` and update it appropriately for any OC4J instance that will use mail attachments.

Any single application can support server-side attachments or client-side attachments, but not both.

To enable server-side attachments, change the setting to `server`, as follows:

```
sendmail.attachment=server
```

To enable client-side attachments, change the setting to `client`:

```
sendmail.attachment=client
```

Having multiple settings is an error condition.

> **Note:** The absence of a `sendmail.properties` file is treated as equivalent to the presence of `sendmail.properties` with a setting of `none`. Mail attachments are disabled in this case.

Sending Attachments

For the mail tag, if server-side attachments are enabled, use the `serverAttachment` tag attribute if you want to specify one or more server-side files to attach to a message. If client-side attachments are enabled, use the `clientAttachment` tag attribute if you want to specify a client-side file to attach to a message (maximum of one file). See "The sendMail Tag Description" on page 8-24. Note that either one of the two attachment modes, but not both, can be supported for any single application.

For both the server attachment mode and the client attachment mode, the mail bean includes methods to specify or retrieve the name (or names) of the file (or files) to attach. See information about `setServerAttachment()`, `getServerAttachment()`, `setClientAttachment()`, and `getClientAttachment()` in "SendMailBean Method Descriptions" on page 8-21.

With either the mail tag or mail bean, a list of server-side files to attach can be either comma-delimited or semicolon-delimited, but not space-delimited (given that spaces are allowed in file names in some operating systems).

Attachment Usage Notes

Be aware of the following usage notes for mail attachments, applying to both the mail tag and mail bean.

- For a client-side file attachment, the file-access `httpUpload` tag is used behind the scenes. The file is uploaded to a temporary location on the OC4J server machine, then deleted once the message has been sent. Any limitations or requirements of the `httpUpload` tag apply to a client-side mail attachment as well. See "File Upload and Download Tag Descriptions" on page 8-12.

- Many e-mail servers have somewhat restrictive size limitations, often approximately 4 MB for any one attachment. The only restrictions for the mail tag or bean are according to disk or memory limitations of the server machine.

- If a problem is encountered with any attachment, the e-mail message is terminated.

- Path names are not exposed to the mail recipient in either the server attachment or client attachment mode. Only the file name itself is indicated.

- For server-side attachments, attaching multiple files of the same name (but obviously with different paths) is supported. How this is handled at the recipient end, regarding any possible file renaming to avoid conflict, is according to the mail client being used. Similarly, in either attachment mode, the mail client might rename a file if an attachment has the same name as an attachment to a previous message. This is all beyond the scope and control of the OC4J mail attachment feature.

- You cannot use wild-card characters for file names.

SendMailBean Description

The `oracle.jsp.webutil.email.SendMailBean` JavaBean is supplied with OC4J to support e-mail functionality from servlet or JSP applications. To use it in a JSP page, you can instantiate it through the standard `jsp:useBean` tag. (For JSP applications, however, you would typically use the `sendMail` tag instead of `SendMailBean`. See "The sendMail Tag Description" on page 8-24.)

SendMailBean Requirements

To use `SendMailBean`, verify that the files `ojsputil.jar`, `mail.jar`, and `activation.jar` are installed and in your classpath. These files are supplied with OC4J.

When you use `SendMailBean` in your code, you must provide the following:

- Message sender

  Use the `setSender()` method to specify the sender.

- Primary recipient (or recipients) of the message

  Use the `setRecipient()` method to specify the primary recipient (or recipients).

- Valid JavaMail session object (`javax.mail.Session`), either directly or indirectly

  There are three ways to supply a JavaMail session:

  - Use the `setHost()` method to specify a host system. In this case, a JavaMail session object will be created automatically.

  - Use the `setMailSession()` method to provide a JavaMail session object directly.

  - For JSP applications, use the `setSession()` method to specify the name of a JavaMail session object that already exists and is accessible through a "session string, `javax.mail.Session` object" pair in the JSP page context. In this case, you must supply the page context instance as an input parameter when you call the `sendMessage()` method to send the e-mail message.

All other `SendMailBean` attributes are optional.

SendMailBean Method Descriptions

This section lists and describes `SendMailBean` methods to send mail messages, close mail sessions, and set or get bean attributes.

> **Note:** To comply with the JavaBean specification, `SendMailBean` has a no-argument constructor.

Here are the public `SendMailBean` methods:

- `void sendMessage()`
- `void sendMessage(javax.servlet.jsp.PageContext)`

  Use the `sendMessage()` method to send the e-mail message.

  If you use the `setSession()` method to supply a JavaMail session, then you must use the `sendMessage(PageContext)` signature and provide the page context instance that holds the specified mail session instance.

If you use the `setMailSession()` or `setHost()` method to supply a JavaMail session, then you do *not* have to provide a page context in using the `sendMessage()` method.

Also be aware, however, that specifying a page context instance might be relevant in determining the character set of an e-mail message with a "text" content type. If you provide no page context when invoking the `sendMessage()` method, then the default character set is `ISO-8859-1`. If you *do* provide a page context, then the default character set is that of the `response` object of the page context. Also note that you can specify the content type and character set directly through the `setContentType()` method.

- `void close()`

  Use this method if you want to release the resources of the JavaMail session instance from the `SendMailBean` instance. This method does not actually close the session.

- `void setBcc(String s)`

  Specify a space-delimited or comma-delimited list of any IDs (e-mail addresses or aliases) to receive blind copies of the message. These IDs will be suppressed from the message Cc field.

- `String getBcc()`

  Retrieve the list of IDs to receive blind copies of the message.

- `void setCc(String s)`

  Specify a space-delimited or comma-delimited list of any IDs (e-mail addresses or aliases) to receive copies of the message. These IDs will appear in the message Cc field.

- `String getCc()`

  Retrieve the list of IDs to receive copies of the message.

- `void setContent(String s)`

  Specify the contents of the e-mail message.

- `String getContent()`

  Retrieve the contents of the e-mail message.

- `void setContentEncoding(String s)`

  Specify the content encoding of the e-mail message. Specify "`base64`" or "`B`" for base64 encoding, "`quoted-printable`" or "`Q`" for quoted-printable encoding, "`7bit`" for seven-bit encoding, or "`8bit`" for eight-bit encoding. These content encodings are part of the JavaMail and RFC 2047 standards. Entries are case-insensitive.

  The default content encoding setting is "`null`", in which case the encoding of the message and headers will be determined by the content. If most characters to be encoded are in ASCII, then quoted-printable encoding will be used; otherwise, `base64` encoding will be used.

- `String getContentEncoding()`

  Retrieve the content encoding of the message.

- `void setContentType(String s)`

Specify the MIME type and optionally the character set of the message, such as in the following examples:

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

setContentType("text/html; charset=US-ASCII");
```

The default MIME type setting is `"text/plain"`, but you cannot specify a character set without explicitly specifying that or some other `"text/xxxx"` MIME type setting.

The default character set depends on whether you provide a JSP page context instance when you call the `sendMessage()` method to send the e-mail message. If you provide no page context, then the default character set is `ISO-8859-1`. If you do provide a page context, then the default character set is that of the `response` object of the page context.

- `String getContentType()`

  Retrieve the MIME type (and character encoding, if applicable) of the message.

- `void setHost(String s)`

  One of the ways to supply a JavaMail session is to specify a mail server host name, in which case `SendMailBean` will obtain a session automatically. Use the `setHost()` method for this purpose, providing a mail host name such as `"gmail.oraclecorp.com"`.

  See "SendMailBean Requirements" on page 8-21 for an overview of supplying the JavaMail session.

- `String getHost()`

  Retrieve the specified mail server host name.

- `void setMailSession(javax.mail.Session sessobj)`

  One of the ways to supply a JavaMail session is to provide the session object directly. Use the `setMailSession()` method for this purpose, providing a `javax.mail.Session` instance.

  See "SendMailBean Requirements" on page 8-21 for an overview of supplying the JavaMail session.

- `javax.mail.Session getMailSession()`

  This returns a JavaMail session that you had previously set.

- `void setRecipient(String s)`

  Specify a space-delimited or comma-delimited list of IDs (e-mail addresses or aliases) of the primary recipients of the message. These IDs will appear in the To field of the message. You must specify at least one recipient.

- `String getRecipient()`

  Retrieve the list of IDs of the primary recipients of the message.

- `void setSender(String s)`

  Specify the ID (e-mail address or alias) of the message sender. This ID will appear in the From field of the message. You must specify the sender.

- `String getSender()`

  Retrieve the ID of the message sender.

- `void setSession(String s)`

  One of the ways to supply a JavaMail session is to provide the name of a `javax.mail.Session` instance that already exists in the JSP page context object. Use the `setSession()` method for this purpose, specifying the name of the session instance.

  In this case, when you use the `sendMessage()` method to send the e-mail message, you must provide the `javax.servlet.jsp.PageContext` instance as input.

  See "SendMailBean Requirements" on page 8-21 for an overview of supplying the JavaMail session.

- `String getSession()`

  Retrieve the name of the session instance.

- `void setSubject(String s)`

  Specify the subject line of the message.

- `String getSubject()`

  Retrieve the subject line of the message.

- `void setServerAttachment(String s)`

  Specify a comma-delimited or semicolon-delimited list of file names (including paths), for server-side files to attach to an e-mail message. These must be files on the OC4J server machine. Server-side attachments must be enabled in the `sendmail.properties` file.

- `String getServerAttachment()`

  Retrieve the file name list for server-side files to attach to the message. This might be useful in presenting a user confirmation page, for example.

- `void setClientAttachment(String s)`

  Specify the path and file name of the client-side file to attach to the e-mail message (maximum of one). This must be a file on the user's client machine. Client-side attachments must be enabled in the `sendmail.properties` file.

- `String getClientAttachment()`

  Retrieve the name of the client-side file to attach to the message. This might be useful in presenting a user confirmation page, for example.

  > **Note:** Regarding mail attachments, see "Mail Attachments" on page 8-19 for related information. Be aware that for any one application, you can use server-side attachments or client-side attachments but not both.

The sendMail Tag Description

As a convenience for JSP developers, OC4J supplies the `sendMail` tag to provide e-mail functionality for a JSP page. The following sections describe the tag:

- The sendMail Tag Syntax

- The sendMail Tag Attribute Descriptions

- Sample Application for sendMail Tag

Be aware of the following requirements for the `sendMail` tag:

- Verify that the files `ojsputil.jar`, `mail.jar`, and `activation.jar` are installed and in your classpath. These files are supplied with OC4J; `ojsputil.jar` is in the "well-known" tag library directory.

- In the current implementation, the `sendMail` tag has its own tag library descriptor, `email.tld`. This must be available to the application, and any JSP page using the tag must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `email.tld` is the following:

  ```
  http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/email.tld
  ```

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

The sendMail Tag Syntax

The `sendMail` tag has the following syntax:

