

Ajax and Java development made simpler, Part 3: Build UI features based on DOM, JavaScript, and JSP tag files

Implement user-friendly and safe validation for your Web forms,
using JavaScript and JSP tag files

Skill Level: Intermediate

[Andrei Cioroianu](#)

Senior Java Developer and Consultant
Devsphere

22 Jul 2008

In the first part of this series, you saw how to generate JavaScript code for sending Ajax requests and processing Ajax responses. The second part showed how to create HTML forms, using conventions and JSP tag files to minimize setup and configuration. In this third part of the series, you'll learn how to develop client-side validators based on JavaScript as well as server-side validators, which are implemented as JSP tag files backing up their JavaScript counterparts. You'll also learn how to use resource-bundles that are reloaded automatically when changed, without requiring the restart of the application.

Client-side validation is useful because it can reduce or even eliminate the cases when the form is returned to the user for correcting errors. Nevertheless, the JavaScript-based validation is not sufficient because it can be bypassed if the data is submitted to the server with a program or if JavaScript is disabled in the user's browser. This article shows how to implement both client-side and server-side validation.

Creating a hierarchy of client-side validators

This section shows you how to use constructors and prototypes in JavaScript for

building a hierarchy of objects that validate the user input in the Web browser. The code presented throughout this section can be found in the `valid.js` file of the sample application (see [Download](#)).

Developing the base validator

In JavaScript, any function may be used with the `new` operator to create objects. The so-called "constructor" must use `this` to properly initialize the object's properties. Listing 1 shows the `Validator()` function, which takes four parameters: a form name, an element name, an ID, and a message. The values of these parameters are stored into the current object as properties.

Objects created with the same constructor can share properties, which are kept in a `prototype` object. In Listing 1, `defaultMsgIndex` is a property shared by all instances created with `new Validator(...)`. When you try accessing a property or a method of an object, the JavaScript engine looks first among the members of the object. If the property or method isn't found there, the engine verifies the object's `prototype`, which can have its own `prototype`, and so on. Therefore, the `prototype` chain can be used to implement inheritance, as you'll see later. In the samples of this article, `Validator` is the base of an object hierarchy.

Listing 1. The `Validator()` constructor

```
function Validator(formName, elemName, outId, msg) {
  if (formName)
    this.formName = formName;
  if (elemName)
    this.elemName = elemName;
  if (outId)
    this.outId = outId;
  if (msg)
    this.msgList = ["", msg];
}

Validator.prototype.defaultMsgIndex = 0;
```

Any JavaScript function can become the method of any object. You just have to use `this.methodName = functionName` in the object's constructor. Because a method is normally shared by all instances created with the same constructor, you can associate the method with the `prototype` of the constructor, using `constructorName.prototype.methodName` as shown in Listing 2:

Listing 2. The `getFormValues()` method of `Validator`

```
Validator.prototype.getFormValues = function() {
  var elemValues = new Array();
  var form = document.forms[this.formName];
  var formElements = form.elements;
  for (var i = 0; i < formElements.length; i++) {
    var element = formElements[i];
    if (element.name == this.elemName) {
```

```

var elemType = element.type.toLowerCase();
if (elemType == "text" || elemType == "textarea"
    || elemType == "password" || elemType == "hidden")
    elemValues[elemValues.length] = element.value;
else if (elemType == "checkbox" && element.checked)
    elemValues[elemValues.length]
        = element.value ? element.value : "On";
else if (elemType == "radio" && element.checked)
    elemValues[elemValues.length] = 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) {
            elemValues[elemValues.length]
                = option.value ? option.value : option.text;
        }
    }
}
}
return elemValues;
}

```

The `getFormValues()` method in Listing 2 returns the values of the form elements whose names coincide with the value of the `elemName` property. The DOM object representing the HTML form in the Web browser is obtained with `document.forms[this.formName]`. Then, `getFormValues()` iterates over the form's elements, treating each element according to its HTML type.

In the case of a text, password, or hidden field, the `getFormValues()` method just adds the element's value to the returned array. If the element's type is checkbox or radio, the value is added only if the HTML element is checked. Finally, if the element is a list, the values of the selected items are stored into the returned array.

Each validator of the hierarchy will need a method to verify a single value. In the case of the base `Validator`, the `verify()` method (shown in Listing 3) returns 0, but the other validators will replace this method with their own. The value returned by `verify()` is used to lookup for a message in `msgList`.

