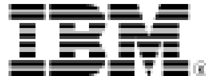


developerWorks.

JavaScript EE, Part 2: Call remote JavaScript functions with Ajax

Implementing RPC for JavaScript using Ajax and Java code

Skill Level: Intermediate

Andrei Cioroianu
Senior Java Developer and Consultant
Devsphere

31 Mar 2009

In Part 1 of this series, you learned how to use the javax.script API in Asynchronous JavaScript and XML (Ajax) and Java™ Platform, Enterprise Edition (Java EE) applications, and how to build a Java servlet that lets you run server-side JavaScript files. This article shows how to implement a Remote Procedure Call (RPC) mechanism for Web applications that use JavaScript on both servers and clients. You'll also learn several interesting techniques, such as implementing Java interfaces with JavaScript, building an XMLHttpRequest wrapper, making Ajax debugging easier, and using JSP tag files to generate JavaScript code.

The RPC mechanism is very simple. You have a set of JavaScript functions on the server and you want to be able to call them from the Web browser as if you had a single JavaScript engine executing both the client code and the server code. Therefore, you'll need some client-side routines that should have the same names and parameters as their server-side counterparts.

The client-side routines will use Ajax to transmit their parameters to the server where the actual processing takes place. A Java servlet will invoke the server-side functions and will return the results to the client, using the JSON format. The client-side routines will then evaluate the Ajax responses, converting the JSON strings back to JavaScript objects, which are returned to the application.

As an application developer, you can focus on building the user interface and the functions that are executed on the server. You won't have to deal with Ajax or RPC

© Copyright IBM Corporation 2009

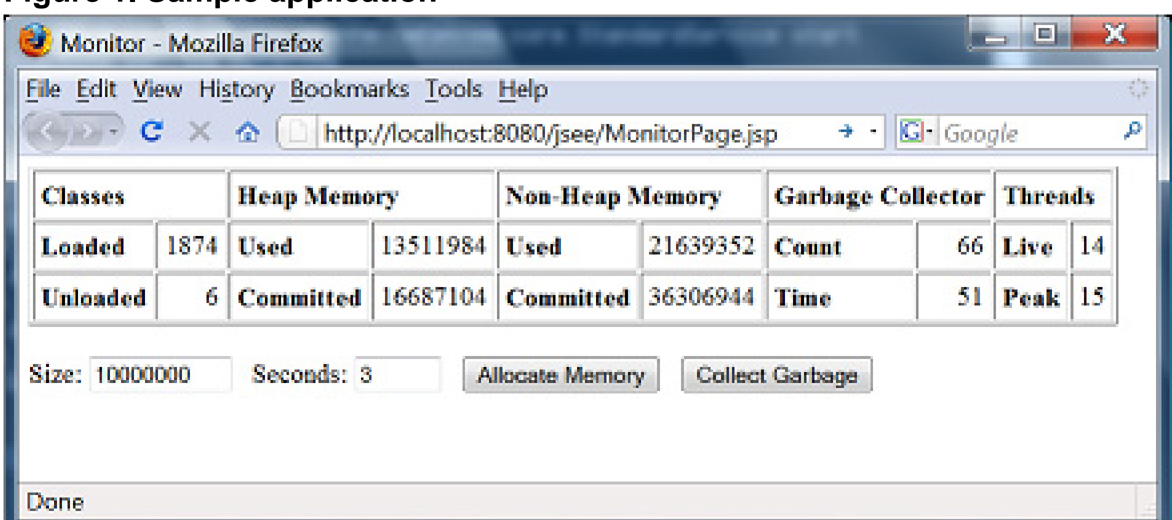
issues because this article provides a JavaScript code generator in the form of a tag file, which you can use in your JSP pages to produce the client-side routines automatically. To understand how this works, let's start with a sample application.

Developing a JVM-monitoring application

The sample application in this article uses the Java Management API to monitor the JVM running the Java EE server that hosts the application. The user interface consists of a single Web page that shows various indicators, such as the number of Java classes, memory consumption, garbage collector's activity, and the number of threads.

This information is retrieved with Ajax and is inserted into an HTML table (see Figure 1; click here for a larger version of Figure 1). To make things more interesting, the Web page contains a form that lets you allocate some memory for a given number of seconds. You can also invoke the JVM's garbage collector (GC).