```
<mail:sendMail host = "SMTP_host_name" | session = "JavaMail_session_name"
               sender = "sender_address"
               recipient = "primary_recipient_IDs"
             [ cc = "cc_recipient_IDs" ]
             [ bcc = "bcc_recipient_IDs" ]
             [ subject = "subject_line" ]
             [ contentType = "MIME_type; [charset=charset]" ]
             [ contentEncoding = "B"|"base64"|"Q"|"quoted-printable"|
                                 "7bit"|"8bit" ]
             [ serverAttachment = "server_file_list" |
               clientAttachment = "client_file" ] >
...
E-mail body
...
</mail:sendMail>
```

sendMail Tag Usage Notes Be aware of the following when using the `sendMail` tag:

- The `sender` and `recipient` attributes are required, and either the `host` or `session` attribute is required.

- Multiple recipients, cc targets, or bcc targets are space-delimited or comma-delimited.

- Use of `serverAttachment` assumes server-side attachments are enabled in the `sendmail.properties` file. Similarly, use of `clientAttachment` assumes client-side attachments are enabled in `sendmail.properties`. Only one mode can be enabled for a single application. See "Enabling Attachments" on page 8-19.

- File names in the `serverAttachment` setting can be comma-delimited or semicolon-delimited, but not space-delimited.

- The e-mail body can contain JSP syntax, which will be processed by the JSP translator.

- Attributes used by the tag are typically input by the user in form fields. All attributes accept request-time expressions.

- The prefix "mail:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.

- See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

The sendMail Tag Attribute Descriptions

The `sendMail` tag supports the following attributes:

- `host` (required if `session` is not specified): This is the appropriate mail host name, such as `"gmail.oraclecorp.com"`. This is used in creating a JavaMail session object for the mail message. Alternatively, you can determine a JavaMail session through the `session` attribute.

- `session` (required if `host` is not specified): This is the name of an existing JavaMail session object that can be retrieved from the JSP page context. Alternatively, you can determine a JavaMail session through the `host` attribute.

- `sender` (required): This is the ID (e-mail address or alias) of the sender of the message. This ID will appear in the From field of the message.

- `recipient` (required): This is a space-delimited or comma-delimited list of IDs of the primary recipients of the message. These IDs will appear in the To field of the message.

- `cc` : This is a space-delimited or comma-delimited list of IDs to receive a copy of the message. These IDs will appear in the Cc field of the message.

- `bcc` : This is a space-delimited or comma-delimited list of IDs to receive a blind copy of the message. These IDs will be suppressed from the Cc field.

- `subject`: This is the subject line of the message.

- `contentType`: This is for the MIME type of the message, and optionally a character set as well, as in the following examples:

```
contentType="text/html"

contentType="text/html; charset=US-ASCII"
```

The default MIME type setting is `"text/plain"`, but you cannot specify a character set without explicitly specifying that or some other text/*xxxx* MIME type.

The default character set is that of the `response` object of the JSP page context.

- `contentEncoding`: Specify `"B"` or `"base64"` for base64 encoding, `"Q"` or `"quoted-printable"` for quoted-printable encoding, `"7bit"` for seven-bit encoding, or `"8bit"` for eight-bit encoding. These are standard JavaMail and RFC 2047 encodings. Entries are case-insensitive.

The default content encoding setting is `"null"`, in which case the encoding of the message and headers will be determined by the content. If most characters to be encoded are in ASCII, then quoted-printable encoding will be used. Otherwise, `base64` encoding will be used.

- `serverAttachment`: This is a comma-delimited or semicolon-delimited list of server-side files to attach to the e-mail message. Server-side attachments must be enabled in the `sendmail.properties` file.

Here is an example:

```
serverAttachment="/tmp/confirm.pdf,/home/schedule.doc"
```

- clientAttachment: This is the name of a client-side file (maximum of one) to attach to the e-mail message. Client-side attachments must be enabled in the sendmail.properties file.

  Here is an example:

  ```
  clientAttachment="c:\finance\budget02.xls"
  ```

Note: Regarding e-mail attachments, see "Mail Attachments" on page 8-19 for related information. Be aware that for any one application, you can use server-side attachments or client-side attachments but not both.

Sample Application for sendMail Tag

This sample application illustrates use of the sendMail tag with no attachments. During the first execution cycle through the page, before the user has specified the sender (or anything else), the HTML form is displayed for user input. During the next execution cycle through the page, after the user has sent the input, the sendMail tag is executed. This page also uses an error page, error.jsp (shown below), to display any exceptions that are thrown.

```
<%@ page language="java" errorPage="error.jsp" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/email.tld"
         prefix="mail" %>
<%
if (request.getParameter("sender")==null) {
%>
<HTML>
<HEAD><TITLE>SendMail Sample</TITLE></HEAD>
<FORM METHOD=post>
<TABLE BORDER=0 CELLSPACING=0 CELLPADDING=0 WIDTH="20%">
<TR><TD>Host:</TD><TD><INPUT TYPE="text" name="host" ></TD></TR>
<TR><TD>From:</TD><TD><INPUT TYPE="text" name="sender" ></TD></TR>
<TR><TD>To:</TD><TD><INPUT TYPE="text" name="recipient"  ></TD></TR>
<TR><TD>Cc:</TD><TD><INPUT TYPE="text" name="cc" ></TD></TR>
<TR><TD>Bcc:</TD><TD><INPUT TYPE="text" name="bcc" ></TD></TR>
<TR><TD>Subject:</TD><TD><INPUT TYPE="text" name="subject"
VALUE="Hi"></TD></TR>
</TABLE><br>
<TEXTAREA name="body" ROWS=4 COLS=30>"How are you!"</TEXTAREA><br><br>
<INPUT TYPE="submit" value="Send">
</FORM>
<%
}
else{
%>
<BODY BGCOLOR="#FFFFFF">
<P>Result:
      <HR>
      <mail:sendMail host='<%=request.getParameter("host")%>'
            sender='<%=request.getParameter("sender")%>'
            recipient='<%=request.getParameter("recipient")%>'
            cc='<%=request.getParameter("cc")%>'
            bcc='<%=request.getParameter("bcc")%>'
            subject='<%=request.getParameter("subject")%>'>
            <%=request.getParameter("body")%>
      </mail:sendMail>
```

```
Sent out Successfully!
        <HR>
</BODY>
<%
}
%>
</HTML>
```

Here is the error page, `error.jsp`:

```
<%@ page language="java" isErrorPage="true"%>
<HTML>
Error: <%= exception.getMessage() %>
</HTML>
```

When you run this application, you will initially see the following default screen:
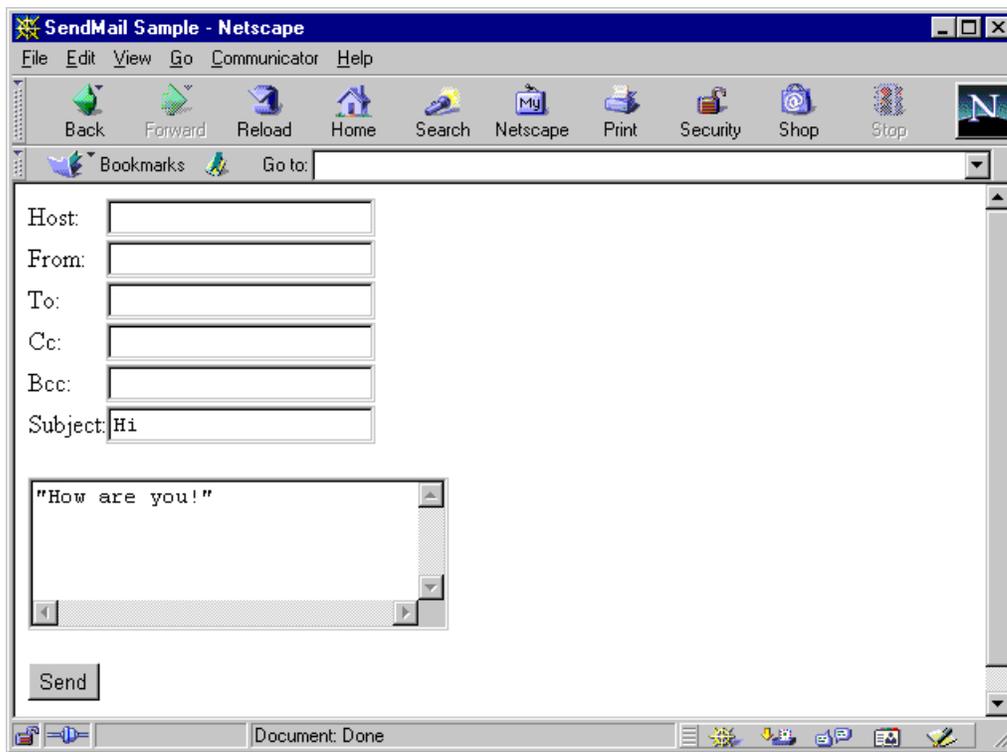
And here is sample user input for a message from `brian.wright@oracle.com` to `blodney.treehut@oracle.com` through the host `gmail.oraclecorp.com`:

9
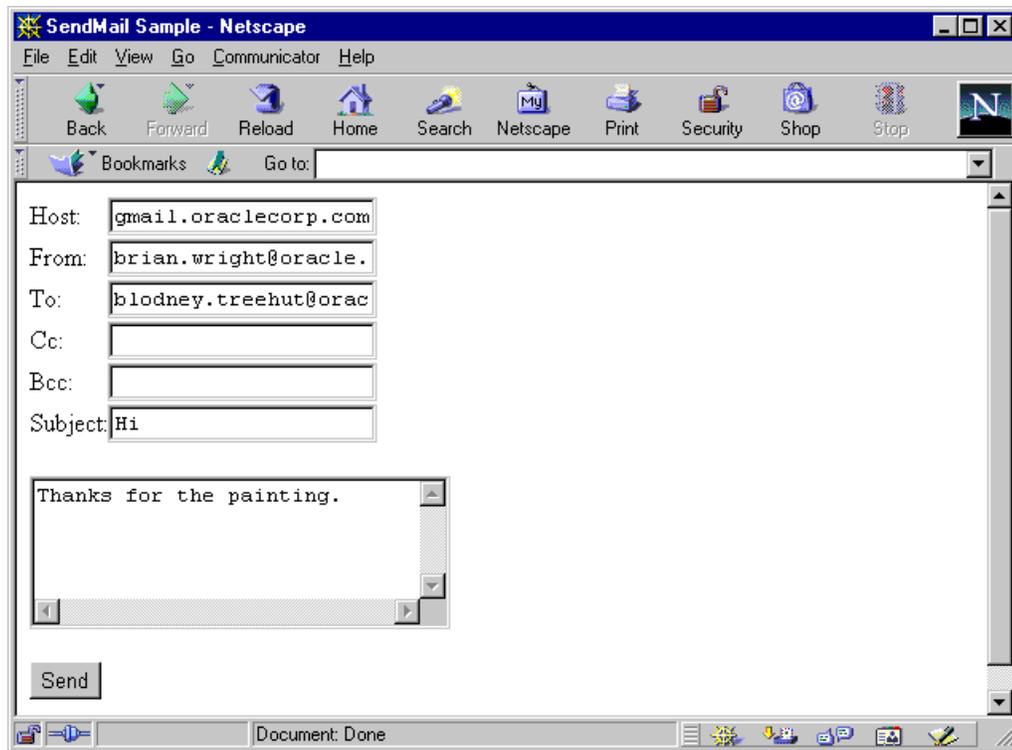
JSP Utilities and Utility Tags

This chapter, consisting of the following sections, documents various OC4J utility features for JSP pages:

- JSP Event-Handling with JspScopeListener
- EJB Tags
- General Utility Tags

JSP Event-Handling with JspScopeListener

In standard servlet and JSP technology, only session-based events are supported. Oracle extends this support to page-based, request-based, and application-based events through the `JspScopeListener` interface and `JspScopeEvent` class in the `oracle.jsp.event` package.

`JspScopeListener` functionality is documented in the following sections, concluding with examples:

- General Use of JspScopeListener
- Use of JspScopeListener in OC4J and Other Servlet 2.3 Environments
- Examples Using JspScopeListener

General Use of JspScopeListener

For Java objects 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 notified. The JSP container accomplishes this by sending a `JspScopeEvent` instance to such objects through the `outOfScope()` method specified in the `JspScopeListener` interface.

This 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.

Properties of the `JspScopeEvent` object include the following:

- Scope that is ending, represented by one of the `int` constants `PAGE_SCOPE`, `REQUEST_SCOPE`, `SESSION_SCOPE`, or `APPLICATION_SCOPE`

  You can retrieve this scope with the following `JspScopeEvent` method:

