

Ajax Java development made simpler, Part 2: Use conventions to minimize setup and configuration

Create customizable Web components using JSTL and JSP tag files

Skill Level: Intermediate

[Andrei Cioroianu](#)

Senior Java Developer and Consultant
Devsphere

20 May 2008

Most Web frameworks try to be as flexible and extensible as possible to accommodate different application needs and development styles. Unfortunately, sometimes this leads to complexity, processing overheads, and large configuration files. This article shows how to use JSP Standard Tag Library (JSTL) and JSP tag files to implement data binding, page navigation, and style conventions, which make both development and maintenance easier. You will learn how to build custom JSP tags with dynamic attributes to facilitate rapid application changes. In addition, the last section of the article contains an example that uses Ajax to submit a Web form.

First of all, you have to control the HTML produced by your framework and be able to adapt the Web components for your application if you want to implement conventions that minimize configuration. There are highly customizable Web frameworks, such as JavaServer Faces (JSF), but their components are not always easy to customize. For example, if you want to change the HTML produced by a JSF component, you normally have to recode the component's renderer and implement a new custom tag. It would be much simpler if you just had to change the HTML in a JSP file. This article shows that it is possible to let the developer be in charge of the framework by creating JSP-based components.

Using JSP tag files to build Web components

JSP tag files are the ideal solution for simplifying Web component development because they let you create custom tag libraries, using the JSP syntax. In addition, tag files are deployed just like JSP pages and do not need a Tag Library Descriptor (TLD) because they use naming and setup conventions defined by the JSP standard, which also provides directives for declaring the tag's attributes within the JSP tag file.

When changed, a JSP tag file is recompiled and reloaded by the application server without having to restart the application, which makes both development and testing much easier. JSP tag files are fast because they are backed by automatically generated Java™ classes much like JSP pages are translated into Servlet classes.

The developerWorks Ajax resource center

Check out the [Ajax resource center](#), your one-stop shop for free tools, code, and information on developing Ajax applications. The [active Ajax community forum](#), hosted by Ajax expert Jack Herrington, will connect you with peers who might just have the answers you're looking for right now.

This article demonstrates how you can build customizable Web components, using JSP tag files and JSTL instead of JSF. The main goals are to be able to change the code that dynamically generates HTML easily, control how the HTTP (or Ajax) requests are processed, and implement conventions that simplify development.

The samples presented throughout the article can be grouped into a mini-framework that you can use instead of Struts or JSF to build Web forms. If you are just starting to use Java for Web applications, you'll appreciate its simplicity and you don't need to learn anything new because the framework's tags have the same names and same attributes as the HTML tags.

If you are an experienced developer, you'll find the framework useful for applications that must exploit the full power of Ajax and DHTML. You won't be restricted by any application model, you'll be able to change the framework's 250 lines of JSP code to generate the HTML that produces the best results in the Web browser, and you'll have the freedom to process the HTTP requests any way that is suitable for your application.

In addition, there are no framework-specific configuration files and no extra classes to manage. Each page can use a plain old Java object (POJO) as a data model or you may even use a Map object instead of the JavaBean instance if the data processing is simple enough to be done with JSP code instead of Java code.

Binding form elements to JavaBean properties

One of the main functions that a Web framework must provide is to bind the UI components to the properties of a data model. This means the framework must get

the data from the JavaBean objects and include it into the HTML form when a Web page is requested. When the user submits the form, the framework must get the request parameters and store the updated values back into the data model. For example, the JSF framework lets you specify the data binding for an input component with the `value` attribute (see Listing 1):

Listing 1. JSF data binding

```
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
...
<h:inputTextarea value="#{dataModel.address}" rows="3" cols="30"/>
```

The data model must be configured for JSF in an XML file as shown in Listing 2:

Listing 2. JSF configuration for the data model

```
<managed-bean>
  <managed-bean-name>dataModel</managed-bean-name>
  <managed-bean-class>formsdemo.AddressBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

This section demonstrates how to implement the same input component, using JSTL and JSP tag files. You might wonder why to do this if the component is already available in the JSF framework. You'll fully understand the benefits of JSP tag files if you compare the `textarea.tag` file of this article with the source code of the Java classes that implement the equivalent component in the JSF framework. Then, think how much work you have to put into other components that are not provided by the JSF framework or by another third-party library.

Processing Web forms with JSTL

Listing 3 shows a simple Web form containing a `<textarea>` element whose value is retrieved from an `AddressBean` instance. The `<jsp:useBean>` tag creates the JavaBean object and places it into the JSP `request` scope. The value of the `address` property is included in the Web page with the `<c:out>` tag of JSTL.

Listing 3. JSP page using JSTL to code data binding

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<jsp:useBean id="dataModel" scope="request" class="formsdemo.AddressBean"/>

<form method="POST">
  <c:if test="${pageContext.request.method == 'POST' && !empty param.address}">
    <c:set target="${dataModel}" property="address" value="${param.address}"/>
  </c:if>
  <textarea name="address" rows="3" cols="30"
    ><c:out value="${dataModel.address}"/></textarea>
  <br><input type="submit" value="Submit"/>
```

```
</form>
```

When the user clicks the Submit button, the Web browser sends the user's input back to the same page because the `<form>` element has no `action` attribute. Then, the application server executes the JSP page, which uses the `<c:set>` tag of JSTL to store the value of the `address` parameter into the `dataModel` bean.

Building the `<df:textarea>` component

If you compare Listings 1 and 3, you'll notice the JSF page defines the text area component with a single line while the JSTL-based page needs 5 lines to bind the `<textarea>` element to the JavaBean property. This code fragment can be reduced back to a single line by moving the JSTL code into a reusable tag file that outputs the `<textarea>` element.

The `textarea.tag` file (see Listing 4) declares only the `name` attribute that is used for the data-binding convention (the form element and the JavaBean property keeping the element's value must have the same name). Any other attributes, such as `rows` or `cols`, will be stored in a `Map` object whose `dynAttr` identifier is specified through the `dynamic-attributes` feature of the `<%@tag%>` directive. These dynamic attributes are outputted in a loop controlled with the `<c:forEach>` tag of JSTL. Any `"`, `&`, `<` and `>` characters are replaced with `"`, `&`, `<` and `>` by the `fn:escapeXml()` function of JSTL.

Listing 4. The `textarea.tag` file

```
<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ 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" %>

<c:if test="${pageContext.request.method == 'POST' && !empty param[name]}">
  <c:set target="${dataModel}" property="${name}" value="${param[name]}" />
</c:if>

<textarea name="${name}"
  <c:forEach var="attr" items="${dynAttr}">
    ${attr.key}="${fn:escapeXml(attr.value)}"
  </c:forEach>
><c:out value="${dataModel[name]}" /></textarea>
```

The Web page shown in Listing 5 declares the `df` prefix for the JSP library containing the tag file. Then, the page uses the `<df:textarea>` component to produce the `<textarea>` element that lets the user enter a value for the `address` property of the `dataModel` object.

Listing 5. JSP page using the tag file

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

<jsp:useBean id="dataModel" scope="request" class="formsdemo.AddressBean"/>

<form method="POST">
  <df:textarea name="address" rows="3" cols="30"/>
  <br><input type="submit" value="Submit"/>
</form>
```

Using skins and style conventions

In a previous article of mine titled "Enhance the appearance of your JSF pages" (see [Resources](#)), I presented a technique for setting the default attributes of the standard JSF components. A custom JSF component was used to traverse the view tree and set the `styleClass` attribute of each component, which works very well for simple JSF components that render a single HTML element.

A non-standard JSF component producing a larger HTML fragment, such as a tree or a table, may not let you set the style of any HTML element. Assuming that the JSF component uses Java code to generate the HTML, your only option is to re-code the JSF component's renderer so you can change the styles of the HTML elements. If you use JSP tag files and JSTL, however, you have full access to the JSP code that outputs the HTML, which means you can change it and adapt it for your application any way you like.

Moving reusable code fragments into separate JSP tag files

The previous section presented the `<df:textarea>` component, which uses JSTL to produce a `<textarea>` element that is bound to a JavaBean property. If you want to build additional Web components, it is a good idea to place the common code fragments into reusable tag files.

The `<c:forEach>` loop that outputs the dynamic attributes can be moved into a separate tag file named `attrList.tag` (see [Listing 6](#)). In addition to making the code more compact, this change allows you to implement features that are useful to all components, such as adding default style classes to the HTML tags.

The `attrList.tag` file declares an attribute named `tag`, which is used in combination with the `skin` parameter to build the class names that are included within the `class` attribute. The `<c:forEach>` loop outputs all attributes that the given `map` contains except the `class` attribute, which is added after the loop.

Listing 6. The `attrList.tag` file

