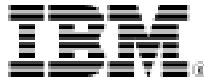


developerWorks.

JavaScript EE, Part 3: Use Java scripting API with JSP

Build JavaScript code that runs on the Web server or in the Web browser

Skill Level: Intermediate

Andrei Cioroianu
Senior Java Developer and Consultant
Devsphere

02 Jun 2009

In the previous two parts of this series, you've seen how to run JavaScript files on the server and how to call remote JavaScript functions with Ajax. This article explains how to use server-side JavaScript code with the Java™Server Pages (JSP) technology and how to build Asynchronous JavaScript and XML (Ajax) user interfaces that remain functional when JavaScript is disabled in the Web browser. The sample code consists of a small JSP tag library that you can reuse in your own applications as well as a dynamic Web form, which is generated with a piece of JavaScript code that can be executed on the Web server or in the Web browser.

The ability to run the same JavaScript code on both servers and clients has significant advantages; it allows you to maintain a single code base for both Ajax and non-Ajax clients and gives you more flexibility. For example, if you develop some JavaScript code that you don't want others to analyze, you can run it on the server, protecting your intellectual property and minimizing security risks. If later you become comfortable releasing the code, you could move the JavaScript code to the client, improving the application's performance.

Page contexts and script contexts

Java Scripting API and JavaServer Pages are two separate Java technologies that can be integrated easily. Each of them executes code in a well-defined context. With

© Copyright IBM Corporation 2009. All rights reserved.

JSP technology, you have access to a set of JSP implicit objects: `pageContext`, `page`, `request`, `response`, `out`, `session`, `config` and `application`. In Part 1, you've already seen how to export these objects to a JavaScript file executed by a servlet. In this article, you'll learn how to do the same for scripts that are executed by a JSP page.

The JavaScript engine uses a different type of context for maintaining script variables and functions defined in the application code. If you run a script that sets variables or contains functions, subsequent scripts executed within the same context can use the variables and functions of the previous scripts. Therefore, it makes sense to use a single script context during the processing of an HTTP request, as you'll see in the following section.

Scripts written in the JavaScript language can access the public fields and call the public methods of any Java object. In addition, you can obtain and modify the value of a JavaBean property, using the `object.property` syntax instead of calling the get and set methods. Because using Java objects in your JavaScript code is so easy, the only thing that is missing is a set of custom tags for exchanging objects between the JSP page context and the JavaScript context. They can be implemented with only a few lines of code as demonstrated in this article.

Using server-side JavaScript code in Web pages

This section shows how to manage a JavaScript context throughout an Ajax/HTTP request and how to exchange variables between the JSP page context and the JavaScript context.

Making JSP objects available in your JavaScript code

The first part of this series presented a Java servlet based on the Java Scripting API, which was used to execute JavaScript files on the server. This section describes a class named `JSUtil`, which uses the same Java Scripting API to run JavaScript code fragments during the execution of a JSP page. First of all, you need to create a `ScriptEngineManager` object and then obtain a `ScriptEngine` instance, as shown in Listing 1.

Listing 1. The getScriptEngine() method of JSUtil

```
package jsee.util;

import javax.script.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.io.*;

public class JSUtil {
    private static ScriptEngine engine;
```


© Copyright IBM Corporation 2009. All rights reserved.

```
    public static synchronized ScriptEngine getScriptEngine() {
        if (engine == null) {
            ScriptEngineManager manager = new ScriptEngineManager();
            engine = manager.getEngineByName("JavaScript");
        }
        return engine;
    }
    ...
}
```

Listing 2 contains the `createScriptContext()` method, which initializes a `ScriptContext` instance, getting the JSP implicit objects from a page context and setting these objects as the variables of the script context. This operation makes the implicit objects accessible from the JavaScript code that will be executed in the script context:

- The `pageContext` object is a repository for page-scoped variables and has methods for obtaining all the other implicit objects.
- The `page` object is the instance of the servlet class processing the current request.
- The `request` object lets you obtain the HTTP request parameters and headers.
- The `response` object lets you set HTTP response headers and provides a writer identified with `out` in the JSP code.
- The `out` object is used for the output of the JSP page.
- The `session` object maintains user-related state between requests.
- The `config` object represents the configuration for the servlet implementing the JSP.
- The `application` object is used to store bean instances shared by all users and obtain initialization parameters specified in the web.xml file.

