Oracle® Fusion Middleware

Developing JSP Tag Extensions for Oracle WebLogic Server



26 (26.1.0.0.0)
G56955-01
November 2026

ORACLE

Oracle Fusion Middleware Developing JSP Tag Extensions for Oracle WebLogic Server, 26 (26.1.0.0.0)

G56955-01

Copyright © 2007, 2026, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documentation i
Conventions ii

1 Introduction

2 Understanding and Creating Custom JSP Tags

Tag Handler API and Implementation 1
Custom Tag Library 1
Custom Tag Format 2
What Can You Do with Custom Tags? 3
Creating and Configuring a JSP Tag Library: Example Procedures 4

3 Creating a Tag Library Descriptor

Overview of Tag Library Descriptors 1
Writing the Tag Library Descriptor 1
Sample Tag Library Descriptor 4
The DynamicAttributes Interface 5
Tag Handler Support of Dynamic Attributes 5
Dynamic Attributes Example 6
    Dynamic Attributes Syntax 6

4 Implementing the Tag Handler

Simple Tag Handler Life Cycle (SimpleTag Interface) 1
    Events in the Simple Tag Handler Life Cycle 1
    JSP Fragments 2
Tag Handler Life Cycle (Tag and BodyTag Interfaces) 3

Copyright © 2007, 2026, Oracle and/or its affiliates.

Iteration Over a Body Tag (IterationTag Interface) 4
Handling Exceptions within a Tag Body 4
Using Tag Attributes 4
Defining New Scripting Variables 5
Dynamically Named Scripting Variables 6
Defining Variables in the Tag Library Descriptor 6
Writing Cooperative Nested Tags 6
Using a Tag Library Validator 7

5 Administration and Configuration

Configuring JSP Tag Libraries 1
Packaging a JSP Tag Library as JAR File 1

Copyright © 2007, 2026, Oracle and/or its affiliates.

Preface

This document is a resource for software developers who want to create and use custom tags in JavaServer Pages (JSPs) and Web authors who want to use custom tags in JSPs.

Audience

Any issues that deal with production phase administration, monitoring, or performance tuning topics are not addressed in this documentation. See Related Documentation for links to WebLogic Server documentation and resources for these topics.

It is assumed that the reader is familiar with Jakarta EE platform and JSP concepts.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documentation

This document contains JSP extension-specific design and development information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server* is a guide to developing WebLogic Server applications.
- *Developing Applications for Oracle WebLogic Server* is a guide to developing WebLogic Server applications.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications.
- Jakarta Pages 4.0 Specification, Chapter 7, Tag Extensions: https://jakarta.ee/specifications/pages/4.0/jakarta-server-pages-spec-4.0.html
- Jakarta Pages 4.0 API reference for the jakarta.servlet.jsp.tagext package: https://jakarta.ee/specifications/pages/4.0/apidocs/jakarta.servlet.jsp/jakarta/servlet/jsp/tagext/package-summary.html

New and Changed WebLogic Server Features

See *What's New in Oracle WebLogic Server* for a comprehensive listing of the new WebLogic Server features introduced in this release.

Conventions

The following text conventions are used in this document:

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

1

Introduction

The Jakarta Pages 4.0 specification enables you to create and use custom tags in Jakarta Server Pages (JSPs). Custom tags let developers put reusable presentation behavior behind a tag name that a page author can use without repeating the Java implementation.
You can use custom JSP tag extensions to generate dynamic content in JSP pages. WebLogic Server supports the Jakarta Pages 4.0 tag-extension mechanism. For the portable rules, see the Jakarta Pages 4.0 Specification at https://jakarta.ee/specifications/pages/4.0/jakarta-server-pages-spec-4.0.html.

> **Note**
>
> Jakarta Pages 4.0 retains classic and simple custom tags but removes APIs deprecated in 3.1. For a classic handler that buffers its body, return BodyTag.EVAL_BODY_BUFFERED from doStartTag(); return IterationTag.EVAL_BODY_AGAIN from doAfterBody() only when another iteration is needed, and SKIP_BODY otherwise. BodyTag.EVAL_BODY_TAG is no longer in the 4.0 API. Replace calls to JspException.getRootCause() with getCause(). For the complete 3.1-to-4.0 change list, see Appendix C.1 of the Jakarta Pages 4.0 Specification.

Copyright © 2007, 2026, Oracle and/or its affiliates.

2

Understanding and Creating Custom JSP Tags

The following topics provide an overview of custom JSP tag functionality, format, and components, as well as procedures for creating and configuring a tag library.

Tag Handler API and Implementation

You write a custom JSP tag by writing a Java class that is called a tag handler.

The Jakarta Pages 4.0 API defines the classes and interfaces for custom tag handlers. See the Jakarta Pages 4.0 Specification at https://jakarta.ee/specifications/pages/4.0/jakarta-server-pages-spec-4.0.html and the jakarta.servlet.jsp.tagext API reference at https://jakarta.ee/specifications/pages/4.0/apidocs/jakarta.servlet.jsp/jakarta/servlet/jsp/tagext/package-summary.html.

A custom tag handler is either a classic tag handler or a simple tag handler:

- Classic tag handlers use the Tag, IterationTag, or BodyTag interface. Tag defines the basic start/end lifecycle and can include a body without buffering it. IterationTag extends Tag and adds doAfterBody() to decide whether to evaluate the body again. BodyTag extends IterationTag and adds access to buffered body content through BodyContent. Use BodyTag only when the handler needs to inspect or transform buffered body content.
- Simple tag handlers use the SimpleTag interface. SimpleTag does not extend Tag. Its doTag() method runs once per tag invocation; the handler can evaluate its body, if present, through the JspFragment supplied by the container.

Implement the appropriate interface directly, or extend a support class that supplies default implementations. SimpleTagSupport implements SimpleTag; TagSupport implements IterationTag (and therefore Tag); BodyTagSupport implements BodyTag. These support classes are concrete classes, not abstract base classes. Choose Tag when no body iteration or buffering is needed, IterationTag when the body may repeat without buffering, BodyTag when the handler needs buffered body content, or SimpleTag for the simpler doTag() lifecycle.

