

Understanding the New Features of JSP 2.0

JSP 2.0 is an upgrade to JSP 1.2 with several new and interesting features that make the life of web application developers and designers easier. The objective of JSP 2.0 is to make JSP easier to use than ever, and more importantly to be used without having to learn the Java programming language itself. This tutorial is aimed at helping you understand some of the new features of JSP 2.0. Specifically, we will be looking at the following features - the Simple Expression Language (EL), JSP Fragments, Tag files and Simple Tag handlers.

Contents

1. [Simple Expression Language \(EL\)](#)
2. [JSP Fragments](#)
3. [Tag Files](#)
4. [Simple Tag handlers](#)
5. [Resources](#)
6. [Feedback](#)

Simple Expression Language

Sun Microsystems introduced the Servlet API, in the later half of 1997, positioning it as a powerful alternative for CGI developers who were looking around for an elegant solution that was more efficient and portable than CGI (Common Gateway Interface) programming. However, it soon became clear that the Servlet API had its own drawbacks, with developers finding the solution difficult to implement, from the perspective of code maintainability and extensibility. It is in some ways, this drawback that prompted the community to explore a solution that would allow embedding Java Code in HTML - JavaServer Pages (JSP) emerged as a result of this exploration.

It was not long before that developers realized the incomprehensibility and lack of maintainability associated with complex JSP pages that mixed presentation and business logic together. Another issue that surfaced for page authors, who could not write scriptlets, was JSP's limitation in terms of standard tag sets. These limitations set the ball rolling to create JSP custom tags, leveraging on JSP's mechanism for implementing custom tags.

The JSP Standard Tag Library (JSTL) is a collection of custom tag libraries that encapsulates, as simple tags, core functionality common to many JSP applications. It eliminates the need to use JSP scriptlets and expressions and uses a higher-level syntax for expressions. It also implements general-purpose functionality such as iteration and conditionalization, data management formatting, manipulation of XML, database access, internationalization and locale-sensitive formatting tags, and SQL tags. JSTL 1.0 introduced the concept of the EL but it was constrained to only the JSTL tags. With JSP 2.0 you can use the EL with template text and even get programmatic access via `javax.servlet.jsp.el`.

Following from our understanding of how JSTL fits into the picture, and the constraints associated with the JSTL expression language, we will look at one of the important nuggets of JSP 2.0 - the JSP Expression Language (EL). We will specifically cover the following:

[JSP Expression language defined](#)
[Mechanisms to enable EL in scriptless JSP pages](#)
[Expression language Syntax](#)
[Valid Expressions in the JSP EL](#)
[Using EL Expressions](#)

JSP Expression language defined

The Expression Language, inspired by both ECMAScript and the XPath expression languages, provides a way to simplify expressions in JSP. It is a simple language that is based on available namespace (the PageContext attributes), nested properties and accessors to collections, operators - arithmetic, relational and logical, extensible functions mapping into static methods in Java classes, and a set of implicit objects.

EL provides the ability to use run-time expressions outside JSP scripting elements. Scripting elements are the elements in a page that can be used to embed Java code in the JSP file. They are commonly used for object manipulation and performing computation that affects the generated content. JSP 2.0 adds EL expressions as a scripting element.

Scripting elements have three subforms:

- Declaration
- Scriptlets
- Expressions.

Let's look at these three subforms in code:

```
<%! int i = 1; %> <% -- Declaration --%>  
<% for (int i =0; i < 10; i++) { %> <% -- Scriptlets --%>  
<jsp:expression> table.getColumn( ) </jsp:expression> <% -- Expression --%>
```

With the addition of EL to the JSP toolkit, the above code can be written using a simpler syntax yet achieving the same results as the JSP elements above. Another advantage of EL expressions is its use in scriptless JSP pages that do not permit the usage of any of the above scripting element subforms. However, it must be noted that JSP pages can be written without the usage of any of the three scripting element subforms, and that the choice of whether a JSP page should be scriptless is entirely based on the requirements and needs of your application.