Listing 2. The createScriptContext() method of JSUtil

```
public class JSUtil {
    ...
    public static ScriptContext createScriptContext(PageContext pageContext) {
        ScriptContext scriptContext = new SimpleScriptContext();
        int scope = ScriptContext.ENGINE_SCOPE;
        scriptContext.setAttribute("pageContext", pageContext, scope);
        scriptContext.setAttribute("page", pageContext.getPage(), scope);
        scriptContext.setAttribute("request", pageContext.getRequest(), scope);
        scriptContext.setAttribute("response", pageContext.getResponse(), scope);
        scriptContext.setAttribute("out", pageContext.getOut(), scope);
        scriptContext.setAttribute("session", pageContext.getSession(), scope);
        scriptContext.setAttribute("config", pageContext.getServletConfig(), scope);
        scriptContext.setAttribute("application",
                pageContext.getServletContext(), scope);
```


© Copyright IBM Corporation 2009. All rights reserved.

```
        scriptContext.setWriter(pageContext.getOut());
        return scriptContext;
    }
    ...
}
```

During the processing of an HTTP request, you'll want to use a single script context so that any JavaScript fragment can use the variables and functions defined by the previously executed scripts. An easy way to satisfy this requirement is to store the script context as an attribute of the `request` object. The `getScriptContext()` method (see Listing 3) uses `jsee.ScriptContext` as the attribute name.

Listing 3. The getScriptContext() method of JSUtil

```
public class JSUtil {
    ...
    public static ScriptContext getScriptContext(PageContext pageContext)
            throws IOException {
        ServletRequest request = pageContext.getRequest();
        synchronized (request) {
            ScriptContext scriptContext
                = (ScriptContext) request.getAttribute("jsee.ScriptContext");
            if (scriptContext == null) {
                scriptContext = createScriptContext(pageContext);
                request.setAttribute("jsee.ScriptContext", scriptContext);
            }
            return scriptContext;
        }
    }
    ...
}
```

The `runScript()` method (shown in Listing 4) executes the given JavaScript fragment in the context of the current HTTP request, using the `eval()` method of the script engine. If a script exception is thrown, `runScript()` outputs the source code followed by the stack trace and throws a `ServletException` to stop the execution of the JSP page.

Listing 4. The runScript() method of JSUtil

```
public class JSUtil {
    ...
    public static void runScript(String source, PageContext pageContext)
            throws ServletException, IOException {
        try {
            getScriptEngine().eval(source, getScriptContext(pageContext));
        } catch (ScriptException x) {
            ((HttpServletResponse) pageContext.getResponse()).setStatus(500);
            PrintWriter out = new PrintWriter(pageContext.getOut());
            out.println("<pre>" + source + "</pre>");
            out.println("<pre>");
            x.printStackTrace(out);
            out.println("</pre>");
            out.flush();
            throw new ServletException(x);
        }
    }
```


© Copyright IBM Corporation 2009. All rights reserved.

```
}
```

Using a custom tag to execute JavaScript code fragments

The script.tag file (see Listing 5) lets you execute a piece of JavaScript code on the server, on the client, or on both, depending on the value of the `runat` attribute. If `runat` is `client` or `both`, the tag file outputs an HTML `<script>` element containing the code that you place between `<js:script>` and `</js:script>` in your Web page. If the `runat` attribute is `server` or `both`, the JavaScript code fragment is obtained as a JSP variable named `source`, whose value is passed to the `runScript()` method of the `JSUtil` class.