```
<%@ attribute name="tag" required="true" rtexprvalue="true" %>
<%@ attribute name="map" required="true" rtexprvalue="true"
  type="java.util.Map" %>
```

```

<%@ tag 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" %>

<c:if test="${!empty initParam.skin}">
  <c:set var="classAttr" value="${initParam.skin}_${tag}"/>
</c:if>

<c:forEach var="attr" items="${map}">
  <c:if test="${attr.key != 'class'}">
    ${attr.key}="${fn:escapeXml(attr.value)}"
  </c:if>
  <c:if test="${attr.key == 'class'}">
    <c:set var="classAttr" value="${classAttr} ${attr.value}"/>
  </c:if>
</c:forEach>

<c:if test="${!empty classAttr}">
  class="${fn:escapeXml(classAttr)}"
</c:if>

```

The `skin` parameter is specified in the `web.xml` file (see Listing 7) of the Web application:

Listing 7. Configuring the skin parameter in web.xml

```

<web-app ...>
  <context-param>
    <param-name>skin</param-name>
    <param-value>default</param-value>
  </context-param>
</web-app>

```

Updating the <df:textarea> component

Listing 8 shows the changed version of the `textarea.tag` file, which uses `<dfu:attrList>` to output the dynamic attributes of the `<textarea>` element:

Listing 8. The updated version of textarea.tag

```

<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ 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="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>

<c:if test="${pageContext.request.method == 'POST' && !empty param[name]}">
  <c:set target="${dataModel}" property="${name}" value="${param[name]}/>
</c:if>

<textarea name="${name}" <dfu:attrList tag="textarea" map="${dynAttr}"/>
><c:out value="${dataModel[name]}/></textarea>

```

The style rules can be defined in a CSS file. For example, if you want to change the border of every `<textarea>` element produced by `<df:textarea>`, you would

just have to code a style rule like the one from Listing 9:

Listing 9. Defining the default style for <textarea> elements

```
.default_textarea
{ border-color: #A0A0A0; border-style: solid; border-width: thin; }
```

Coding form-processing and page-navigation rules

The JSF framework has a complicated request processing life cycle, which is split into 6 distinct phases. Few Java developers fully understand it and are able to customize it. For a high-traffic Ajax application, you might want a simpler mechanism for processing the HTTP requests to avoid the CPU overload. This section shows how easily you can implement this with the help of JSTL and JSP tag files.

Creating the <df:form> component

The `form.tag` file (shown in Listing 10) produces a `<form>` element and uses JSTL to set variables that are accessed by its nested tags as you'll see later in this article. Instead of using the `dataModel` object as in the previous JSP examples, the tag file takes a `model` attribute, which is placed into the `page` scope like any other attribute of a JSP tag file.

The `form.tag` file copies some of its attributes into the `request` scope with the `<c:set>` tag of JSTL so that other tag files can get the values of the `name`, `model`, and `action` attributes, using the `formName`, `formModel` and `formAction` variables. In addition, the `formPost` variable indicates if the HTTP method is `POST` and `<jsp:useBean>` creates a `HashMap` instance that will store any processing errors.

Listing 10. The form.tag file

```
<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="action" required="false" rtexprvalue="true" %>
<%@ attribute name="method" required="false" rtexprvalue="true" %>
<%@ attribute name="model" required="true" rtexprvalue="true"
    type="java.lang.Object" %>
<%@ 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="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>

<c:set var="formName" scope="request" value="{name}"/>
<c:set var="formModel" scope="request" value="{model}"/>
<c:if test="{!empty action}">
    <c:set var="formAction" scope="request" value="{action}"/>
</c:if>
<c:set var="formPost" scope="request"
    value="{fn:toUpperCase(pageContext.request.method) == 'POST'}"/>
```

```

<jsp:useBean id="formErrors" scope="request" class="java.util.HashMap" />
<form name="${name}" method="POST" <df:attrList tag="form" map="${dynAttr}" />>
  <jsp:doBody/>
</form>

<c:if test="${formPost && empty formErrors && !empty formAction}">
  <c:set var="forwardURL" value="${formAction}" />
</c:if>

<c:remove var="formName" scope="request" />
<c:remove var="formModel" scope="request" />
<c:remove var="formAction" scope="request" />
<c:remove var="formPost" scope="request" />
<c:remove var="formErrors" scope="request" />