If you want a clear separation between your presentation and business logic, then you also have the choice to force the page to go scriptless. By enforcing scriptless pages, the dynamic behavior of JSP pages must be provided through other elements such as JavaBeans, EL expressions, Custom actions and standard tag libraries.

Mechanisms to enable EL in scriptless JSP pages

There are two mechanisms that are available to ensure that a page does not contain any scripting elements. Each of these mechanisms also provide a way to enable EL in scriptless JSP pages.

- **Using a page directive:**

When using a page directive, you can specify whether EL is enabled or not by setting the value of the isELEnabled directive to "true" or "false" respectively, as shown below:

```
<%@ page isScriptingEnabled="true|false" isELEnabled="true|false"%>
```

- **Using an element of the deployment descriptor:**

When using an element of the deployment descriptor, you can specify whether EL is enabled or not by including the boolean value "true" or "false" in between <el-enabled> tags, as shown below:

```
...
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-enabled>true</scripting-enabled>
    <scripting-enabled>true</scripting-enabled>
  </jsp-property-group>
</jsp-config>
....
```

Expression Language Syntax

The JSP expression language allows a page author to access a bean using a simple syntax such as:

```
#{expr}
```

In the above syntax, expr stands for a valid expression. It must be noted that this expression can be mixed with static text, and may also be combined with other expressions to form larger expressions.

Valid Expressions in the JSP EL

Valid expressions can include literals, operators, variables (object references), and function calls. We will look at each of these valid expressions separately:

Literals

The JSP Expression language defines the following literals that can be used in expressions:

Literals	Literal values
Boolean	true and false
Integer	Similar to Java. It can include any positive or negative number e.g, 24, -45, 567
Floating Point	Similar to Java. It can include any positive or negative floating point numbers e.g, -1.8E-45, 4.567
String	Any string delimited by single or double quotes. For single quotes, double quotes, and backslashes use the backslash character as an escape sequence. It must be noted that if double quotes are used around the string, the single quote need not be escaped.
Null	null

Let us look at some examples that use Literals as valid expressions:

```
#{false} <%-- evaluates to false --%>
#{3*8}
```

Operators

The JSP expression language provides the following operators, most of which are the usual operators available in Java:

Term	Definition
Arithmetic	+, - (binary), *, /, div, %, mod, - (unary)
Logical	and, &&, or, , !, not
Relational	==, eq, !=, ne, <, lt, >, gt, <=, le, >=, ge. Comparisons may be made against other values, or against boolean, string, integer, or floating point literals.
Empty	The empty operator is a prefix operation that can be used to determine if a value is null or empty.
Conditional	A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.

Let us look at some examples that use Operators as valid expressions:

`$((6 * 5) + 5) <!-- evaluates to 35 -->`

`$(empty name)`

Implicit Objects

The JSP expression language defines a set of implicit objects, many of which are available in JSP scriptlets and expressions:

Term	Definition
pageContext	The context for the JSP page. It can be used to access the JSP implicit objects such as request, response, session, out, servletContext etc. For example, <code>\$(pageContext.response)</code> evaluates to the response object for the page.

In addition, several implicit objects are available that allow easy access to the following objects:

Term	Definition
param	maps a request parameter name to a single String parameter value (obtained by calling <code>ServletRequest.getParameter (String name)</code>). The <code>getParameter (String)</code> method returns the parameter with the given name. The expression <code>\$(param.name)</code> is equivalent to <code>request.getParameter (name)</code> .
paramValues	maps a request parameter name to an array of values (obtained by calling <code>ServletRequest.getParameter (String name)</code>). It is very similar to the param implicit object except that it retrieves a string array rather than a single value. The expression <code>\$(paramvalues.name)</code> is equivalent to <code>request.getParamterValues(name)</code> .
header	maps a request header name to a single String header value (obtained by calling <code>ServletRequest.getHeader(String name)</code>). The expression <code>\$(header.name)</code> is equivalent to <code>request.getHeader(name)</code> .
headerValues	maps a request header name to an array of values (obtained by calling <code>ServletRequest.getHeaders(String)</code>). It is very similar to the header implicit object. The expression <code>\$(headerValues.name)</code> is equivalent to <code>request.getHeaderValues (name)</code> .
cookie	maps cookie names to a single cookie object. A client request to the server can contain one or more cookies. The expression <code>\$(cookie.name.value)</code> returns the value of the first cookie with the given name. If the request contains multiple cookies with the same name, then you should use the <code>\$(headerValues.name)</code> expression.