Listing 5. The script.tag file

```
<%@ attribute name="runat" required="true" rtexprvalue="true" %>
<%@ tag body-content="scriptless" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:if test="${runat == 'client' or runat == 'both'}">
    <script type="text/javascript">
        <jsp:doBody/>
    </script>
</c:if>

<c:if test="${runat == 'server' or runat == 'both'}">
    <jsp:doBody var="source"/>
    <%
        jsee.util.JSUtil.runScript(
                (String) jspContext.getAttribute("source"),
                (PageContext) jspContext);
    %>
</c:if>
```

An easy way to understand how the `<js:script>` tag works is to look at a simple example. Listing 6 contains the ScriptDemo.jsp page, which defines a function named `link()` that returns an `<a>` element. This function can be executed on the Web server and in the Web browser because the `runat` attribute is `both`.

During the processing of the JSP page on the server side, the `println(link(request.requestURL))` call outputs a link to the current page. The `request` object is accessible from the JavaScript code (on the server) thanks to the context initialization performed by the `JSUtil` class whose `runScript()` method is invoked by the tag file. When the JavaScript engine evaluates the `request.requestURL` expression, it actually calls the `getRequestURL()` method of the `HttpServletRequest` instance.

On the client side, the `document.writeln(link(location))` call will output another link to the ScriptDemo.jsp page whose URL is obtained with the `location` property, which is available only in the Web browser.

© Copyright IBM Corporation 2009. All rights reserved.

Listing 6. The ScriptDemo.jsp example

```
<%@ taglib prefix="js" tagdir="/WEB-INF/tags/js" %>

<js:script runat="both">
    function link(url) {
        return '<a href="' + url + '">' + url + '</a>';
    }
</js:script>

<js:script runat="server">
    println(link(request.requestURL));
</js:script>

<br>

<js:script runat="client">
    document.writeln(link(location));
</js:script>
```

Listing 7 shows the HTML output produced by the ScriptDemo.jsp page, which contains the `link()` function, the `<a>` element generated on the server side, and the JavaScript code that outputs the second link on the client side.

Listing 7. The output generated by ScriptDemo.jsp

```
<script type="text/javascript">
    function link(url) {
        return '<a href="' + url + '">' + url + '</a>';
    }
</script>

<a href="http://localhost:8080/jsee/ScriptDemo.jsp">
http://localhost:8080/jsee/ScriptDemo.jsp</a>

<br>

<script type="text/javascript">
    document.writeln(link(location));
</script>
```

Getting and setting variables

When using server-side JavaScript code in Web pages, you'll need to exchange variables between the JSP context and the script context. The get.tag file (see Listing 8) implements a custom tag named `<js:get>`, which exports a script variable as a JSP variable whose name is provided through the `var` attribute.

The variable's name is obtained from the JSP context because it is passed to the tag file as a JSP attribute. The script variable's value is retrieved with the `getAttribute()` method of the script context. Then, a JSP variable named `varAlias` is created in the page scope of the tag file with `jspContext.setAttribute()`. The `<%@variable%>` directive that is used in the tag file instructs the JSP container (for example, Tomcat) to export the `varAlias`

© Copyright IBM Corporation 2009. All rights reserved.

variable to the page invoking the tag file, using the variable name specified with the `var` attribute. This technique allows the invoked tag file to define JSP variables in the `page` scope of the JSP using the `<js:get>` tag.

Listing 8. The get.tag file

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

<%
    String var = (String) jspContext.getAttribute("var");
    Object value = jsee.util.JSUtil.getScriptContext(
            (PageContext) jspContext).getAttribute(var);
    jspContext.setAttribute("varAlias", value);
%>
```

The `<js:set>` tag, which is implemented by the set.tag file (shown in Listing 9), takes two attributes (a variable name and a value) that are used to set a JavaScript variable with the `setAttribute()` method of the script context. If the optional `value` attribute is not provided, the tag file uses `<jsp:doBody var="value"/>` to execute the code placed between `<js:set>` and `</js:set>` in the JSP page.

Listing 9. The set.tag file