Listing 3. The `verify()` method of `Validator`

```

Validator.prototype.verify = function(value) {
    return 0;
}

```

The `showMsg()` method (see Listing 4) inserts a message into an HTML element by setting the `innerHTML` property. The object representing the HTML element is obtained with `document.getElementById(this.outId)`. The message is encoded with the `htmlEncode()` function, which was described in [Part 1](#) of the series. The code of this auxiliary function can be found in `encode.js`.

Listing 4. The `showMsg()` method of `Validator`

```
Validator.prototype.showMsg = function(index) {
    document.getElementById(this.outId).innerHTML
        = htmlEncode(this.msgList[index]);
}
```

Listing 5 contains the `execute()` method, which iterates over the values returned by `getFormValues()` and invokes `verify()` for each value. The user input is considered valid if `msgIndex` is 0 at the end of the `execute()` method. In this case, `showMsg()` clears any previous messages by storing an empty string into the `innerHTML` property of the output element. Otherwise, the error message appears in the Web page after the call of `showMsg()`.

Listing 5. The `execute()` method of `Validator`

```
Validator.prototype.execute = function() {
    var msgIndex = this.defaultMsgIndex;
    var elemValues = this.getFormValues();
    for (var i = 0; i < elemValues.length; i++)
        if (elemValues[i]) {
            msgIndex = this.verify(elemValues[i]);
            if (msgIndex != 0)
                break;
        }
    this.showMsg(msgIndex);
    return msgIndex == 0;
}
```

Implementing concrete validators

The `valid.js` file contains a validator for required fields. The `RequiredValidator()` constructor (see Listing 6) takes the same parameters as the `Validator()` function. In order to initialize the object properties with `Validator()`, the `RequiredValidator()` constructor sets a method named `base()`. After calling `Validator()` with the help of `base()`, `RequiredValidator()` deletes the `base()` method from the current object because it is not needed anymore.

Listing 6. The `RequiredValidator()` constructor and its prototype settings

```
function RequiredValidator(formName, elemName, outId, msg) {
    this.base = Validator;
    this.base(formName, elemName, outId, msg);
    delete this.base;
}

RequiredValidator.prototype = new Validator();

RequiredValidator.prototype.defaultMsgIndex = 1;

RequiredValidator.prototype.verify = function(value) {
    return value.length > 0 ? 0 : 1;
}
```

The prototype for all `RequiredValidator` objects is a `Validator` instance, which has no properties because no parameters are passed to the `Validator()` constructor. By setting this prototype, `RequiredValidator` "inherits" the properties and methods defined for the `Validator`'s prototype.

For example, when you call the `execute()` method of a `RequiredValidator` object, the JavaScript engine will look first among the object's methods. Because the object doesn't have an `execute()` method, the next place to look for it is the object's prototype, followed by the prototype's prototype and so on. In this example, the `RequiredValidator`'s prototype is a `Validator` instance whose prototype has the `execute()` method.

The `defaultMsgIndex` property of the `RequiredValidator`'s prototype is set to 1 because an error should be signaled if the element's value is missing. The `verify()` method returns 0 if the value is not empty or 1 if the value of the required field has not been entered.

Listing 7 shows `LengthValidator`, which is very similar to `RequiredValidator`. The `LengthValidator()` constructor takes seven parameters: the form's name, the name of the element whose value must be validated, the ID of the element used to output error messages, minimum and maximum lengths for the validated value, and two error messages. The `LengthValidator`'s prototype is a new `Validator` instance just like in the case of `RequiredValidator`. The `verify()` method returns 0 if the value is valid, 1 if it's too short, or 2 in case it's too long.

Listing 7. The `LengthValidator()` constructor and its prototype settings

```
function LengthValidator(formName, elemName, outId, min, max, msgMin, msgMax) {
    this.base = Validator;
    this.base(formName, elemName, outId);
    delete this.base;
    this.min = min;
    this.max = max;
    this.msgList = ["", msgMin, msgMax];
}

LengthValidator.prototype = new Validator();

LengthValidator.prototype.verify = function(value) {
    if (value.length < this.min)
        return 1;
    if (value.length > this.max && this.max > 0)
        return 2;
    return 0;
}
```

To easily manage the validators of a Web page, the `valid.js` file defines the `ValidatorList()` constructor (see Listing 8), which creates an object wrapping an array of validators. The `add()` method appends a validator instance to the array. The `execute()` method iterates over the list of validators, invoking the `execute()` method of each validator whose properties match the given form name and the

optional element name. Therefore, the `execute()` method of `ValidatorList` can be used to validate a whole form or just an element of the form.