You can include one or more custom JSP tags in a *tag library*. You define a tag library by a tag library descriptor (.tld) file. The TLD describes the syntax for each tag and ties it to the Java classes that execute its functionality.

Custom Tag Library

JSP tag libraries include one or more custom JSP tags and are defined in a tag library descriptor (.tld) file. To use a custom tag library from a JSP page, reference its tag library descriptor with a `<%@ taglib %>` directive.

For example:

```
<%@ taglib uri="myTLD" prefix="mytaglib" %>
```

- uri

  The uri value in a taglib directive identifies the tag library. In this example, the directive uses myTLD:

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<%@ taglib uri="myTLD" prefix="mytaglib" %>
```

> **Note**
>
> If /WEB-INF/library.tld contains <uri>myTLD</uri>, the JSP container discovers that TLD automatically; no taglib entry in web.xml is required. Ordinary TLD files may be placed under /WEB-INF, but not under /WEB-INF/tags, which is reserved for tag files and an optional implicit.tld. A TLD inside a JAR in /WEB-INF/lib is discovered when it is under META-INF in that JAR.

An explicit mapping is also allowed. If you use one, place this fragment inside the existing <web-app> element in /WEB-INF/web.xml:

```
<jsp-config>
  <taglib>
    <taglib-uri>myTLD</taglib-uri>
    <taglib-location>/WEB-INF/library.tld</taglib-location>
  </taglib>
</jsp-config>
```

- `prefix`

  The `prefix` attribute assigns a label to the tag library. You use this label to reference its associated tag library when writing your pages using custom JSP tags.

  For example, if the library (called mytaglib) from the example above defines a new tag called `newtag`, you would use the tag in your JSP page like this:

  ```
  <mytaglib:newtag>
  ```

See Creating a Tag Library Descriptor.

Custom Tag Format

A custom tag format can be empty, called an *empty tag*, or can contain a body, called a *body tag*. Both types of tags can accept a number of attributes that are passed to the Java class that implements the tag.

See Handling Exceptions within a Tag Body for more information.

An empty tag takes the following form:

```
<mytaglib:newtag attr1="aaa" attr2="bbb" ... />
```

A body tag takes the following form:

```
<mytaglib:newtag attr1="aaa" attr2="bbb" ... >
  body
</mytaglib:newtag>
```

A tag body can include more JSP syntax, and even other custom JSP tags that also have nested bodies. Tags can be nested within each other to any level.

For example:

```
<mytaglib:tagA>
   <h2>This is the body of tagA</h2>
   You have seen this text <mytaglib:counter /> times!
  <p>
  <mytaglib:repeater repeat=4>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    <p>Hello World!
  </mytaglib:repeater>
</mytaglib:tagA>
```

The preceding example uses three custom tags to illustrate the ability to nest tags within a body tag. The tags function like this:

- The body tag `<mytaglib:tagA>` only sees the HTML output from its evaluated body. That is, the nested JSP tags `<mytaglib:counter>` and `<mytaglib:repeater>` are first evaluated and their output becomes part of the evaluated body of the `<mytaglib:tagA>` tag.
- The body of a body tag is first evaluated as JSP and all tags that it contains are translated, including nested body tags, whose bodies are recursively evaluated. The result of an evaluated body can then be used directly as the output of a body tag, or the body tag can determine its output based on the content of the evaluated body.
- The output generated from the JSP of a body tag is treated as plain HTML. That is, the *output is not further interpreted as JSP*.

What Can You Do with Custom Tags?

Custom tags can perform the following tasks:

- An empty tag can perform presentation-related server-side work based on its attributes, such as selecting a display option. Do not use a tag running inside a JSP as the only login or permission check for that page. Protect the JSP URL and other relevant resources through the web application's authentication and authorization configuration, such as Servlet security constraints for the URL patterns.
  - If the tag is included directly in the JSP page, then the surrounding scope is the JSP page output.
  - If the tag is nested within another parent tag, then the output becomes part of the evaluated body of its parent tag.
- Define new objects that can be referenced and used as scripting variables in the JSP page. A tag can introduce fixed-named scripting variables, or can define a dynamically named scripting variable with the `id` attribute.
- Iterate over body content of the tags until a certain condition is met. Use iteration to create repetitive output, or to repeatedly invoke a server side action.
- Determine whether the rest of the JSP page should be processed as part of the request, or skipped.
- An empty tag can perform server-side work based on the tag's attributes. The action that the tag performs can determine whether the rest of the page is interpreted or some other action is taken, such as a redirect. This function is useful for checking that users are logged in before accessing a page, and redirecting them to a login page if necessary.
- An empty tag can insert content into a page based on its attributes. You can use such a tag to implement a simple page-hits counter or another template-based insertion.
- An empty tag can define a server-side object that is available in the rest of the page, based on its attributes. You can use this tag to create a reference to an EJB, which is queried for data elsewhere in the JSP page.
- A body tag has the option to process its output before the output becomes part of the HTML page sent to the browser, evaluate that output, and then determine the resulting HTML that is sent to the browser. This functionality could be used to produce "quoted HTML," reformatted content, or used as a parameter that you pass to another function, such as an SQL query, where the output of the tag is a formatted result set.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- A body tag can repeatedly process its body until a particular condition is met.

Creating and Configuring a JSP Tag Library: Example Procedures

To create and use a custom JSP tag:

1. Write the tag handler class. It implements SimpleTag, Tag, IterationTag, or BodyTag, or extends the corresponding support class described above.
2. Write a Jakarta Pages 4.0 TLD that declares the tag, names the handler class, and contains <uri>myTLD</uri>. See Creating a Tag Library Descriptor.
3. Put the TLD at /WEB-INF/library.tld and the compiled handler class files under /WEB-INF/classes in directories matching their Java package names. Chapter 5 also explains how to package both in a JAR under /WEB-INF/lib.
4. At the top of the JSP page, add <%@ taglib uri="myTLD" prefix="mytaglib" %>.
5. Use a tag declared by the TLD, for example <mytaglib:code>text to format</mytaglib:code> when using the code tag in the Chapter 3 sample TLD.
6. Deploy the web application. No explicit web.xml taglib mapping is needed for the /WEB-INF/library.tld example because the TLD declares the same URI, myTLD. For an explicit mapping, use the Chapter 5 example.

Copyright © 2007, 2026, Oracle and/or its affiliates.

3

Creating a Tag Library Descriptor

The following topics describe how to create a tag library descriptor (TLD).

Overview of Tag Library Descriptors

A tag library allows a developer to group together tags with related functionality. A tag library uses a tag library descriptor (TLD) file that describes the tag extensions and relates them to their Java classes. WebLogic Server and some authoring tools use the TLD to get information about the extensions. TLD files have the file extension .tld and are written in XML notation.

Writing the Tag Library Descriptor

Order the elements in the tag library descriptor file as they are defined in the XSD. This ordering is used in the following procedure. The XML parser throws an exception if you incorrectly order the TLD elements.

The body of the TLD contains additional nested elements inside of the `<taglib> ... </taglib>` element. These nested elements are also described in the steps below. For display in this document, nested elements are indented from their parent elements, but indenting is not required in the TLD.

The example in Sample Tag Library Descriptor declares a new tag called code. The functionality of this tag is implemented by the Java class `weblogic.taglib.quote.CodeTag`.

To create a Tag Library Descriptor:

1. Create a text file with an appropriate name and the extension .tld, and save it in the WEB-INF directory of the Web application containing your JSP(s). Content beneath the WEB-INF directory is non-public and is not served over HTTP by WebLogic Server.
2. Include the following header:

   ```
   <taglib xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
   xmlns.jcp.org/xml/ns/javaee/web-jsptaglibrary_2_2.xsd"
       version="3.1"
       uri="http://www.example.com/mytaglib">
   ```

3. Add the contents of the TLD, embedded in a `<taglib>` element, as indicated in steps 4-7. The contents include elements containing information about the tag library and elements that define each tag. For example:

   ```
   <taglib>
     ... body of taglib descriptor ...
   </taglib>
   ```

4. Identify the tag library:

   ```
   <tlib-version>version_number</tlib-version>
   ```

   (Required) The version number of the tag library.

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<jsp-version>version_number</jsp-version>
```