Figure 1. Sample application



On the server side, the application uses a JavaScript file whose functions are invoked from the Web browser with the help of Ajax, using the script runner that was presented in Part 1 of this series. This is a simple servlet that handles all requests whose URLs end with the .jss extension. The servlet looks for the corresponding JavaScript file on the server and executes it, using the Java Scripting API.

Creating the Web page

Listing 1 shows the table and the form of the MonitorPage.jsp file. Each data cell has an ID so that the table's content can be updated with JavaScript. The `onload` attribute of the `<body>` tag is used to call a JavaScript function named `init()`, which will initialize the client-side portion of the application. Two other functions

© Copyright IBM Corporation 2009

named `allocMem()` and `gc()` are called when the user clicks the buttons.

Listing 1. The HTML code of MonitorPage.jsp

```
<html>
<head>
    ...
    <style type="text/css">
        th { text-align: left; }
        td { text-align: right; }
        .space { margin: 10px; }
    </style>
</head>
<body onload="init()">
    <table border="1" cellpadding="5">
        <tr>
            <th colspan=2>Classes</th>
            <th colspan=2>Heap Memory</th>
            <th colspan=2>Non-Heap Memory</th>
            <th colspan=2>Garbage Collector</th>
            <th colspan=2>Threads</th>
        </tr>
        <tr>
            <th>Loaded</th>
            <td id="info.classes.loaded"></td>
            <th>Used</th>
            <td id="info.memory.heap.used"></td>
            <th>Used</th>
            <td id="info.memory.nonHeap.used"></td>
            <th>Count</th>
            <td id="info.gc.count"></td>
            <th>Live</th>
            <td id="info.threads.live"></td>
        </tr>
        <tr>
            <th>Unloaded</th>
            <td id="info.classes.unloaded"></td>
            <th>Committed</th>
            <td id="info.memory.heap.committed"></td>
            <th>Committed</th>
            <td id="info.memory.nonHeap.committed"></td>
            <th>Time</th>
            <td id="info.gc.time"></td>
            <th>Peak</th>
            <td id="info.threads.peak"></td>
        </tr>
    </table>
    <br>
    <form name="monitorForm">
        Size: <input name="size" size="10">
        <span class="space"></span>
        Seconds: <input name="seconds" size="4">
        <span class="space"></span>
        <button type="button"
            onclick="allocMem(this.form.size.value, this.form.seconds.value)"
        >Allocate Memory</button>
        <span class="space"></span>
        <button type="button" onclick="gc()">Collect Garbage</button>
    </form>
</body>
</html>
```

The MonitorPage.jsp file (see Listing 2) uses a custom tag named `<js:rpc>` to generate the client-side JavaScript functions that call their server-side counterparts.

© Copyright IBM Corporation 2009

The <js:rpc> tag has the following attributes:

script	the URL of the script that will be executed on the server
function	the signature of the JavaScript function that is called remotely
validator	an optional expression that is evaluated in the Web browser to verify the function's parameters
jsonVar	the name of an optional JavaScript variable that will store the JSON response
method	the HTTP method, which can be GET or POST
async	a boolean attribute indicating whether XMLHttpRequest should be used asynchronously or synchronously

Listing 2. The JavaScript code of MonitorPage.jsp

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

<html>
<head>
    <title>Monitor</title>
    <script src="xhr.js" type="text/javascript">
    </script>
    <script type="text/javascript">
        <js:rpc function="getInfo()" script="MonitorScript.jss"
                method="GET" async="true" jsonVar="json">
            showInfo(json, "info");
        </js:rpc>

        <js:rpc function="allocMem(size, seconds)" script="MonitorScript.jss"
                validator="valid('Size', size) && valid('Seconds', seconds)"
                method="POST" async="true"/>

        <js:rpc function="gc()" script="MonitorScript.jss"
                method="POST" async="false">
            alert("Garbage Collected");
        </js:rpc>

        function showInfo(obj, id) {
            if (typeof obj == "object") {
                for (var prop in obj)
                    showInfo(obj[prop], id + "." + prop);
            } else {
                var elem = document.getElementById(id);
                if (elem)
                    elem.innerHTML = htmlEncode(String(obj));
            }
        }

        function valid(name, value) {
            if (!value || value == "") {
                alert(name + " is required");
                return false;
            }
            var n = new Number(value);
            if (isNaN(n) || n <= 0 || Math.floor(n) != n) {
                alert(name + " must be a positive integer.");
```


© Copyright IBM Corporation 2009