```
public int getScope()
```

- Container object that is the repository for objects at this scope, one of the implicit objects `page`, `request`, `session`, or `application`

  This is the object that manages the relevant scope. You can retrieve this object with the following `JspScopeEvent` method:

  ```
  public java.lang.Object getContainer()
  ```

- Name of the object to which the notification pertains

  This is the name of the instance of the class that implements `JspScopeListener`. The instance of this class is an attribute of either the `page`, `request`, `session`, or `application` object (as applicable), so this instance name is the attribute name. You can retrieve this name with the following `JspScopeEvent` method:

  ```
  public String getName()
  ```

- JSP implicit `application` object

  You can retrieve this with the following `JspScopeEvent` method:

  ```
  public ServletContext getApplication()
  ```

The `JspScopeEvent` class has a constructor as follows:

```
public JspScopeEvent (ServletContext sc, Object container, String name,
                      int scope)
```

Use of JspScopeListener in OC4J and Other Servlet 2.3 Environments

`JspScopeListener` uses different mechanisms to support the different scopes, though all are implemented according to servlet and JSP standards.

For pages running in an OC4J environment, there is also an OC4J-specific runtime implementation for `page` scope, for convenience.

These features are covered in the following sections:

- Requirements for JspScopeListener
- Runtime and Tag Implementations to Support Page Scope
- Servlet Filter Implementation to Support Request Scope
- Listener Class Implementation to Support Application Scope
- Integration with HttpSessionBindingListener to Support Session Scope

Requirements for JspScopeListener

The `JspScopeListener` implementation requires the following:

- The `oracle.jsp.event.JspScopeListener` interface and `JspScopeEvent` class, and the classes of the `oracle.jsp.event.impl` package, all of which are supplied in the `ojsp.jar` file

- A servlet 2.3 or higher environment (such as OC4J)

Runtime and Tag Implementations to Support Page Scope

For OC4J environments, there is support for `page` scope functionality through an Oracle-specific runtime implementation. Enable this by setting the JSP

check_page_scope configuration parameter to `true`. The default is `false`, for performance reasons.

For portability to other environments, there is also an implementation to support `page` scope through a custom tag, `checkPageScope`. Put the appropriate code between the `checkPageScope` start-tag and end-tag. This tag, with no attributes, is defined as follows:

```
<!-- The checkPageScope tag -->
<tag>
   <name>checkPageScope</name>
   <tagclass>oracle.jsp.jml.tagext.CheckPageScopeListenerTag</tagclass>
   <bodycontent>JSP</bodycontent>
   <info>
      To provide the notification logic for any
      JspScopeListener stored in page scope.
      This tag is not needed on OC4J.
   </info>
</tag>
```

Here is an example of its use:

```
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/jml.tld"
           prefix="jml" %>
<jml:checkPageScope>
pagescope.jsp
<jsp:useBean id="tb" class="testpkg.TestData" />
<%
   /* testpkg.TestData implements oracle.jsp.event.JspScopeListener.
      checkPageScope tag will provide the notification logic for any
      JspScopeListener stored in page scope.
      This tag is not needed on OC4J.
   */
   // some more JSP / code here ...
%>
<%= new java.util.Date() %>
</jml:checkPageScope>
```

> **Note:** The `checkPageScope` tag is currently part of the Oracle JML tag library, which is included in the `ojsputil.jar` file and requires the `jml.tld` tag library descriptor file. An appropriate `taglib` directive is shown in the preceding example. See "Overview of the JSP Markup Language (JML) Tag Library" on page 3-1 for related information.

Servlet Filter Implementation to Support Request Scope

Objects of `request` scope are supported through a servlet filter. The filtering applies to any servlets matching a specified URL pattern.

For support of event-handling for request-scope objects, add an entry such as the following to the `web.xml` file for your application, or to `orion-web.xml` or `global-web-application.xml` as appropriate. To ensure proper operation of the `JspScopeListener` functionality, this setting must be *after* any other `filter` settings.

```
<filter>
   <filter-name>Request Filter</filter-name>
   <filter-class>oracle.jsp.event.impl.RequestScopeFilter</filter-class>
```

```
</filter>
<!-- Define filter mappings for the defined filters -->
<filter-mapping>
    <filter-name>Request Filter</filter-name>
    <url-pattern>/jsp/*</url-pattern>
</filter-mapping>
```

> **Note:** In this particular example, `"/jsp/*"` is the URL pattern covered by the filter. Users can choose other patterns instead, such as `"/*.jsp"` or `"/*"`.

Listener Class Implementation to Support Application Scope

Objects with `application` scope are supported through a servlet context listener implementation class, in accordance with the servlet specification.

For support of event-handling for application-scope objects, add an entry such as the following to the `web.xml` file for your application. To ensure proper operation of the `JspScopeListener` functionality, this setting must be *after* any other `listener` settings.

```
<listener>
    <listener-class>oracle.jsp.event.impl.AppScopeListener</listener-class>
</listener>
```

For an application-scope object, in addition to notification upon the conclusion of the application and servlet context, there is notification when an attribute is replaced in the servlet context or removed from the servlet context. For example, the listener `outOfScope()` method of an application-scope object is called in either of the following circumstances, assuming a servlet context object `ctx`:

```
ctx.setAttribute("name", "Smith");
...
ctx.setAttribute("name, "Jones");
```

or:

```
ctx.setAttribute("name", "Smith");
...
ctx.removeAttribute("name");
```

> **Note:** This functionality was not available prior to Oracle9*i*AS Release 2.

Integration with HttpSessionBindingListener to Support Session Scope

For session-scope objects, you can write a class that implements both the `JspScopeListener` interface and the standard `javax.servlet.http.HttpSessionBindingListener` interface. This would give you the flexibility of supporting instances of this class for other scopes as well. If instances would never be used outside of `session` scope, however, there is no need to implement `JspScopeListener`.

In the integration scenario, the `valueUnbound()` method, specified in the `HttpSessionBindingListener` interface, should call the `outOfScope()` method that is specified in the `JspScopeListener` interface.

Following is a basic example:

```
import oracle.jsp.event.impl.*;
import javax.servlet.*;
import javax.servlet.http.*;

class SampleObj implements HttpSessionBindingListener,JspScopeListener
{
   public void valueBound(HttpSessionBindingEvent e)
   {
      System.out.println("The object implements the JspScopeListener also");
   }

   public void valueUnBound(HttpSessionBindingEvent e)
   {
      try
      {
         outOfScope(new JspScopeEvent(null,(Object)e.getSession(),
             e.getName(),javax.servlet.jsp.PageContext.SESSION_SCOPE));
      } catch (Throwable e) {}
   ...........
   }
   public void outOfScope(JspScopeEvent e)
   {...}
}
```

Examples Using JspScopeListener

This section provides two examples of `JspScopeListener` usage: a JSP page and accompanying JavaBean, then a servlet.

Example: JSP Page Using JspScopeListener

This example consists of a JavaBean, `ScopeDispatcher`, that implements the `JspScopeListener` interface, and a JSP page that uses `ScopeDispatcher` instances for request-scope and application-scope functionality.

bookcatalog.jsp   The `bookcatalog.jsp` page allows users to search for a book in the catalog or insert a new book entry. The catalog is kept in a hashtable that is initially read from the local file stream.

At the end of a request, if a new book has been submitted: 1) the book is entered into the application-level `catalog` hashtable; 2) the book count is incremented.

At the end of execution of the application, the `catalog` hashtable is sent back to the local file stream, the number of newly inserted books is shown, and query results are displayed if there was a book search.

```
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<%! static int newbookCount = 0; %>
<%! static Hashtable catalog; %>
<%! boolean bookAdded = false; %>
<html>
<head>
<title> BookStore Price catalog </title>
</head>
<body bgcolor="white">
<font size=5 color="red">
<table color="#FFFFCC" width="100%" border="1" cellspacing="0" cellpadding="0" >
<tr>
```

```
<td>
<form action="bookcatalog.jsp">
<b> BookName </b>
<input type="text" name="bookname">
<input type="submit" value="Get the Price">
</form>
</td>
<td>
<form action="bookcatalog.jsp">
<b>BookName</b>
<input type="text" name="new_book">
<br>
<b>Price</b>
<input type="text" name="price">
<input type="submit" value="Add to Catalog">
</form>
</td>
</tr>
</table>

<%
  String bookname = request.getParameter("bookname");
  catalog = (Hashtable) application.getAttribute("pricelist");
  if (catalog == null)
  {
   try{
    ObjectInputStream oin = new ObjectInputStream
                            (new FileInputStream("bookcatalog.out"));
    Object obj =  oin.readObject();
    catalog = (Hashtable) obj;
    oin.close();
   }
  catch(Exception e) {
   catalog = new Hashtable();}
  application.setAttribute("pricelist",catalog);
 }
  if (bookname != null)
  {
    String price = (String) catalog.get(bookname.trim());
    if (price != null)
    {
     out.println("<h2>Book : " +bookname+ "</h2>");
     out.println("<h2>Price: "+price +"</h2>");
    }
    else
      out.println("<h2> Sorry, the  Book : " + bookname + " is not available in
                     the catalog</h2>");
  }
%>

<%-- declare the event dispatchers --%>
<jsp:useBean id = "requestDispatcher"
             class = "oracle.jsp.sample.event.ScopeDispatcher"
             scope = "request" >
   <jsp:setProperty name = "requestDispatcher" property = "page"
                     value = "<%= this %>" />
   <jsp:setProperty name = "requestDispatcher" property = "methodName"
                     value = "request_OnEnd" />
</jsp:useBean>
```

```
<jsp:useBean id = "appDispatcher"
             class = "oracle.jsp.sample.event.ScopeDispatcher"
             scope = "application" >
   <jsp:setProperty name = "appDispatcher" property = "page"
                     value = "<%= this %>" />
   <jsp:setProperty name = "appDispatcher" property = "methodName"
                     value = "application_OnEnd" />
</jsp:useBean>
<%!
   // request_OnEnd Event Handler
   public void request_OnEnd(HttpServletRequest request) {
      // acquire beans
               String newbook  = request.getParameter("new_book");
               bookAdded = false;
               if ((newbook != null) && (!newbook.equals("")))
               {
                  catalog.put(newbook,request.getParameter("price"));
                  newbookCount++;
                  bookAdded = true;
               }
   }
%>


<%!
  public void application_OnEnd(ServletContext application)
  {
    try
    {
      ObjectOutputStream os = new ObjectOutputStream(
                                  new FileOutputStream("bookcatalog.out"));
      os.writeObject(catalog);
      os.flush();
      os.close();
    }
    catch (Exception e)
    {}
  }
%>


<%
if (bookAdded)
   out.println("<h2> The New book is been added in the catalog </h2>");
%>
<%-- Page implementation goes here --%>
<h2> Total number of books added is <%= newbookCount %></h2>
</font>
</body>
</html>
```

ScopeDispatcher.java

```
package oracle.jsp.sample.event;
import java.lang.reflect.*;
import oracle.jsp.event.*;

public class ScopeDispatcher extends Object implements JspScopeListener {
    private Object page;
    private String methodName;
    private Method method;

    public ScopeDispatcher() {
```

```
        }

        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.APPLICATION_SCOPE)
                 && method != null) {
                try {
                    Object args[] = {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")};

            try {
                return c.getDeclaredMethod(m, pTypes);
            } catch (NoSuchMethodException nsme) {
                // fall through and try the request signature
            }

            pTypes[0] = Class.forName("javax.servlet.http.HttpServletRequest");
            return c.getDeclaredMethod(m, pTypes);
        }
    }
```

Example: Servlet Using JspScopeListener

This section contains a sample servlet that uses `JspScopeListener` functionality for a request-scope object. The nested class `DBScopeObj` implements the `JspScopeListener` interface.