(Required) Describes the JSP specification version (number) this tag library requires in order to function. The default is 2.0.

```
<short-name>TagLibraryName</short-name>
```

(Required) Assigns a short name to this tag library. This element is not used by WebLogic Server.

```
<uri>unique_string</uri>
```

(Required) Defines a public URI that uniquely identifies this version of the tag library.

```
<display-name>display_name</display-name>
```

(Optional) Contains a short name that is intended to be displayed by tools.

```
<icon>
  <small-icon>icon.jpg</small-icon>
```

(Optional) Contains the name of a file containing a small (16 x 16) icon image. The file name is a relative path within the tag library. The image must be either in the JPEG or GIF format, and the file name must end with the suffix ".jpg" or ".gif" respectively. The icon can be used by tools.

```
  <large-icon>icon.jpg</large-icon>
```

(Optional) Contains the name of a file containing a large (32 x 32) icon image. The file name is a relative path within the tag library. The image must be either in the JPEG or GIF format, and the file name must end with the suffix ".jpg" or ".gif" respectively. The icon can be used by tools.

```
</icon>
<description>...text...</description>
```

(Required) Defines an arbitrary text string describing the tag library.

```
<validator>unique_string</validator>
```

(Optional) Provides information on the scripting variables defined by this tag. A translation-time error occurs if a tag that has one or more variable subelements has a TagExtraInfo class that returns a non-null object.

```
<listener>unique_string</listener>
```

(Optional) Defines an optional event listener object to be instantiated and registered automatically.

5. Define a tag library validator (Optional).

```
<validator>
```

Top level element for a validator.

```
  <validator-class>my.validator</validator-class>
```

(Required) The Java class that performs the validation.

```
  <init-param>
```

(Optional) Defines initialization parameters for the validator class.

```
  <param-name>param</param-name>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

Defines the name of this parameter.

```
<param-value>value</param-value>
```

Defines the name of this parameter.

6. Define a tag.

   Use a separate `<tag>` element to define each new tag in the tag library. The `<tag>` element takes the following nested tags:

   ```
   <name>tag_name</name>
   ```

   (Required) Defines the name of the tag. This is used when referencing the tag in a JSP file, after the ":" symbol, For example:

   ```
   <mytaglib:tag_name>
   ```

   See What's New in Oracle WebLogic Server for a comprehensive listing of the new and changed JSP features in this release.

   ```
   <tag-class>package.class.name</tag-class>
   ```

   (Required) Declares the tag handler class that implements the functionality of this tag. Specify the fully qualified package name of the class.

   Locate the class file under the WEB-INF/classes directory, in a directory structure reflecting the package name. You can also package the classes in a tag library jar file.

   See Packaging a JSP Tag Library as JAR File.

   ```
   <tei-class>package.class.name</tei-class>
   ```

   (Optional) Declares the subclass of `TagExtraInfo` that describes the scripting variables introduced by this tag. If your tag does not define new scripting variables, it does not use this element. Specify the fully qualified package name of the class. You can perform validation of the tag's attributes in this class.

   Place the class files under the WEB-INF/classes directory of your Web application, under a directory structure reflecting the package name. You can also package the classes in a tag library jar file.

   See Packaging a JSP Tag Library as JAR File.

   ```
   <body-content>empty | JSP | scriptless | tagdependent</body-content>
   ```

   (Optional) Defines the content of the tag body.

   `empty` means that you use the tag in the *empty tag* format in the JSP page. For example:

   ```
   <taglib:tagname/>
   ```

   `JSP` means that the contents of the tag can be interpreted as a JSP and that you must use the tag in the *body tag* format. For example:

   ```
   <taglib:tagname>...</taglib:tagname>
   ```

   `scriptless` means that the contents of the tag do not contain any scripts or scripting elements.

   `tagdependent` means that your tag will interpret the contents of the body as a non-JSP (for example, an SQL statement).

   ```
   <attribute>
   ```


Copyright © 2007, 2026, Oracle and/or its affiliates.

(Optional) Defines the name of the attribute as it appears in the tag element in the JSP page. For example:

```
<taglib:mytag myAttribute="myAttributeValue">
```

Use a separate `<attribute>` element to define each attribute that the tag can take. Tag attributes allow the JSP author to alter the behavior of your tags.