<c:if test="${!empty forwardURL}">
  <jsp:forward page="${forwardURL}" />
</c:if>

```

The `<form>` element will have the same HTML attributes as the `<df:form>` tag, except `method`, which is always `POST`, and `action`, which is omitted so the data is posted back to the same page. The `<form>` element will also contain the output generated with `<jsp:doBody>`, which executes the JSP code placed between `<df:form>` and `</df:form>` (see Listing 11). If the HTTP method is `POST` and there are no errors, the `form.tag` file forwards the request for further processing to the page whose URL is specified in the `action` attribute. Before that, the tag file removes its variables from the `request` scope with the `<c:remove>` tag of JSTL.

Listing 11. Using `<df:form>` in a JSP page

```

<df:form name="..." model="${...}" action="...">
  ...
  <df:textarea .../>
  ...
</df:form>

```

Handling user errors and application exceptions

The `textarea.tag` file presented earlier didn't handle any exceptions that might occur while setting the JavaBean property that is bound to the form element. This is remedied by the `setProp.tag` file (shown in Listing 12), which uses the `<c:catch>` tag of JSTL to catch any exception that could be thrown by the `set` method of the JavaBean property or even by the `<c:set>` tag if there is a conversion error, such as `NumberFormatException`:

Listing 12. The `setProp.tag` file

```

<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="array" required="false" rtexprvalue="true"
  type="java.lang.Boolean" %>
<%@ attribute name="bool" required="false" rtexprvalue="true"
  type="java.lang.Boolean" %>
<%@ tag 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="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>

<c:set var="propValue" value="{null}"/>
<c:if test="{formPost && empty formErrors[name]}">
  <c:if test="{!array && !bool && !empty param[name]}">
    <c:set var="propValue" value="{param[name]}/>
  </c:if>
  <c:if test="{!array && bool}">
    <c:set var="propValue" value="{!empty param[name]}/>
  </c:if>
  <c:if test="{array && fn:length(paramValues[name]) > 0}">
    <c:set var="propValue" value="{paramValues[name]}/>
  </c:if>
</c:if>

<c:if test="{propValue != null}">
  <c:catch var="exception">
    <c:set target="{formModel}" property="{name}" value="{propValue}"/>
  </c:catch>
  <c:if test="{exception != null}">
    <dfu:addError name="{name}" msg="{exception.message}"
      exception="{exception}"/>
  </c:if>
</c:if>

```

The `setProp` tag file can be used to set properties that have a type accepted by `<c:set>`, such as `String`, `int`, `float`, `boolean` as well as indexed properties, which can be `String` arrays. If a type conversion error occurs or an exception is thrown, the `addError` tag file (see Listing 13) is used to put the error message into the `formErrors` map:

Listing 13. The `addError` tag file

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="msg" required="true" rtexprvalue="true" %>
<%@ attribute name="exception" required="false" rtexprvalue="true"
  type="java.lang.Throwable" %>

<c:if test="{!empty formErrors[name]}">
  <c:set var="msg" value="{formErrors[name]}; {msg}"/>
</c:if>
<c:set target="{formErrors}" property="{name}" value="{msg}"/>

<c:if test="{exception != null}">
  <% ((Throwable) jspContext.getAttribute("exception")).printStackTrace(); %>
</c:if>

```

The final version of the `textarea` tag file (shown in Listing 14) uses `<dfu:setProp>` to set the JavaBean property. In addition, the tag file lets you specify a default value in the JSP page, between `<df:textarea>` and `</df:textarea>`. If the bean property has an empty value, the tag file uses `<jsp:doBody>` to obtain the contents of the custom tag from the JSP page.

Listing 14. The final version of textarea.tag

```
<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ 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="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>

<dfu:setProp name="${name}" />

<c:set var="textareaValue" value="${formModel[name]}" />
<c:if test="${empty textareaValue}">
  <jsp:doBody var="textareaValue" />
</c:if>

<textarea name="${name}" <dfu:attrList tag="textarea" map="${dynAttr}" />
><c:out value="${textareaValue}" /></textarea>
```

Developing more UI components

So far, you've seen how to create your own text area and form components and how to process the HTTP requests. This section presents additional components, such as lists, input fields, checkboxes, radio and Submit buttons so that you can build fully functional Web forms.