Listing 8. The `ValidatorList()` constructor and its prototype settings

```
function ValidatorList() {
    this.validatorArray = new Array();
}

ValidatorList.prototype.add = function(validator) {
    this.validatorArray[this.validatorArray.length] = validator;
}

ValidatorList.prototype.execute = function(formName, elemName) {
    var valid = true;
    for (var i = 0; i < this.validatorArray.length; i++) {
        var validator = this.validatorArray[i];
        if (validator.formName == formName)
            if (!elemName || validator.elemName == elemName)
                if (!validator.execute())
                    valid = false;
    }
    return valid;
}

var pageValidators = new ValidatorList();
```

Building a library of server-side validators

The previous section shows how to validate the user input on the client side, using JavaScript. Once the form data is submitted, it is a good practice to revalidate the user input on the server-side for security reasons or just in case JavaScript is disabled in the user's browser. This section discusses a set of JSP tag files acting as validators.

Tag files are an easy way to implement the validation on the server because they use the JSP syntax and are reusable across pages. In addition, the same tag files that verify the user input can generate the JavaScript code that does the validation in the Web browser. This technique reduces coding because you specify the validators' attributes only once in the Web page.

Using JSP tag files to implement validators

The JavaScript validators presented in the previous section need `` elements in the Web page to show their messages with the help of `innerHTML`. These elements are generated with the `errMsg.tag` file (shown in Listing 9):

Listing 9. The `errMsg.tag` file

```
<%@ attribute name="id" required="true" rtexprvalue="true" %>
<%@ tag body-content="scriptless" %>
```

```
<span id="${id}" class="ValidationError"><jsp:doBody/></span>
```

The `valid.css` file (see Listing 10) defines the `ValidationError` style class that is used for the error messages:

Listing 10. The `valid.css` file

```
.ValidationError { color: red; }
```

The `required.tag` file (see Listing 11) implements the server-side validation for the required fields and generates the JavaScript code fragments that setup the `RequiredValidator` objects on the client-side. The tag file defines two attributes: the name of the required field and an optional error message.

If the `msg` attribute is not provided, `required.tag` invokes another tag file with `<dvu:useDefaultMsg/>` to set the `defaultValidMsg` map, which contains the default messages of the validators. Then, the `required` message is stored into the `msg` variable.

Next, the `required.tag` file uses `<dvu:errMsg>` to generate the `` element that will wrap the error message, if any. If the request parameter with the given name is missing, meaning the user left the required field blank, the tag file outputs the message with `<c:out>` and adds the error to a map with the `<dfu:addError>` tag, which was presented in [Part 2](#) of this series.

Then, the `required.tag` file generates a bit of JavaScript code that will be executed in the Web browser. This code creates a `RequiredValidator` object, which is added to a list of JavaScript validators named `pageValidators`. The previous section of this article described `RequiredValidator`. The string parameters that are passed to the `RequiredValidator()` constructor are encoded with the `<da:jstring>` tag, which was presented in [Part 1](#) of this series.

Listing 11. The `required.tag` file

```
<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="msg" required="false" rtexprvalue="true" %>
<%@ tag body-content="empty" %>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="da" tagdir="/WEB-INF/tags/dynamic/ajax" %>
<%@ taglib prefix="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>
<%@ taglib prefix="dvu" tagdir="/WEB-INF/tags/dynamic/valid/util" %>

<c:if test="${empty msg}">
  <dvu:useDefaultMsg/>
  <c:set var="msg" value="${defaultValidMsg.required}"/>
</c:if>

<c:set var="outId" value="${name}Required"/>
```

```

<dvu:errMsg id="{outId}">
  <c:if test="{formPost && empty param[name]}">
    <c:out value="{msg}" />
    <dfu:addError name="{name}" msg="{msg}" />
  </c:if>
</dvu:errMsg>

<script type="text/javascript">
  pageValidators.add(new RequiredValidator(
    <da:jstring value="{formName}" />, <da:jstring value="{name}" />,
    <da:jstring value="{outId}" />, <da:jstring value="{msg}" />));
</script>

```

The `length.tag` file (shown in Listing 12) is another validator that verifies the length of a request parameter. The validated parameter is normally the submitted value of a form element. The tag file accepts five attributes: the element's name, the minimum and maximum lengths, and two optional error messages. The tag file outputs any error messages within a `` element generated with `<dvu:errMsg>` just like in the case of `required.tag`. The JavaScript code generated by `length.tag` creates a `LengthValidator` object on the client side.