```
<name>myAttribute</name>
<required>true | false</required>
```

(Optional) Defines whether this attribute has optional use in the JSP page. If not defined here, the default is `false` — that is, the attribute is optional by default. If `true` is specified, and the attribute is not used in a JSP page, a translation-time error occurs.

```
<rtexprvalue>true | false</rtexprvalue>
```

(Optional) Defines whether this attribute can take a scriptlet expression as a value, allowing it to be dynamically calculated at request time. If this element is not specified, the value is presumed to be `false`.

```
</attribute>
```

7. Define scripting variables (optional).

   In the `<tag>` element, you can define scripting variables.

   ```
   <variable>
   ```

   Top level element for declaring a variable.

   ```
   <name-given>someName</name-given>
   ```

   Defines the name of the variable, or you can define the name from an attribute using:

   ```
   <name-from-attribute>attrName</name-from-attribute>
   ```

   Names the variable with the value of `attrName`.

   ```
   <variable-class>some.java.type</variable-class>
   ```

   The Java type of this variable.

   ```
   <declare>true</declare>
   ```

   (Optional) If set to true, indicates that the variable is to be defined.

   ```
   <scope>AT_BEGIN</scope>
   ```

   The scope of the scripting variable. Valid options are:

   - `NESTED` (The variable is only available inside the tag body)
   - `AT_BEGIN` (The variable is defined just before executing the body)
   - `AT_END` (The variable is defined just after executing the body.)

   ```
   </variable>
   ```

Sample Tag Library Descriptor

The following example represents a listing of a tag library descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="https://jakarta.ee/xml/ns/jakartaee"
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://
jakarta.ee/xml/ns/jakartaee/web-jsptaglibrary_4_0.xsd"
        version="4.0">
  <description>Formatting tags for HTML content.</description>
  <tlib-version>1.0</tlib-version>
  <short-name>quote</short-name>
  <uri>myTLD</uri>
  <tag>
    <name>code</name>
    <tag-class>examples.webapp.jsp.tags.ext.quote.CodeTag</tag-class>
    <body-content>tagdependent</body-content>
    <attribute>
      <name>defaultFont</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>commentColor</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>quoteColor</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

The DynamicAttributes Interface

Jakarta Pages 4.0 supports the DynamicAttributes interface for custom tags whose attribute names are not all known when the tag library is authored.

For example, if you want to customize the `width` portion of the `<table>` HTML tag by determining its value at runtime, you could make use of the `DynamicAttributes` interface to customize only the portion of the tag you want to change, without needing to specify anything about the other portions of the `<table>` tag (`border`, `cellspacing`, and so on). Without the `DynamicAttributes` interface, you would have needed to write a Tag Handler or other complicated code to accomplish the task of determining the value of `width` dynamically at runtime.

Tag Handler Support of Dynamic Attributes

The TLD is what ultimately determines whether a tag handler accepts dynamic attributes or not. If a tag handler declares that it supports dynamic attributes in the TLD but it does not implement the `DynamicAttributes` interface, the tag handler must be considered invalid by the container.

If the `dynamic-attributes` element (a child element to the `tag` element for the tag library being authored) for a tag being invoked contains the value true, the following requirements apply:

- For each attribute specified in the tag invocation that does not have a corresponding attribute element in the TLD for this tag, a call must be made to `setDynamicAttribute()`, passing in the namespace of the attribute (or null if the attribute does not have a namespace or prefix), the name of the attribute without the namespace prefix, and the final value of the attribute.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Dynamic attributes must be considered to accept request-time expression values.
- Dynamic attributes must be treated as though they were of type `java.lang.Object`.
- The JSP container must recognize dynamic attributes that are passed to the tag handler using the <jsp:attribute> standard action.
- If the `setDynamicAttribute()` method throws `JspException`, the `doStartTag()` or `doTag()` method is not invoked for this tag, and the exception must be treated in the same manner as if it came from a regular attribute setter method.
- For a JSP document in either standard or XML syntax, if a dynamic attribute has a prefix that does not map to a namespace, a translation error must occur. In standard syntax, only namespaces defined using taglib directives are recognized.

Dynamic Attributes Example

In this example, attributes a and b are declared with <attribute> entries in the TLD. Attributes d1, d2, d3, and d4 are not declared, so they are dynamic attributes. The invokeDynamic tag handler implements DynamicAttributes, and its <tag> entry in the TLD contains <dynamic-attributes>true</dynamic-attributes>. The mytag namespace URI in this JSP document must identify the TLD for the tag library containing invokeDynamic.

You set the attributes using the calls:

- `setA( "1" ),`
- `setDynamicAttribute( null, "d1", "2" ),`
- `setDynamicAttribute( "http://www.foo.com/jsp/taglib/mytag.tld", "d2", "3" ),`
- `setB( "4" ),`
- `setDynamicAttribute( null, "d3", "5" ), and`
- `setDynamicAttribute( "http://www.foo.com/jsp/taglib/mytag.tld", "d4", "6" ).`

Example 3-1 Dynamic Attribute Example

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:mytag="http://www.foo.com/jsp/taglib/mytag.tld"
          version="4.0">
  <mytag:invokeDynamic a="1" d1="2" mytag:d2="3">
    <jsp:attribute name="b">4</jsp:attribute>
    <jsp:attribute name="d3">5</jsp:attribute>
    <jsp:attribute name="mytag:d4">6</jsp:attribute>
  </mytag:invokeDynamic>
</jsp:root>
```