Creating the <df:select> and <df:option> components

In JSP pages, the <df:option> tags will be nested within <df:select> just like in the case of the <option> and <select> elements of HTML. Therefore, the select.tag file (see Listing 15) copies its name and multiple attributes into the request scope so they can be accessed in option.tag. Then, the select.tag file uses <dfu:setProp> to set the model property that has the same name as the form element. If the multiple attribute is true, the property's type should be an array.

Listing 15. The select.tag file

```
<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="multiple" required="false" rtexprvalue="true"
  type="java.lang.Boolean" %>
<%@ 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="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>

<c:set var="selectName" scope="request" value="${name}" />
<c:set var="selectMultiple" scope="request" value="${multiple}" />

<dfu:setProp name="${name}" array="${multiple}" />

<select name="${name}"
  <c:if test="${multiple}">
```

```

        multiple
    </c:if>
    <dfu:attrList tag="select" map="{dynAttr}"/>
>
    <jsp:doBody/>
</select>

<c:remove var="selectName" scope="request"/>
<c:remove var="selectMultiple" scope="request"/>

```

In a JSP page, the `<df:select>` tag can have dynamic attributes, such as `id` or `class`, which are passed to the outputted `<select>` element with the help of `<dfu:attrList>`. The `<jsp:doBody>` tag of JSP executes the contents of `<df:select>`, which would normally consist of several `<df:option>` tags.

Before generating the `<option>` element, the `option.tag` file (shown in Listing 16) sets the `optionLabel` and `optionValue` variables. The `<dfu:isSelected>` tag, which is used in `option.tag`, outputs its body consisting of the selected string only if the option's value is equal to the value of the model property that has the same name as the `<select>` element. Therefore, the property of the data model determines the initial selection of the list produced by `<df:select>` and `<df:option>`.

Listing 16. The option.tag file

```

<%@ attribute name="value" required="false" rtexprvalue="true" %>
<%@ 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="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>

<jsp:doBody var="optionLabel"/>

<c:set var="optionValue" value="{value}"/>
<c:if test="{empty optionValue}">
    <c:set var="optionValue" value="{optionLabel}"/>
</c:if>

<option
    <c:if test="{!empty value}">
        value="{fn:escapeXml(value)}"
    </c:if>
    <dfu:isSelected name="{selectName}" value="{optionValue}"
        array="{selectMultiple}">
        selected
    </dfu:isSelected>
    <dfu:attrList tag="option" map="{dynAttr}"/>
><c:out value="{optionLabel}"/></option>

```

Listing 17 shows the `isSelected.tag` file, which is invoked from `option.tag`. If the `array` attribute is `false`, the `isSelected.tag` file compares the `value` attribute with `formModel[name]`. If the `array` attribute is `true`, `formModel[name]` is expected to be an array and `isSelected.tag` compares each element with the `value` attribute.

Listing 17. The isSelected.tag file

```

<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="value" required="true" rtexprvalue="true" %>
<%@ attribute name="array" required="false" rtexprvalue="true"
    type="java.lang.Boolean" %>
<%@ tag 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" %>

<c:set var="selected" value="false"/>

<c:if test="${!array}">
    <c:set var="selectedValue" value="{formModel[name]}/>
    <c:if test="${!empty selectedValue}">
        <c:if test="{selectedValue == value}">
            <c:set var="selected" value="{true}"/>
        </c:if>
    </c:if>
</c:if>

<c:if test="{array}">
    <c:set var="selectedValues" value="{formModel[selected]}/>
    <c:if test="{fn:length(selectedValues) > 0}">
        <c:forEach var="selectedValue" items="{selectedValues}">
            <c:if test="{selectedValue == value}">
                <c:set var="selected" value="{true}"/>
            </c:if>
        </c:forEach>
    </c:if>
</c:if>

<c:if test="{selected}">
    <jsp:doBody/>
</c:if>

```

The `<jsp:doBody>` tag from `isSelected.tag` executes the JSP code placed between `<dfu:isSelected>` and `</dfu:isSelected>` in `option.tag`, which basically outputs the `selected` string in this case. The `<dfu:isSelected>` component is also used in `input.tag` to verify if a radio button should be checked.

Building the `<df:input>` component

The `input.tag` file (see Listing 18) outputs an `<input>` element. Depending on the value of the `type` attribute, the tag file executes different JSP fragments, which set the model property with the same name as the HTML element.

If `type` is `text`, `password` or `hidden`, the `<dfu:setProp>` tag stores the request parameter into the JavaBean object when the HTTP method is POST. Next, the value of the bean property is retrieved with `formModel[name]` and stored into the `inputValue` variable whether the HTTP method is GET or POST.