```
<%@ attribute name="var" required="true" rtexprvalue="false" %>
<%@ attribute name="value" required="false" rtexprvalue="true"
    type="java.lang.Object"%>
<%@ tag body-content="scriptless" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:if test="${empty value}">
    <jsp:doBody var="value"/>
</c:if>

<%
    String var = (String) jspContext.getAttribute("var");
    Object value = jspContext.getAttribute("value");
    jsee.util.JSUtil.getScriptContext((PageContext) jspContext)
        .setAttribute(var, value, javax.script.ScriptContext.ENGINE_SCOPE);
%>
```

Listing 10 contains the VarDemo.jsp example that uses the `<js:set>` tag to initialize two script variables, which are modified in a JavaScript code fragment executed on the server with the help of `<js:script>`. Then, the new values of the two variables are obtained with the `<js:get>` tag, and they are included in the page's output with the `<c:out>` tag of JSTL.

Listing 10. The VarDemo.jsp example

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


© Copyright IBM Corporation 2009. All rights reserved.

```
<js:set var="x" value="1"/>
<js:set var="y">2</js:set>

<js:script runat="server">
    x++;
    y--;
</js:script>

<js:get var="x"/>
<js:get var="y"/>

<c:out value="x = ${x}"/> <br>
<c:out value="y = ${y}"/>
```

Building Ajax UIs that work in any browser

This section presents a dynamic form based on Ajax and DHTML. If JavaScript is disabled in the Web browser, the user interface produced by the DynamicForm.jsp page remains functional by running the scripting code on the server side. A server-side JavaScript file named BackendScript.jss is used to process the submitted form data.

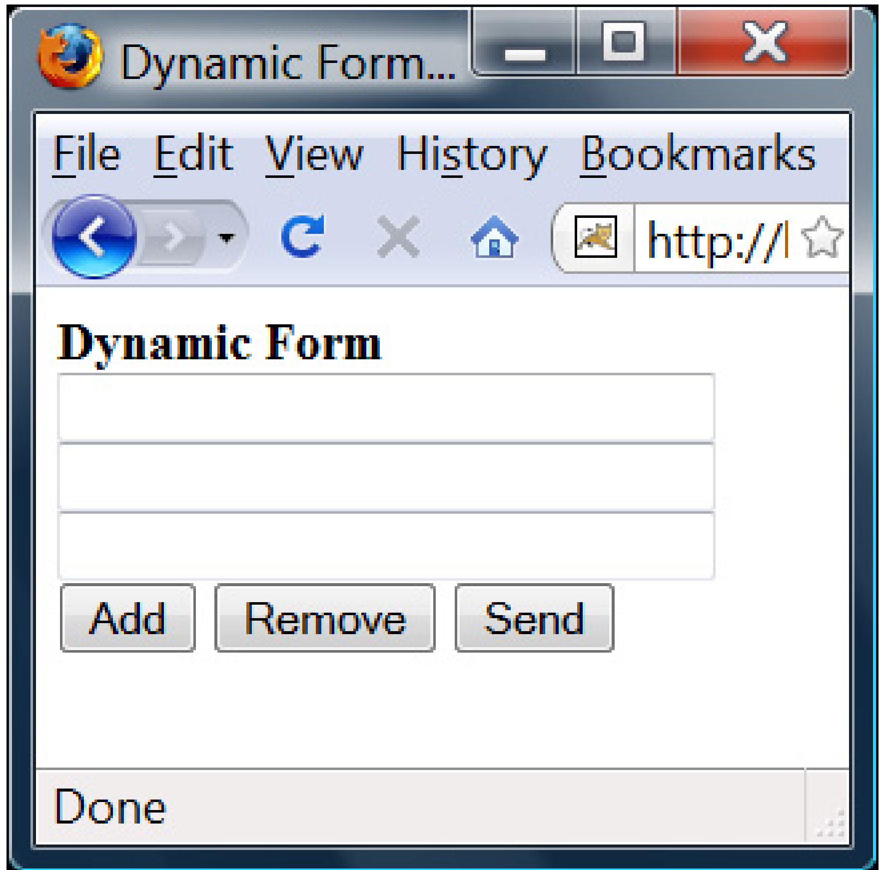
The form is kept simple so that the code can be understood very easily. The main objective of this example is to demonstrate several Web techniques, which you can use in complex real-world applications.

Developing dynamic forms

The screenshot included in Figure 1 shows the sample form, which consists of a variable number of input fields and three buttons labeled Add, Remove, and Send. The Add button appends a new input field to the form, the Remove button deletes the last field, and Send submits the form data to the server.

Figure 1. Simple dynamic form

© Copyright IBM Corporation 2009. All rights reserved.



When DynamicForm.jsp is executed on the server the initial input fields are generated with a JavaScript function named `outputForm()`. As shown in Listing 11, these fields are placed within a `<div>` element whose content will be modified with the `updateForm()` function, which is called in the Web browser when the user clicks Add or Remove. When the user clicks Send, the user input is submitted to the server with the `submitForm()` function, which uses `XMLHttpRequest`.

No function is called when the user clicks a button if JavaScript is disabled, in which case the Web browser sends the data to the server. This doesn't happen when JavaScript is enabled because the `updateForm()` and `submitForm()` functions return `false`.

Listing 11. The form of DynamicForm.jsp