Listing 12. The `length.tag` file

```

<%@ attribute name="name" required="true" rtexprvalue="true" %>
<%@ attribute name="min" required="false" rtexprvalue="true"
  type="java.lang.Long" %>
<%@ attribute name="max" required="false" rtexprvalue="true"
  type="java.lang.Long" %>
<%@ attribute name="msgMin" required="false" rtexprvalue="true" %>
<%@ attribute name="msgMax" required="false" rtexprvalue="true" %>
<%@ 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="da" tagdir="/WEB-INF/tags/dynamic/ajax" %>
<%@ taglib prefix="dfu" tagdir="/WEB-INF/tags/dynamic/forms/util" %>
<%@ taglib prefix="dvu" tagdir="/WEB-INF/tags/dynamic/valid/util" %>

<dvu:useDefaultMsg/>
<c:if test="{empty msgMin}">
  <c:set var="msgMin" value="{defaultValidMsg.too_short}" />
</c:if>
<c:if test="{empty msgMax}">
  <c:set var="msgMax" value="{defaultValidMsg.too_long}" />
</c:if>

<c:set var="outId" value="{name}Length" />
<dvu:errMsg id="{outId}">
  <c:if test="{formPost && !empty param[name]}">
    <c:if test="{min != null && fn:length(param[name]) < min}">
      <c:out value="{msgMin}" />
      <dfu:addError name="{name}" msg="{msgMin}" />
    </c:if>
    <c:if test="{max != null && fn:length(param[name]) > max}">
      <c:out value="{msgMax}" />
      <dfu:addError name="{name}" msg="{msgMax}" />
    </c:if>
  </c:if>
</dvu:errMsg>

<script type="text/javascript">
  pageValidators.add(new LengthValidator(<da:jstring value="{formName}" />,

```

```

    <da:jstring value="\${name}"/>, <da:jstring value="\${outId}"/>,
    \${min != null ? min : 0}, \${max != null ? max : 0},
    <da:jstring value="\${msgMin}"/>, <da:jstring value="\${msgMax}"/>));
</script>

```

Creating resource bundles with JSP tag files

In Java™ applications, messages are normally stored in resource bundles, which are typically coded as `.properties` files placed in `CLASSPATH`. When you change something in a resource bundle, you have to restart the application to ensure the refresh of the `.properties` file. If you want to avoid the application restarts, you could place the messages and other text resources in JSP tag files. The application server automatically reloads the tag files when they are changed.

Listing 13 shows the `useDefaultMsg.tag` file, which defines the validators' default messages, grouping them in `java.util.HashMap` objects that are created with `<jsp:useBean>` in the JSP request scope. Messages are placed into the maps with the `<c:set>` tag of JSTL. For every request, the `useDefaultMsg.tag` file chooses the message map that will be used by the validators, depending on the user's preferred locale, which is obtained with `\${pageContext.request.locale}`.

Listing 13. The `useDefaultMsg.tag` file

```

<%@ 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:if test="\${defaultValidMsg_en == null}">
  <jsp:useBean id="defaultValidMsg_en" scope="request" class="java.util.HashMap">
    <c:set target="\${defaultValidMsg_en}" property="required" value="Required"/>
    <c:set target="\${defaultValidMsg_en}" property="too_short" value="Too short"/>
    <c:set target="\${defaultValidMsg_en}" property="too_long" value="Too long"/>
  </jsp:useBean>
  <c:if test="\${!initParam.debug}">
    <c:set var="defaultValidMsg_en" scope="application"
      value="\${requestScope.defaultValidMsg_en}"/>
  </c:if>
</c:if>
...
<c:if test="\${defaultValidMsg == null}">
  <c:choose>
    <c:when test="\${fn:startsWith(pageContext.request.locale, 'en')}">
      <c:set var="defaultValidMsg" scope="request"
        value="\${defaultValidMsg_en}"/>
    </c:when>
    ...
  </c:choose>
</c:if>

```

This example demonstrates how to implement automatically reloadable resources. For a real application, you'll probably enhance this mechanism so that you can use a simpler syntax for defining the resources. For example, you could use a custom tag

instead of `<c:set>` to avoid specifying the `target` attribute for each message. You might also want to be able to place the resources for different locales in separate files to make translation easier.