In case of a radio button, the `<dfu:isSelected>` tag presented earlier is used to compare the `value` attribute with the value of the model property that has the same name as the radio button. If the two values coincide, the `inputChecked` variable is made `true`. If the `type` attribute is `checkbox`, the bean property's type is expected

to be boolean.

Listing 18. The input.tag file

```
<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="type" required="true" rtexprvalue="true" %>
<%@ attribute name="value" required="false" rtexprvalue="true" %>
<%@ attribute name="action" required="false" rtexprvalue="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="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>

<c:set var="inputType" value="{fn:toLowerCase(type) }"/>
<c:set var="inputValue" value="{value}"/>
<c:set var="inputChecked" value="{false}"/>

<c:choose>
  <c:when test="{inputType=='text' || inputType=='password' || inputType=='hidden'}">
    <dfu:setProp name="{name}"/>
    <c:set var="inputValue" value="{formModel[name]}"/>
  </c:when>
  <c:when test="{inputType == 'radio'}">
    <dfu:setProp name="{name}"/>
    <dfu:isSelected name="{name}" value="{value}">
      <c:set var="inputChecked" value="{true}"/>
    </dfu:isSelected>
  </c:when>
  <c:when test="{inputType == 'checkbox'}">
    <dfu:setProp name="{name}" bool="true"/>
    <c:if test="{formModel[name]}">
      <c:set var="inputChecked" value="{true}"/>
    </c:if>
  </c:when>
  <c:when test="{inputType=='submit'}">
    <c:if test="{formPost && !empty param[name] && !empty action}">
      <c:set var="formAction" scope="request" value="{action}"/>
    </c:if>
  </c:when>
</c:choose>

<input name="{name}" type="{type}"
  <c:if test="{!empty inputValue}">
    value="{fn:escapeXml(inputValue) }"
  </c:if>
  <c:if test="{inputChecked}">
    checked
  </c:if>
  <dfu:attrList tag="input_{type}" map="{dynAttr}"/>
>
```

For submit buttons, the input.tag file accepts the action attribute, which can be used to specify the URL of a page that should replace the form's action when the submit button is clicked. Therefore, a form can have multiple submit buttons, and each button can take the request to a different page for generating the HTML response. This is much simpler than specifying navigation rules in a configuration file as in the case of the JSF framework.

Submitting Web forms with Ajax

In another article of mine titled "Use XMLHttpRequest to submit JSF forms" (see [Resources](#)), I showed how to send the data of a Web form with Ajax. That article contains a JSF example named `SupportForm.jsp`. A similar example is presented here so that you can compare the form of the JSF article with the Web form based on the JSP tag files of this article.

Creating forms backed by JavaBeans or Java Maps

The `SupportBean` class (in Listing 19) is a simple bean containing several properties that are bound to the elements of the example Web form:

Listing 19. The `SupportBean` class

```
package formsdemo;

public class SupportBean implements java.io.Serializable {
    private String name;
    private String email;
    private String versions[];
    private String platform;
    private String browser;
    private boolean crash;
    private String problem;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    ...
}
```

Listing 20 shows the `SupportForm.jsp` page, which uses the library of tag files presented in the previous sections of this article. The Web components have the same attributes as the HTML elements that they produce. The `<df:form>` tag has the additional `model` attribute, which specifies the JavaBean object where the submitted form data will be stored.

Listing 20. The `SupportForm.jsp` example

```
<%@ taglib prefix="df" tagdir="/WEB-INF/tags/dynamic/forms" %>

<jsp:useBean id="supportBean" scope="request" class="formsdemo.SupportBean"/>

<html>
<head>
    <title>Support Form</title>
    <link rel="stylesheet" href="forms.css" type="text/css">
</head>
<body>

    <h1>Support Form</h1>

    <df:form name="supportForm" model="${supportBean}"
```

```

        action="SupportConfirm.jsp">

<p>Name: <br>
<df:input name="name" type="text" size="40" />

<p>Email: <br>
<df:input name="email" type="text" size="40" />

<p>Versions: <br>
<df:select name="versions" multiple="true" size="5">
  <df:option>2.0.1</df:option>
  <df:option>2.0.0</df:option>
  <df:option>1.1.0</df:option>
  <df:option>1.0.1</df:option>
  <df:option>1.0.0</df:option>
</df:select>

<p>Platform: <br>
<df:input name="platform" type="radio" value="Windows" /> Windows <br>
<df:input name="platform" type="radio" value="Linux" /> Linux <br>
<df:input name="platform" type="radio" value="Mac" /> Mac <br>

<p>Browser: <br>
<df:select name="browser" size="1">
  <df:option value=""></df:option>
  <df:option value="IE">IE</df:option>
  <df:option value="Firefox">Firefox</df:option>
  <df:option value="Netscape">Netscape</df:option>
  <df:option value="Mozilla">Mozilla</df:option>
  <df:option value="Opera">Opera</df:option>
  <df:option value="Safari">Safari</df:option>
</df:select>

<p><df:input name="crash" type="checkbox" /> Causes browser crash

<p>Problem: <br>
<df:textarea name="problem" rows="10" cols="40" />

<p><df:input name="submit" type="submit" value="Submit" />

</df:form>

</body>
</html>