```
<html>
<head>
<title>Dynamic Form</title>
    ...
</head>
<body>
    <b>Dynamic Form</b><br>
    <form name="dynamicForm" method="POST" action="DynamicForm.jsp">
        <div id="inputDiv">
            <js:script runat="server">
```


© Copyright IBM Corporation 2009. All rights reserved.

```
                outputForm();
            </js:script>
        </div>
        <input type="submit" name="add" value="Add"
                onclick="return updateForm('add')">
        <input type="submit" name="remove" value="Remove"
                onclick="return updateForm('remove')">
        <input type="submit" name="send" value="Send"
                onclick="return submitForm()">
    </form>
</body>
</html>
```

The DynamicForm.jsp page contains several functions that can run on the Web server or in the Web browser. The `htmlEncode()` and `buildInput()` routines (see Listing 12) are used to generate a single `<input>` element.

Listing 12. The htmlEncode() and buildInput() functions of DynamicForm.jsp

```
<%@ taglib prefix="js" tagdir="/WEB-INF/tags/js" %>
...
<js:script runat="both">
    function htmlEncode(value) {
        return value ? value.replace(/&/g, "&amp;").replace(/"/g, "&quot;")
            .replace(/</g, "&lt;").replace(/>/g, "&gt;") : "";
    }

    function buildInput(name, value) {
        var content = '<input type="text"';
        if (name)
            content += ' name="' + htmlEncode(name) + '"';
        if (value)
            content += ' value="' + htmlEncode(value) + '"';
        content += ' size="30">';
        return content;
    }
    ...
</js:script>
```

The `buildForm()` function (shown in Listing 13) takes two parameters: a list of values and a command. The last value is ignored when the given command is `remove`. If the `cmd` parameter is `add`, an empty field is added at the end of the returned HTML string.

Listing 13. The buildForm() function of DynamicForm.jsp

```
<js:script runat="both">
    ...
    function buildForm(values, cmd) {
        var n = values.length;
        if (cmd == 'remove' && n > 1)
            n--;
        var content = '';
        for (var i = 0; i < n; i++)
            content += buildInput("inputField", String(values[i])) + '<br>';
        if (cmd == 'add')
            content += buildInput("inputField") + '<br>';
        return content;
```


© Copyright IBM Corporation 2009. All rights reserved.

