Tag Libraries Tutorial

THE NETWORK IS THE COMPUTER™

901 San Antonio Road
Palo Alto, California 94303
650 960-1300

July 27, 2000

Please
Recycle

Contents

Preface

This tutorial describes how to use and develop JavaServer Pages™ (JSP™) tag libraries. The tutorial assumes that you know how to develop servlets and JSP pages and are familiar with packaging servlets and JSP pages into Web application archives. For information on these topics, see the resources and technical resources areas on the Sun Microsystems servlet and JSP technology Web sites.

What is a Tag Library?

In JavaServer Pages technology, *actions* are elements that can create and access programming language objects and affect the output stream. The JSP specification defines 6 standard actions that must be provided by any compliant JSP implementation.

In addition to the standard actions, JSP v1.1 technology supports the development of reusable modules called *custom actions*. A custom action is invoked by using a *custom tag* in a JSP page. A *tag library* is a collection of *custom tags*.

Some examples of tasks that can be performed by custom actions include form processing, accessing databases and other enterprise services such as email and directories, and flow control. Before the availability of custom actions, JavaBeans components in conjunction with scriplets were the main mechanism for performing such processing. The disadvantage of using this approach is that it makes JSP pages more complex and difficult to maintain.

Custom actions alleviate this problem by bringing the benefits of another level of componentization to JSP pages. Custom actions encapsulate recurring tasks so that they can be reused across more than one application and increase productivity by encouraging division of labor between library developers and library users. JSP tag libraries are created by developers who are proficient at the Java programming language and expert in accessing data and other services. JSP tag libraries are used by Web application designers who can focus on presentation issues rather than being concerned with how to access databases and other enterprise services.

Some features of custom tags are:

- They can be customized via attributes passed from the calling page.
- They have access to all the objects available to JSP pages.
- They can modify the response generated by the calling page.
- They can communicate with each other. You can create and initialize a JavaBeans component, create a variable that refers to that bean in one tag, and then use the bean in another tag.
- They can be nested within one another, allowing for complex interactions within a JSP page.

The next two sections describe the tasks involved in using and defining tags. The tutorial concludes with a discussion of two tag library examples. The examples include complete binary and source code in two Web application archives.

Using Tags

This section describes how a page author specifies that a JSP page is using a tag library and introduces the different types of tags.

Declaring Tag Libraries

You declare that a JSP page will use tags defined in a tag library by including a `taglib` directive in the page before any custom tag is used:

```
<%@ taglib uri="/tlt" prefix="tlt" %>
```

The `uri` attribute refers to a URI that uniquely identifies the tag library. This URI can be relative or absolute. If it is relative it must be mapped to an absolute location in the `taglib` element of a Web application deployment descriptor, the configuration file associated with Web applications developed according to the Java Servlet and JavaServer Pages specifications. The `prefix` attribute defines the prefix that distinguishes tags provided by a given tag library from those provided by other tag libraries.

Types of Tags

JSP custom actions are expressed using XML syntax. They have a start tag and end tag, and possibly a body:

```
<tlt:tag>
   body
</tlt:tag>
```

A tag with no body can be expressed as follows:

```
<tlt:tag />
```

Simple Tags

The following simple tag invokes an action that creates a greeting:

```
<tlt:greeting />
```

Tags With Attributes

The start tag of a custom action can contain attributes in the form `attr="value"`. Attributes serve to customize the behavior of a tag just as parameters are used to affect the outcome of executing a method on an object.

Tag attributes can be set from one or more parameters in the request object or from a `String` constant. The only types of attributes that can be set from request parameter values and `String` constants are those listed in Table 1; the conversion applied is that shown in the table. When assigning values to indexed attributes the value must be an array; the rules just described apply to the elements.

Table 1 Valid Tag Attribute Assignments

Property Type	Conversion on String Value
boolean or Boolean	As indicated in java.lang.Boolean.valueOf(String)
byte or Byte	As indicated in java.lang.Byte.valueOf(String)
char or Character	As indicated in java.lang.Character.valueOf(String)
double or Double	As indicated in java.lang.Double.valueOf(String)
int or Integer	As indicated in java.lang.Integer.valueOf(String)
float or Float	As indicated in java.lang.Float.valueOf(String)
long or Long	As indicated in java.lang.Long.valueOf(String)

An attribute value of the form `<%= scriptlet_expression %>` is computed at request time. The value of the expression depends on the type of the attribute's value, which is specified in the object that implements the tag (called a *tag handler*). Request-time expressions can be assigned to attributes of any type; no automatic conversions will be performed.

The following tag has an attribute named `date`, which accepts a `String` value obtained by evaluating the variable `today`:

```
<tlt:greeting date="<%= today %>" />
```

Tags With a Body

A tag can contain custom and core tags, scripting elements, HTML text, and tag-dependent body content between the start and end tag. In the following example, the date information is provided in the body of the tag, instead of as an attribute:

```
<tlt:greeting>
   <%= today %>
</tlt:greeting>
```

Choosing Between Passing Information as Attributes or Body

As shown in the last two sections, it is possible to pass a given piece of data as an attribute of the tag or to the tag's body. Generally speaking, any data that is a simple string or can be generated by evaluating a simple expression is best passed as an attribute.