initParam	maps a context initialization parameter name to a single value (obtained by calling ServletContext.getInitparameter (String name)).
-----------	---

Outside of the above two types of implicit objects, there are also objects that allow access to the various scoped variables such as Web context, session, request, page:

Term	Definition
pageScope	maps page-scoped variable names to their values. For e.g., an EL expression can access an object, with a page scope in the JSP, with <code>#{pageScope.objectName}</code> and an attribute of the object can be accessed using <code>#{pageScope.objectName.attributeName}</code> .
requestScope	maps request-scoped variable names to their values. This object allows for access to the attributes of the request object. For e.g., an EL expression can access an object, with a request scope in the JSP, with <code>#{requestScope.objectName}</code> and an attribute of the object can be accessed using <code>#{requestScope.objectName.attributeName}</code> .
sessionScope	maps session-scoped variable names to their values. This object allows for access to the attributes of the session object. For example: <pre><% session.put (name", "John Doe"); %> \${sessionScope.name} <%-- evaluates to John Doe --%> <%= session.get("name"); %> <%-- This is an equivalent scripting expression --%></pre>
applicationScope	maps application-scoped variable names to their values. This implicit object allows for access to objects with application scope.

It must be noted that when an expression references one of these objects by name, the appropriate object is returned instead of the corresponding attribute. For example: `#{pageContext}` returns the PageContext object, even if there is an existing pageContext attribute containing some other value.

Using EL Expressions

EL expressions can be used in two situations:

- As attribute values in standard and custom actions
- In template text, such as HTML or non-JSP elements, in the JSP file - In this situation, the value of the expression in template text is evaluated and inserted into the current output. However, it must be noted that an expression will not be evaluated if the body of the tag is declared to be tagdependent.

JSP Fragments

In this section we will look at another new and interesting feature of JSP 2.0, JSP Fragments. We will specifically look at the following topic areas:

[JSP Fragments Defined](#)
[Methods for creating a JSP Fragment](#)
[Invoking a JSP Fragment](#)
[JspFragment Syntax](#)

JSP Fragments Defined

JSP Fragments is a new feature of JSP 2.0 that allows page authors to create custom action fragments that can be invoked. It allows a portion of JSP code to be encapsulated into a Java object that can be passed around and evaluated zero or more times. Template text and expression evaluations can be included in JSP fragments.

JSP Fragments are created when providing the body of a `<jsp:attribute>` standard action for an attribute that is defined as a fragment or of type `JspFragment`, or when providing the body of a tag invocation handled by a Simple Tag Handler. Depending on the configuration of the action being invoked, the body of the element either specifies a value that is evaluated only once, or it specifies the body as a JSP fragment.

For example, in **Code Snippet A** where we have defined an attribute named `x` and assigned a value of `tutorial1`, the fragment may be invoked only once. On the other hand in **Code Snippet B** the fragment may be invoked several times for evaluation.

Code Snippet A

```
<jsp:attribute name="x" >
  tutorial1
</jsp:attribute>
```

Code Snippet B

```
<jsp:attribute name="stringFrag">
This is the output of chaining 2 defined strings: ${result}
</jsp:attribute>
```

It is worth noting that the fragments themselves are represented by the JSP container in Java, by an instance of the `javax.servlet.jsp.tagext.JspFragment` interface, even though the JSP Fragment is coded in JSP syntax. In the context of a tag invocation, pieces of JSP code are translated into JSP fragments.

