

Ajax and Java development made simpler, Part 4: Create JSF-like components, using JSP tag files

Learn how to use deferred values and deferred methods in custom JSP tag

Skill Level: Intermediate

[Andrei Cioroianu](#)

Senior Java Developer and Consultant
Devsphere

29 Jul 2008

JavaServer Pages (JSP) and JavaServer Faces (JSF) used to have different variants of the Expression Language (EL). Their unification in JSP 2.1 opened new possibilities, allowing you to use deferred values and deferred method attributes in your custom JSP tags. This article shows how to develop Java™ Web components based on JSP tag files, which are much simpler and easier to build than the JSF components.

JSP and JSF are the most important Java standards that we use today for building Web applications. The philosophies of these two technologies are quite different, however. With JSP, the underlying mechanisms are simple, you control what is happening, and you have the freedom to do things how you want. JSF adds some complexity and processing overhead, but the application model is standardized, which means tools can do more for you, vendors can provide component libraries, and developers can focus on building applications.

As [Parts 2](#) and [3](#) of this series demonstrated, you don't really need a complex framework for building Web forms. If you like simplicity or if you need to develop a Web application quickly, you can use the JSP Standard Tag Library (JSTL) in combination with tag files and dynamic attributes. You'll be able to customize your components very easily, controlling the HTML output and the HTTP requests.

Using the JSF framework together with a good component library may be the right

choice if you are building a sophisticated Web application. Because no Web framework can satisfy every need of every application, you might still have to build a few components of your own. In this case, you can use the techniques presented in this article to create JSF-like components without using the JSF APIs. You'll be able to mix your non-JSF components with the JSF components in the same Web page and you'll avoid much of the JSF complexity.

Using deferred values

Defined initially as a main feature of JSTL, the expression language was later incorporated into the JSP 2.0 standard. Those early EL versions allowed you to query JavaBeans and collections, using `#{ . . . }` expressions, which were evaluated by the JSP container during the execution of the JSP pages.

The JSF framework defined its own EL variant, which supported additional capabilities for its `#{ . . . }` expressions, such as deferred evaluation and the option to set the value of an EL expression. These features enabled data bindings and allowed the JSF framework to control when an expression is evaluated.

The JSP 2.1 standard unified the two EL variants. You can still use `#{ . . . }` expressions whose evaluation time is determined by the JSP container, but now you can also use `#{ . . . }` expressions within the attributes of your custom JSP tags, which don't have to be based on JSF. This section shows how to use deferred-value expressions for non-JSF tags, which are much easier to build than their JSF counterparts.

Creating and configuring tag handlers

JSP 2.1 lets you use `#{ . . . }` expressions within the attributes of any custom tag as long as the attribute's setter method takes a `javax.el.ValueExpression` parameter. Listing 1 shows what the handler of the custom tag should look like. It normally extends the `SimpleTagSupport` class of the `javax.servlet.jsp.tagext` package and it must have a `set` method for every attribute of the custom tag. In addition, the tag handler should provide a `doTag()` method that is called when the JSP page containing the custom tag is executed.

Listing 1. The setter of a deferred-value attribute

```
package mylib;

public class MyTag extends javax.servlet.jsp.tagext.SimpleTagSupport {
    private javax.el.ValueExpression myAttr;

    public void setMyAttr(javax.el.ValueExpression myAttr) {
        this.myAttr = myAttr;
    }

    public void doTag() throws javax.servlet.jsp.JspException,
```

```

        java.io.IOException {
    }
}

```

Custom tags are defined in a tag library descriptor (TLD) file, which contains the version, prefix, and URI of the tag library, as well as a definition of each custom tag, specifying the tag name, the tag handler class, the type of the body content, and the tag's attributes. For every attribute of type `javax.el.ValueExpression`, you must include `<deferred-value>` within the `<attribute>` element as shown in Listing 2:

Listing 2. Defining the custom tag and its deferred-value attribute

```

<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee" ... version="2.1">
  <tlib-version>1.0</tlib-version>
  <short-name>mylib</short-name>
  <uri>/mylib.tld</uri>
  <tag>
    <name>myTag</name>
    <tag-class>mylib.MyTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <name>myAttr</name>
      <deferred-value>
        <type>...</type>
      </deferred-value>
    </attribute>
    ...
  </tag>
</taglib>

```

If a JSP page contains `<mylib:myTag myAttr="#{...}">`, the JSP container wraps the expression in a `javax.el.ValueExpression` object, which is passed to the `setMyAttr()` method of the tag handler instance whose `doTag()` method can use the EL API to evaluate the expression and change its value (see Listing 3):