Tags That Define Scripting Variables

A tag can define a variable that can be used in scripts within a page. The following example illustrates how to define and use a scripting variable that contains an object returned from a JNDI lookup. Examples of such objects include enterprise beans, transactions, databases, environment entries, and so on:

```
<tlt:lookup id="tx" type="UserTransaction"
   name="java:comp/UserTransaction" />
<% tx.begin(); %>
```

Cooperating Tags

Tags cooperate with each other by means of shared objects.

In the following example, `tag1` creates a named object called `obj1`, which is then reused by `tag2`. The convention encouraged by the JSP specification is that a tag with attribute named `id` creates and names an object and the object is then referenced by other tags with an attribute named `name`.

```
<tlt:tag1 id="obj1" attr2="value" />
<tlt:tag2 name="obj1" />
```

In the next example, an object created by the enclosing tag of a group of nested tags is available to all inner tags. Since the object is not named, the potential for naming conflicts is reduced. The following example illustrates how a set of cooperating nested tags would appear in a JSP page.

```
<tlt:outerTag>
   <tlt:innerTag />
</tlt:outerTag>
```

Defining Tags

To define a tag, you need to:

- Develop a tag handler and helper classes for the tag
- Declare the tag in a tag library descriptor

This section describes the properties of tag handlers and tag library descriptors and explains how to develop tag handlers and library descriptor elements for each type of tag introduced in the previous section.

Tag Handlers

A *tag handler* is an object invoked by a JSP container to evaluate a custom tag during the execution of the JSP page that references the tag. Tag handler methods are called by the JSP page implementation class at various points during the evaluation of the tag.

When the start tag of a custom tag is encountered, the JSP page implementation class calls methods to initialize the appropriate handler and then invokes the handler's `doStartTag` method. When the custom end tag is encountered, the handler's `doEndTag` method is invoked. Additional methods are invoked in between when a tag handler needs to interact with the body of the tag. For further information, see "How Is a Tag Handler Invoked?" on page 35.

In order to provide a tag handler implementation, you must implement the methods that are invoked at various stages of processing the tag. The methods are summarized in Table 2.

Table 2 Tag Handler Methods

Tag Handler Type	Methods
Simple	`doStartTag, doEndTag, release`
Attributes	`doStartTag, doEndTag, set/getAttribute1...N`
Body, No Interaction	`doStartTag, doEndTag, release`
Body, Interaction	`doStartTag, doEndTag, release, doInitBody, doAfterBody`

A tag handler has access to an API that allows it to communicate with the JSP page. The entry point to the API is the page context object through which a tag handler can access to all the other implicit objects (request, session, and application) accessible from a JSP page. Implicit objects can have attributes associated with them. Such attributes are accessed using the appropriate `[set/get]Attribute` method.

If the tag is nested, a tag handler also has access to the handler (called the *parent*) associated with the enclosing tag.

Tag handlers must implement either the `Tag` or `BodyTag` interfaces. Interfaces can be used to take an existing Java object and make it a tag handler. For newly created handlers, you can use the `TagSupport` and `BodyTagSupport` classes as base classes. You can download documentation that describes these interfaces and classes from the JSP specification download page.

Tag Library Descriptors

A *tag library descriptor* (TLD) is an XML document that describes a tag library. A TLD contains information about a library as a whole and about each tag contained in the library. TLDs are used by a JSP container to validate the tags and by JSP development tools.

The following TLD elements are used to define a tag library:

`<taglib>`

`<tlibversion>` - The tag library's version

`<jspversion>` - The JSP specification version the tag library depends on

`<shortname>` - A simple default name that could be used by a JSP page authoring tool to create names with a mnemonic value; for example, `shortname` may be used as the preferred prefix value in taglib directives and/or to create prefixes for IDs.

`<uri>` - A URI that uniquely identifies the tag library

`<info>` - Descriptive information about the tag library

```
    <tag>
    ...
    </tag>
    ...
</taglib>
```

The TLD element required for all tags is the one used to specify a tag handler's class:

```
    <tag>
       <tagclass>classname</tagclass>
       ...
    </tag>
```

The following sections will describe the methods and tag library descriptor elements that you need to develop for each type of tag introduced in "Using Tags" on page 6.

Simple Tags

Tag Handlers

The handler for a simple tag must implement the `doStartTag` and `doEndTag` methods of the `Tag` interface. The `doStartTag` method is invoked when the start tag is encountered. This method returns `SKIP_BODY` because a simple tag has no body. The `doEndTag` method is invoked when the end tag is encountered. The `doEndTag` method needs to return `EVAL_PAGE` if the rest of the page needs to be evaluated; otherwise it should return `SKIP_PAGE`.

The following simple tag:

```
    <tlt:simple />
```

would be implemented by the following tag handler:

```
public SimpleTag extends Tag Support {
    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().print("Hello.");
        } catch (Exception ex) {
            throw new JspTagException("SimpleTag: " +
                e.getMessage());
        }
        return SKIP_BODY;
    }
    public int doEndTag() {
        return EVAL_PAGE;
    }
}
```

TLD `bodycontent` Element

Tags without bodies must declare that their body content is empty:

```
<tag>
    ...
    <bodycontent>empty</bodycontent>
</tag>
```

Tags With Attributes

Defining Attributes in a Tag Handler

For each tag attribute, you must define a property and JavaBeans style get and set methods in the tag handler. For example, the tag handler for the tag

```
<tlt:twa attr1=”value1”>
```

where `value1` is of type `AttributeClass`, must contain the following declaration and methods:

```
private AttributeClass attr1;
setAttr1(AttributeClass ac) { ... }
```

```
AttributeClass getAttr1() { ... }
```

Note that if your attribute is named `id`, and your tag handler inherits from the `TagSupport` class, you do not need to define the property and set and get methods because these are already defined by `TagSupport`.

