Oracle® Fusion Middleware

Developing JSP Tag Extensions for Oracle WebLogic Server

12c (12.2.1.4.0)
E90834-02
November 2022

ORACLE

Oracle Fusion Middleware Developing JSP Tag Extensions for Oracle WebLogic Server, 12c (12.2.1.4.0)

E90834-02

Contents

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 Java 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 customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

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.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications.
- "*JavaServer Pages Tutorial*" at `http://docs.oracle.com/javaee/7/tutorial`
- JavaServer Pages (JSP) product overview at `http://www.oracle.com/technetwork/java/javaee/jsp/index.html`

Conventions

The following text conventions are used in this document:

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

1

Introduction and Roadmap

The following topics describe the contents and organization of this guide—*Developing JSP Tag Extensions for Oracle WebLogic Server*:

- Guide to this Document
- New and Changed JSP Features In This Release

Guide to this Document

- This chapter, Introduction and Roadmap introduces the organization of this guide.
- Understanding and Creating Custom JSP Tags provides an overview of custom JSP tag functionality, format, and components, as well as procedures for creating and configuring a tag library.
- Creating a Tag Library Descriptor describes how to create a tag library descriptor (TLD) file.
- Implementing the Tag Handler describes how to write Java classes that implement the functionality of an extended tag.
- Administration and Configuration provides an overview of administration and configuration tasks for using JSP tag extensions.

New and Changed JSP Features In This Release

See *What's New in Oracle WebLogic Server 12.2.1.3.0* for a comprehensive listing of the new WebLogic Server features introduced in this release.

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 JSP 2.1 API defines a set of classes and interfaces that you use to write custom tag handlers.

See the JSP 2.1 Specification at `http://java.sun.com/products/jsp/reference/api/index.html`.

Documentation for the `javax.servlet.jsp.tagext` API is available at `http://docs.oracle.com/docs/cd/E17802_01/products/products/jsp/2.1/docs/jsp-2_1-pfd2/index.html`.

Your tag handler must be of one of the following two types:

- Classic Tag Handlers implement one of three interfaces:

  Tag—Implement the `javax.servlet.jsp.tagext.Tag` interface if you are creating a custom tag that does not need access to its interface. The API also provides a convenience class `TagSupport` that implements the `Tag` interface and provides default empty methods for the methods defined in the interface.

  `BodyTag`—Implement the `javax.servlet.jsp.tagext.BodyTag` interface if your custom tag needs to use a body. The API also provides a convenience class `BodyTagSupport` that implements the `BodyTag` interface and provides default empty methods for the methods defined in the interface. Because `BodyTag` extends `Tag` it is a super set of the interface methods.

  `IterationTag`—Implement the `javax.servlet.jsp.tagext.IterationTag` interface to extend `Tag` by defining an additional method `doAfterBody()` that controls the reevaluation of the body.

- Simple Tag Handlers (`SimpleTag` interface):

  Implement the `javax.servlet.jsp.tagext.SimpleTag` interface if you wish to use a much simpler invocation protocol. The `SimpleTag` interface does not extend the

`javax.servlet.jsp.tagext.Tag` interface as does the `BodyTag` interface. Therefore, instead of supporting the `doStartTag()` and `doEndTag()` methods, the `SimpleTag` interface provides a `simple doTag()` method, which is called once and only once for each tag invocation.

You write the tag handler class by doing one of the following:

- Implement one of three interfaces, `SimpleTag`, `Tag`, or `BodyTag`, which define methods that are invoked during the life cycle of the tag.
- Extend an abstract base class that implements the `SimpleTag`, `Tag`, or `BodyTag` interfaces.

Extending an abstract base class relieves the tag handler class from having to implement all methods in the interfaces and also provides other convenient functionality. The `SimpleTagSupport`, `TagSupport`, and `BodyTagSupport` classes implement the `SimpleTag`, `Tag` or `BodyTag` interfaces and are included in the API.

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 JSP engine attempts to find the tag library descriptor by matching the `uri` attribute to a `uri` that is defined in the web application deployment descriptor (web.xml) with the `<taglib-uri>` element.