Methods for creating a JSP Fragment

There are two ways to create a JSP Fragment:

- **Providing the body of a `<jsp:attribute>`:**

For an attribute that is defined as a fragment (or of type `JspFragment`) in a tag file, a JSP Fragment can be created by providing the body of a `<jsp:attribute>` by specifying `fragment = "true"` in the attribute directive:

```
<%@ attribute name="stringFrag" fragment="true" %>
```

- **Providing the body of a tag invocation:**

Another way to create a JSP fragment is to provide the body of a tag invocation handled by a [Simple Tag Handler](#). The `JspFragment` instance is associated with the `JspContext` of the surrounding page, before being passed to a tag handler. All information that is not specific to servlets is abstracted by the `JspContext`, which also serves as the base class for the `PageContext` class. This abstraction is affected because fragments can not only work with JSP but also with other technologies, and also because the abstraction allows for an implementation-dependent access. It must be noted that the collaboration of the fragment and the `JspContext` is preserved for the duration of the tag invocation in which it is being used.

Invoking a JSP Fragment

After employing any of the two methods above to create a JSP fragment, it is passed to a tag handler for later invocation. The invocation of JSP fragments can be done either programmatically from a tag handler written in Java, or from a tag file using the `<jsp:invoke>` or `<jsp:doBody>` standard action. A bean property of type `Jsp-Fragment` is used to pass JSP fragments to tag handlers. These fragments can be invoked by calling the `invoke()` method in the `JspFragment` interface. It is worth noting that it is possible for a fragment to recursively invoke (indirectly) itself.

The responsibility for setting the values, of all declared `AT_BEGIN` and `NESTED` variables in the `JspContext` of the calling page/tag, is taken over by the tag handler to which the JSP Fragment is passed. This happens before invocation of the JSP Fragment. In the case where the fragment is invoked using `<jsp:invoke>` or `<jsp:doBody>` with the specification of the `var` attribute, a custom `java.io.Writer` is created that can expose the result of the invocation as a `java.lang.String` object. If the `varReader` attribute is specified, a custom `java.io.Reader` object is created that can expose the resulting invocation as a `java.io.Reader` object.

JspFragment Syntax

The simple syntax for a `JspFragment` is:

public interface JspFragment

JSP Syntax is used to define JSP Fragments as the body of a tag for an invocation to a SimpleTag handler, or as the body of a `<jsp:attribute>` standard action specifying the value of an attribute that is declared as a fragment, or to be of type JspFragment in the TLD.

Note that tag library developers and page authors should not generate Jsp-Fragment implementations manually. Also it is worth noting that it is not necessary to generate a separate class for each fragment. One possible implementation is to generate a single helper class for each page that implements JspFragment. Upon construction, a discriminator can be passed to select which fragment that instance will execute.

Tag Files

In this section, we will look at another new and interesting feature of JSP 2.0, Tag Files. We will specifically look at the following topics:

[Tag Files Defined](#)

[Tag File Syntax](#)

[Standard action in Tag files](#)

[Advantages of Using Tag Files](#)

Tag Files Defined

With the introduction of JSP 2.0, knowledge of Java is no longer a prerequisite to create a custom tag action. JSP 2.0's Simple tag extension allows page authors to write tag extensions using only JSP syntax. Tag files bring the power of reuse to the basic page author, who is presentation-focused and is not required to know Java.

Tag files are essentially source files that facilitate the abstraction of a fragment of JSP code, making it reusable via a custom action. Even for page authors or tag library developers who know Java, writing tag files is more convenient when developing tags that primarily output template text. When used together with JSP Fragments and Simple Tag Handlers, these concepts have the ability to simplify JSP development substantially, even for developers.

The required file extension for a tag file is `.tag` or `.tagx`. As is the case with JSP files, the actual tag may be composed of a top file that includes other files that contain either a complete tag or a segment of a tag file. Similar to the recommended extension, `.jspx`, for a segment of a JSP file, the recommended extension for a segment of a tag file is `.tagf`. Tag files can contain nothing more than template text, or they can contain JSP and markup code.

A tag file can forward to a page via the standard action, with the forward being handled through the request dispatcher (similar to JSP). When `RequestDispatcher.forward()` returns, the decision to stop processing the tag file and throw a `javax.servlet.jsp.SkipPage-Exception` is left to the discretion of the container.