```
                return false;
            } else
                return true;
        }

        function init() {
            getInfo();
            setInterval(getInfo, 1000);
        }
    </script>
    ...
</head>
...
</html>
```

The JavaScript code included between `<js:rpc>` and `</js:rpc>` will be used to process the Ajax responses. For example, in the case of the `getInfo()` function, the `json` response is passed to another function named `showInfo()`, which traverses the object tree recursively, inserting the information within the data cells of the HTML table. Because `async` is `true`, the generated `getInfo()` function will return immediately after sending the request, and the `showInfo()` function will be invoked by the Ajax callback.

The `allocMem()` function uses the `validator` attribute to verify the user input. If the `valid()` function returns `false` for any of the two parameters, the remote call is canceled and an alert message is shown. The `gc()` function will wait for the response before returning the control because `async` is `false` in its case. The `init()` function calls `getInfo()` to initialize the UI and passes the same function to `setInterval()` so that it is called every second to refresh the Web page's information.

Listing 3 contains the code produced with the `<js:rpc>` custom tag, which is implemented as a tag file named rpc.tag. Each generated function uses an `XHR` object whose prototype can be found in the xhr.js file that `MonitorPage.jsp` imports in its header. The source code of both rpc.tag and xhr.js files will be presented later in this article.

Listing 3. Generated JavaScript functions

```
var getInfoXHR = new XHR("GET", "MonitorScript.jss", true);

function getInfo() {
    var request = getInfoXHR.newRequest();
    getInfoXHR.addHeader("Ajax-Call", "getInfo()");
    function processResponse() {
        if (getInfoXHR.isCompleted()) {
            var json = eval(request.responseText);
            showInfo(json, "info");
        }
    }
    getInfoXHR.sendRequest(processResponse);
}


var allocMemXHR = new XHR("POST", "MonitorScript.jss", true);
```


© Copyright IBM Corporation 2009

```
function allocMem(size, seconds) {
    if (!(valid('Size', size) && valid('Seconds', seconds)))
        return;
    var request = allocMemXHR.newRequest();
    allocMemXHR.addHeader("Ajax-Call", "allocMem(size, seconds)");
    allocMemXHR.addParam("size", size);
    allocMemXHR.addParam("seconds", seconds);
    function processResponse() {
        if (allocMemXHR.isCompleted()) {
        }
    }
    allocMemXHR.sendRequest(processResponse);
}


var gcXHR = new XHR("POST", "MonitorScript.jss", false);

function gc() {
    var request = gcXHR.newRequest();
    gcXHR.addHeader("Ajax-Call", "gc()");
    function processResponse() {
        if (gcXHR.isCompleted()) {
            alert("Garbage Collected");
        }
    }
    gcXHR.sendRequest(processResponse);
}
```

Coding the server-side script

The MonitorScript.jss file contains three JavaScript functions that are executed on the server. The `getInfo()` function (shown in Listing 4) uses the Java Management API to obtain JVM information related to classes, memory, and threads. The entire data is packed into an object tree, which will be returned as JSON in response to an Ajax request. The `getInfo()` function doesn't have to do anything special. You'll see how the RPC mechanism is implemented in the following sections.

Listing 4. The getInfo() function of MonitorScript.jss

```
importClass(java.lang.management.ManagementFactory);

function getInfo() {
    var classes = ManagementFactory.getClassLoadingMXBean();
    var memory = ManagementFactory.getMemoryMXBean();
    var heap = memory.getHeapMemoryUsage();
    var nonHeap = memory.getNonHeapMemoryUsage();
    var gc = ManagementFactory.getGarbageCollectorMXBeans();
    var threads = ManagementFactory.getThreadMXBean();

    var gcCount = 0;
    var gcTime = 0;
    for (var i = 0; i < gc.size(); i++) {
        gcCount += gc.get(i).collectionCount;
        gcTime += gc.get(i).collectionTime;
    }

    return {
        classes: {
            loaded: classes.loadedClassCount,
```


© Copyright IBM Corporation 2009