A tag attribute whose value is a `String` can name an attribute of one of the implicit objects available to tag handlers. An implicit object attribute would be accessed by passing the tag attribute value to the `[set/get]Attribute` method of the implicit object. This is a good way to pass scripting variable names to a tag handler where they are associated with objects stored in the page context (See "Tags That Define Scripting Variables" on page 17).

TLD `attribute` Element

For each tag attribute you must specify whether the attribute is required, and whether the value can be determined by an expression:

```
<tag>
   ...
   <attribute>
        <name>attr1</name>
        <required>true|false|yes|no</required>
        <rtexprvalue>true|false|yes|no</rtexprvalue>
   </attribute>
</tag>
```

If a tag attribute is not required, a tag handler should provide a default value.

Attribute Validation

The documentation for a tag library should describe valid values for tag attributes. When a JSP page is translated, a JSP container will enforce any constraints contained in the TLD element for each attribute.

The attributes passed to a tag can also be validated at translation time with the `isValid` method of a class derived from `TagExtraInfo`. This class is also used to provide information about scripting variables defined by the tag (See "Tags That Define Scripting Variables" on page 17).

The `isValid` method is passed the attribute information in a `TagData` object, which contains attribute-value tuples for each of the tag's attributes. Since the

validation occurs at translation time, the value of an attribute that is computed at request time will be set to `TagData.REQUEST_TIME_VALUE`.

The tag `<tlt:twa attr1="value1" />` has the following TLD `attribute` element:

```
<attribute>
   <name>attr1</name>
   <required>true</required>
   <rtexprvalue>true</a>
</attribute
```

This declaration indicates that the value of `attr1` can be determined at runtime.

The following `isValid` method checks that the value of `attr1` is a valid boolean value. Note that since the value of `attr1` can be computed at runtime, `isValid` must check whether the tag user has chosen to provide a runtime value.

```
public class TwaTEI extends TagExtraInfo {
   public boolean isValid(Tagdata data) {
        Object o = data.getAttribute("attr1");
        if (o != null && o != TagData.REQUEST_TIME_VALUE) {
             if (o.toLowerCase().equals("true") ||
                  o.toLowerCase().equals("false") )
                  return true;
             }
             else
                  return false;
        }
        else
             return false;
        }
   }
}
```

Tags With a Body

Tag Handlers

A tag handler for a tag with a body is implemented differently depending on whether the tag handler needs to interact with the body or not. By interact, we mean that the tag handler reads or modifies the contents of the body or causes iterative evaluation of the body.

Tags That Do Not Interact With the Body. If the tag handler does not need to interact with the body, the tag handler should implement the `Tag` interface (or be derived from `TagSupport`). If the body of the tag needs to be evaluated, the `doStartTag` method needs to return `EVAL_BODY_INCLUDE`; otherwise it should return `SKIP_BODY`.

Tags That Interact With the Body. If the tag handler needs to interact with the body, the tag handler must implement `BodyTag` (or be derived from `BodyTagSupport`). Such handlers typically implement the `doInitBody` and the `doAfterBody` methods. These methods interact with body content passed to the tag handler by the JSP page implementation class.

A body content supports several methods to read and write its contents. A tag handler can use the body content's `getString` or `getReader` methods to extract information from the body and the `writeOut(out)` method to write the body contents to an out stream. The writer supplied to the `writeOut` method is obtained using the tag handler's `getPreviousOut` method. This method is used to ensure that a tag handler's results are available to an enclosing tag handler.

If the body of the tag needs to be evaluated, the `doStartTag` method needs to return `EVAL_BODY_TAG`; otherwise it should return `SKIP_BODY`.

`doInitBody` Method

The `doInitBody` method is called after the body content is set but before it is evaluated. You generally use this method to perform any initialization that depends on the body content.

`doAfterBody` Method

The `doAfterBody` method is called *after* the body content is evaluated.

Like the `doStartTag` method, `doAfterBody` must return an indication of whether to continue evaluating the body. Thus, if the body should be evaluated again, as would be the case if you were implementing an iteration tag, `doAfter-`

`Body` should return `EVAL_BODY_TAG`; otherwise `doAfterBody` should return `SKIP_BODY`.

`release` Method

A tag handler should reset its state and release any private resources in the `release` method.