Tag File Syntax

The syntax for tag files is similar to that of the JSP Syntax, save for the following:

- While some tag file specific directives are available, some directives are unavailable or have limited availability
- The `<jsp:doBody>` and `<jsp:invoke>` standard actions can only be used in tag files. We will look at these separately in the section titled [Standard action in Tag files](#)

Following is a tabular listing of directives available to tag files:

Directive	Description/Limitations
Page	This directive is not available to tag files. Since a tag file is not considered a page, a tag file needs to be used instead. Note that if this directive is used in a tag file, it'll result in a translation error.
Taglib	Available to tag files, this directive is similar to JSP pages

include	Available to tag files, this directive is similar to JSP pages. The included file must comply with syntax valid for a tag file. Otherwise, a translation error will occur.
Tag	Available and applicable only to tag files. Using this directive in a JSP will cause a translation error.
attribute	Available and applicable only to tag files. Using this directive in a JSP will cause a translation error.
variable	Available and applicable only to tag files. Using this directive in a JSP will cause a translation error.

Using classic tag handlers indirectly from a tag file binds the use of the tag file to Servlet environments, because the *tag* interface relies on PageContext (which is Servlet centric). So it pays to wary when a classic tag handler that implements the *tag* interface is invoked from a tag file. Note that SimpleTag extensions can be used in environments other than Servlet.

Standard action in Tag files

There are two standard actions that can only be used in tag files - they are `<jsp:invoke>` and `<jsp:doBody>`. In this section, we will look into these two standard actions in some detail:

`<jsp:doBody>`

As mentioned earlier, the `<jsp:doBody>` standard action can only be used in tag files, and must result in a translation error if used in a JSP. The standard action of `<jsp:doBody>` is very similar to `<jsp:invoke>`, with the only difference being that `<jsp:doBody>` operates on the body of the tag instead of a specific fragment passed as an attribute. It invokes the body of the tag, sending the output of the result to the `JspWriter`, or to a scoped attribute that can be examined and manipulated. The body of a tag is passed to the simple tag handler as a `JspFragment` object. If the `<jsp:doBody>` contains a non-empty body, a translation error will occur.

Following is a tabular listing of the directives available to the `<jsp:doBody>` standard action:

Attribute	Description
var	This attribute is the name used to identify a fragment during the tag invocation. This directive is mandatory
varReader	This is the name of a scoped attribute to store the result of the fragment invocation in, as a <code>java.lang.String</code> object. A translation error must occur if both <code>var</code> and <code>varReader</code> are specified. If neither <code>var</code> nor <code>varReader</code> are specified, the result of the fragment goes directly to the <code>JspWriter</code> , as described earlier in this section. This directive is optional.
scope	This is the name of a scoped attribute to store the result of the fragment invocation in, as a <code>java.io.Reader</code> object. A translation error must occur if both <code>var</code> and <code>varReader</code> are specified. If neither <code>var</code> nor <code>varReader</code> is specified, the result of the fragment invocation goes directly to the <code>JspWriter</code> , as described earlier in this section.

Since `<jsp:doBody>` only operates on the body of the tag, there is no name attribute for this standard action. The `var`, `varReader`, and `scope` attributes are all supported with the same semantics as for `<jsp:invoke>`. Fragments are provided access to variables the same way for `<jsp:doBody>` as they are for `<jsp:invoke>`. If no body was passed to the tag, `<jsp:doBody>` will behave as though a body was passed in that produces no output.