```

The `<df:form>` tag used in `SupportForm.jsp` forwards the HTTP request to the `SupportConfirm.jsp` page (shown in Listing 21) whose URL is specified in the `action` attribute. This confirmation page outputs the properties of the data model.

Listing 21. The `SupportConfirm.jsp` page

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>Support Confirmation</title>
</head>
<body>

  <h1>Support Confirmation</h1>

  <p>Your information was received.

  <p>Name: <c:out value="\${supportBean.name}" /><br>

```

```

<p>Email: <c:out value="\${supportBean.email}"/><br>
<p>Versions: <c:forEach var="version" items="\${supportBean.versions}">
  <c:out value="\${version}"/></c:forEach><br>
<p>Platform: <c:out value="\${supportBean.platform}"/><br>
<p>Browser: <c:out value="\${supportBean.browser}"/><br>
<p>Causes browser crash: <c:out value="\${supportBean.crash}"/>
<p>Problem: <c:out value="\${supportBean.problem}"/><br>

</body>
</html>

```

The JSP EL and the JSTL tags, such as `<c:set>`, have the same syntax for accessing the properties of a JavaBean and the elements of a Map instance. Therefore, it is possible to use a `java.util.HashMap` instead of the `formsdemo.SupportBean` object in the previous example. The only thing you have to change is the class name in the `<jsp:useBean>` tag (see Listing 22):

Listing 22. The SupportForm2.jsp example

```

<jsp:useBean id="supportBean" scope="request" class="java.util.HashMap"/>
...
<df:form name="supportForm" model="\${supportBean}" action="SupportConfirm.jsp">
...
</df:form>

```

Adding the Ajax code to the Web form

The `AutoSaveScript.js` file, which you can find in the source code archive, was fully described in "Use XMLHttpRequest to submit JSF forms" (see [Resources](#)). The JavaScript file contains several functions that are reused in this article. The `getFormData()` function (shown in Listing 23) takes a `form` object and returns the form's data encoded in a string:

Listing 23. The getFormData() function of AutoSaveScript.js

```

function getFormData(form) {
    var dataString = "";

    function addParam(name, value) {
        dataString += (dataString.length > 0 ? "&" : "")
            + escape(name).replace(/\+/g, "%2B") + "="
            + escape(value ? value : "").replace(/\+/g, "%2B");
    }

    var elemArray = form.elements;
    for (var i = 0; i < elemArray.length; i++) {
        var element = elemArray[i];
        var elemType = element.type.toUpperCase();
        var elemName = element.name;
        if (elemName) {
            if (elemType == "TEXT"
                || elemType == "TEXTAREA"
                || elemType == "PASSWORD"
                || elemType == "HIDDEN")
                addParam(elemName, element.value);
            else if (elemType == "CHECKBOX" && element.checked)

```

```

        addParam(elemName, element.value ? element.value : "On");
    else if (elemType == "RADIO" && element.checked)
        addParam(elemName, element.value);
    else if (elemType.indexOf("SELECT") != -1)
        for (var j = 0; j < element.options.length; j++) {
            var option = element.options[j];
            if (option.selected)
                addParam(elemName,
                    option.value ? option.value : option.text);
        }
    }
}
return dataString;
}

```

The `submitFormData()` function of the `AutoSaveScript.js` file (see Listing 24) has two parameters: a form object whose data is sent to the server with Ajax and a callback function that is invoked to process the Ajax response:

Listing 24. The `submitFormData()` function of `AutoSaveScript.js`