During development, you should set a `debug` parameter to `true` in the `web.xml` file (see Listing 14) so that all message maps are recreated for every request, which allows you to make changes without having to restart the Web application. When moving to a production environment, you should switch `debug` to `false` to enable the caching of the maps in the `application` scope.

Listing 14. The `web.xml` file

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

  <context-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </context-param>

</web-app>
```

Enabling the JavaScript validators for the Web forms

You've seen so far how to validate the user input on both client and server sides. Next, you'll learn how to activate the JavaScript-based validation in the Web browser. The `onblur` attribute of a form element allows you to execute some JavaScript code when the element loses the keyboard focus, which is a good time to verify the value that the user has just entered. In addition, the `onsubmit` attribute of the `<form>` tag allows you to invoke a JavaScript function right after the click of a Submit button, and you can even cancel the form's post by returning `false` if the user's input is not valid.

[Part 2](#) of this series showed how to generate HTML forms, using a set of JSP tag files. You don't have to change them for adding the validation support because the form tag files use the `attrList.tag` file (shown in Listing 15) to generate the dynamic attributes of the form elements. In [Part 2](#), you've already seen how to add default styles in `attrList.tag`. This file is modified here to add the `onblur` and `onsubmit` attributes, which contain the function calls for executing the JavaScript validators.

The `class`, `onblur`, and `onsubmit` attributes that must be added to an HTML element are placed in a `java.util.HashMap` instance named `addMap`. Then, `attrList.tag` iterates over the elements of the `map` and `addMap` collections, generating the HTML attributes of the element.

Listing 15. 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" %>

<jsp:useBean id="addMap" class="java.util.HashMap"/>
<c:if test="${!empty initParam.skin}">
    <c:set target="${addMap}" property="class"
        value="${initParam.skin}_${tag}"/>
</c:if>
<c:if test="${tag=='input_text' || tag=='input_password' || tag=='input_radio'
    || tag=='textarea' || tag=='select'}">
    <c:set target="${addMap}" property="onblur"
        value="pageValidators.execute('${formName}', this.name)"/>
</c:if>
<c:if test="${tag=='form'}">
    <c:set target="${addMap}" property="onsubmit"
        value="return pageValidators.execute('${formName}')"/>
</c:if>

<c:forEach var="attr" items="${map}">
    <c:if test="${addMap[attr.key] == null}">
        ${attr.key}="${fn:escapeXml(attr.value)}"
    </c:if>
    <c:if test="${addMap[attr.key] != null}">
        <c:set var="sep" value="${fn:startsWith(attr.key, 'on') ? ';' : ' ' }"/>
        <c:set target="${addMap}" property="${attr.key}"
            value="${attr.value}${sep}${addMap[attr.key] }"/>
    </c:if>
</c:forEach>

<c:forEach var="attr" items="${addMap}">
    ${attr.key}="${fn:escapeXml(attr.value)}"
</c:forEach>

```

Adding validation to the sample application

Part 2 of this series presented a simple JSP example based on the form tag files. This section shows how to add validation to the Web form.

Updating the Web form

Listing 16 contains the modified version of the `SupportForm.jsp` page. The first change that you'll notice is the `<%@page%>` directive, which has the `autoFlush="false"` and `buffer="64kb"` attributes. By default, JSP pages have an 8kb buffer, which is flushed automatically when it's full of the dynamically generated HTML. Once the buffer is flushed, the page cannot use `<jsp:forward>` anymore.

The `form.tag` file, which implements the `<df:form>` tag used in the `SupportForm.jsp` page, uses `<jsp:forward>` to dispatch the HTTP request to

another page named `SupportConfirm.jsp` when the user input is valid. With the added validation code, the size of the HTML generated by `SupportForm.jsp` is greater than 8kb. Therefore, the page's buffer has been increased to ensure the forwarding can work properly. In addition, the `autoFlush` has been disabled to allow the easy debugging when the buffer isn't large enough to contain the entire HTML produced by the Web page.