Listing 3. Using the methods of `javax.el.ValueExpression` in `doTag()`

```

package mylib;

public class MyTag extends javax.servlet.jsp.tagext.SimpleTagSupport {
    private javax.el.ValueExpression myAttr;
    ...
    public void doTag() throws javax.servlet.jsp.JspException,
        java.io.IOException {
        javax.el.ELContext elContext = getJspContext().getELContext();
        Object value = myAttr.getValue(elContext); // get current value
        value = ... // change value
        if (!myAttr.isReadOnly(elContext))
            myAttr.setValue(elContext, value); // set new value
    }
}

```

If you look at the Java source code that a JSP container generates from a JSP page, you'll observe how the attribute values are passed to the setter methods of the tag handler instance. After that, the handler's `doTag()` method is called to execute the custom tag. This is the mechanism that works behind the custom tags that appear in the JSP page.

The code from Listing 3 uses the EL API to work with deferred-value expressions. The `getValue()` method of `javax.el.ValueExpression` evaluates the expression in the given EL context, `isReadOnly()` returns `true` if the expression's value cannot be set, and the `setValue()` method changes the value of the bean property that is specified in the EL expression.

For example, if a JSP page contains `<mylib:myTag myAttr="#{someBean.someProp}">`, the `getValue()` call from Listing 3 will return the property's value and `setValue()` will change the value of the bean property. The `isReadOnly()` method would have returned `true` only if the bean class didn't have a set method for its property.

Using the EL API in JSP tag files

As you can see in the previous examples, it is fairly simple to build custom tags whose attributes accept deferred-value expressions. If you choose to develop JSP tag files, the custom tag implementation is even easier because the JSP container generates the tag handler classes for you and you don't have to configure the custom tags in a TLD file anymore. You just use the `<%@ attribute name="myAttr" deferredValue="true" %>` directive in the tag file and the JSP container does the rest.

The `set.tag` file (see Listing 4) can be used in a scriptless JSP page to set the value of a `#{...}` expression. The JSP container will generate a tag handler class, which will have setter methods for all three attributes: `expr`, `value`, and `type`. The `doTag()` method of the generated class will contain the Java code placed between `<%` and `%>`. This code gets the attribute values from the JSP context, converts the given value to the specified type, and then calls the `setValue()` method of the expression object.

Listing 4. The `set.tag` file

```
<%@ attribute name="expr" required="true" deferredValue="true" %>
<%@ attribute name="value" required="true" rtexprvalue="true"
    type="java.lang.Object" %>
<%@ attribute name="type" required="false" rtexprvalue="true" %>
<%@ tag body-content="empty" %>

<%
    javax.el.ELContext elContext = jspContext.getELContext();
    javax.el.ValueExpression valueExpr =
        (javax.el.ValueExpression) jspContext.getAttribute("expr");
    Object valueAttr = jspContext.getAttribute("value");
```

```

String valueType = (String) jspContext.getAttribute("type");

if (!valueExpr.isReadOnly(elContext)) {
    Class valueClass = null;
    if (valueType != null && valueType.length() > 0)
        valueClass = Class.forName(valueType);
    else
        valueClass = valueExpr.getExpectedType();
    javax.servlet.jsp.JspFactory jspFactory
        = javax.servlet.jsp.JspFactory.getDefaultFactory();
    valueAttr = jspFactory.getJspApplicationContext(application)
        .getExpressionFactory().coerceToType(valueAttr, valueClass);
    valueExpr.setValue(elContext, valueAttr);
}
%>

```

For compatibility reasons, tag files are expected to use the JSP 2.0 version by default. Because deferred-values are a JSP 2.1 feature, you have to specify the proper JSP version in a file named `implicit.tld` (shown in Listing 5). This file must be placed in each directory that contains tag files using JSP 2.1 features.

Listing 5. The `implicit.tld` file

```

<taglib>
  <jsp-version>2.1</jsp-version>
</taglib>

```

The sample application of this article has two directories containing tag files:

- `/WEB-INF/tags/dynamic/comp` groups a few Web components that can be used in JSP pages:
 - `inputText.tag`
 - `inputData.tag`
 - `action.tag`
- `/WEB-INF/tags/dynamic/comp/util` contains helper tag files that are invoked from the Web components:
 - `set.tag`
 - `invoke.tag`

Building a simple Web component

The following example uses the `set.tag` file to build an input component. The `inputText.tag` file (see Listing 6) declares two attributes with the `<%@attribute%>` directive. The name attribute has `rtexprvalue="true"` so that you may use `#{...}` expressions within the value of this attribute. For the value attribute, you can use `#{...}` expressions because its `<%@attribute%>` directive contains `deferredValue="true"`. The tag file also contains `<%@ tag`

`dynamic-attributes="dynAttr" ... %>` allowing you to provide additional attributes, which are stored in a Java Map named `dynAttr`.