```
    }
</js:script>
```

Implementing the server-side code

The DynamicForm.jsp page also contains two JavaScript functions that are executed only on the Web server. Listing 14 shows the `getInputValues()` routine, which returns the values of the `inputField` parameter if the HTTP method is `POST`.

Listing 14. The getInputValues() function of DynamicForm.jsp

```
<js:script runat="server">
    function getInputValues() {
        if (request.getMethod().toUpperCase().equals("POST"))
            return request.getParameterValues("inputField");
        else
            return null;
    }
    ...
</js:script>
```

The other server-side function is `outputForm()` (see Listing 15). As mentioned earlier, if JavaScript is disabled when the user clicks a button, the Web browser submits the form data to the server. In this case, `outputForm()` detects which button was clicked and sets the `cmd` variable.

If the command is `send`, the DynamicForm.jsp page includes the output of a script named BackendScript.jss, which is executed by the `JSServlet` class that was presented in the first part of the series. The script returns a list of values encoded with the JSON format.

The output of BackendScript.jss becomes part of a JavaScript fragment executed on the server side because `<jsp:include page="BackendScript.jss"/>` is placed between `<js:script runat="server">` and `</js:script>`.

At the end, the `outputForm()` function generates the list of HTML input fields with `buildForm()` and prints the resulted string.

Listing 15. The outputForm() function of DynamicForm.jsp

```
<js:script runat="server">
    ...
    function outputForm() {
        var cmd = "";
        if (request.getParameter("add") != null)
            cmd = "add";
        else if (request.getParameter("remove") != null)
            cmd = "remove";
        else if (request.getParameter("send") != null)
            cmd = "send";
        var values = null;
        if (cmd == "send") {
```


© Copyright IBM Corporation 2009. All rights reserved.

```
            values = <jsp:include page="BackendScript.jss"/>;
        } else {
            values = getInputValues();
            if (!values)
                values = ["", "", ""]; // default input values
        }
        println(buildForm(values, cmd));
    }
</js:script>
```

Listing 16 shows BackendScript.jss, which gets the input values with `paramValues.inputField` and sorts the array, which is then converted to JSON with `toSource()`. The `paramValues` variable is initialized with the values of the request parameters in the init.jss script that is executed by `JSServlet` as explained in Part 1 of the series.

Listing 16. The BackendScript.jss file

```
var values = paramValues.inputField;
if (values)
    println(values.sort().toSource());
else
    println("[]");
```

Using JavaScript on the client-side

The DynamicForm.jsp page has several routines that are designed to run only in the Web browser. The `getInputValues()` function (see Listing 17) uses the `document.forms.dynamicForm.inputField` expression to get the array of DOM objects representing the input fields. This expression may actually return a DOM object if there is a single input field. Therefore, `getInputValues()` verifies if `fields` is an array by testing the availability of the `length` property. The values of the input fields are returned as an array of strings.

Listing 17. The getInputValues() function of DynamicForm.jsp

```
<js:script runat="client">
    function getInputValues() {
        var values = new Array();
        var fields = document.forms.dynamicForm.inputField;
        if (typeof fields.length == "number")
            for (var i = 0; i < fields.length; i++)
                values[i] = fields[i].value;
        else if (fields)
            values[0] = fields.value;
        return values;
    }
    ...
</js:script>
```

The `setInputValues()` function (shown in Listing 18) sets the values of the form's input fields. After invoking this function, the user will see the new values in the

© Copyright IBM Corporation 2009. All rights reserved.

Web browser.

Listing 18. The setInputValues() function of DynamicForm.jsp

```
<js:script runat="client">
    ...
    function setInputValues(values) {
        var fields = document.forms.dynamicForm.inputField;
        if (typeof fields.length == "number")
        for (var i = 0; i < fields.length; i++)
            fields[i].value = values[i];
        else if (fields)
            fields.value = values[0];
    }
    ...
</js:script>
```

When the user clicks Add or Remove, the Web browser calls the `updateForm()` function (see Listing 19), which is used with the `onclick` attribute of those buttons. The list of input fields is regenerated with the `buildForm()` function, which preserves the current values of the fields, and the new HTML content is inserted in the Web page, using the `innerHTML` property of the `inputDiv` element.

Listing 19. The updateForm() function of DynamicForm.jsp