```

function submitFormData(form, callback) {
    var xhr;
    if (window.ActiveXObject)
        xhr = new ActiveXObject("Microsoft.XMLHTTP");
    else if (window.XMLHttpRequest)
        xhr = new XMLHttpRequest();
    else
        return null;

    var method = form.method ? form.method.toUpperCase() : "GET";
    var action = form.action ? form.action : document.URL;
    var data = getFormData(form);

    var url = action;
    if (data && method == "GET")
        url += "?" + data;
    xhr.open(method, url, true);

    function submitCallback() {
        if (callback && xhr.readyState == 4 && xhr.status == 200)
            callback(xhr);
        ...
    }
    xhr.onreadystatechange = submitCallback;

    xhr.setRequestHeader("Ajax-Request", "Auto-Save");
    if (method == "POST") {
        xhr.setRequestHeader("Content-Type",
            "application/x-www-form-urlencoded");
        xhr.send(data);
    } else
        xhr.send(null);

    return xhr;
}

```

Listing 25 shows the `submitAllForms()` function, which sends the data of all forms that the current Web page contains. In addition, this function deletes the previous `XMLHttpRequest` objects from the memory to prevent a memory leak in the Web browser.

Listing 25. The submitAllForms() function of AutoSaveScript.js

```

var autoSaveXHR = new Array();

function submitAllForms(callback) {
    var formArray = document.forms;
    for (var i = 0; i < formArray.length; i++) {
        if (autoSaveXHR[i]) {
            var oldXHR = autoSaveXHR[i];
            oldXHR.onreadystatechange = function() { };
            oldXHR.abort();
            delete oldXHR;
        }
        autoSaveXHR[i] = submitFormData(formArray[i], callback);
    }
}

```

The SupportForm3.jsp page (shown in Listing 26) is a modified version of the Web form presented earlier. This JSP page imports AutoSaveScript.js in its header and defines a callback function that shows the Ajax response in a separate window. The form also contains a second button labeled **Submit with Ajax**, which invokes the submitSupportForm() function that returns false so the form isn't resubmitted by the Web browser.

Listing 26. The SupportForm3.jsp example

```

<%@ taglib prefix="df" tagdir="/WEB-INF/tags/dynamic/forms" %>

<jsp:useBean id="supportBean" scope="request" class="formsdemo.SupportBean"/>

<html>
<head>
    ...
    <script type="text/javascript" src="AutoSaveScript.js">
    </script>
    <script type="text/javascript">
        function ajaxCallback(xhr) {
            confirmWindow=window.open("", "_blank",
                "menubar=no, resizable=yes, scrollbars=yes, width=600, height=600");
            confirmWindow.document.write(xhr.responseText);
            confirmWindow.document.close();
        }

        function submitSupportForm() {
            submitAllForms(ajaxCallback);
            return false;
        }
    </script>
</head>
<body>
    ...
    <df:form name="supportForm" model="${supportBean}"
        action="SupportConfirm.jsp">
        ...
        <df:input name="ajaxSubmit" type="submit" value="Submit with Ajax"
            onclick="return submitSupportForm()"/>
    </df:form>
</body>
</html>

```

Conclusion

In this article, you have learned how to build lightweight components that use conventions to simplify Web development. It took less than 10K of JSP code to implement a set of tag files that produce the basic form elements: lists, text fields, radio buttons, check boxes, and submit buttons. The components are fully functional, but some essential features, such as data validation and error reporting, are missing. Stay tuned for the next article of this series, which will show how to create JSP tag files doing validation on the server-side and producing JavaScript code that performs validation on the client-side as well.

Downloads

Description	Name	Size	Download method
Sample application for this article	wa-aj-simplejava216	216KB	HTTP

[Information about download methods](#)

Resources

- [Participate in the discussion forum for this content.](#)
- "[Enhance the appearance of your JSF pages](#)" (Andrei Cioroianu, developerWorks, January 2008) shows how to implement default styles for the standard JSF components. This article is the first part of a series titled "Craft Ajax applications using JSF with CSS and JavaScript."
- "[Use XMLHttpRequest to submit JSF forms](#)" (Andrei Cioroianu, developerWorks, August 2007) contains the JSF-based version of the SupportForm.jsp example and fully describes the JavaScript functions of the AutoSaveScript.js file. This article is the first part of a series titled "Auto-save JSF forms with Ajax."
- 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 Technology home 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.