After importing the used tag libraries with the `<%@taglib%>` directive, the `inputText.tag` file invokes `set.tag` with `<dcu:set>` if the HTTP request contains a parameter with the given name. If present, the value of this parameter is stored into the bean property specified within the `value` attribute. For example, if you use `<dc:inputText name="someParam" value="#{someBean.someProp}" ... />` in a JSP page, the value of the `someParam` request parameter will be stored into the `someProp` property of the bean instance.

Next, the `inputText.tag` file generates an `<input type="text">` element with the given name and `value` attributes. The `value` attribute of the `<input>` element will contain the value of the bean property. The dynamic attributes from the `dynAttr` map are also added to the `<input>` element. For example, if the JSP page contains `<dc:inputText ... class="someClass"/>`, the generated HTML element will contain the `class` attribute.

Listing 6. The `inputText.tag` component

```
<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="value" required="true" deferredValue="true" %>
<%@ tag dynamic-attributes="dynAttr" body-content="empty" %>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="dcu" tagdir="/WEB-INF/tags/dynamic/comp/util" %>

<c:if test="${param[name] != null}">
  <dcu:set expr="#{value}" value="${param[name]}" />
</c:if>

<input type="text" name="${fn:escapeXml(name)}" value="${fn:escapeXml(value)}"
  <c:forEach var="attr" items="${dynAttr}">
    <c:if test="${attr.key != 'type'}">
      ${attr.key}="${fn:escapeXml(attr.value)}"
    </c:if>
  </c:forEach>
>
```

Listing 7 shows how you can invoke the `inputText.tag` file in a JSP page, using `<dc:inputText>`. First of all, you must import the library of tag files with the `<%@taglib%>` directive. Then, you would normally create or obtain a bean instance with `<jsp:useBean>`. The `<dc:inputText>` component should be placed within a `<form>` element. In this example, the generated `<input>` element will contain the value of the bean property. When the user clicks the **Submit** button, the form data is sent to the server where the `inputText.tag` file stores the user's input into the bean property with the help of `<dcu:set>`.

Listing 7. Using `<dc:inputText>` in a JSP page

```
<%@ taglib prefix="dc" tagdir="/WEB-INF/tags/dynamic/comp" %>
<jsp:useBean id="someBean" class="someapp.SomeBean" scope="request"/>
<form method="POST">
  <dc:inputText name="someParam" value="#{someBean.someProp}"
    class="someClass" size="20" maxlength="40"/>
  <input type="submit" value="Submit"/>
</form>
```

Using deferred methods

The initial EL variant, which was defined as part of JSTL 1.0, did not have any mechanism for calling Java methods from JSP pages. When EL became part of JSP 2.0, the syntax was enhanced to support EL functions that were mapped to static Java methods. In many cases, however, you want to call instance methods of the JavaBean objects.

Recognizing the need to specify action and validation methods for a Web component, the JSF EL variant added another feature allowing you to use method expressions within an attribute. The unified EL of JSP 2.1 enabled this feature for any custom JSP tag that may or may not use the JSF API. This section shows how to use deferred-method expressions for non-JSF tags, which are much simpler than their JSF counterparts.

Invoking a method with the EL API

If an attribute of a custom tag must accept a method expression, you have to use the `javax.el.MethodExpression` type for the parameter of the attribute setter as shown in [Listing 8](#). If you use `<mylib:myTag methodAttr="#{someBean.someMethod}">` in a Web page, the JSP container will wrap the expression into an object that is passed to `setMethodAttr()`.

Then, the JSP container will invoke the tag handler's `doTag()` method where you can call the bean method, passing the EL context and an array of parameters to the `invoke()` method of `javax.el.MethodExpression`. You can make the returned value available in the JSP page by creating a JSP variable with the `setAttribute()` method of the JSP context.

Listing 8. Using `javax.el.MethodExpression`

```
package mylib;

public class MyTag extends javax.servlet.jsp.tagext.SimpleTagSupport {
    private javax.el.MethodExpression methodAttr;

    public void setMethodAttr(javax.el.MethodExpression methodAttr) {
        this.methodAttr = methodAttr;
    }
}
```

```

    }

    public void doTag() throws javax.servlet.jsp.JspException,
                           java.io.IOException {
        javax.el.ELContext elContext = getJspContext().getELContext();
        Object paramArray[] = new Object[] {...};
        Object returnValue = methodAttr.invoke(elContext, paramArray);
        getJspContext().setAttribute("returnVar", returnValue);
    }
}

```

When you build custom tags backed by your own Java tag handlers, you must configure them in a TLD file (see Listing 9). The method attribute must be marked with a `<deferred-method>` element whose `<method-signature>` sub-element should contain the signature of the invoked bean method.