The following example reads the content of the body (which contains an SQL query) and passes it to a object that executes the query. Since the body does not need to be reevaluated, `doAfterBody` returns `SKIP_BODY`.

```
public class QueryTag extends BodyTagSupport {
    public int doAfterBody() throws JspTagException {
        BodyContent bc = getBodyContent();
        // get the bc as string
        String query = bc.getString();
        // clean up
        bc.clearBody();
        try {
            Statement stmt = connection.createStatement();
            result = stmt.executeQuery(query);
        } catch (SQLException e) {
            throw new JspTagException("QueryTag: " +
                e.getMessage());
        }
        return SKIP_BODY;
    }
}
```

The following example reads the content of the body, transforms that content, and then writes the modified version to the out stream.

```
public class TransformTag extends BodyTagSupport {
    public int doAfterBody() throws JspTagException {
        BodyContent bc = getBodyContent();
        String body = bc.getString();
        bc.clearBody();
        try {
```

```
                getPreviousOut().print(body.transform());
            } catch (IOException e) {
                throw new JspTagException("TransformTag: " +
                    e.getMessage());
            }
            return SKIP_BODY;
        }
    }
```

TLD `bodycontent` Element

For tags that have a body, you must specify the type of the body content:

```
<tag>
    ...
    <bodycontent>JSP|tagdependent</bodycontent>
</tag>
```

Body content containing custom and core tags, scripting elements, and HTML text is categorized as `JSP`; all other types of body content are `tagdependent`. Note that the value of this element does not affect the interpretation of the body. The `bodycontent` element is only intended to be used by an authoring tool to present the content of the body.

Tags That Define Scripting Variables

Tag Handlers

A tag handler is responsible for creating and setting the object referred to by the scripting variable into a context accessible from the page. It does this by using the `pageContext.setAttribute(name, value, scope)` or `pageContext.setAttribute(name, value)` methods. Typically an attribute passed to the custom tag specifies the `name` of the scripting variable object; this name can be retrieved by invoking the attribute's get method described in "Defining Attributes in a Tag Handler" on page 12.

If the value of the scripting variable is dependent on an object present in the tag handler's context it can retrieve the object using the `pageContext.getAttribute(name, scope)` method.

The usual procedure is that the tag handler retrieves a scripting variable value object, performs some processing on the object, and then sets the scripting variable's value using the `pageContext.setAttribute(name, object)` method.

The scope that an object can have is summarized in Table 3. The scope constrains the accessibility and lifetime of the object.

Table 3 Scope of Objects

Name	Accessible From	Lifetime
`page`	Current page	Until the response has been sent back to the user or the request is passed to a new page
`request`	Current page and any included or forwarded pages	Until the response has been sent back to the user
`session`	Current request and any subsequent request from the same browser (subject to session lifetime).	The life of the user's session
`application`	Current and any future request from the same Web application	The life of the application

In addition to setting the value of the variable within the tag handler, you must define a class derived from `TagExtraInfo` that provides information to the JSP container about the nature of the variable. A `TagExtraInfo` must implement the method `getVariableInfo` to return an array of `VariableInfo` objects containing the following information:

- Variable name
- Variable class
- Whether the variable refers to a new or existing object value.
- The availability of the variable

Table 4 describes the availability of the scripting variable and the methods where the value of the variable must be set or reset.

Table 4 Scripting Variable Availability

Value	Availability	Methods
NESTED	Between the start tag and the end tag.	In doInitBody and doAfterBody for a tag handler implementing BodyTag; otherwise in doStartTag.
AT_BEGIN	From the start tag until the end of the page.	In doInitBody, doAfterBody, and doEndTag for a tag handler implementing BodyTag; otherwise in doStartTag and doEndTag.
AT_END	After the end tag until the end of the page.	In doEndTag.

The JSP container passes a parameter called data to the getVariableInfo method that contains an attribute-value tuples for each of the tag's attributes. These attributes can be used to provide the VariableInfo object with a scripting variable's name and class.

Recall the scripting variable example described in the first section:

```
<tlt:lookup id="tx" type="UserTransaction"
   name="java:comp/UserTransaction" />
<% tx.begin(); %>
```

The object retrieved from the JNDI lookup is stored as a page context attribute with the name of the scripting variable.

```
public LookupTag extends TagSupport {
   private String type;
   private String name;
   public int doStartTag() {
        return SKIP_BODY;
   }
   public int doEndTag() throws JspException {
   try {
        InitialContext context = new InitialContext();
```

```
            Object obj = (Object)context.lookup(name);
            pageContext.setAttribute(getId(), obj);
            } catch(javax.naming.NamingException e) {
                throw new JspException("Unable to look up " + name
                    + " due to " + e.getMessage());
            }
            return EVAL_PAGE;
        }
    }
```

The scripting variable `tx` is defined in the following tag extra info class. Since the name (`tx`) and class (`UserTransaction`) of the scripting variable were passed in as tag attributes, they are retrieved with the `data.getAttributeString` method and used to fill in the `VariableInfo` constructor. To allow the scripting variable `tx` to be used in the rest of the page, the scope of `tx` is set to be `AT_END`.

```
public class LookupTagTEI extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data) {
        VariableInfo info1
            = new VariableInfo(
                data.getAttributeString("id"),
                data.getAttributeString("type"),
                true,
                VariableInfo.AT_END);
        VariableInfo[] info = { info1 } ;
        return info;
    }
}
```

TLD `teiclass` Element

The `TagExtraInfo` class defined for each scripting variable must be declared in the tag library descriptor as follows:

```
<tag>
    ...
    <teiclass>LookupTagTEI<teiclass>
</tag>
```

Cooperating Tags

Tags cooperate by sharing objects. JSP technology supports two styles of object sharing.

The first style requires that a shared object be named and stored in the page context (one of the implicit objects accessible to both JSP pages and tag handlers). To access objects created and named by another tag, a tag handler uses the `pageContext.getAttribute(name, scope)` method.