```java
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import javax.servlet.*;
import javax.servlet.http.*;
import oracle.jsp.event.*;
import oracle.jsp.event.impl.*;

public class RequestScopeServlet extends HttpServlet {

    PrintWriter out;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        out         = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title> RequestScopeServlet! </title>");
        out.println("</head>");
        response.setContentType("text/html");
        DBScopeObj aobj = new DBScopeObj();
        request.setAttribute("dbcon",aobj);
        request.setAttribute("name","scott");
        request.setAttribute("company","oracle");
        request.setAttribute("city","sanmateo");
        Enumeration en = request.getAttributeNames();
        out.println("<BR> Request Attributes : <BR> <BR>");
        while (en.hasMoreElements()) {
            String key = (String)en.nextElement();
            Object value = request.getAttribute(key);
            out.println(key + "   : " + value+"<BR>");
         }
        out.println("</body>");
        out.println("</html>");
    }

    class DBScopeObj implements JspScopeListener
    {
        public void initDBConnection()
        {
            //  can create a minimum number of predefined
            // DBConnections
        }

        DBScopeObj()
        {
            // if DBconnection is available in the connection
            // pool then pickup from the pool and give the handle.
        }

        public void outOfScope(JspScopeEvent e)
        {
         ServletContext ctx = e.getApplication();
```

```
                           out.println
                              ("<BR>*******************************************************");
                           out.println("<BR> JspScopeEvent <BR>");
                           out.println("<BLINK>");
                           out.println
                              ("<BR> In outOfScope method for the Request Attribute <BR>");
                           out.println("Name =  " +e.getName() + "<BR>");
                           out.println("</BLINK>");
                           out.println
                              ("*******************************************************<BR>");
                           // logging in the context also

                           ctx.log("*****************************************************");
                           ctx.log(" JspScopeEvent ");
                           ctx.log(" In outOfScope method for the Request Attribute ");
                           ctx.log("Name =  " +e.getName());
                           ctx.log("*****************************************************");
                           returnDBConnection();
                        }

                        public void returnDBConnection()

                        {
                            //Can return the handle to the connection pool
                        }
                    }
                }
```

EJB Tags

OC4J provides a custom tag library to simplify the use of Enterprise JavaBeans in JSP pages. The library includes tags to create a home instance, create an EJB instance, and iterate through a collection of EJBs.

The functionality of the OC4J EJB tags follows the J2EE specification. The tags allow you to instantiate EJBs by name, using configuration information in the web.xml file. One of the tags is a useBean tag, with functionality similar to that of the jsp:useBean tag for invoking a regular JavaBean.

The following sections document the tags, concluding with examples:

- EJB Tag Configuration
- EJB Tag Descriptions
- EJB Tag Examples

EJB Tag Configuration

Use an <ejb-ref> element in your application web.xml file for each EJB you will use, as in the following example:

```
<ejb-ref>
   <ejb-ref-name>ejb/DemoSession</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>ejbdemo.DemoSessionHome</home>
   <remote>ejbdemo.DemoSession</remote>
</ejb-ref>
```

The <ejb-ref> element and its subelements, or <ejb-local-ref> to use local interfaces, are used according to the servlet specification. Briefly, this is as follows:

- The `<ejb-ref-name>` subelement specifies a reference name that can be used by other components of a J2EE application to access this component. For example, this name could be used in a location value.

- The `<ejb-ref-type>` subelement specifies the category of EJB.

- The `<home>` subelement specifies the package and type of the EJB home interface. Alternatively, use the `<local-home>` subelement for EJB local interfaces.

- The `<remote>` subelement specifies the package and type of the EJB remote interface. Alternatively, use the `<local>` subelement for EJB local interfaces.

These values are reflected in attribute values of the EJB tags.

See the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for additional information about EJB development and configuration.

EJB Tag Descriptions

This section provides syntax and attribute descriptions for the OC4J EJB tags. Be aware of the following requirements:

- Verify that the file `ojsputil.jar` is installed and in your classpath. This file is provided with OC4J, in the "well-known" tag library directory.

- The tag library descriptor, `ejbtaglib.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `ejbtaglib.tld` is the following:

```
http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/ejbtaglib.tld
```

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

> **Notes:**
>
> - The prefix "ejb:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.
>
> - See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

The following sections provide information about the EJB tags:

- EJB useHome Tag

- EJB useBean Tag

- EJB createBean Tag

- EJB iterate Tag

When first creating an EJB instance, you will have to use a `useHome` tag to create a home interface instance. Then use the following as appropriate:

- To create a single EJB instance: a `useBean` tag, and either the `useBean` tag `value` attribute or a nested `createBean` tag

- To create a collection of EJB instances and iterate through them (more typical for entity beans): an `iterate` tag

After an EJB instance is created, it is placed in the appropriate scope object. You will need only a `useBean` tag to access it subsequently.

EJB useHome Tag

The `useHome` tag looks up the home interface for the EJB and creates an instance of it.

Syntax

```
<ejb:useHome id = "home_instance_name"
            type = "home_interface_type"
            location = "home_lookup_name"
          [ local = "true" | "false" ] />
```

This tag uses no body.

Attributes

- `id` (required): Specify a name for the home interface instance. This can be for either a local or remote home interface, depending on the setting of the `local` attribute. The instance is accessible from the start-tag to the end of the page.

- `type` (required): This is for the name (Java type) of the home interface.

- `location` (required): This is a JNDI name used to look up the home interface of the desired EJB within the application.

- `local`: Set this to "true" to use the local home interface. The default value is "false", to use the remote home interface. If `local="true"` for the `useHome` tag, this must also be the case for the `useBean` tag.

Example

```
<ejb:useHome id="aomHome" type="com.acme.atm.ejb.AccountOwnerManagerHome"
            location="java:comp/env/ejb/accountOwnerManager" />
```

EJB useBean Tag

Use the EJB `useBean` tag for instantiating and using the EJB. The `id`, `type`, and `scope` attributes are used as in a standard `jsp:useBean` tag that instantiates a regular JavaBean.

You can use one of two mechanisms when you first instantiate the EJB:

- The `value` attribute

or:

- A nested EJB `createBean` tag

When using a nested `createBean` tag, the EJB instance is implicitly returned into the `value` attribute of the parent `useBean` tag. Once the EJB is instantiated, `value` attributes and nested `createBean` tags are unnecessary for subsequent `useBean` tags using the same EJB instance.

> **Note:** See "EJB iterate Tag" on page 9-13 for how to use a collection of EJB instances.

Syntax

```
<ejb:useBean id = "EJB_instance_name"
            type = "EJB_class_name"
```

```
        [ value = "<%=Object%>" ]
        [ scope = "page" | "request" | "session" | "application" ]
        [ local = "true" | "false" ] >
```

... nested createBean tag for first instantiation, if no value attribute ...

```
</ejb:useBean>
```

Attributes

- id (required): Specify an instance name for the EJB.

- type (required): Specify the class name for the EJB.

- value: When first instantiating the EJB, if you do not use a nested createBean tag, you can use the value attribute to return an EJBObject instance to narrow. This is a mechanism for instantiating the EJB.

- scope: Specify the scope of the EJB instance. The default scope setting is "page".

- local: Set this to "true" to use the local home interface. The default value is "false", to use the remote home interface. If local="true" for the useBean tag, this must also be the case for the useHome tag.

> **Note:** You cannot use local="true" if scope="session" in a distributable application.

Example This example shows the use of an EJB that has already been instantiated.

```
<ejb:useBean id="bean" type="com.acme.MyBean" scope="session" />
```

EJB createBean Tag

For first instantiating an EJB, if you do not use the value attribute of the EJB useBean tag, you must nest an EJB createBean tag within the useBean tag to do the work of creating the EJB instance. This will be an EJBObject instance. The instance is implicitly returned into the value attribute of the parent useBean tag.

Syntax

```
<ejb:createBean instance = "<%=Object%>" />
```

This tag uses no body.

Attributes

- instance (required): This is to return the EJB, a created EJBObject instance.

Example In this createBean tag, the create() method of the EJB home interface instance creates an instance of the EJB.

```
<ejb:useBean id="bean" type="com.acme.MyBean" scope="session">
    <ejb:createBean instance="<%=home.create()%>" />
</ejb:useBean>
```

EJB iterate Tag

Use this tag to iterate through a collection of EJB instances. This is more typical for entity beans, because standard finder methods for entity beans return collections.

In the start-tag, obtain the collection through finder results from the home interface. In the tag body, iterate through the collection as appropriate.

> **Note:** See "EJB useBean Tag" on page 9-12 for how to use a single EJB instance.

Syntax

```
<ejb:iterate id = "EJB_instance_name"
             type = "EJB_class_name"
             collection = "<%=Collection%>"
          [ max = "<%=Integer%>" ] >

... body ...

</ejb:iterate>
```

The body is evaluated once for each EJB in the collection.

Attributes

- `id` (required): This is an iterator variable, the EJB instance name for each iteration.
- `type` (required): This is the EJB class name.
- `collection` (required): This is to return the EJB collection.
- `max`: Optionally specify a maximum number of beans to iterate through.

Example

```
<ejb:iterate id="account" type="com.acme.atm.ejb.Account"
             collection="<%=accountManager.getOwnerAccounts()%>"
             max="100">
     <jsp:getProperty name="account" property="id" />
</ejb:iterate>
```

EJB Tag Examples

This section provides examples of EJB tag usage, one using a session bean and one using an entity bean.

EJB Tag Session Bean Example

This example relies on the following configuration in the application `web.xml` file:

```
<ejb-ref>
   <ejb-ref-name>ejb/DemoSession</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>ejbdemo.DemoSessionHome</home>
   <remote>ejbdemo.DemoSession</remote>
</ejb-ref>
```

Here is the sample code:

```
<%@ page import="ejbdemo.*" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/ejbtaglib.tld"
           prefix="ejb" %>
<html>
<head> <title>Use EJB from JSP</title> </head>
<body>
```

```
<ejb:useHome id="home" type="ejbdemo.DemoSessionHome"
            location="java:comp/env/ejb/DemoSession" />
<ejb:useBean id="demo" type="ejbdemo.DemoSession" scope="session" >
    <ejb:createBean instance="<%=home.create()%>" />
</ejb:useBean>
<heading2>      Enterprise Java Bean:  </heading2>
 <p><b> My name is "<%=demo.getName()%>". </b></p>
</body>
</html>
```

This sample code accomplishes the following:

- It creates the `home` instance of the EJB home interface. Note that the `type` value of the `useHome` tag matches the value of the `<home>` subelement of the `<ejb-ref>` element in the `web.xml` file. Also, the `location` value of `useHome` reflects the value of the `<ejb-ref-name>` subelement of the `<ejb-ref>` element.

- It uses the `home.create()` method to create the `demo` instance of the EJB. Note that the `type` value of the `useBean` tag matches the value of the `<remote>` subelement of the `<ejb-ref>` element in the `web.xml` file.

- It uses the `demo.getName()` method to print a user name.

EJB Tag Entity Bean Example

This example relies on the following configuration in the application `web.xml` file:

```
<ejb-ref>
   <ejb-ref-name>ejb/DemoEntity</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <home>ejbdemo.DemoEntityHome</home>
   <remote>ejbdemo.DemoEntity</remote>
</ejb-ref>
```

Here is the sample code:

```
<%@ page import="ejbdemo.*" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/ejbtaglib.tld"
         prefix="ejb" %>
<html>
<head> <title>Iterate over EJBs from JSP</title> </head>
<body>