Listing 9. Defining the custom tag and its deferred-method attribute

```

<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee" ... version="2.1">
  <tlib-version>1.0</tlib-version>
  <short-name>mylib</short-name>
  <uri>/mylib.tld</uri>
  <tag>
    <name>myTag</name>
    <tag-class>mylib.MyTag</tag-class>
    <body-content>scriptless</body-content>
    ...
    <attribute>
      <name>methodAttr</name>
      <deferred-method>
        <method-signature>...</method-signature>
      </deferred-method>
    </attribute>
  </tag>
</taglib>

```

It's useful to know how to implement custom tags as shown in Listings 8 and 9, but it is easier to develop JSP tag files because the JSP container generates the tag handler classes for you and you don't have to configure the custom tags in a TLD file anymore. You just use the `<%@attribute%>` directive in the tag file as shown in Listing 10:

Listing 10. Defining a deferred-method attribute in a JSP tag file

```

<%@ attribute name="methodAttr" deferredMethod="true"
  deferredMethodSignature="..." %>

```

Because multiple methods of the same class can have the same name, the JSP container needs the full signature to identify the method that must be called. You can, however, treat `javax.el.MethodExpression` as any regular Java class if you want to build generic custom tags that can work with any method no matter what its signature looks like.

The `invoke.tag` file (see [Listing 11](#)) defines a method attribute of type `javax.el.MethodExpression`. If you already have an instance of this class stored in a JSP variable, you can pass it to the tag file with `<dcu:invoke method="{methodVar}" .../>`. The custom tag also has an optional `params` attribute, which can be a single object or an array of objects. These parameters are passed to the `invoke()` method of the expression object.

The returned value is stored into the `varAlias` variable, which is exported with the `<%@variable%>` directive. For example, if the JSP page contains `<dcu:invoke method="{methodVar}" params="{paramArray}" var="returnVar"/>`, the `invoke.tag` file will call the method with the given parameters and will store the returned value into `returnVar`.

Listing 11. The `invoke.tag` file

```
<%@ attribute name="method" required="true" rtexprvalue="true"
    type="javax.el.MethodExpression" %>
<%@ attribute name="params" required="false" rtexprvalue="true"
    type="java.lang.Object" %>
<%@ attribute name="var" required="true" rtexprvalue="false" %>
<%@ variable name-from-attribute="var"
    variable-class="java.lang.Object"
    alias="varAlias" scope="AT_END" %>
<%@ tag body-content="empty" %>

<%
    javax.el.ELContext elContext = jspContext.getELContext();
    javax.el.MethodExpression methodExpr =
        (javax.el.MethodExpression) jspContext.getAttribute("method");
    Object paramArray = jspContext.getAttribute("params");
    if (paramArray == null)
        paramArray = new Object[0];
    else if (!(paramArray instanceof Object[]))
        paramArray = new Object[] { paramArray };
    Object returnValue = methodExpr.invoke(
        elContext, (Object[]) paramArray);
    jspContext.setAttribute("varAlias", returnValue);
%>
```

Calling action methods from JSP pages

The JSF framework lets you attach action methods to command buttons. These methods have no parameters and are invoked on the server side after clicking a submit button in the Web browser. Calling an action method from a non-JSF page is now possible with the help of the deferred-method feature that was added to JSP 2.1. You've already seen how to use the EL API in the `invoke.tag` file, which is used in `action.tag` (shown in [Listing 12](#)) to call an action method:

Listing 12. The `action.tag` component

```
<%@ attribute name="method" required="true" deferredMethod="true"
    deferredMethodSignature="String action()" %>
<%@ attribute name="var" required="true" rtexprvalue="false" %>
```

```

<%@ variable name-from-attribute="var"
      variable-class="java.lang.String"
      alias="varAlias" scope="AT_END" %>
<%@ tag body-content="empty" %>

<%@ taglib prefix="dcu" tagdir="/WEB-INF/tags/dynamic/comp/util" %>

<dcu:invoke method="{method}" var="varAlias"/>

```

Listing 13 demonstrates how to use the `<dc:action>` tag in a regular JSP page to call the action method. You would typically invoke such methods during the processing of a `POST` request. In this example, the result returned by the action method is stored in a JSP variable named `outcome`.