In the second style of object sharing, an object created by the enclosing tag handler of a group of nested tags is available to all inner tag handlers. This form of object sharing has the advantage that it uses a private namespace for the objects, thus reducing the potential for naming conflicts.

To access an object created by an enclosing tag, a tag handler must first obtain its enclosing tag with the static method `TagSupport.findAncestorWithClass(from, class)` or the `TagSupport.getParent()` method. The former method should be used when a specific nesting of tag handlers cannot be guaranteed. Once the ancestor has been retrieved, a tag handler can access any statically or dynamically created objects. Statically created objects are members of the parent. Private objects can also be created dynamically created. Such objects can be stored in a tag handler with the `setValue` method and retrieved with the `getValue method`.

The following example illustrates a tag handler that supports both the named and private object approaches to sharing objects. In the example, the handler for a query tag checks whether an attribute named `connection` has been set in the `doStartTag` method. If the connection attribute has been set, the handler retrieves the connection object from the page context. Otherwise, the tag handler first retrieves the tag handler for the enclosing tag, and then retrieves the connection object from that handler.

```
public class QueryTag extends BodyTagSupport {
   private String connectionId;
   public int doStartTag() throws JspException {
        String cid = getConnection();
        if (cid != null) {
        // there is a connection id, use it
             connection =(Connection)pageContext.
                  getAttribute(cid);
        } else {
```

```
                ConnectionTag ancestorTag =
                    (ConnectionTag)findAncestorWithClass(this,
                        ConnectionTag.class);
                if (ancestorTag == null) {
                    throw new JspTagException("A query without
                        a connection attribute must be nested
                        within a connection tag.");
                }
                connection = ancestorTag.getConnection();
            }
        }
    }
```

The query tag implemented by this tag handler could be used in either of the following ways:

```
<tlt:connection id="con01" ....> ... </tlt:connection>
<tlt:query id="balances" connection="con01">
    SELECT account, balance FROM acct_table
        where customer_number = <%= request.getCustno()%>
</tlt:query>

<tlt:connection ...>
    <x:query id="balances">
        SELECT account, balance FROM acct_table
            where customer_number = <%= request.getCustno()%>
    </x:query>
</tlt:connection>
```

The tag library descriptor for the tag handler must indicate that the connection attribute is optional with the following declaration:

```
<tag>
    <attribute>
    <name>connection</name>
    <required>false</required>
</tag>
```

Examples

The examples described in this section demonstrate solutions to two recurring problems in developing JSP applications: minimizing the amount of Java programming in JSP pages and ensuring a common look and feel across applications. In doing so, they illustrate many of the styles of tags discussed in the first section.

The complete binary and source code for the examples is contained in two Web application archives: `iteration.war` and `template.war`. You can unpack the archives with the command `jar xvf` *webapp*`.war`.

When an archive is unpacked, its contents are deposited into the directories listed in the following table. This directory layout is required by the Java Servlet specification and is one that you usually will use while developing an application.

Table 5 Web Application Directory Structure

Directory	Contents
webapp	JSP and HTML files
webapp`/WEB-INF`	`web.xml` (Web application deployment descriptor) and `taglib.tld` (tag library descriptor)
webapp`/WEB-INF/classes`	classes accessed by JSP files and servlet implementations
webapp`/WEB-INF/lib`	JAR files containing the binary and source of tag library handler and tag extra info classes

You can run the examples on Tomcat, the freely available implementation of the Java Servlet and JavaServer Pages technologies, by performing the following steps:

1. Install Tomcat.
2. Download the Web application archives into the directory *TOMCAT_HOME*`/webapps`. When an archived Web application is accessed, Tomcat 3.2 automatically unpacks it into the directory *TOMCAT_HOME*`/docBase` (where `docBase` is the application directory specified in `server.xml`) and adds the context for each archive to the server startup file. If you are using an earlier version of Tomcat you will

need to add the following lines to the file *TOMCAT_HOME*/conf/server.xml:

```
<Context    path="/iteration"    docBase="webapps/iteration"
debug="0" reloadable="true" />
<Context     path="/template"     docBase="webapps/template"
debug="0" reloadable="true" />
```

3. Start Tomcat.
4. Invoke the examples by following the links:

```
http://localhost:8080/iteration

http://localhost:8080/template/example/home
```

An Iteration Tag

Constructing page content dependent on dynamically generated data often requires the use of flow control scripting statements. By moving the flow control logic to tag handlers, flow control tags reduce the amount of scripting needed in JSP pages.

The `iteration` tag retrieves objects from a collection stored in a JavaBeans component and assigns them to a scripting variable. The body of the tag retrieves information from the scripting variable. While elements remain in the collection, the `iteration` tag causes the body to be reevaluated.