Dynamic Attributes Syntax

A handler accepts dynamic attributes when it implements jakarta.servlet.jsp.tagext.DynamicAttributes and its TLD <tag> entry contains <dynamic-attributes>true</dynamic-attributes>. The handler must implement this method:

You must also configure an entry for the tag in the TLD to indicate dynamic attributes are accepted. For any attribute that is not declared in the TLD for this tag, instead of getting an error at translation time, the `setDynamicAttribute()` method is called, with the name and value of the attribute. You configure the tag to remember the names and values of the dynamic attributes.

Copyright © 2007, 2026, Oracle and/or its affiliates.

The `setDynamicAttribute()` method is called when a tag declared to accept dynamic attributes passes an attribute that is not declared in the TLD. The syntax is as follows:

```
public void setDynamicAttribute(String uri, String localName, Object value)
        throws JspException;
```

The parameter values are as follows:

- `uri` - the namespace of the attribute, or null if in the default namespace.
- `localName` - the name of the attribute being set.
- `value` - the value of the attribute.

A `JspException` is thrown if the tag handler signals that it does not accept the given attribute. The container must not call `doStartTag()` or `doTag()` for this tag.

For the full method contract, see the Jakarta Pages 4.0 DynamicAttributes API at https://jakarta.ee/specifications/pages/4.0/apidocs/jakarta.servlet.jsp/jakarta/servlet/jsp/tagext/DynamicAttributes.html.

Copyright © 2007, 2026, Oracle and/or its affiliates.

4

Implementing the Tag Handler

The following topics describe how to write Java classes that implement the functionality of an extended tag.

Simple Tag Handler Life Cycle (SimpleTag Interface)

Simple tag handlers present a much simpler invocation protocol than do classic tag handlers. The tag library descriptor maps tag library declarations to their physical underlying implementations. A simple tag handler is represented in Java by a class that implements the `SimpleTag` interface.

The life cycle of a simple tag handler is straightforward and is not complicated by caching semantics. Once a simple tag handler is instantiated by the container, it is executed and then discarded. The same instance must not be cached and reused. Initial performance metrics show that caching a tag handler instance does not necessarily lead to greater performance. To accommodate such caching makes writing portable tag handlers difficult and makes the tag handler prone to error.

In addition to being simpler to work with, simple tag extensions rely on JspContext rather than directly on the servlet-specific PageContext. PageContext extends JspContext. JspContext provides access to the JspWriter and scoped attributes, while PageContext adds facilities for JSP pages running in a servlet environment. The classic Tag interface uses PageContext; SimpleTag uses JspContext. If a SimpleTag has a body, the container passes it to setJspBody() as a JspFragment. The handler can invoke that fragment as many times as needed.

See JSP Fragments.

Unlike classic tag handlers, the `SimpleTag` interface does not extend `Tag`. Instead of supporting `doStartTag()` and `doEndTag()`, the `SimpleTag` interface provides a simple `doTag()` method, which is called once and only once for any given tag invocation. All tag logic, iteration, body evaluations, and so on are performed in this single method. Thus, simple tag handlers have the equivalent power of `BodyTag`, but with a much simpler life cycle and interface.

To support body content, the container calls setJspBody() with a JspFragment representing the tag body. The handler can call invoke() on that fragment to evaluate the body. SimpleTagSupport provides getJspBody() and other convenience methods.

Events in the Simple Tag Handler Life Cycle

The following describes the life cycle of simple tag handlers, from creation to invocation. When a simple tag handler is invoked, the following events occur in order:

1. Simple tag handlers are created initially using a zero argument constructor on the corresponding implementation class. Unlike classic tag handlers, a simple tag handler instance must never be pooled by the container. A new instance must be created for each tag invocation.
2. The container calls setJspContext() and, when the parent is non-null, setParent() on the tag handler. A tag file can receive a JspContext wrapper to provide its own page scope. In that case, getJspContext() returns the wrapped JspContext.

Copyright © 2007, 2026, Oracle and/or its affiliates.

3. The attributes specified as XML element attributes (if any) are evaluated next, in the order in which they are declared, according to the following rules (referred to as "evaluating an XML element attribute"). The appropriate bean property setter is invoked for each. If no setter is defined for the specified attribute but the tag accepts dynamic attributes, the `setDynamicAttribute()` method is invoked as the setter.

   If the attribute is a scripting expression (for example, "`<%= 1+1 %>`" in JSP syntax, or "`%= 1+1 %`" in XML syntax), the expression is evaluated, and the result is converted and passed to the setter.

   Otherwise, if the attribute contains any expression language expressions (for example, "`Hello ${name}`"), the expression is evaluated, and the result is converted and passed to the setter.

   Otherwise, the attribute value is taken verbatim, converted, and passed to the setter.

4. The container evaluates each <jsp:attribute> element in the order it appears and passes its value to the matching attribute setter. If the tag accepts dynamic attributes and no declared setter matches, the container calls setDynamicAttribute(). For an attribute whose declared type is JspFragment, the container creates a JspFragment and passes it to the setter. For other types, the container evaluates the <jsp:attribute> body, converts the result to the declared type, and passes that value to the setter.

5. If this invocation has a nonempty body, the container creates a JspFragment for it and calls setJspBody() on the handler. The body comes from <jsp:body>, or from the custom action body when there is no <jsp:body> or <jsp:attribute>. For tagdependent body content, the fragment supplies the body text verbatim; for scriptless content, it evaluates the body as scriptless JSP. The container does not call setJspBody() when the tag declares empty body content or this invocation has no body.