<ejb:useHome id="home" type="ejbdemo.DemoEntityHome"
            location="java:comp/env/ejb/DemoEntity" />
<% int i=0; %>
<ejb:iterate id="demo" type="ejbdemo.DemoEntity"
            collection="<%=home.findAll()%>" max="3" >
<li> <heading2> Bean #<%=++i%>:  </heading2>
 <b> My name is "<%=demo.getName()+"_"+ demo.getId()%>". </b> </li>
</ejb:iterate>
</body>
</html>
```

This sample code accomplishes the following:

- It creates the `home` instance of the EJB home interface. Note that the `type` value of the `useHome` tag matches the value of the `<home>` subelement of the `<ejb-ref>` element in the `web.xml` file. Also, the `location` value of `useHome` reflects the value of the `<ejb-ref-name>` subelement of the `<ejb-ref>` element.

- It uses the `home.findAll()` method to return a collection of EJBs. Note that the `type` value in the `iterate` tag matches the value of the `<remote>` subelement of the `<ejb-ref>` element in the `web.xml` file.

- It iterates through the collection, always using `demo` for the current instance, and using the `demo.getName()` and `demo.getId()` methods to output information from each EJB.

General Utility Tags

OC4J provides miscellaneous utility tags to perform a number of operations. The following sections contain details about the tags:

- Display Tags

- Miscellaneous Utility Tags

Note the following requirements for the utility tags:

- Verify that the file `ojsputil.jar` is installed and in your classpath. This file is provided with OC4J, in the "well-known" tag library directory.

- The tag library descriptor, `utiltaglib.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `utiltaglib.tld` is the following:

  ```
  http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/utiltaglib.tld
  ```

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

> **Notes:**
>
> - The prefix "util:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.
>
> - See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

Display Tags

The following sections document syntax and attributes of the display tags:

- Utility displayCurrency Tag

- Utility displayDate Tag

- Utility displayNumber Tag

Utility displayCurrency Tag

This tag displays a specified amount of money, formatted as currency appropriate for the locale. If no locale is specified, then the `request` object will be searched for a locale. If none is found there, the system default locale is used.

Syntax

```
<util:displayCurrency amount = "<%=Double%>"
```

```
                              [ locale = "<%=Locale%>" ] />
```

This tag uses no body.

Attributes

- `amount` (required): Specify the amount to format.

- `locale`: Optionally specify a locale, as a `java.util.Locale` instance.

Example

```
<util:displayCurrency amount="<%=account.getBalance()%>"
                      locale="<%=account.getLocale()%>" />
```

Utility displayDate Tag

This tag displays a specified date, formatted appropriately for the locale. If no locale is specified, the system default locale is used.

Syntax

```
<util:displayDate date = "<%=Date%>"
                 [ locale = "<%=Locale%>" ] />
```

This tag uses no body.

Attributes

- `date` (required): Specify the date to format, as a `java.util.Date` instance.

- `locale`: Optionally specify a locale, as a `java.util.Locale` instance.

Example

```
<util:displayDate date="<%=account.getDate()%>"
                  locale="<%=account.getLocale()%>" />
```

Utility displayNumber Tag

This displays the specified number appropriately for the locale and optionally in the specified format. If no locale is specified, the system default locale is used.

Syntax

```
<util:displayNumber number = "<%=Double%>"
                   [ locale = "<%=Locale%>" ]
                   [ format = "<%=Format%>" ] />
```

This tag uses no body.

Attributes

- `number` (required): Specify the number to format.

- `locale`: Optionally specify the locale, as a `java.util.Locale` instance.

- `format`: Optionally specify a format, as a `java.text.Format` instance.

Example

```
<util:displayNumber number="<%=shoe.getSize()%>" />
```

Miscellaneous Utility Tags

The following sections document syntax and attributes of the general utility tags:

- Utility ifInRole Tag
- Utility lastModified Tag

Utility iterate Tag

Use this tag to iterate through a collection. Obtain the collection in the start-tag and iterate through the collection in the body.

Syntax

```
<util:iterate id = "instance_name"
              type = "class_name"
              collection = "<%=Collection%>"
            [ max = "<%=Integer%>" ] >

... body ...

</util:iterate>
```

The body is evaluated once for each element in the collection.

Attributes

- `id` (required): This is an iterator variable, the instance name for each iteration.
- `type` (required): This is the class name; the collection is a set of instances of this type.
- `collection` (required): This is the collection itself.
- `max`: Optionally specify a maximum number of elements to iterate through.

Example

```
<util:iterate id="contact" type="com.acme.connections.Contact"
              collection="<%=company.getContacts()%>" >
    <jsp:getProperty name="contact" property="name"/>
</util:iterate>
```

Utility ifInRole Tag

Use this tag to evaluate the tag body and include it in the body of the JSP page, depending on whether the user is in the specified application role. The tag handler executes the `isUserInRole()` method of the `request` object.

The concept of "role" is according to the servlet specification. Roles are defined in `<role>` elements in the application `web.xml` file.

Syntax

```
<util:ifInRole role = "<%=String%>"
            [ include = "true" | "false" ] >

   ... body to include ...

</util:ifInRole>
```

Attributes

- `role` (required): Specify the role to check, to see if the user included in this role.

- `include`: Use a `"true"` setting (the default) to include the body only if the user is in the role. Use a `"false"` setting to include the body only if the user is *not* in the role.

Example

```
<util:ifInRole role="users" include="true">
     Logged in as <%=request.getRemoteUser()%><br>
     <form action="logout.jsp">
     <input type="submit" value="Log out"><br>
     </form>
</util:ifInRole>
<util:ifInRole role="users" include="false">
     <form method="POST">
     Username: <input name="j_username" type="text"><br>
     Password: <input name="j_password" type="password"><br>
     <input type="submit" value="Log in">
     </form>
</util:ifInRole>
```

Utility lastModified Tag

This tag displays the date of the last modification of the current file, appropriately formatted for the locale. If no locale is specified, then the `request` object will be searched for a locale. If none is found there, the system default locale is used.

Syntax

```
<util:lastModified
             [ locale = "<%=Locale%>" ] />
```

This tag uses no body.

Attributes

- `locale`: Optionally specify the locale, as a `java.util.Locale` instance.

Example

```
<util:lastModified />
```

10

Web Services Tags

Oracle furnishes a tag library with OC4J that enables developers to create JSP pages for use as client programs for Web services. This chapter describes the tag library and consists of the following sections:

- Overview of Web Services

- OC4J Web Services Tags

This chapter is written with the assumption that you are already familiar with Web services, Simple Object Access Protocol (SOAP), and the Web Services Description Language (WSDL); however, some overview is provided here. There are also references to additional documents, including related specifications from the World Wide Web Consortium (W3C).

The OC4J Web services tag library is based on Oracle Application Server Web Services. See the *Oracle Application Server Web Services Developer's Guide* for information.

Overview of Web Services

The following sections provide a quick overview of Web services concepts:

- General Web Services Overview

- Overview of SOAP and Related Features

- Overview of Web Services Description Language Key Elements

- Overview of Web Service Messages and XML Schema Definitions

- Web Service Example

General Web Services Overview

Web services are sets of procedures, or actions, that can be invoked by a client over the Internet, regardless of the computing platform. Web services consist of loosely coupled components over a distributed computing environment following a widely adopted set of standards such as SOAP, WSDL, and UDDI (all discussed later in this chapter). As an example, there might be a "World Cup Soccer" service that consists of actions to get scores, schedules, and standings.

A Web service must have the following features:

- It must be able to describe itself, such as its functionality and its input and output attributes. A Web service describes itself through an XML-style WSDL document. See "Overview of Web Services Description Language Key Elements" on page 10-3.

- It must make itself generally available so that client applications can access it. The standard way to do this is to be listed in a Universal Description, Discovery, and Integration (UDDI) directory. Public UDDI directories are available to aggregate groups of businesses or users (or perhaps to anyone on the Internet), while private UDDI directories are available only within a particular business or group.

- A client application must be able to invoke it through a standard protocol once the application has found and examined it. A leading protocol for Web services is Simple Object Access Protocol (SOAP). With SOAP, the Web service is behind a SOAP server at the server end and the client application goes through a SOAP server at the client end. Data exchanges are "SOAP-enveloped" and can gain access through firewalls. This SOAP exchange is conceptually similar to a Remote Method Invocation (RMI) exchange, except that RMI exchanges cannot go through firewalls. See "Overview of SOAP and Related Features" on page 10-2 for a brief overview of SOAP.

- Once invoked, it must return a response to provide requested results to the client application. This is performed through the same standard protocol, such as SOAP.

For more information about Web services, particularly OracleAS Web Services, you can refer to the *Oracle Application Server Web Services Developer's Guide*.

For related specifications, refer to the following Web sites for the W3C SOAP specification, W3C WSDL specification, and UDDI specification, respectively:

http://www.w3.org/TR/SOAP

http://www.w3.org/TR/wsdl

http://www.uddi.org/specification.html

Overview of SOAP and Related Features

This section offers a brief overview of SOAP. See the W3C *Simple Object Access Protocol (SOAP) 1.1* specification for details.

SOAP is a lightweight, XML-based protocol for exchanging typed and structured data over the Internet or other distributed environments. Among other features, SOAP supports *remote procedure call* (RPC) and message-oriented data exchanges.

In a message-oriented implementation, data is exchanged through a modular packaging and encoding model. A *message* is a WSDL component that specifies input data parts and output data parts associated with an operation. See "Overview of Web Service Messages and XML Schema Definitions" on page 10-4 for more information.

RPC is an alternative to sockets, with the communication interface being at the level of procedure calls. It is as though you are calling a local procedure, but arguments of the call are actually packaged and sent to a remote target. The RPC mechanism uses a request/response methodology, where an end-point receives a procedure-oriented message and sends back a corresponding response.

Using SOAP with RPC is independent of the protocol binding. Where HTTP is the protocol binding, HTTP requests correspond to RPC calls, and HTTP responses correspond to RPC responses.

Key aspects of SOAP include the following.

- SOAP *envelope* construct: The envelope encloses a SOAP header and SOAP body and indicates what is in a message, whether it is required, and who should process it.

- SOAP encoding rules: Encoding rules define serialization mechanisms for the exchange of instances of the datatypes used in an application.

- SOAP RPC representation: The RPC representation specifies a convention for representing RPC calls and responses.

Overview of Web Services Description Language Key Elements

A Web service is described using the XML-based Web Services Description Language in a WSDL (`.wsdl`) document.

Following are some key WSDL terms.

- *Operation*: An operation is a particular action performed by a service, such as any of the "get scores", "get schedules", and "get standings" examples for a World Cup service.

- *Message*: A message is an abstract definition that specifies the data that is being input and output for an operation.

- *Port type*: A port type is an abstract definition of the operations supported by a service.

- *Binding*: A binding is a protocol and data format specification for one or more operations supported by a service. A binding mechanism maps the generic or abstract definition of a Web service to a concrete implementation, including data encoding, message protocol, and communication protocol.

- *Port*: A port is a single end-point, a combination of a binding and a network address. Essentially, a port is the concrete manifestation of the capabilities described by a port type. In a SOAP-based implementation, a port is a SOAP location.

  To be more precise than previously, a Web service is really a collection of related ports, or end-points, not just a collection of abstract actions or operations.

The WSDL specification outlines the general structure of a WSDL document, which includes the following key elements. Refer to the W3C *Web Services Description Language (WSDL) 1.1* specification for complete information.

- A `<types>` element , through one or more `<schema>` subelements, contains descriptions of the data that is exchanged in messages used by the operations of the service.

- A `<message>` element provides an abstract definition of data being sent as input or output for an operation.

- A `<portType>` element, through one or more `<operation>` subelements, contains abstract definitions of the operations of the Web service. An `<operation>` element specifies the message that is used for input and the message that is used for output for the operation.

- A `<binding>` element, also through `<operation>` subelements, binds each operation to the particular protocol and data formats to be used.

- A `<service>` element defines the ports, or end-points, of the Web service. Within the `<service>` element is one or more `<port>` subelements, where each `<port>` element ties a binding to an address to define the end-point.

Overview of Web Service Messages and XML Schema Definitions

Messages define parameters used by the operations, or methods, of a Web service. A message is a typed definition of the data being communicated, consisting of one or more parts. Each part corresponds to a logical entity, such as a "Purchase Order" part and an "Invoice" part. For each part, there are type specifications for the associated data items.

In a SOAP-based implementation, such as for OracleAS Web Services, the datatypes used by a message are defined through the XML Schema Definition (XSD) language, which supports predefined simple types as well as user-defined complex types.

With an implementation that uses XSD, the syntax for defining a message is as follows:

```
<message name="nmtoken">
    <part name="nmtoken" [type="qname"] [element="qname"] />
</message>
```

In this syntax, the `element` attribute refers to where an XSD complex type is defined using XSD syntax, the `type` attribute indicates an XSD simple type, `"nmtoken"` indicates a standard XML name token, and `"qname"` indicates a standard XML qualified name. There can be zero or more messages, and zero or more parts for each message.

For a SOAP encoding style of `encoded`, only simple types are allowed, so the `element` attribute is not used. For an encoding style of `literal`, you can have simple types or complex types, so a `<part>` element can use either the `type` attribute or the `element` attribute, but not both.

Here is an example of a message definition, from "Example: WSDL Definition", which follows shortly:

```
<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>
```

`GetLastTradePriceInput` is the name of the message, which is an input message (as the name implies). In this case, the `element` attribute refers to a namespace where a complex type, `TradePriceRequest`, is defined. Here is an example of such a definition (also part of "Example: WSDL Definition" below):

```
<element name="TradePriceRequest">
   <complexType>
      <all>
          <element name="tickerSymbol" type="string"/>
          <element name="companyName" type="string"/>
      </all>
   </complexType>
</element>
```

An XML schema primer is available from W3C at the following location:

http://www.w3.org/TR/xmlschema-0/

Web Service Example

This example shows the WSDL definition of a Web service, illustrating the input and output messages embedded in an HTTP request and HTTP response, respectively.

Example: WSDL Definition

The W3C *Web Services Description Language (WSDL) 1.1* specification provides the following example of a WSDL document that defines a stock quote service taking a ticker symbol as input and returns the current stock price as output. Note this uses a SOAP encoding style of `literal`, so complex types are allowed (and used).

```
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
        <schema targetNamespace="http://example.com/stockquote.xsd"
                xmlns="http://www.w3.org/2000/10/XMLSchema">
            <element name="TradePriceRequest">
                <complexType>
                    <all>
                        <element name="tickerSymbol" type="string"/>
                    </all>
                </complexType>
            </element>
            <element name="TradePrice">
                <complexType>
                    <all>
                        <element name="price" type="float"/>
                    </all>
                </complexType>
            </element>
        </schema>
    </types>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>

    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
            <input message="tns:GetLastTradePriceInput"/>
            <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
            <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
                <input>
                    <soap:body use="literal"/>
                </input>
                <output>
                    <soap:body use="literal"/>
```

```
                </output>
        </operation>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
            <port name="StockQuotePort" binding="tns:StockQuoteBinding">
                <soap:address location="http://example.com/stockquote"/>
            </port>
    </service>

</definitions>
```

This WSDL definition first specifies the GetLastTradePriceInput and
GetLastTradePriceOutput input and output messages, then ties them to the
operation GetLastTradePrice, then defines a binding and a port for that operation.

Notes:

- This example has all aspects of the Web service definition,
  including the XML schema definitions for data exchanges, in
  the same document. Alternatively, stockquote.xsd, for
  example, could be a separate XSD document instead of a
  namespace within this document. The W3C WSDL
  specification illustrates this. Be aware, however, that the OC4J
  Web services tag library does *not* support WSDL documents
  that use <import> elements to import other WSDL
  documents.

- The example uses a document-style binding. The OC4J 10.1.2
  implementation of the Web services tag library supports
  RPC-style and document-style bindings. In the document-style
  case, the output response object is an XML document of type
  XMLElement. In the RPC-style case, the output object might be
  of any type.

Example: SOAP Messages Embedded in HTTP Request and Response

Corresponding to the Web service defined in the preceding example, this section
shows what the messages would look like, with the soap-enveloped input message
embedded in an HTTP request, and the soap-enveloped output message embedded in
an HTTP response. These examples are also from the W3C *Web Services Description
Language (WSDL) 1.1* specification.

Here is a request:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "SOAP_URI"

    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
        <soapenv:Body>
            <m:GetLastTradePrice xmlns:m="xmlns_URI">
                <m:tickerSymbol>DIS</m:tickerSymbol>
            </m:GetLastTradePrice>
        </soapenv:Body>
    </soapenv:Envelope>
```

In this example, *xmlns_URI* is a URI value used to identify the namespace where the `GetLastTradePrice` operation and its messages are defined, such as the WSDL document in the preceding "Example: WSDL Definition". This is also where `tickerSymbol` is defined. The request is for a stock quote for Walt Disney Company. *SOAP_URI* is the URI for the SOAP action HTTP header for the HTTP binding of SOAP.

And here is the response:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <soapenv:Body>
         <m:GetLastTradePriceResponse xmlns:m="Some_URI">
            <m:price>34.5</m:price>
         </m:GetLastTradePriceResponse>
      </soapenv:Body>
   </soapenv:Envelope>
```

By convention, the response for an operation *Xxxx* is called *Xxxx*`Response`. *Some_URI* is a URI value used to identify the namespace where the `GetLastTradePriceResponse` operation is defined.

OC4J Web Services Tags

The following sections provide an overview and details of the Web services tag library, as well as an overview of OracleAS Web Services, upon which the tag library implementation is based.

- Overview of OracleAS Web Services and the Tag Library Implementation

- Overview of Functionality of Web Services Tags

- Web Services Tag Descriptions

- Web Services Tag Examples

Overview of OracleAS Web Services and the Tag Library Implementation

The Web services tag library provided with OC4J enables developers to conveniently create JSP pages for Web service client applications. The implementation uses a SOAP-based mechanism. A client application would access the WSDL document, then use the WSDL information to access the operations of a Web service.

The tag library also uses the Oracle implementation of the dynamic invocation API, described in the *Oracle Application Server Web Services Developer's Guide.* When a client application acquires a WSDL document at runtime, the dynamic invocation API is the vehicle for invoking any SOAP operation described in the WSDL document. The tag handler uses the API when sending a SOAP request that invokes a Web service and when handling the SOAP response.

The Oracle dynamic invocation API consists of classes and interfaces in the `oracle.j2ee.ws.client` and `oracle.j2ee.ws.client.wsdl` packages.

The `oracle.j2ee.ws.client` package includes the following.

- `WebServiceProxyFactory`: Given a WSDL document (through a Java input stream that contains the document or through the URL of the document), a

`WebServiceProxyFactory` instance can use the name of a service and the name of one of its ports, as specified in the WSDL document, to create a `WebServiceProxy` instance (an instance of a class that implements the `WebServiceProxy` interface).

- `WebServiceProxy`: Use this interface in representing a service defined in a WSDL document. Each `WebServiceProxy` instance is based on the location of the WSDL document and, optionally, on additional qualifiers that identify which service and port should be used. A `WebServiceProxy` class exposes methods to determine the WSDL port type, including the syntax and signatures of all operations exposed by the WSDL document, and to invoke the defined operations.

- `WebServiceMethod`: Use this interface in invoking a Web service method, or operation.

The `oracle.j2ee.ws.client.wsdl` package includes the following.

- `Operation`: This interface represents a WSDL operation.

- `Message`: This interface represents a message used in the input or output of an operation.

- `Part`: This interface represents a message part.

- `Input`: This interface represents an input message.

- `Output`: This interface represents an output message.

> **Note:** The dynamic invocation API is packaged in `dsv2.jar` in the *ORACLE_HOME*/lib directory. Also note that the SOAP implementation requires `soap.jar` in the *ORACLE_HOME*/soap directory.

Overview of Functionality of Web Services Tags

This section provides an overview of the OC4J Web services tag library and its functionality. The tag library includes support for the following:

- Binding to a Web service

- Using a Web service operation through SOAP requests and SOAP responses

- Defining input and output message parts

- Mapping SOAP/XML datatypes to Java types

- Setting custom properties for use by the client application

The tag library supports invoking operations defined in WSDL documents that use the W3C XML schema version whose namespace is the following:

http://www.w3.org/2001/XMLSchema

The Web services tag library includes the `webservice` tag, optionally with nested `map` and `property` tags, and the `invoke` tag, optionally with nested `part` tags. They are used as follows.

- `webservice`: Use this tag to create a Web service proxy. The tag requires the URL of a WSDL document and then uses one of the following combinations:

  - A binding and SOAP location, useful for a WSDL document identified in a UDDI registry

– A service name and port, provided through tag attributes or through the first service and its first port from the WSDL document

■ `map`: The Web service proxy uses this tag, if specified, to add an entry to the SOAP mapping registry, which is a registry that maps local SOAP/XML types to Java types. Any number of `map` tags can be nested within a `webservice` tag, one tag for each desired type mapping.

■ `property`: Optionally, use this tag to define any of several supported custom properties for use by the Web service client application. Each `property` tag must be nested within the `webservice` tag; the property will have the same scope as the parent Web service.

■ `invoke`: Use this tag to invoke an operation of the Web service. An `invoke` tag accesses a Web service proxy either by being nested within a `webservice` tag, or through a scripting variable.

■ `part`: If an operation has input message parts, use `part` tags, nested within an `invoke` tag, to define the message parts. Use one `part` tag for each part.

Notes:

■ The tag library does not support the use of `<import>` elements within WSDL documents to import other WSDL documents.

■ Custom bindings, including custom HTTP bindings or custom MIME bindings, are not supported.

Because the OC4J Web services tag library implementation is based on the OracleAS Web Services implementation, any additional limitations of OracleAS Web Services also apply to the tag library.

Web Services Tag Descriptions

The following sections supply detailed descriptions of the OC4J Web services tags, a standards-compliant JavaServer Pages tag library implementation, including syntax documentation:

■ Web Services webservice Tag

■ Web Services map Tag

■ Web Services property Tag

■ Web Services invoke Tag

■ Web Services part Tag

Note the following requirements for the Web services tag library:

■ The Web services tag library is included in the `ojsputil.jar` file. This file is provided with OC4J, in the "well-known" tag library directory. Verify that this file is installed and in your classpath.

■ The tag library descriptor, `wstaglib.tld`, must be available to the application, and any JSP page using the library must have an appropriate `taglib` directive. In an Oracle Application Server installation, the TLD is in `ojsputil.jar`. The `uri` value for `wstaglib.tld` is the following:

```
http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/wstaglib.tld
```

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for information about `taglib` directives, the well-known tag library directory, TLD files, and the meaning of `uri` values.

For an example that uses the tags described in this section, see "Web Services Tag Examples" on page 10-15.

Notes:

- The prefix "ws:" is used in the tag syntax here. This is by convention, but is not required. You can specify any desired prefix in the `taglib` directive.

- See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

Web Services webservice Tag

Use this tag to create a Web service proxy, an instance of a class that implements the `oracle.j2ee.ws.client.WebServiceProxy` interface. The tag requires the URL of a WSDL document and uses a binding and SOAP location or a service name and port, as follows:

1. First, if tag attributes provide a binding and SOAP location, the tag handler uses them in creating the proxy. Tag attributes for service name and port are ignored in this case.

2. If no binding and SOAP location are provided, the tag handler uses a service name and port, as follows:

   a. If a service name and port are provided through tag attributes, then the tag handler uses them in creating the proxy.

   b. If no service name and port are provided, the tag handler uses the first service in the WSDL document and the first port listed for that service.

Using a binding and SOAP location is particularly useful for a Web service whose WSDL document is accessed through a UDDI registry. In that case, the binding and location can be determined through UDDI queries and supplied to the tag through request-time expressions.

After the Web service proxy is created, it will use any nested `map` tags to add entries to the SOAP mapping registry. See the next section, "Web Services map Tag".

Syntax

```
<ws:webservice wsdlUrl = "WSDL_URL_of_Web_service"
            [ id = "variable_name_for_Web_service_proxy" ]
            [ scope = "page" | "request" | "session" | "application" ]
            [ binding = "SOAP_binding_information" ]
            [ soapLocation = "SOAP_endpoint_URL" ]
            [ service = "service_name_in_WSDL" ]
            [ port = "port_name_for_service" ] >

...body / nested tags...