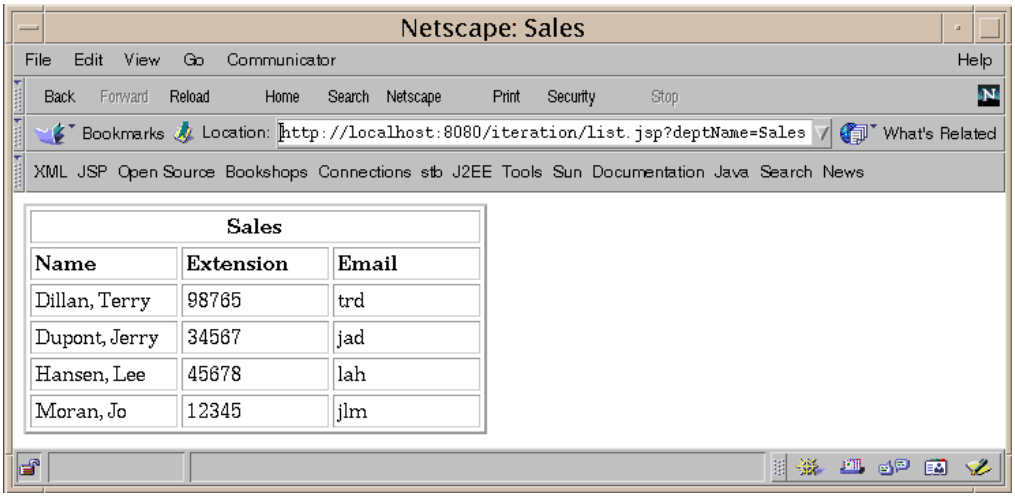
JSP Page

The iteration example application contains two JSP pages that uses the iterator tag; one of the pages, `index.jsp`, is shown below. The page initializes the iteration tag with a collection maintained by a JavaBeans component that represents an organization. The iteration tag populates a table with the names of departments in the organization. The other jsp page, `list.jsp`, uses the iterator tag to display the members of a selected department.

```
<%@ taglib uri="/tlt" prefix="tlt" %>
<html>
   <head>
   <title>Organization</title>
   </head>
   <body bgcolor="white">
   <jsp:useBean id="org" class="Organization"/>
```

```
    <table border=2 cellspacing=3 cellpadding=3>
    <tlt:iteration name="departmentName" type="String"
        group="<%= org.getDepartmentNames()%>">
        <tr>
            <td><b>Departments</b></td>
        </tr>
        <tr>
            <td><a href="list.jsp?deptName=
                <%= departmentName %>">
                <%= departmentName %></a></td>
        </tr>
        </tlt:iteration>
    </table>
    </body>
</html>
```

The following figure shows the result of executing `list.jsp`:



Tag Handler

The iteration tag uses an iterator initialized from the collection provided via the `group` tag attribute. If the iterator contains more elements, `doStartTag` sets the value of the scripting variable to the next element and then indicates that the body should be evaluated.

After the body has been evaluated, the doAfterBody method retrieves the body content and writes it to the out stream. The body content is cleared in preparation for another body evaluation. If the iterator contains more elements, doAfterBody again sets the value of the scripting variable to the next element and indicates that the body should be evaluated again, which causes the reexecution of doAfterBody. When there are no remaining elements, doAfterBody terminates the process by returning SKIP_BODY.

```
private Iterator iterator;
public void setGroup(Collection members) {
    if(members.size() > 0)
        iterator = members.iterator();
}

public int doStartTag() {
    if(iterator == null) {
        return SKIP_BODY;
    }
    if(iterator.hasNext()) {
        pageContext.setAttribute(name, iterator.next());
        return EVAL_BODY_TAG;
    } else {
        return SKIP_BODY;
    }
}

public int doAfterBody() throws JspTagException {
    BodyContent body = getBodyContent();
    try {
        body.writeOut(getPreviousOut());
    } catch (IOException e) {
        throw new JspTagException("IterationTag: " +
            e.getMessage());
    }

    // clear up so the next time the body content is empty
    body.clearBody();
```

```
    if (iterator.hasNext()) {
        pageContext.setAttribute(name, iterator.next());
        return EVAL_BODY_TAG;
    } else {
        return SKIP_BODY;
    }
}
```

Tag Extra Info Class

The scripting variable is defined in the following tag extra info class. Since the name (`member`) and class (`Member`) of the scripting variable were passed in as tag attributes, they are retrieved with the `data.getAttributeString method` and used to fill in the `VariableInfo` constructor.

```
public class IterationTEI extends TagExtraInfo {
    ...
    public VariableInfo[] getVariableInfo(TagData data) {
        VariableInfo info1
            = new VariableInfo(
                data.getAttributeString("name"),
                data.getAttributeString("type"),
                true,
                VariableInfo.NESTED);
        VariableInfo [] info = { info1 };
        return info;
    }
}
```

A Template Mechanism

A template mechanism provides a way to separate the common elements that are part of each screen from the elements that change with each screen of an application. Putting all the common elements together into one file makes it easier to maintain and enforce a consistent look and feel in all the screens. It also makes development of individual screens easier since the designer can focus on portions of a screen that are specific to that screen while the template takes care of the rest.

The template is a JSP page, with place holders for the parts that need to change with each screen. Each of these place holders is referred to as a parameter of the template. For example, a simple template could include a title parameter for the top of the generated screen and a body parameter to refer to a JSP page for the custom content of the screen.

Once you have a template, you can generate different presentation screens from it simply by passing it different parameters.