  > **Note:**
  >
  > You do not need to mention the `<taglib>` tag in the web.xml if the value of uri in `<%@ taglib uri='myTLD' prefix='mytaglib' %>` is the same as the uri specified in the `.tld` file, provided the `.tld` file is in the default location (`/WEB-INF/` or `/WEB-INF/tags/`). See the JSP 2.1 specification.

  For example, `myTLD` in the above the `taglib` directive would reference its tag library descriptor (library.tld) in the web application deployment descriptor like this:

  ```
  <taglib>
   <taglib-uri>myTLD</taglib-uri>
   <taglib-location>library.tld</taglib-location>
  </taglib>
  ```

- `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>
     <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:

- Produce output. The output of the tag is sent to the surrounding scope. The scope can be one of the following:
  - 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.
- 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 custom JSP tags:

1. Write a tag handler class.

   When you use a custom tag in your JSP, this class executes the functionality of the tag. A tag handler class implements one of three interfaces:

```
javax.servlet.jsp.tagext.BodyTag
javax.servlet.jsp.tagext.Tag
javax.servlet.jsp.tagext.SimpleTag
```

Your tag handler class is implemented as part of a *tag library*.

2. Reference the tag library in your JSP source using the JSP `<taglib>` directive. A tag library is a collection of JSP tags. Include this directive at the top of your JSP source.

   See Configuring JSP Tag Libraries.

3. Write the tag library descriptor (TLD). The TLD defines the tag library and provides additional information about each tag, such as the name of the tag handler class, attributes, and other information about the tags.

   See Creating a Tag Library Descriptor.

4. Reference the TLD in the Web application deployment descriptor (web.xml).

5. Use your custom tag in your JSP.

   See Configuring JSP Tag Libraries.

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 version="2.0" xmlns="http://java.sun.com/xml/j2ee" xmlns:xsi="http://
   www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
   j2ee/web-jsptaglibrary_2_0.xsd">
   ```

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.

```
<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>
```

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 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>
```

(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:

```
<taglib version="2.0" xmlns="http://java.sun.com/xml/j2ee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
j2ee/web-jsptaglibrary_2_0.xsd">
<taglib>
  <tlib-version>2.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>quote</short-name>
<uri>tag lib version id</uri>
  <description>
     This tag library contains several tag extensions
     useful for formatting content for HTML.
  </description>

  <tag>
    <name>code</name>
    <tag-class>weblogic.taglib.quote.CodeTag</tag-class>
    <body-content>tagdependent</body-content>
    <attribute>
      <name>fontAttributes</name>
    </attribute>
    <attribute>
      <name>commentColor</name>
    </attribute>
    <attribute>
      <name>quoteColor</name>
    </attribute>
  </tag>
</taglib>
```

The DynamicAttributes Interface

You can use the `DynamicAttributes` interface, supported by the JSP 2.0 container, as tags with values that are treated in a consistent manner but with names that are not necessarily known at development time.

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.
- 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 the following example attributes `a` and `b` are declared by using the attribute element in the TLD, attributes `d1` and `d2` are not declared, and the `dynamic-attributes` element is set to true.

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:mytag="http://www.foo.com/jsp/taglib/mytag.tld" version="2.0">
<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

For a tag to declare that it accepts dynamic attributes, it must implement the `DynamicAttributes` interface. The syntax is as follows:

```
public interface DynamicAttributes
```

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.