</ws:webservice>
```

Note: The `scope` attribute *cannot* take request-time expressions.

Attributes

- `wsdlUrl` (required): Use this attribute to specify a URL where the WSDL for the desired Web service can be accessed.

- `id`: If the Web service is to be accessed by an `invoke` tag that is *not* nested within the `webservice` tag, use the `id` attribute to specify the name for a `WebServiceProxy` scripting variable so that the variable can be referenced by the `invoke` tag. The specified name must be a valid Java identifier. When you use the `id` attribute, the specified variable will be declared automatically with scope `AT_END` (available from the `webservice` end-tag to the end of the JSP page).

- `scope`: Optionally, specify the scope of the `webservice` tag. The default setting is `"page"`.

- `binding`: In scenario #1 above, use the `binding` attribute to specify the SOAP binding information for a SOAP location (end-point URL) that you specify through the `soapLocation` attribute. You must use these attributes together. The binding information is as defined in the WSDL document, specifying concrete protocol and data format specifications for the operations and messages defined by a particular port type.

- `soapLocation`: In scenario #1 above, use `soapLocation` to specify a SOAP location (end-point URL) as defined in the WSDL document, for which the binding information specified through the `binding` attribute applies. You must use these attributes together.

- `service`: In scenario #2a above, use the `service` attribute to specify the name of a service defined in the WSDL document. You must use this attribute with the `port` attribute, but both are ignored if you use `binding` and `soapLocation`.

- `port`: In scenario #2a above, use the `port` attribute to specify a port for the service that is specified through the `service` attribute. You must use these attributes together. The Web service proxy will use the specified port. The port address will be as specified in the corresponding `<service>` element in the WSDL document. The `service` and `port` attributes are ignored if you use `binding` and `soapLocation`.

Web Services map Tag

For interoperability, a mapping mechanism is necessary to map WSDL-defined SOAP/XML datatypes to the Java types used in JSP pages of a Java client application. This is possible through the OracleAS Web Services SOAP mapping registry.

You can have any number of `map` tags nested within a `webservice` tag, to have the Web service proxy add entries to the registry. Use one `map` tag for each desired type mapping.

The registry is an instance of the `XMLJavaMappingRegistry` class of the `org.apache.soap.util.xml` package. A `WebServiceProxy` instance has a `getXMLMappingRegistry()` method to access the registry.

The `map` tag includes attributes to specify the encoding style, serializer, deserializer, and namespace URI to facilitate the type mapping. The Web services tag library supports custom serializers and deserializers, if you want to create your own.

> **Important:** When using a `map` tag, you must nest it within a `webservice` tag.

Syntax

```
<ws:map localName = "local_name_of_SOAPXML_type"
        namespaceUri = "URI_of_namespace_for_SOAPXML_type"
        javaType = "Java_type_to_map"
        encodingStyle = "URL_of_SOAP_encoding_style"
        java2xmlClassName = "Java_to_XML_serializer"
        xml2javaClassName = "XML_to_Java_deserializer" />
```

Attributes

- `localName` (required): Specify the local name of the SOAP/XML datatype, such as `SOAPStruct`, for example.

- `namespaceUri` (required): Specify a valid URI for the namespace of the SOAP/XML datatype. The following is an example:

  `http://soapinterop.org/xsd`

- `javaType` (required): Specify the Java type which you want to map to the SOAP/XML type. The types must be legally mappable.

- `encodingStyle` (required): Specify a valid URI for a SOAP encoding style. The following is an example:

  `http://schemas.xmlsoap.org/soap/encoding`

- `java2xmlClassName` (required): Specify the class name with the functionality for serializing the data for Java-to-XML conversion. This can be a custom class. The following is an example:

  `org.apache.soap.encoding.soapenc.BeanSerializer`

- `xml2javaClassName` (required): Specify the class name with the functionality for deserializing the data for XML-to-Java conversion. This can be a custom class. The following is an example:

  `org.apache.soap.encoding.soapenc.BeanSerializer`

Web Services property Tag

You can optionally use this tag to specify a name/value pair that defines any of several supported custom properties for use by the Web service client application. For example, you could use `property` tags to specify an HTTP proxy host and proxy port if a proxy is required for access through a network firewall. The following properties are supported:

- `http.proxyHost`: Use this property to specify the host name of an HTTP proxy server.

- `http.proxyPort`: Use this property to specify a port number of an HTTP proxy server.

- `javax.net.ssl.KeyStore`: Use this property to specify the full path of an Oracle security wallet file.

> **Important:** When using a `property` tag, you must nest it within a `webservice` tag. The property will have the same scope as the parent Web service.

Syntax

```
<ws:property name="http.proxyHost" | "http.proxyPort" | "javax.net.ssl.KeyStore"
              value = "property_value" />
```

Attributes

- `property` (required): Specify the property you want to set; it must be one of the supported properties listed in the tag syntax.

- `value` (required): Specify the desired value of the property—a host name, port number, or full path to an Oracle wallet file.

Web Services invoke Tag

Use this tag to invoke an operation of the Web service. The tag handler will call the remote Web service operation by passing an input message in a SOAP request, then will wait for the SOAP response. You must specify the operation, as well as an object ID for the object that will contain the returned response. The tag handler uses the operation name to find the operation in the WSDL document.

The `invoke` tag gains access to a Web service proxy in one of two ways:

- The `invoke` tag is nested within the `webservice` tag that establishes the proxy.

- The `invoke` tag uses its `webservice` attribute to access a `WebProxyService` scripting variable created through a `webservice` tag `id` attribute.

In a situation where there are overloaded operations (two operations of the same name using different I/O messages), the `invoke` tag has attributes to specify the input and output message names for the desired operation. In this case, for RPC-style bindings, the specified input and output message names are used to form the RPC signature of the operation. Otherwise, the RPC signature is the default according to the WSDL document.

If the output message has multiple parts, then the returned result is an array of message parts (all within a single SOAP response).

The `invoke` tag can act as an XML producer, supporting explicit passing of an XML output object through the `toXMLObjName` attribute. This is useful if the `invoke` tag is nested inside other kinds of tags such as Web Object Cache tags or the XML `transform` tag. Also note that an XML output object can be written to the `JspWriter` object of the JSP page for output directly to the user's browser. This is enabled through the `xmlToWriter` attribute.

> **Notes:**
>
> - Waiting for the SOAP response is a blocking function.
>
> - The scope of the output result object, identified by the `id` attribute, is the same as the scope of the proxy object defined in the `webservice` tag for the Web service. For an `invoke` tag nested within a `webservice` tag, this scope is from the `webservice` start-tag to the `webservice` end-tag. However, the `id` object can still be accessed outside the `webservice` tag through use of the `findAttribute()` method of the JSP page context object.

Syntax

```
<ws:invoke id = "variable_name_for_output_result"
           operation = "operation_to_invoke"
```

```
              [ webservice = "variable_name_of_Web_service_proxy" ]
              [ inputMsgName = "name_of_input_message" ]
              [ outputMsgName = "name_of_output_message" ]
              [ xmlToWriter = "true" | "false" ]
              [ toXMLObjName = "objectname" ] >
```

...body / nested tags...

```
</ws:invoke>
```

Attributes

- `id` (required): Specify a scripting variable name for the output result object. The specified name must be a valid Java identifier. See the note preceding the tag syntax above for information about the scope of the `id` object.

- `operation` (required): Specify an operation to be executed (an operation from the WSDL document).

- `webservice`: Use this attribute if you want to specify the name of a `WebServiceProxy` scripting variable corresponding to the service to invoke. This is not necessary if the `invoke` tag is nested inside the `webservice` tag that accesses the desired service.

- `inputMsgName`: Optionally specify the input message name—the name of a `wsdl:input` tag in the WSDL document—for the operation. This is only necessary if there are overloaded operations (operations with the same name that use different message names).

- `outputMsgName`: Optionally specify the output message name—the name of a `wsdl:output` tag in the WSDL document—for the operation. This is necessary only if there are overloaded operations (operations with the same name that use different message names).

- `xmlToWriter`: For a document-style Web service, and if the output is an XML object, set this attribute to `"true"` if you want to output the XML to the `JspWriter` output object of the JSP page. The default setting is `"false"`.

- `toXMLObjName`: For a document-style Web service, and if you want to explicitly pass the output in an XML object, use this attribute to specify the name of the object.

Web Services part Tag

Use this tag, nested within an `invoke` tag, if the operation being performed requires input message part values, using one `part` tag for each input part.

How to specify the part value might depend on whether you are using an RPC-style or document-style Web service. For RPC-style, you must use the `value` attribute. For document-style, you have the option of passing the value through an XML request element in the tag body.

> **Note:** If you use both a tag body and the `value` attribute, the tag body is ignored.

Syntax

```
<ws:part name = "part_name"
        [ value = "part_value" ] >
```

...optional body, with request element, for document-style...

```
</ws:part>
```

Attributes

- `name` (required): Specify the name of the input part (a valid Java identifier).

- `value`: Specify the value of the input part. This is required for an RPC-style Web service. For a document-style Web service, you can use the tag body instead.

Web Services Tag Examples

This section provides a template for use of the Web services tag library, a sample JSP page to invoke an RPC-style Web service, and a sample page to invoke a document-style Web service.

Web Services Example: Usage Template

```
<HTML>
<HEAD>
<TITLE>Title</TITLE>
</HEAD>
<BODY>
<H2>This is sample HTML text.</H2>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/wstaglibrary.tld"
           prefix="ws" %>
<ws:webservice id="myws"
               wsdlUrl="wsdlurl"
               {
                   binding="" soapLocation="" | service="" port=""
               }
               {
                   scope="page | request | session | application"
               }
                >
   <ws:property name="property" value="string"/>

   <ws:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           localname="SOAPStruct"
           namespaceUri="http://soapinterop.org/xsd"
           javaType="MySoapStructBean"
           java2xmlClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
           xml2javaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
    />

</ws:webservice>

<ws:invoke id="result" webservice="myws" operation="add" inputMsgName=""
           outputMsgName="">
   <ws:part name="part_name" value="{string | <%= expression %>}"/>
</ws:invoke>

<% =result %>
</BODY>
</HTML>
```

Web Services Example: Sample JSP Page for RPC-Style Web Service

```
<%@ page contentType="text/html"%>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/wstaglib.tld"
```