JSP Page

The entry page of the example, `main.jsp`, is shown below. The first part of the page uses a set of nested tags—`definition`, `screen`, and `parameter`—to define a table of screen definitions for an application and select a specific definition based on the request attribute `selectedScreen`.

```
<%@ taglib uri="/tlt" prefix="tlt" %>
<tlt:definition name="tutorial"
   screen="<%= (String)request.
        getAttribute(\"selectedScreen\") %>">
   <tlt:screen id="/home">
        <tlt:parameter parameter="title" value="Home Page"
             direct="true"/>
        <tlt:parameter parameter="banner" value="/banner.jsp"
             direct="false"/>
        <tlt:parameter parameter="body" value="/home.jsp"
             direct="false"/>
   </tlt:screen>
   <tlt:screen id="/first">
        <tlt:parameter parameter="title" value="First Page"
             direct="true"/>
        ...
   <tlt:screen id="/second">
        ...
</tlt:definition>
```

The second part of the page uses the `insert` tag to insert parameters from the selected definition into the application screen.

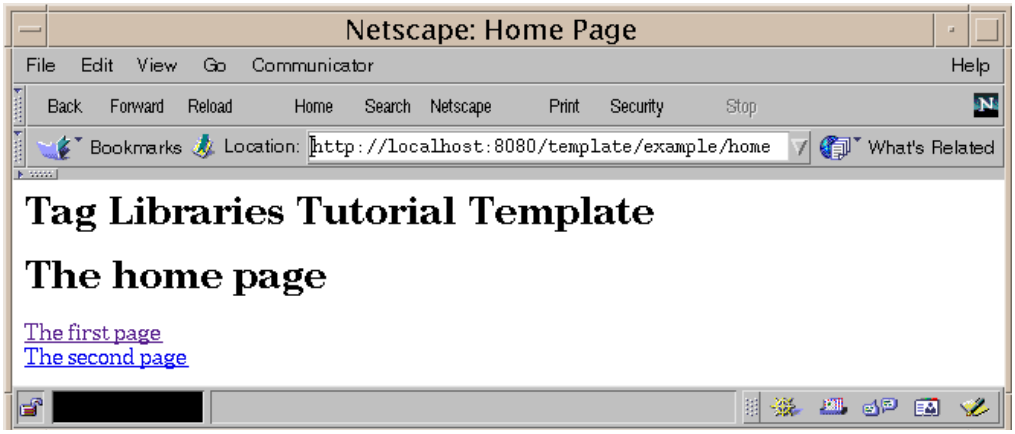
```
<html>
```

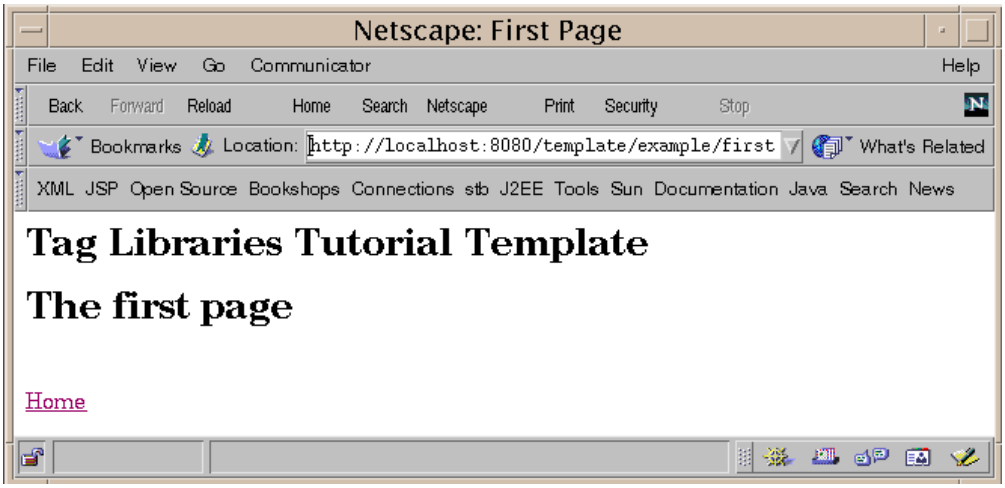
```
    <head>
        <title>
            <tlt:insert definition="tutorial"
                parameter="title"/>
        </title>
    </head>
    <body bgcolor="white">
        <tlt:insert definition="tutorial" parameter="banner"/>
        <tlt:insert definition="tutorial" parameter="body"/>
    </body>
</html>
```

The template is instantiated by the `Dispatcher` servlet. This servlet first gets requested screen and stores as an attribute of the request. This is necessary because when the request is forwarded to `main.jsp`, the request URL info no longer reflects the original request (`/home/example/*.jsp`), but instead reflects the path (`/template/main.jsp`) of the forwarded page. Finally the servlet dispatches the request to `main.jsp`:

```
public class Dispatcher extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException {
        request.setAttribute("selectedScreen",
            request.getPathInfo());
        try {
            getServletConfig().
            getServletContext().getRequestDispatcher(
            "/main.jsp").forward(request, response);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

The following figures show the home and first pages of the application:





Tag Handlers

The template tag library contains four tag handlers: `DefinitionTag`, `ScreenTag`, `ParameterTag`, and `InsertTag`. These tag handlers demonstrate the use of cooperating tags. `DefinitionTag`, `ScreenTag`, and `ParameterTag` comprise a set of nested tags handlers in which private objects are shared between parent and child tags. `DefinitionTag` creates a named object called `definition` that is used by `InsertTag`.

In `doStartTag`, `DefinitionTag` creates a private object that contains a hashtable of screen definitions. A screen definition consists of a screen identifier and a set of parameters associated with the screen. The table of screen definitions is filled

in by `ScreenTag` and `ParameterTag` from text provided as attributes to these tags. Table 6 shows the contents of the screen definitions hashtable for the JSP page discussed in the previous section.

Table 6 Screen Definitions

Screen ID	Title	Banner	Body
/home	Home Page	/banner.jsp	/home.jsp
/first	First Page	/banner.jsp	/first.jsp
/second	Second Page	/banner.jsp	/second.jsp

In `doEndTag`, `DefinitionTag` creates a public object of class `Definition`, selects a screen definition based on the URL passed in the request, and uses the definition to initialize the `Definition`. If the URL passed in the request is /home, the `Definition` contains the items from the first row of Table 6:

Title	Banner	Body
Home Page	/banner.jsp	/home.jsp