6. The `doTag()` method is invoked.

7. The implementation of `doTag()` performs its function, potentially calling other tag handlers (if the tag handler is implemented as a tag file) and invoking fragments.

8. The `doTag()` method returns, and the tag handler instance is discarded. If `SkipPageException` is thrown, the rest of the page is not evaluated and the request is completed. If this request was forwarded or included from another page (or servlet), only the current page evaluation stops.

9. For each tag scripting variable declared with scopes `AT_BEGIN` or `AT_END`, the appropriate scripting variables and scoped attributes are declared, as with classic tag handlers.

JSP Fragments

A JSP fragment is a portion of JSP code passed to a tag handler that can be invoked as many times as needed. You can think of a fragment as a template that is used by a tag handler to produce customized content. Thus, unlike simple attributes which are evaluated by the container, fragment attributes are evaluated by tag handlers during tag invocation.

JSP fragments can be parameterized using the JSP expression language (JSP EL) variables in the JSP code that composes the fragment. The JSP EL variables are set by the tag handler, thus allowing the handler to customize the fragment each time it is invoked.

See WebLogic JSP Reference in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

During the simple tag handler life cycle, the body of a `SimpleTag`, if present, is translated into a JSP fragment and passed to the `setJspBody` method. During the translation phase, various sections of the page are translated into implementations of the `jakarta.servlet.jsp.tagext.JspFragment` abstract class, before being passed to a tag

Copyright © 2007, 2026, Oracle and/or its affiliates.

handler. This is done automatically for any JSP code in the body of a named attribute (one that is defined by `<jsp:attribute>`) that is declared to be a fragment, or of type `JspFragment`, in the tag library descriptor (TLD). This is also automatically done for the body of any tag handled by a simple tag handler. Once passed in, the tag handler can then evaluate and re-evaluate the fragment as many times as needed, or even pass it along to other tag handlers, in the case of tag files.

A JSP fragment can be parameterized by a tag handler by setting page-scoped attributes in the `JspContext` associated with the fragment. These attributes can then be accessed by way of the expression language.

A translation error must occur if a piece of JSP code that is to be translated into a JSP fragment contains scriptlets or scriptlet expressions.

Tag Handler Life Cycle (Tag and BodyTag Interfaces)

The methods inherited from the `Tag` or `BodyTag` interfaces and implemented by the tag handler class are invoked by the JSP engine at specific points during the processing of the JSP page. These methods signify points in the life cycle of a tag and are executed in the following sequence:

1. When the JSP container encounters a classic tag, it obtains a handler instance. The instance may be new or one that the container has reused from an earlier invocation. Before calling doStartTag(), the container calls setPageContext() and setParent() to provide the current page and enclosing tag. If the handler extends TagSupport or BodyTagSupport, those support classes provide the basic implementations of these methods.
2. The `setXXXX()` JavaBean-like methods for each tag attribute are invoked.
3. The JSP container calls doStartTag(). The handler returns one of these values:
   - SKIP_BODY: Do not evaluate the body; proceed to doEndTag(). A tag declared with <body-content>empty</body-content> must return SKIP_BODY.
   - EVAL_BODY_INCLUDE: Evaluate a nonempty body directly to the current JspWriter. A Tag, IterationTag, or BodyTag handler may return this value. For IterationTag and BodyTag, the container calls doAfterBody() after evaluating the body.
   - BodyTag.EVAL_BODY_BUFFERED: For a BodyTag handler only, evaluate a nonempty body into a BodyContent buffer. The container first calls setBodyContent() and doInitBody().

   Jakarta Pages 4.0 does not define BodyTag.EVAL_BODY_TAG. Do not use that constant in source code or documentation examples. If the action invocation has no body, the container proceeds to doEndTag() without the body callbacks even when doStartTag() returned an evaluation value.
4. Only on the EVAL_BODY_BUFFERED path for a nonempty action body, the container calls setBodyContent(BodyContent). The handler can retain that buffer to inspect or transform evaluated body text. BodyTagSupport already retains it and exposes getBodyContent(). The container does not call setBodyContent() for SKIP_BODY, EVAL_BODY_INCLUDE, or an empty action body.
5. Immediately after setBodyContent(), the container calls doInitBody() before the first buffered body evaluation. Use this callback for preparation that needs access to the active BodyContent. The container does not call doInitBody() on the EVAL_BODY_INCLUDE path.
6. After each nonempty body evaluation for an IterationTag or BodyTag handler, the container calls doAfterBody(). Return IterationTag.EVAL_BODY_AGAIN to evaluate the body again,

Copyright © 2007, 2026, Oracle and/or its affiliates.

or Tag.SKIP_BODY to stop iterating and proceed to doEndTag(). On the buffered path, the handler can inspect getBodyContent(); on the inclusive path, body output has already gone directly to the current JspWriter, so there is no BodyContent buffer for that evaluation.

7. On the buffered path, a handler that wants to emit its buffered result can write it to the enclosing writer obtained from BodyContent.getEnclosingWriter() or BodyTagSupport.getPreviousOut(). Do not write the entire accumulated buffer after every iteration or earlier iterations will appear repeatedly. On the inclusive path, no buffered-body copy is needed because the body output has already been written to the current writer.
8. The `doEndTag()` method is invoked. Your tag handler can implement this method to perform post-tag, server side work, write output to the parent scope `JspWriter`, or close resources such as database connections.

   Your tag handler writes output directly to the surrounding scope using the JspWriter obtained from pageContext.getOut() in doEndTag(). On the buffered path, the JSP container restores the enclosing writer before calling doEndTag(). Do not manipulate the page-context writer stack directly in the handler.

   You can control the flow for evaluation of the rest of the JSP page by returning one of the following values from the `doEndTag()` method:

   `EVAL_PAGE`—Directs the JSP engine to continue processing the rest of the JSP page.

   `SKIP_PAGE`—Directs the JSP engine to skip the rest of the JSP page.
9. After all invocations on this classic tag handler instance have finished, the container calls release() before discarding the instance. release() is for long-term state and is not called after every doEndTag(). A classic handler may have handled several invocations before this callback.