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(java.lang.String uri, java.lang.String localName,
java.lang.Object value)
```

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.

See the `DynamicAttributes` API at `http://docs.oracle.com/javaee/7/api/javax/servlet/jsp/tagext/DynamicAttributes.html` for more information about these interfaces.

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)
- Tag Handler Life Cycle (Tag and BodyTag Interfaces)
- Iteration Over a Body Tag (IterationTag Interface)
- Handling Exceptions within a Tag Body
- Using Tag Attributes
- Defining New Scripting Variables
- Writing Cooperative Nested Tags
- Using a Tag Library Validator

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 do not rely directly on any servlet APIs, which allows for potential future integration with other technologies. This capability is facilitated by the `JspContext` class, which `Page-Context` extends. `JspContext` provides generic services such as storing the `JspWriter` and keeping track of scoped attributes, whereas `PageContext` has functionality specific to serving JSPs in the context of servlets. Whereas the Tag interface relies on `PageContext`, `SimpleTag` only relies on `JspContext`. The body of `SimpleTag`, if present, is translated into a JSP fragment and passed to the `setJspBody` method. The tag can then execute the 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 `setJspBody()` method is provided. The container invokes the `setJspBody()` method with a `JspFragment` object encapsulating the body of the tag. The tag handler implementation can call `invoke()` on that fragment to evaluate the body. The `SimpleTagSupport` convenience class provides `getJsp-Body()` and other useful methods to make this even easier.

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 `setJspContext()` and `setParent()` methods are invoked on the tag handler. The `setParent()` method need not be called if the value being passed in is `null`. In the case of tag files, a `JspContext` wrapper is created so that the tag file can appear to have its own page scope. Calling `getJspContext()` must return the wrapped `Jsp-Context`.
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 value for each `<jsp:attribute>` element is evaluated, and the corresponding bean property setter methods are invoked for each, in the order in which they appear in the body of the tag. If no setter is defined for the specified attribute but the tag accepts dynamic attributes, the `setDynamicAttribute()` method is invoked as the setter.

   Otherwise, if the attribute is not of type `JspFragment`, the container evaluates the body of the `<jsp:attribute>` element. This evaluation can be done in a container-specific manner. Container implementors should note that in the process of evaluating this body, other custom actions may be invoked.

   Otherwise, if the attribute is of type `JspFragment`, an instance of a `Jsp-Fragment` object is created and passed in.
5. The value for the body of the tag is determined, and if a body exists the `setJsp-Body()` method is called on the tag handler.

   If the tag is declared to have a `body-content` of empty or no body or an empty body is passed for this invocation, then `setJspBody()` is not called.

Otherwise, the body of the tag is either the body of the `<jsp:body>` element, or the body of the custom action invocation if no `<jsp:body>` or `<jsp:attribute>` elements are present. In this case, an instance of a `Jsp-Fragment` object is created and it is passed to the setter. If the tag is declared to have a `body-content` of `tagdependent`, the `JspFragment` must echo the body's contents verbatim.

Otherwise, if the tag is declared to have a `body-content` of type `scriptless`, the `JspFragment` must evaluate the body's contents as a JSP scriptless 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 `Skip-PageException` 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 `javax.servlet.jsp.tagext.JspFragment` abstract class, before being passed to a tag 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 engine encounters a tag in a JSP page, a new tag handler is initialized. The `setPageContext()` and `setParent()` methods of the `javax.servlet.jsp.tagext.Tag` interface are invoked to set up the environment context for the tag handler. As a tag developer, you need not implement these methods if you extend the `TagSupport` or `BodyTagSupport` base classes.
2. The `setXXXX()` JavaBean-like methods for each tag attribute are invoked.
3. The `doStartTag()` method is invoked. You can define this method in your tag handler class to initialize your tag handler or open connections to any resources it needs, such as a database.

   At the end of the `doStartTag()` method, you can determine whether the tag body should be evaluated by returning one of the following value constants from your tag handler class:

   `SKIP_BODY`—Directs the JSP engine to skip the body of the tag. Return this value if the tag is an empty-body tag. The body-related parts of the tag's life cycle are skipped, and the next method invoked is `doEndTag()`.

   `EVAL_BODY_INCLUDE`—Directs the JSP engine to evaluate and include the content of the tag body. The body-related parts of the tag's life cycle are skipped, and the next method invoked is `doEndTag()`.

   You can only return this value for tags that implement the `Tag` interface. This allows you to write a tag that can determine whether its body is included, but is not concerned with the contents of the body. You cannot return this value if your tag implements the `BodyTag` interface (or extends the `BodyTagSuport` class).