`<jsp:invoke>`

To reiterate, the `<jsp:invoke>` standard action can only be used in tag files and will result in a translation error if used in a JSP. Following is a tabular listing of the directives available to the `<jsp:invoke>` standard action:

Attribute	Description
fragment	This attribute is the name used to identify a fragment during the tag invocation. This directive is mandatory.

var	This is the name of a scoped attribute to store the result of the fragment invocation in, as a java.lang.String object. A translation error must occur if both var and varReader are specified. If neither var nor varReader are specified, the result of the fragment goes directly to the JspWriter, as described in the beginning of this section. This directive is optional.
varReader	This is the name of a scoped attribute to store the result of the fragment invocation in, as a java.io.Reader object. A translation error must occur if both var and varReader are specified. If neither var nor varReader is specified, the result of the fragment invocation goes directly to the JspWriter, as described in the beginning of this section. This directive is optional.
scope	This is the scope in which to store the resulting variable. A translation error must result if the value is not one of page, request, session, or application. A translation error will result if this attribute appears without specifying either the var or varReader attribute as well. Note that a value of session should be used with caution since not all calling pages may be participating in a session. A container must throw an IllegalStateException at runtime if scope is session and the calling page does not participate in a session. Defaults to page. This directive is optional.

The `<jsp:invoke>` standard action is used to invoke a fragment (which is determined by the name attribute) and send the output of the result to the `JspWriter`. The fragment is invoked using the `JspFragment.invoke()` method. Null is passed as the `Writer` parameter to force the write (for the results to be sent to, that is) to the `JspWriter` of the `JspContent` associated with the `JspFragment` object. An example of using `invoke` is shown below:

```
<jsp:invoke fragment="Frag1" />
```

It's also possible to send the output to a page-scoped variable that can be used later on the page for other manipulations, by using the `var` or `varReader` attribute. When using either of these attributes a custom `java.io.Writer` is passed instead of null. You can further tweak the `Writer` to specify whether you want a `String` or a `Reader`.

The `var` attribute is a `java.lang.String` while the `varReader` attribute is a `java.io.Reader` object. While `String` objects contain the content sent by the fragment to the `Writer`, the `Reader` object can produce the content sent by the fragment to the `Writer`. The `Reader` is resettable - if the `reset()` method is called, the result of the invoked fragment can be re-read without re-executing the fragment.

The optional `scope` attribute can be used to set the resulting scoped variable. `Scope` can be set to the standard JSP scopes including: `page`, `request`, `session` or `application`. Here are two examples of how to use the `var`, `varReader` and the `scope` attribute:

```
<jsp:invoke fragment="Frag2" var="resultString" scope="request"/>
<jsp:invoke fragment="Frag3" varReader="resultReader" scope="session"/>
```

Advantages of Using Tag Files

Some of the advantages of using tag files in JSP development:

- *Reuse mechanism for page authors* - Tag files constitute a flexible and efficient reuse mechanism for page authors by allowing better customization of the included content as well as the nesting of tags.
- *Compatibility with tag handlers* - Tag files are very well suited for tag handlers that primarily output HTML content, similar to how JSPs are well suited for replacing Servlets that primarily output HTML content
- *Scriptlet free JSP pages* - Tag files make for a great way to hide ugly scriptlets, by helping to change the script-based JSP code in a web application to much cleaner JSTL-style code with no scriptlets, and with EL expressions instead of scripting expressions. Page authors can easily abstract scriptlets into tag files and then invoke the tags. Later, the scriptlet-based tag files can be converted to JSTL-style code or encapsulated into Java tag handlers.
- *Rapid development* - Dynamic recompilation of tag files are supported in some containers like Tomcat 5.0 (worth noting that the 5.x releases implement the Servlet 2.4 and JSP 2.0 specifications) With this feature, you simply need to deploy your tag file in /

WEB-INF/tags/ or a subdirectory, and tweak it until it works, without having to recompile and redeploy for each and every change during development.

- *Flexible Packaging* - Tag files provide the advantage of flexible packaging. The directory /WEB-INF/tags/ is now a standard directory that is recognized by compliant containers. The JSP container will process any file with the .tag extension that is present in this directory, or a subdirectory of tags. The container will create an implicit TLD file as well as a simple tag handler.
Options for packaging - Tag files can be packaged in one of three ways:

In /WEB-INF/tags/ with no TLD. The custom actions are then imported into the JSP using `<%@ taglib prefix="..." tagdir="/WEB-INF/tags" %>`

In /WEB-INF/tags/ with a supplementary TLD. This allows for greater customization of the tag file, and makes it transparent to the caller that the tag was implemented as a tag file. The TLD would be imported using `<%@ taglib prefix="..." uri="..." %>`

In /META-INF/tags/ in a JAR file with a TLD. This is ideal for tag files that are part of a tag library in a JAR file that can simply be "dropped in" to your web application. Note that Tag files that are bundled in a JAR require a TLD, and the one's that are not defined in a TLD but appear in a JAR are ignored by the web container.