You'll find additional changes in the header of the `SupportForm.jsp` page, which declares the library of validator tags (prefix `dv`), the `valid.css` file, and two JavaScript files (`encode.js` and `valid.js`), which contain the JavaScript functions that are used for HTML encoding and client-side validation. Throughout the `SupportForm.jsp` page, most of the form elements are preceded by `<dv:required>` and some of them use `<dv:length>` for verifying the length of the entered strings.

Listing 16. The `SupportForm.jsp` example

```
<%@ page autoFlush="false" buffer="64kb"%>
<%@ taglib prefix="df" tagdir="/WEB-INF/tags/dynamic/forms" %>
<%@ taglib prefix="dv" tagdir="/WEB-INF/tags/dynamic/valid" %>

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

<html>
<head>
  <title>Support Form</title>
  <link rel="stylesheet" href="forms.css" type="text/css">
  <link rel="stylesheet" href="valid.css" type="text/css">
  <script src="encode.js" type="text/javascript">
  </script>
  <script src="valid.js" type="text/javascript">
  </script>
</head>
<body>

  <h1>Support Form</h1>

  <df:form name="supportForm" model="{supportBean}"
    action="SupportConfirm.jsp">

    <p>Name:
    <dv:required name="name"/>
    <dv:length name="name" max="60"/> <br>
    <df:input name="name" type="text" size="40"/>

    <p>Email:
    <dv:required name="email"/>
    <dv:length name="email" max="60"/> <br>
    <df:input name="email" type="text" size="40"/>

    <p>Versions:
    <dv:required name="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>

  </df:form>

</body>
</html>
```

```

<p>Platform:
<dv:required name="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:
<dv:required name="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:
<dv:required name="problem"/>
<dv:length name="problem" min="100" max="2000"
  msgMin="Cannot have less than 100 characters"
  msgMax="Cannot have more than 2000 characters"/> <br>
<df:textarea name="problem" rows="10" cols="40"/>

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

</df:form>

</body>
</html>

```

Analyzing the generated HTML

If you look at the HTML generated by the `SupportForm.jsp` page (partially shown in Listing 17), you'll observe the following attributes and code fragments that are related to the client-side validation:

- The `<form>` element has the `onsubmit` attribute, which contains a JavaScript call that returns `true` only if the whole form is valid.
- The `<input>`, `<textarea>`, and `<select>` elements use the `onblur` attribute to validate their values as soon as the keyboard focus is lost.
- Each validator tag generates:
 - A `` element where the error messages are inserted.
 - A piece of JavaScript code that creates the validator object.

Listing 17. HTML generated by SupportForm.jsp

```

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

```

```
<script src="encode.js" type="text/javascript">
</script>
<script src="valid.js" type="text/javascript">
</script>
</head>
<body>

  <h1>Support Form</h1>

  <form name="supportForm" method="POST" class="default_form"
        onsubmit="return pageValidators.execute('supportForm')">
    ...
    <p>Email:
    <span id="emailRequired" class="ValidationError">
    </span>
    <script type="text/javascript">
      pageValidators.add(new RequiredValidator(
        "supportForm", "email", "emailRequired", "Required"));
    </script>
    <span id="emailLength" class="ValidationError">
    </span>
    <script type="text/javascript">
      pageValidators.add(new LengthValidator(
        "supportForm", "email", "emailLength", 0, 60,
        "Too short", "Too long"));
    </script>
    <br>
    <input name="email" type="text" size="40" class="default_input_text"
          onblur="pageValidators.execute('supportForm', this.name)">
    ...
  </form>

</body>
</html>
```

Conclusion

In this article, you learned how to build a hierarchy of client-side validators, using JavaScript constructors and prototypes. You also learned how to create server-side validators and resource-bundles based JSP tag files. Stay tuned for the next article of the series, where you'll find out how to create JSF-like components, using JSP tag files.

Downloads

Description	Name	Size	Download method
Sample application for this article	wa-aj-simplejava	321KB	HTTP

[Information about download methods](#)

Resources

- [Part 1](#) of this series demonstrated how to generate JavaScript code for sending Ajax requests and processing Ajax responses.
- [Part 2](#) showed how to create HTML forms, using conventions and JSP tag files to minimize setup and configuration.
- 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.
- Visit developer.mozilla.org if you are looking for JavaScript tools and documentation.
- Go to the [JSP 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.

Trademarks

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