```
<js:script runat="client">
    ...
    function updateForm(cmd) {
        document.getElementById('inputDiv').innerHTML
            = buildForm(getInputValues(), cmd);
        return false;
    }
    ...
</js:script>
```

If JavaScript is enabled, the Web browser invokes the `submitForm()` function (shown in Listing 20) when the user clicks the Send button whose `onclick` attribute contains `return submitForm()`. This JavaScript function uses a small Ajax library that was presented in the second part of this series. You can find the library's source code in the xhr.js file.

In the DynamicForm.jsp page, an `XHR` object is created and stored in the `xhr` variable. The `XHR()` constructor takes three parameters:

- The HTTP method, which is `POST` in this example
- The URL of the HTTP request, which is `BackendScript.jss`
- A boolean parameter indicating whether the requests should be asynchronous or not. In this example they are synchronous, meaning the function waits for the HTTP response before returning the control.

© Copyright IBM Corporation 2009. All rights reserved.

Listing 20. The submitForm() function of DynamicForm.jsp

```
<script src="xhr.js" type="text/javascript">
</script>
...
<js:script runat="client">
    ...
    var xhr = new XHR("POST", "BackendScript.jss", false);

    function submitForm() {
        var request = xhr.newRequest();
        if (!request)
            return true;
        var values = getInputValues();
        for (var i = 0; i < values.length; i++)
            xhr.addParam("inputField", values[i]);
        function processResponse() {
            if (xhr.isCompleted()) {
                var newValues = eval(request.responseText);
                setInputValues(newValues);
            }
        }
        xhr.sendRequest(processResponse);
        return false;
    }
</js:script>
```

The `submitForm()` function creates a new `XMLHttpRequest` instance with `xhr.newRequest()`. If this object cannot be created (that is, the Web browser doesn't support Ajax), `submitForm()` returns `true` so that the `onclick` expression of the Send button can return `true`, which means the Web browser will submit the form data to the server.

If the new `XMLHttpRequest` object is successfully created, the next step is to get the values of the input fields, which will be added as request parameters with the help of `xhr.addParam()`.

The `processResponse()` inner function is used as a callback. If the HTTP request is completed and the status code is 200, the text response is evaluated and the obtained array is passed to the `setInputValues()` function, which updates the values of the input fields.

The `xhr.sendRequest()` call invokes the `send()` method of the `XMLHttpRequest` instance. Then, `submitForm()` returns `false` so that the `onclick` expression of the Send button can return `false`, indicating the Web browser must not submit the form data to the server. This behavior is correct because the input values are sent with `XMLHttpRequest` when JavaScript is enabled and the browser shouldn't resubmit the same data.

Conclusion

In this article, you've learned how to run server-side JavaScript code that is

© Copyright IBM Corporation 2009. All rights reserved.

embedded within JSP pages and how to exchange variables between the JSP page context and the JavaScript context. You've also learned how to build an Ajax user interface that can still work when JavaScript is disabled in the Web browser. You can use the same techniques when your application must support non-Ajax client devices in addition to Ajax-capable Web browsers.

© Copyright IBM Corporation 2009. All rights reserved.

Downloads

Description	Name	Size	Download method
Sample application for this article	jsee_part3_src.zip	26KB	HTTP

Information about download methods

© Copyright IBM Corporation 2009. All rights reserved.

Resources

- "JavaScript EE, Part 1: Run JavaScript files on the server side" (developerWorks, December 2008) shows how to compile and execute JavaScript files using the javax.script API, how to export Java objects as script variables, and how to build JavaScript files that are executed on the server-side.
- "JavaScript EE, Part 2: Call remote JavaScript functions with Ajax" (developerWorks, March 2009) shows how to use Remote Procedure Calls (RPC) in Ajax/Java applications, how to implement Java interfaces with JavaScript, and how to manage the lifecycle of the Ajax requests when connecting to data feeds.
- JSR-223 Scripting for the Java Platform is the specification containing all the details for using scripting languages in Java applications.
- Java Scripting Programmer's Guide presents the Java Scripting API and provides more code examples.
- 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.

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 over 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.

© Copyright IBM Corporation 2009. All rights reserved.