Iteration Over a Body Tag (IterationTag Interface)

A tag that implements the `jakarta.servlet.jsp.tagext.IterationTag` interface has a method available called `doAfterBody()` that allows you to conditionally re-evaluate the body of the tag. If `doAfterBody()` returns `IterationTag.EVAL_BODY_AGAIN` the body is re-evaluated, if `doAfterBody()` returns `Tag.SKIP_BODY`, the body is skipped and the `doEndTag()` method is called.

For the complete contract, see the Jakarta Pages 4.0 IterationTag API at https://jakarta.ee/specifications/pages/4.0/apidocs/jakarta.servlet.jsp/jakarta/servlet/jsp/tagext/IterationTag.html.

Handling Exceptions within a Tag Body

You can catch exceptions thrown from within a tag by implementing the `doCatch()` and `doFinally()` methods of the `jakarta.servlet.jsp.tagext.TryCatchFinally` interface.

For the complete contract, see the Jakarta Pages 4.0 TryCatchFinally API at https://jakarta.ee/specifications/pages/4.0/apidocs/jakarta.servlet.jsp/jakarta/servlet/jsp/tagext/TryCatchFinally.html.

Using Tag Attributes

Your custom tags can define any number of attributes that can be specified from the JSP page. You can use these attributes to pass information to the tag handler and customize its behavior.

You declare each attribute name in the TLD in the `<attribute>` element. This declares the name of the attribute and other attribute properties.

Copyright © 2007, 2026, Oracle and/or its affiliates.

For each writable tag attribute declared in the TLD, the handler provides a public setter whose name and parameter type match that attribute. For an attribute named foo of type String, the handler can declare:

```
import jakarta.servlet.jsp.tagext.TagSupport;

public class ExampleTag extends TagSupport {
    private String foo;

    public void setFoo(String value) {
        this.foo = value;
    }
}
```

The first letter of the attribute name is capitalized after set. A matching getter is optional; the JSP container does not require one merely to set the attribute. The handler can store the supplied value in a field for its other methods to use.

The JSP engine invokes the setter methods for each attribute appropriately after the tag handler is initialized and before the `doStartTag()` method is called. Generally, you should implement the setter methods to store the attribute value in a member variable that is accessible to the other methods of the tag handler.

Defining New Scripting Variables

Your tag handler can introduce new scripting variables that can be referenced by the JSP page at various scopes. Scripting variables can be used like implicit objects within their defined scope.

Define a new scripting variable by using the <tei-class> element to identify a Java class that extends `jakarta.servlet.jsp.tagext.TagExtraInfo`. For example:

```
<tei-class>examples.webapp.jsp.tags.ext.session.ListTagExtraInfo</tei-class>
```

Then write the `TagExtraInfo` class.

For example:

```
package examples.webapp.jsp.tags.ext.session;

import jakarta.servlet.jsp.tagext.TagData;
import jakarta.servlet.jsp.tagext.TagExtraInfo;
import jakarta.servlet.jsp.tagext.VariableInfo;

public class ListTagExtraInfo extends TagExtraInfo {
    @Override
    public VariableInfo[] getVariableInfo(TagData data) {
        return new VariableInfo[] {
            new VariableInfo("name", "java.lang.String", true, VariableInfo.NESTED),
            new VariableInfo("value", "java.lang.String", true, VariableInfo.NESTED)
        };
    }
}
```

This class declares two nested scripting variables, name and value, both of type java.lang.String. The JSP container makes them available while evaluating the body of the list tag. The tag handler sets page-scoped attributes with those exact names before each body evaluation.

The constructor for `VariableInfo()` takes four arguments.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- A `String` that defines the name of the new variable.
- A `String` that defines the Java type of the variable. Give the full package name for types in packages other than the `java.lang` package.
- A boolean indicating whether this is a new scripting variable to declare. In this example, both values are true because both variables are new.
- An `int` declaring the scope of the variable. Use a static field from `VariableInfo` shown here:

  `VariableInfo.NESTED`—Available only within the start and end tags of the tag.

  `VariableInfo.AT_BEGIN`—Available from the start tag until the end of the page.

  `VariableInfo.AT_END`—Available from the end tag until the end of the page.

Configure your tag handler to initialize the value of the scripting variables via the page context. For example, the following Java source could be used in the `doStartTag()` method to initialize the values of the scripting variables defined above:

```
pageContext.setAttribute("name", name);
pageContext.setAttribute("value", value);
```

In the release-branch ListTag.doStartTag() example, the local variables name and value are Strings read from the session. The page-context attribute names must match the VariableInfo names exactly. The same two assignments are made before each subsequent body evaluation in ListTag.doAfterBody().

You can also access variables created with the `TagExtraInfo` class by referencing it the same way you access a JavaBean that was created with `useBean`.

Dynamically Named Scripting Variables

You can define the name of a new scripting variable from a tag attribute. This definition enables you to use multiple instances of a tag that define a scripting variable at the same scope, without the scripting variables of the tag clashing. If you want to achieve this from your class that extends `TagExtraInfo`, you must get the name of the scripting variable from the `TagData` that is passed into the `getVariableInfo()` method.

From `TagData`, you can retrieve the value of the attribute that names the scripting variable using the `getAttributeString()` method. There is also the `getId()` method that returns the value of the `id` attribute, which is often used to name a new implicit object from JSP tag.

Defining Variables in the Tag Library Descriptor

You can define variables in the TLD.

For more information, see Define scripting variables.

Writing Cooperative Nested Tags

You can design your tags to implicitly use properties from tags they are nested within. For example, in the code example called SQL Query (see the samples/examples/jsp/tagext/sql directory of your WebLogic Server installation) a `<sql:query>` tag is nested within a `<sql:connection>` tag. The query tag searches for a parent scope connection tag and uses the JDBC connection established by the parent scope.