Simple Tag Handlers

In this section, we will look at another new and interesting feature of JSP 2.0, Simple Tag Handlers. Specifically, we will look at the following:

[Using Simple Tag Extensions](#)
[Simple Tag Interface](#)
[Simple Tag Syntax](#)
[Simple Tag Handler Lifecycle](#)

Using Simple Tag Extensions

The API and invocation protocol for classic tag handlers is necessarily somewhat complex because scriptlets and scriptlet expressions in tag bodies can rely on surrounding context defined using scriptlets in the enclosing page. With the introduction of the Expression Language (EL) and JSP Standard Tag Library (JSTL), JSP page authors can now develop JSP pages that do not need scriptlets or scriptlet expressions. This also has an implication on the requirements that classic tag handlers need to take into consideration, with many of them being irrelevant in most cases. This allows for a definition of a tag invocation protocol that is easier to use for many use cases. The introduction of the simple tag extension in JSP 2.0 signalled an easier way to implement custom actions with a lifecycle that is as easier to work with.

Outside of the fact that simple tag extensions made work easier, they also do not directly depend upon any Servlet APIs, which meant that they opened up a whole new space for further integration with other technologies. This is accomplished because PageContext now extends JspContext. 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. The Tag interface relies on PageContext, whereas SimpleTag only relies on JspContext.

Simple Tag Extensions can be written in one of two ways:

- In Java, by defining a class that implements the `javax.servlet.jsp.tagext.SimpleTag` interface. This class is intended for use by advanced page authors and tag library developers who need the flexibility of the Java language in order to write their tag handlers. The `javax.servlet.jsp.tagext.SimpleTagSupport` class provides a default implementation for all methods in SimpleTag.
- In JSP syntax, using tag files. With the ability to write custom actions in JSP syntax, page authors (with no prior knowledge of Java), advanced page authors or tag library developers (who know Java but are producing tag libraries that are primarily template based presentation) have been benefitted.

SimpleTag Interface

Simple tag handlers are those that implement the SimpleTag interface. It is worth noting that this interface is provided for situations when the flexibility of the Java language is needed in order to write the tag handler.

The invocation protocol used by SimpleTag is simplified from the one used for Classic tag handlers. The `javax.servlet.jsp.tagext.SimpleTagSupport` class provides a default implementation for all methods in SimpleTag. The complete interface definition is shown below:

```

public interface SimpleTag extends JspTag {
    public void doTag()throws JspException, java.io.IOException;
    public void setParent(JspTag parent);
    public JspTag getParent();
    public void setJspContext(JspContext pc);
    public void setJspBody(JspFragment jspBody);
}

```

It is important to note that that the SimpleTag interface extends directly from JspTag and doesn't extend Tag. This implies the fact that SimpleTag doesn't have any inherent JSP/Servlet knowledge embedded within it. Another difference that is worth noticing is that SimpleTag only has one lifecycle method, doTag() defined as:

```

public void doTag()throws JspException, java.io.IOException

```

The doTag() method is called only once for any given tag invocation. This means that all code (includes tag logic, iteration, or body evaluations) related to this tag is contained in one nice and trim method. If you compare this to the IterationTag interface, you will notice how considerably easier it is to get the job done.

The setJspBody() method is provided to support body content. 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 getJspBody() and other useful methods to make this even easier.