   `EVAL_BODY_TAG`—Instructs the JSP engine to evaluate the tag body, then invokes the `doInitBody()` method. You can only return this value if your tag implements the `BodyTag` interface (or extends the `BodyTagSupport` class).
4. The `setBodyContent()` method is invoked. At this point, any output from the tag is diverted into a special `JspWriter` called `BodyContent`, and is not sent to the client. All content from evaluating the body is appended to the `BodyContent` buffer. This method allows the tag handler to store a reference to the `BodyContent` buffer so it is available to the `doAfterBody()` method for post-evaluation processing.

   If the tag is passing output to the JSP page (or the surrounding tag scope if it is nested), the tag must explicitly write its output to the parent-scoped `JspWriter` between this point in the tag life cycle and the end of the `doEndTag()` method. The tag handler can gain access to the enclosing output using the `getEnclosingWriter()` method.

   You do not need to implement this method if you are using the `BodyTagSupport` convenience class, because the tag keeps a reference to the `BodyContent` and makes the reference available through the `getBodyContent()` method.

5. The `doInitBody()` method is invoked. This method allows you to perform some work immediately before the tag body is evaluated for the first time. You might use this opportunity to set up some scripting variables, or to push some content into the `BodyContent` before the tag body. The content you prepend is *not* being evaluated as JSP—unlike the tag body content from the JSP page.

   The significant difference between performing work in this method and performing work at the end of the `doStartTag()` method (once you know you are going to return `EVAL_BODY_TAG`) is that with this method, the scope of the tag's output is nested and does not go directly to the JSP page (or parent tag). All output is now buffered in a special type of `JspWriter` called `BodyContent`.

6. The `doAfterBody()` method is invoked. This method is called after the body of the tag is evaluated and appended to the `BodyContent` buffer. Your tag handler should implement this method to perform some work based on the evaluated tag body. If your handler extends the convenience class `BodyTagSupport`, you can use the `getBodyContent()` method to access the evaluated body. If you are simply implementing the `BodyTag` interface, you should have defined the `setBodyContent()` method where you stored a reference to the `BodyContent` instance.

   At the end of the `doAfterBody()` method, you can determine the life cycle of the tag again by returning one of the following value constants:

   `SKIP_BODY`—Directs the JSP engine to continue, not evaluating the body again. The life cycle of the tag skips to the `doEndTag()` method.

   `EVAL_BODY_TAG`—Directs the JSP engine to evaluate the body again. The evaluated body is appended to the `BodyContent` and the `doAfterBody()` method is invoked again.

   At this point, you may want your tag handler to write output to the surrounding scope. Obtain a writer to the enclosing scope using the `BodyTagSupport.getPreviousOut()` method or the `BodyContent.getEnclosingWriter()` method. Either method obtains the same enclosing writer.

   Your tag handler can write the contents of the evaluated body to the surrounding scope, or can further process the evaluated body and write some other output. Because the `BodyContent` is appended to the existing `BodyContent` upon each iteration through the body, you should only write out the entire iterated body content once you decide you are going to return `SKIP_BODY`. Otherwise, you will see the content of each subsequent iteration repeated in the output.

7. The `out` writer in the `pageContext` is restored to the parent `JspWriter`. This object is actually a stack that is manipulated by the JSP engine on the `pageContext` using the `pushBody()` and `popBody()` methods. Do not, however, attempt to manipulate the stack using these methods in your tag handler.

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 the `doEndTag()` method. The previous step restored `pageContext.out` to the enclosing writer when `popBody()` was invoked on the `pageContext`.

   You can control the flow for evaluation of the rest of the JSP page by returning one of the following values from the `doEngTag()` 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. The `release()` method is invoked. This occurs just before the tag handler instance is de-referenced and made available for garbage collection.

Iteration Over a Body Tag (IterationTag Interface)

A tag that implements the `javax.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. See the Java EE Javadocs for this interface.

You can download the Javadocs at `http://www.oracle.com/technetwork/java/javaee/jsp/index.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 `javax.servlet.jsp.tagext.TryCatchFinally` interface. See the Java EE Javadocs for this interface.

You can download the Javadocs at `http://www.oracle.com/technetwork/java/javaee/jsp/index.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.

Your tag handler must implement *setter* and *getter* methods based on the attribute name, similar to the JavaBean convention. For example, if you declare an attribute named `foo`, your tag handler must define the following public methods:

```
public void setFoo(String f);
public String getFoo();
```

The first letter of the attribute name is capitalized after the set/get prefix.

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 `javax.servlet.jsp.tagext.TagExtraInfo`. For example:

```
<tei-class>weblogic.taglib.session.ListTagExtraInfo</tei-class>
```

Then write the `TagExtraInfo` class.

For example:

```
package weblogic.taglib.session;
import javax.servlet.jsp.tagext.*;
public class ListTagExtraInfo extends TagExtraInfo {

public VariableInfo[] getVariableInfo(TagData data) {
 return new VariableInfo[] {
  new VariableInfo("username","String",true,VariableInfo.NESTED)
  new VariableInfo("dob","java.util.Date",true,VariableInfo.NESTED)
   };
  }
 }
```

The example above defines a single method, `getVariableInfo()`, which returns an array of `VariableInfo` elements. Each element defines a new scripting variable. The example shown above defines two scripting variables called `username` and `dob`, which are of type `java.lang.String` and `java.util.Date`, respectively.

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

- 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` that declares whether the variable must be instantiated before use. Set this argument to "true" unless your tag handler is written in a language other than Java.
- 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", nameStr);
pageContext.setAttribute("dob", bday);
```

Where the first parameter names the scripting variable, and the second parameter is the value assigned. Here, the Java variable `nameStr` is of type `String` and `bday` is of type `java.util.Date`.

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 7.

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.

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.

```
try {
  ConnectionTag connTag = (ConnectionTag)
    findAncestorWithClass(this,
      Class.forName("weblogic.taglib.sql.ConnectionTag"));
  } catch(ClassNotFoundException cnfe) {
    throw new JspException("Query tag connection "+
                           "attribute not nested "+
                           "within connection tag");
}
```

This example returns the closest parent tag class whose tag handler class matched the class given. If the direct parent tag is not of this type, then it is parent is checked and so on until a matching tag is found, or a `ClassNotFoundException` is thrown.

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 user-written Java class that you can use to perform validation of the XML view of a JSP page. `validate(String, String, PageData)` on the validator class is invoked by the JSP compiler at translation time (when the JSP is

converted to a servlet) and returns a `null` string if the page is validated or a string containing error information it the validation fails.

To implement a Tag Library Validator:

1. Write the validator class.

   A validator class extends the `javax.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.

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

   See Packaging a JSP Tag Library as JAR File.

5

Administration and Configuration

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

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

Configuring JSP Tag Libraries

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

To configure JSP tag libraries:

1. Create a tag library descriptor (TLD).

   See Creating a Tag Library Descriptor.

2. Reference this TLD in the web application deployment descriptor, `web.xml`. For example:

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

   In this example the tag library descriptor is a file called `library.tld`. Always specify the location of the `tld` relative to the root of the Web application.

3. Place the tag library descriptor file in the `WEB-INF` directory of the Web application.

4. Reference the tag library in the JSP page.

   In your JSP, reference the tag library with a JSP directive. For example:

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

5. Place the tag handler Java class files for your tags in the `WEB-INF/classes` directory of your Web application.

6. Deploy the Web application on WebLogic Server. See Deploying and Packaging from a Split Development Directory in *Developing Applications for Oracle WebLogic Server*.

Packaging a JSP Tag Library as JAR File

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

To package a JSP tag library as a JAR file:

1. Create a TLD (tag library descriptor) file named `taglib.tld`.

   See Creating a Tag Library Descriptor.

2. Create a directory containing the compiled Java tag handler class files used in your tag library.

3. Create a subdirectory of the directory you created in step;2 and call it `META-INF`.
4. Copy the `taglib.tld` file you created in step;1 into the `META-INF` directory you created in step;3.
5. Archive your compiled Java class files into a jar file by executing the following command from the directory you created in step;2.

   ```
   jar cv0f myTagLibrary.jar
   ```

   (where *myTagLibrary.jar* is a name you provide)
6. Copy the jar file into the `WEB-INF/lib` directory of the Web application that uses your tag library.
7. Reference the tag library in your JSP.

   For example:

   ```
   <%@ taglib uri="taglib.tld" prefix="wl" %>
   ```

   A more flexible alternative would be to perform steps 1 through 6 and then reference the tag library descriptor in the Web application deployment descriptor, `web.xml`.

   For example:

   ```
   . . .
   <taglib>
   ;;;<taglib-uri>myjar.tld</taglib-uri>
   ;;;<taglib-location>
   ;;;;;;/WEB-INF/lib/myTagLibrary.jar
   ;;;</taglib-location>
   </taglib>
   . . .
   ```

   You can then reference the tag library in your JSP.

   For example:

   ```
   <%@ taglib uri="myjar.tld" prefix="wl" %>
   ```