Copyright © 2007, 2026, Oracle and/or its affiliates.

To locate a parent scope tag, your nested tag uses the static `findAncestorWithClass()` method of the `TagSupport` class. The following is an example taken from the `QueryTag` example.

```
ConnectionTag connectionTag = (ConnectionTag)
    jakarta.servlet.jsp.tagext.TagSupport.findAncestorWithClass(
        this, ConnectionTag.class);
if (connectionTag == null) {
    throw new JspException("Query tag must be nested in a connection tag");
}
conn = connectionTag.getConnection();
```

findAncestorWithClass(this, ConnectionTag.class) searches outward through enclosing tag handlers and returns the nearest ConnectionTag instance. It returns null if no such ancestor exists. The null check above turns that condition into a JspException before the handler calls getConnection(). This lookup does not throw ClassNotFoundException because the class is supplied directly as ConnectionTag.class.

Using this feature in your custom tags can simplify the syntax and usage of tags in the JSP page.

Using a Tag Library Validator

A tag library validator is a Java class that extends jakarta.servlet.jsp.tagext.TagLibraryValidator and checks the XML view of a JSP page during translation. The JSP container calls validate(String prefix, String uri, PageData page). That method returns a ValidationMessage[] array containing any validation errors. It returns null or an empty array when the page is valid. The PageData argument supplies the page's XML view; the method does not return a String.

To implement a Tag Library Validator:

1. Write the validator class.

   A validator class extends the `jakarta.servlet.jsp.tagext.TagLibraryValidator` class.

2. Reference the validator in the tag library descriptor.

   For example:

   ```
   <validator>
     <validator-class>
        myapp.tools.MyValidator
     </validator-class>
   </validator>
   ```

3. (Optional) Define initialization parameters.

   Your validator class can get and use initialization parameters.

   For example:

   ```
   <validator>
     <validator-class>
        myapp.tools.MyValidator
     </validator-class>
     <init-param>
       <param-name>myInitParam</param-name>
       <param-value>foo</param-value>
     </init-param>
   </validator>
   ```

4. Package the validator class in the `WEB-INF/classes` directory of a Web application.

Copyright © 2007, 2026, Oracle and/or its affiliates.

You can also package the classes in a tag library jar file.

See Packaging a JSP Tag Library as JAR File.

Copyright © 2007, 2026, Oracle and/or its affiliates.

5

Administration and Configuration

The following topics provide an overview of administration and configuration tasks for using JSP tag extensions.

Configuring JSP Tag Libraries

You can configure a JSP tag library as well as package a tag library as a JAR file.

To deploy the example tag library as files in a web application:

1. Create the Jakarta Pages 4.0 TLD shown in Chapter 3. Save it as /WEB-INF/library.tld in the web application. Its <uri> element is <uri>myTLD</uri>.
2. Compile examples.webapp.jsp.tags.ext.quote.CodeTag and place its class file at /WEB-INF/classes/examples/webapp/jsp/tags/ext/quote/CodeTag.class. Put any additional handler classes required by the TLD under /WEB-INF/classes using their Java package paths.
3. In the JSP page that uses the library, add <%@ taglib uri="myTLD" prefix="mytaglib" %>.
4. Invoke the tag declared by that TLD, for example <mytaglib:code>text to format</mytaglib:code>.
5. Deploy the web application to WebLogic Server. The container discovers /WEB-INF/library.tld and maps its declared URI, myTLD, without a web.xml taglib entry.
6. If this application uses an explicit mapping for the same library, put the “Explicit web.xml mapping” fragment below inside the existing <web-app> element of /WEB-INF/web.xml. Do not add a second <web-app> root.

   Explicit web.xml mapping: The mapping is optional for this example because library.tld already declares myTLD. If the application includes an explicit mapping, use exactly this fragment:

```
<jsp-config>
  <taglib>
    <taglib-uri>myTLD</taglib-uri>
    <taglib-location>/WEB-INF/library.tld</taglib-location>
  </taglib>
</jsp-config>
```

Packaging a JSP Tag Library as JAR File

You can also package a JSP tag library as a JAR file.

To package the Chapter 3 code-tag library as a JAR file:

1. Save the Chapter 3 Jakarta Pages 4.0 TLD as taglib-staging/META-INF/taglib.tld. It contains <uri>myTLD</uri> and names examples.webapp.jsp.tags.ext.quote.CodeTag as the handler.
2. Compile CodeTag and place CodeTag.class at taglib-staging/examples/webapp/jsp/tags/ext/quote/CodeTag.class. Place any other classes named by this TLD under taglib-staging in their matching Java package directories.

Copyright © 2007, 2026, Oracle and/or its affiliates.

3. Before creating the JAR, the staging layout must look like this:

```
taglib-staging/
├──── META-INF/
│     └──── taglib.tld
└──── examples/
      └──── webapp/jsp/tags/ext/quote/CodeTag.class
```

4. From the directory that contains taglib-staging, run these two commands. The first archives the contents of taglib-staging at JAR root; the second lists the resulting entries:

```
jar --create --file myTagLibrary.jar -C taglib-staging .
jar --list --file myTagLibrary.jar
```

5. Confirm that the listing contains META-INF/taglib.tld and examples/webapp/jsp/tags/ext/quote/CodeTag.class. It must not prefix either path with taglib-staging/.

6. Copy myTagLibrary.jar to /WEB-INF/lib/myTagLibrary.jar in the web application.

7. In the JSP page, use <%@ taglib uri="myTLD" prefix="wl" %> and then invoke the declared tag, for example <wl:code>text to format</wl:code>. No web.xml taglib mapping is needed because META-INF/taglib.tld declares the URI myTLD.

```
<%@ taglib uri="myTLD" prefix="wl" %>
```

   The container discovers META-INF/taglib.tld inside the JAR in /WEB-INF/lib and maps the TLD's myTLD URI to that library. No explicit web.xml taglib mapping is required for this example.

Copyright © 2007, 2026, Oracle and/or its affiliates.