Listing 13. Using `<dc:action>` in a JSP page

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="dc" tagdir="/WEB-INF/tags/dynamic/comp" %>

<jsp:useBean id="someBean" class="someapp.SomeBean" scope="request"/>

<c:if test="{fn:toUpperCase(pageContext.request.method) == 'POST'}">
  <dc:action method="{someBean.someAction}" var="outcome"/>
  {outcome}
</c:if>

<form method="POST">
  <input type="submit" value="Submit">
</form>

```

Adding a validator attribute to the input component

An earlier example (shown in [Listing 6](#)) demonstrated how to build an input component. The `inputData.tag` file (see [Listing 14](#)) enhances the component, adding the `valueType` and `validator` attributes, which are used to convert and validate the user's input. The `valueType` attribute is passed to the `<dcu:set>` tag, which takes care of the data conversion.

The `validator` method receives the user's input as a parameter and may return an error message, which is stored in a JSP variable named `validationError`. The `validator` method is called from the `inputData.tag` file with `<dcu:invoke>`.

Listing 14. The `inputData.tag` component

```

<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="value" required="true" deferredValue="true" %>
<%@ attribute name="valueType" required="false" rtexprvalue="true" %>
<%@ attribute name="validator" required="false" deferredMethod="true"
  deferredMethodSignature="java.lang.String validate(java.lang.String)" %>
<%@ variable name-given="validationError" scope="AT_BEGIN"
  variable-class="java.lang.String" %>
<%@ tag dynamic-attributes="dynAttr" body-content="scriptless" %>

```

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="dcu" tagdir="/WEB-INF/tags/dynamic/comp/util" %>

<c:remove var="validationError"/>
<c:if test="${param[name] != null and validator != null}">
  <dcu:invoke method="${validator}" params="${param[name]}" var="validationError"/>
</c:if>

<c:if test="${param[name] != null and validationError == null}">
  <dcu:set expr="#{value}" value="${param[name]}" type="${valueType}"/>
</c:if>

<jsp:doBody/>

<input type="text" name="{fn:escapeXml(name)}"
  value="{fn:escapeXml(!empty param[name] ? param[name] : value)}"
  <c:forEach var="attr" items="{dynAttr}">
    <c:if test="{attr.key != 'type'}">
      {attr.key}="{fn:escapeXml(attr.value)}"
    </c:if>
  </c:forEach>
>

```

The `inputData.tag` file uses `<jsp:doBody/>` to execute the content placed between `<dc:inputData>` and `</dc:inputData>` in the JSP page (see Listing 15). This allows the JSP page to show the validation error before the `<input>` element that is generated by the tag file. The error can also be shown after `<input>`.

Listing 15. Using `<dc:inputData>` in a JSP page

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="dc" tagdir="/WEB-INF/tags/dynamic/comp" %>

<jsp:useBean id="someBean" class="someapp.SomeBean" scope="request"/>

<form method="POST">
  <dc:inputData name="someParam" value="{someBean.someProp}"
    validator="{someBean.someValidator}"
    class="someClass" size="20" maxlength="40">
    <c:out value="{validationError}"/>
  </dc:inputData>
  <c:out value="{validationError}"/>
  <input type="submit" value="Submit">
</form>

```

Conclusion

Throughout this series, you learned how to simplify Ajax and Java development, using JSP tag files, JSP EL, JSTL, dynamic attributes, code generators, conventions, and JavaScript object hierarchies. In this final part of the series, you learned how to use deferred values and deferred methods in custom JSP tags to create JSF-like components.

Downloads

Description	Name	Size	Download method
Sample application for this article	wa-aj-simplejava4121	412KB	HTTP

[Information about download methods](#)

Resources

- [Part 1](#) of this series: Learn how to generate JavaScript code for sending Ajax requests and processing Ajax responses.
- [Part 2](#): Create HTML forms, using conventions and JSP tag files to minimize setup and configuration.
- [Part 3](#): Implement user-friendly and safe validation for your Web forms, using JavaScript and JSP tag files.
- The developerWorks [Web development zone](#) is packed with tools and information for Web 2.0 development.
- The developerWorks [Ajax resource center](#) contains a growing library of Ajax content as well as useful resources to get you started developing Ajax applications today.
- Go to the [JSP Web page](#) for more resources on JavaServer Pages.

About the author

Andrei Cioroianu

Andrei Cioroianu is the founder of [Devsphere](#), a provider of Java EE development and Web 2.0/Ajax consulting services. He's been using Java and Web technologies since 1997 and has 10 years of professional experience in solving complex technical problems and managing the full life cycle of commercial products, custom applications, and open-source frameworks. You can reach Andrei through the contact form at www.devsphere.com.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.