```
                    prefix="ws" %>
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; ">
</HEAD>
<BODY>
<%
 String itemID = request.getParameter("itemID");
%>
<ws:webservice id="ebay"
    wsdlUrl="http://www.xmethods.net/sd/2001/EBayWatcherService.wsdl"
    binding="eBayWatcherBinding"
    soapLocation="http://services.xmethods.net:80/soap/servlet/rpcrouter"
    scope="page">
  <ws:property name="http.proxyHost" value="www-proxy.us.oracle.com"/>
  <ws:property name="http.proxyPort" value="80"/>
</ws:webservice>
<ws:invoke id="price" webservice="ebay" operation="getCurrentPrice">
  <ws:part name="auction_id" value="<%=itemID%>"/>
</ws:invoke>
<B>
Action price for eBay Item # <%=itemID%> is :
</B>
<P>
$<%= price%>
@
<%= new java.util.Date()%>
</P>
</BODY>
</HTML>
```

Web Services Example: Sample JSP Page for Document-Style Web Service

```
<%@ page contentType="text/xml;"%>
<%@ page  import= oracle.xml.parser.v2.XMLElement;"%>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/wstaglib.tld"
           prefix="ws" %>
<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/xml.tld"
           prefix="xml" %>

<ws:webservice id="bookService"
        wsdlUrl="http://hosting.msugs.ch/cheeso9/books/books.asmx?WSDL"
        binding="LookyBookServiceSoap"
        soapLocation ="http://hosting.msugs.ch/cheeso9/books/books.asmx"
        scope="session">
</ws:webservice>

<ws:invoke id="bookResult"
        operation="GetInfo"
        webservice="bookService">
    <ws:part name="parameters">
        <GetInfo xmlns="http://dinoch.dyndns.org/webservices/">
          <ISBN>SomeISBNNumber</ISBN>
        </GetInfo>
    </ws:part>
</ws:invoke>
<%
    XMLNode resultNode = (XMLNode) bookResult;
    resultNode.Error! Bookmark not defined.(new java.io.PrintWriter(out));
%>
```

```
</BODY>
</HTML>
```

A

JML Compile-Time Syntax and Tags

The JSP tag library framework was introduced in the JSP 1.1 specification. Oracle JSP releases prior to the implementation of the JSP 1.1 specification could support JML tags only as Oracle-specific translator extensions. This is referred to as *compile-time tag support* in this manual.

JSP releases with OC4J continue to support the compile-time JML implementation; however, it is generally advisable to use the standards-compliant runtime implementation whenever possible. The runtime implementation is documented in Chapter 3, "JSP Markup Language Tags".

This appendix discusses features of the compile-time implementation that are not in common with the runtime implementation, and consists of the following sections:

- JML Compile-Time Syntax Support

- JML Compile-Time Tag Support

For a general discussion of when it might be advantageous to use a compile-time implementation, refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide.*

JML Compile-Time Syntax Support

The following sections describe Oracle-specific bean reference syntax and expression syntax supported by the compile-time JML implementation for specifying tag attribute values:

- JML Bean References and Expressions, Compile-Time Implementation

- Attribute Settings with JML Expressions

This functionality requires the OC4J JSP translator; it is not portable to other JSP environments.

JML Bean References and Expressions, Compile-Time Implementation

A *bean reference* is any reference to a JavaBean instance 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()
```

The Oracle compile-time JML implementation, however, offers abbreviated syntax, as described in the following subsections.

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. You can omit default property names 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:

```
customer
```

Most JavaBeans do not define a default property. Of those that do, the most significant are the JML datatype JavaBeans described in Chapter 2, "JavaBeans for Extended Types".

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 3-2 notes standards-compliant syntax. 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 the preceding section, "JML Bean References and Expressions, Compile-Time Implementation". Note the requirements that follow.

- Wherever Chapter 3 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 3 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 (presuming `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, where 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 Tag Support

This section presents the following:

- Documentation of the `taglib` directive that you must use for compile-time JML support

- Summary of all compile-time tags, noting which are desupported in the runtime implementation

- Description of tags supported by the compile-time implementation that are desupported in the runtime implementation

  Tags still supported in the runtime implementation are documented in "JSP Markup Language (JML) Tag Descriptions" on page 3-2.

> **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 current JSP specification.

The taglib Directive for Compile-Time JML Support

The Oracle compile-time JML support implementation uses a custom class, 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, instead of specifying a TLD file as in standard JSP tag library usage:

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

For information about usage of the taglib directive for the JML runtime implementation, see "Overview of the JSP Markup Language (JML) Tag Library" on page 3-1.

JML Tag Summary, Compile-Time Versus Runtime

Most JML tags are available in both the runtime model and the compile-time model; however, there are exceptions, as summarized in the following tables.

Table A–1   Bean-Binding Tags: Compile-Time Model Versus Runtime Model

Tag	Supported in Oracle Compile-Time Implementation?	Supported in Oracle Runtime Implementation?
useBean	Yes	No; use jsp:useBean.
useVariable	Yes	Yes
useForm	Yes	Yes
useCookie	Yes	Yes
remove	Yes	Yes

Table A–2   Bean Manipulation Tags: Compile-Time Model Versus Runtime Model

Tag	Supported in Oracle Compile-Time Implementation?	Supported in Oracle Runtime Implementation?
getProperty	Yes	No; use jsp:getProperty.
setProperty	Yes	No; use jsp:setProperty.
set	Yes	No
call	Yes	No
lock	Yes	No

Table A–3   Control Flow Tags: Compile-Time Model Versus Runtime Model

Tag	Supported in Oracle Compile-Time Implementation?	Supported in Oracle Runtime Implementation?
if	Yes	Yes

Table A–3   (Cont.) Control Flow Tags: Compile-Time Model Versus Runtime Model

Tag	Supported in Oracle Compile-Time Implementation?	Supported in Oracle Runtime Implementation?
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`.

Table A–4   XML Tags: Compile-Time Model Versus Runtime Model

Tag	Supported in Oracle Compile-Time Implementation?	Supported in Oracle Runtime Implementation?
transform	Deprecated	Yes
styleSheet	Deprecated	Yes

Table A–5   Utility Tags: Compile-Time Model Versus Runtime Model

Tag	Supported in Oracle Compile-Time Implementation?	Supported in Oracle Runtime Implementation?
print	Yes; use double-quotes to specify a string literal.	No; use JSP expressions.
plugin	Yes	No; use `jsp:plugin`.

> **Note:**   Since Oracle9*i*AS Release 2 (9.0.3), the `transform` and `styleSheet` tags are deprecated in the compile-time implementation.

Descriptions of Additional JML Tags, Compile-Time Implementation

The following sections provide detailed descriptions of JML tags that are still supported by the JML compile-time implementation but are not supported by the JML runtime implementation. The tags supported in the runtime implementation are documented under "JSP Markup Language (JML) Tag Descriptions" on page 3-2.

- 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

Notes:

■ The prefix "jml:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.

■ See "Tag Syntax Symbology and Notes" on page 1-2 for general information about tag syntax conventions in this manual.

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 creates a new instance of the appropriate class and attaches 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.

You can refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* for an overview of the `jsp:useBean` tag.

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

In addition to specifying `id`, you must specify `class`, `type` (or `class` *and* `type`), or `beanName`.

Refer to the Sun Microsystems *JavaServer Pages Specification* for detailed information about `jsp:useBean` 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 *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* or the Sun Microsystems *JavaServer Pages Specification*.

Syntax

```
<jml:getProperty name = "beanInstanceName"
                 property = "propertyName" />
```

Attributes

- `name` (required): This is the name of the bean whose property is being retrieved.

- `property` (required): This is the name of the property being retrieved.

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 and adds functionality to support JML expressions. In particular, you can use JML bean references.

For general information about `setProperty` usage, refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* or the Sun Microsystems *JavaServer Pages Specification.*

Syntax

```
<jml:setProperty name = "beanInstanceName"
                 property = " * " |
                 property = "propertyName" [ param = "parameterName" ] |
                 property = "propertyName"
              [ value = "stringLiteral" | "<%= jmlExpression %>" ] />
```

Attributes

- `name` (required): This is the name of the bean whose property is being set.

- `property` (required): This is the name of the property being set.

- `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 (required): This is a direct reference (JML bean reference) to the bean property to be set.

- value (required): This is the new property value. It is expressed either as a string literal, a JML expression, or a standard JSP expression.

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 (required): 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.

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-scope 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` (required): This is the name of the object that should be locked during execution of code in the `lock` tag body.

Example In the following example, `pageCount` is an application-scope `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 the including page (the page invoking `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.

For general information about `include` usage, refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* or the Sun Microsystems *JavaServer Pages Specification.*

Syntax

```
<jml:include page = "relativeURL" | "<%= jmlExpression %>"
             flush = "true" | "false" />
```

Attributes

For general information about `include` attributes and usage, refer to the Sun Microsystems *JavaServer Pages Specification.*

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.

For general information about `forward` usage, refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide.*

Syntax

```
<jml:forward page = "relativeURL" | "<%= jmlExpression %>" />
```

Attributes

For general information about `forward` attributes, refer to the Sun Microsystems *JavaServer Pages Specification.*

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` (required): Specifies the string or expression to be evaluated and output.

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 general information about `plugin` usage, refer to the *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide* or the Sun Microsystems *JavaServer Pages Specification.*

B

Third Party Licenses

This appendix includes the Third Party License for third party products included with Oracle Application Server and discussed in this manual. Topics include:

- Apache HTTP Server
- Jaxen
- SAXPath

Apache HTTP Server

Under the terms of the Apache license, Oracle is required to provide the following notices. However, the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

The Apache Software License

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000-2002 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
```

```
*
* 4. The names "Apache" and "Apache Software Foundation" must
*    not be used to endorse or promote products derived from this
*    software without prior written permission. For written
*    permission, please contact apache@apache.org.
*
* 5. Products derived from this software may not be called "Apache",
*    nor may "Apache" appear in their name, without prior written
*    permission of the Apache Software Foundation.
*
* THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
* USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
* OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
* ====================================================================
*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Software Foundation.  For more
* information on the Apache Software Foundation, please see
* <http://www.apache.org/>.
*
* Portions of this software are based upon public domain software
* originally written at the National Center for Supercomputing Applications,
* University of Illinois, Urbana-Champaign.
*/
```

Jaxen

Oracle is required to provide the text of the third-party license, but the third-party program will be subject to the Oracle license, and Oracle will NOT provide warranties and technical support for the third-party technology.

This program contains third-party code from Jaxen. Under the terms of the Jaxen license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Jaxen software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Jaxen software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Jaxen.

The Jaxen Software License

Copyright (C) 2000-2002 bob mcwhirter & James Strachan. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "Jaxen" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jaxen.org.

4. Products derived from this software may not be called "Jaxen", nor may "Jaxen" appear in their name, without prior written permission from the Jaxen Project Management (pm@jaxen.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgment equivalent to the following: "This product includes software developed by the Jaxen Project (http://www.jaxen.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.jaxen.org/.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE Jaxen AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Jaxen Project and was originally created by bob mcwhirter and James Strachan . For more information on the Jaxen Project, please see http://www.jaxen.org/.

SAXPath

Oracle is required to provide the text of the third-party license, but the third-party program will be subject to the Oracle license, and Oracle will NOT provide warranties and technical support for the third-party technology.

This program contains third-party code from SAXPath. Under the terms of the SAXPath license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the SAXPath software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the SAXPath software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or SAXPath.

The SAXPath License

Copyright (C) 2000-2002 werken digital. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "SAXPath" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@saxpath.org.

4. Products derived from this software may not be called "SAXPath", nor may "SAXPath" appear in their name, without prior written permission from the SAXPath Project Management (pm@saxpath.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgment equivalent to the following: "This product includes software developed by the SAXPath Project (http://www.saxpath.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.saxpath.org/.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE SAXPath AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the SAXPath Project and was originally created by bob mcwhirter and James Strachan . For more information on the SAXPath Project, please see http://www.saxpath.org/. */

Index