```
public int doStartTag() {
   Hashtable screens = null;
   try {
   // look for the screens object or create if it does not exist
        screens = (Hashtable) getValue("screens");
        if (screens == null)
             setValue("screens", new Hashtable());
        else
             ...
   } catch (Exception e) {
        ...
   }
   return EVAL_BODY_INCLUDE;
}

public int doEndTag()throws JspTagException {
   try {
```

```
            Definition definition = new Definition();
            Hashtable screens = null;
            ArrayList params = null;
            TagSupport screen = null;
            if (getValue("screens") != null)
                screens = (Hashtable) getValue("screens");
            if (screens != null)
                params = (ArrayList) screens.get(screenId);
            else
                ...
            if (params == null)
                ...
            Iterator ir = null;
            if (params != null)
                ir = params.iterator();
            while ((ir != null) && ir.hasNext())
                definition.setParam((Parameter) ir.next());
                // put the definition in the page context
            pageContext.setAttribute(
                definitionName, definition);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return EVAL_PAGE;
    }
```

`InsertTag` uses the `Definition` object to insert parameters of the screen definition into the response. First it retrieves the definition object from the page context. The `isDirect` parameter attribute determines whether the parameter value is directly inserted into the response or treated as the name of a JSP file which is dynamically included into the response.

The definition for the URL `/home` is shown below. The definition specifies that the value of the `Title` parameter should be inserted directly into the output stream, but the values of `Banner` and `Body` should be dynamically included.

Parameter Name	title	banner	body
Parameter Value	Home Page	/banner.jsp	/home.jsp
IsDirect	true	false	false

```
public int doStartTag() {
   // get the definition from the page context
   try {
        definition = (Definition) pageContext.
            getAttribute(definitionName);
   } catch (NullPointerException e) {
        ...
   }
   // get the parameter
   if (parameterName != null && definition != null)
        parameter = (Parameter)definition.
            getParam(parameterName);
   if (parameter != null)
        directInclude = parameter.isDirect();
   return SKIP_BODY;
}

public int doEndTag()throws JspTagException {
   // flush data
   try {
        pageContext.getOut().flush();
   } catch (Exception e) {
        ...
   }
   try {
        // if parameter is direct, print to out
        if (directInclude && parameter != null)
```

```
                pageContext.getOut().print(parameter.getValue());
            // if parameter is indirect,
            // include results of dispatching to page
            else {
                if ((parameter != null) &&
                    (parameter.getValue() != null))
                    pageContext.getRequest().
                    getRequestDispatcher(
                        parameter.getValue()).include(
                        pageContext.getRequest(),
                        pageContext.getResponse());
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return EVAL_PAGE;
    }
```

Web Application Deployment Descriptor

The `Dispatcher` servlet is used to forward requests to the application template. The `servlet-mapping` element in the Web application deployment descriptor maps all URL patterns of the form `/example/*` to the dispatcher servlet. The `servlet` element then maps the dispatcher servlet to an instance of the `Dispatcher` class. The `taglib` element maps the logical name `/tlt` to the absolute location of the tag library descriptor. This allows a page author to use the logical name in the `taglib` page directive.

```
<servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/example/*</url-pattern>
</servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>Dispatcher</servlet-class>
</servlet>
<taglib>
    <taglib-uri>/tlt</taglib-uri>
```

```
    <taglib-location>/WEB-INF/taglib.tld</taglib-location>
</taglib>
```

How Is a Tag Handler Invoked?

The Tag interface defines the basic protocol between a tag handler and JSP page implementation class. It defines the life cycle and the methods to be invoked when the start and end tag of an action are encountered.

The JSP page implementation class invokes the setPageContext, setParent, and attribute setting methods before calling doStartTag. The JSP page implementation class also guarantees that release will be invoked on the tag handler before the end of the page.

Here is a typical tag handler method invocation sequence:

```
ATag t = new ATag();
t.setPageContext(...);
t.setParent(...);
t.setAttribute1(value1);
t.setAttribute2(value2);
t.doStartTag();
t.doEndTag();
t.release();
```

The BodyTag interface extends Tag by defining additional methods that let a tag handler access its body. The interface provides three new methods:

setBodyContent - creates body content and adds to tag handler

doInitBody - called before evaluation of tag body

doAfterBody - called after evaluation of tag body

A typical invocation sequence is:

```
t.doStartTag();
out = pageContext.pushBody();
t.setBodyContent(out);
// perform any initialization needed after body content is set
t.doInitBody();
t.doAfterBody();
```

```
// while doAfterBody returns EVAL_BODY_TAG we
// iterate body evaluation
...
t.doAfterBody();
t.doEndTag();
t.pageContext.popBody();
t.release();
```