Most SimpleTag handlers should extend javax.servlet.jsp.tagext.SimpleTagSupport. This is the convenience class, similar to TagSupport or BodyTagSupport. There are also some helpful methods included in this class that include:

- public JspFragment getJspBody() which returns the body passed in by the container via setJspBody. The JspFragment encapsulates the body of the tag. If the JspFragment is null, it indicates that tag has a body content type of empty.
- public static final JspTag findAncestorWithClass(JspTag from, java.lang.Class klass) which finds the instance of a given class type that is closest to a given instance. This method uses the getParent() method from the Tag and/or Simple Tag interfaces. This method is used for coordination among cooperating tags. While traversing the ancestors, for every instance of TagAdapter (used to allow collaboration between classic Tag handlers and SimpleTag handlers) encountered, the tag handler returned by TagAdapter.getAdaptee() is compared to klass. In a case where the tag handler matches this class, and not its TagAdapter, is returned.

SimpleTag Syntax

The basic syntax for a SimpleTag is:

```

public interface SimpleTag extends JspTag

```

The above syntax defines an interface for Simple Tag Handlers. Simple Tag Handlers differ from Classic Tag Handlers in that 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, etc. are to be performed in this single method. Thus, simple tag handlers have the equivalent power of BodyTag, but with a much simpler lifecycle 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 as many times as it needs.

It is worth noting that a SimpleTag handler must have a public no-args constructor. Most SimpleTag handlers should extend SimpleTagSupport.

Simple Tag Handler Lifecycle

When a simple tag handler is required on a JSP, it is instantiated by the container, it is executed, and then it is discarded. There are no complicated caching semantics when using this interface since nothing is cached or reused. It was decided by the expert group that performance gains that might be made using caching mechanisms dramatically increased the difficulty in writing portable tag handlers and made the handlers error prone.

If performance concerns are critical, I would suggest first implementing your tag handler as a simple tag and then taking some performance metrics to see if your criteria are met before embarking on the more complicated and time-consuming path of writing a Classic handler.

The following lifecycle events take place for the simple tag handler (in the same order):

1. A new tag handler instance is created each time the tag is encountered by the container. This is done by calling the zero argument constructor on the corresponding implementation class. It is important to note that a new instance must be created for each tag invocation.
2. The `setJspContext()` and `setParent()` methods are invoked on the tag handler. If the value being passed is 'null', then the `setParent()` method need not be called. 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 `JspContext`.
3. The container calls the setters for each attribute defined for this tag in the order in which they appear in the JSP page or Tag File. If the attribute value is an expression language expression or a runtime expression, it gets evaluated first and is then passed on to the setter. On the other hand if the attribute is a dynamic-attribute then `setDynamicAttribute()` is called.
4. The `setJspBody()` method is called by the container to set the body of this tag, as a `JspFragment`. A value of null is passed to `setJspBody()` if the tag is declared to have a `<body-content>` of empty.
5. The `doTag()` method is called by the container. All tag logic, iteration, body evaluations, etc. occur in this method.
6. All variables are synchronized after the `doTag()` method returns.

Resources

This resource section is part of the tutorial series titled *'Understanding the New Features of JSP 2.0'*. It lists references like the JSP 2.0 specification along with links to other resources that can help you understand and apply the concepts and techniques presented in the tutorials.

Resources/References	URL
JSP 2.0 Sample Demos	http://otn.oracle.com/sample_code/tech/java/jsps/content.html
JSP 2.0 HowTo's	http://otn.oracle.com/sample_code/tech/java/codesnippet/jsps/content.html
JSP Standard tag Library	http://java.sun.com/products/jsp/jstl/
JSP 2.0 Specification	http://www.jcp.org/aboutJava/communityprocess/first/jsr152/
OTN Sample Code	http://otn.oracle.com/sample_code/content.html

Feedback

If you have questions or comments about this tutorial, you can:

- Post a message in the [OTN Sample Code discussion forum](#). OTN developers and other experts monitor the forum.
- Send email to the author. <mailto:Dilip.Thomas@oracle.com>

If you have suggestions or ideas for future tutorials, you can:

- Post a message in the [OTN Member Feedback forum](#).

- Send email to <mailto:Tom.Hunert@oracle.com>
-