```
            unloaded: classes.unloadedClassCount,
        },
        memory: {
            heap: {
                init: heap.init,
                used: heap.used,
                committed: heap.committed,
                max: heap.max
            },
            nonHeap: {
                init: nonHeap.init,
                used: nonHeap.used,
                committed: nonHeap.committed,
                max: nonHeap.max
            }
        },
        gc: {
            count: gcCount,
            time: gcTime
        },
        threads: {
            live: threads.threadCount,
            peak: threads.peakThreadCount
        }
    };
}
```

The `allocMem()` function (see Listing 5) creates a Java thread that executes a `run()` method implemented in JavaScript. The code creates a `byte` array, using the `newInstance()` method of the `java.lang.reflect.Array` class, and calls `java.lang.Thread.sleep()`. After that, the array becomes eligible for garbage collection.

Listing 5. The allocMem() function of MonitorScript.jss

```
function allocMem(size, seconds) {
    var runnable = new java.lang.Runnable() {
        run: function() {
            var array = new java.lang.reflect.Array.newInstance(
                    java.lang.Byte.TYPE, size);
            java.lang.Thread.sleep(seconds * 1000);
        }
    }
    var thread = new java.lang.Thread(runnable);
    thread.start();
}
```

Listing 6 shows the `gc()` function, which invokes the JVM's garbage collector.

Listing 6. The gc() function of MonitorScript.jss

```
function gc() {
    java.lang.System.gc();
}
```


© Copyright IBM Corporation 2009

Building the XMLHttpRequest wrapper

You saw the JavaScript code produced by the `<js:rpc>` tag in the previous section. To minimize and simplify the generated code, all operations related to the `XMLHttpRequest` API, such as creating a new request object or sending the HTTP request, were placed into a separate JavaScript file named xhr.js. This section presents the methods of the `XHR` object.

Initializing requests

The `XHR()` constructor (see Listing 7) takes three parameters (`method`, `url`, and `async`) and stores them as properties. The `newRequest()` method aborts the previous request if it's still active, frees the memory with the `delete` operator, and creates a new `XMLHttpRequest` instance. This behavior is suitable for applications that use Ajax to connect to a data feed or to save the user's input. In a typical case, you are interested in loading or saving only the latest data.

Listing 7. The XHR() and newRequest() functions of xhr.js

```
function XHR(method, url, async) {
    this.method = method.toUpperCase();
    this.url = url;
    this.async = async;
}

XHR.prototype.newRequest = function() {
    var request = this.request;
    if (request) {
        request.onreadystatechange = function() { };
        if (request.readyState != 4)
            request.abort();
        delete request;
    }
    request = null;
    if (window.ActiveXObject)
        request = new ActiveXObject("Microsoft.XMLHTTP");
    else if (window.XMLHttpRequest)
        request = new XMLHttpRequest();
    this.request = request;
    this.sent = false;
    this.params = new Array();
    this.headers = new Array();
    return request;
}
```

Listing 8 shows the `addParam()` and `addHeader()` methods, which let you add request parameters and headers that are included in the HTTP request when it's sent. You can use these methods as soon as a new request is created. The `checkRequest()` function will throw an exception if the `XMLHttpRequest` object has not been created or if the request has already been sent.

Listing 8. The checkRequest(), addParam(), and addHeader() functions of

© Copyright IBM Corporation 2009

xhr.js

```
XHR.prototype.checkRequest = function() {
    if (!this.request)
        throw "Request not created";
    if (this.sent)
        throw "Request already sent";
    return true;
}

XHR.prototype.addParam = function(pname, pvalue) {
    this.checkRequest();
    this.params[this.params.length] = { name: pname, value: pvalue };
}

XHR.prototype.addHeader = function(hname, hvalue) {
    this.checkRequest();
    this.headers[this.headers.length] = { name: hname, value: hvalue };
}
```

Sending requests

The `sendRequest()` function (see Listing 9) encodes the parameters, opens the request, adds the headers, sets the Ajax callback if `async` is `true`, and sends the HTTP request. If `async` is `false`, the callback is invoked after `send()`.

Listing 9. The sendRequest() function of xhr.js

```
XHR.prototype.sendRequest = function(callback) {
    this.checkRequest();
    var query = "";
    for (var i = 0; i < this.params.length; i++) {
        if (query.length > 0)
            query += "&";
        query += encodeURIComponent(this.params[i].name) + "="
            + encodeURIComponent(this.params[i].value);
    }
    var url = this.url;
    if (this.method == "GET" && query.length > 0) {
        if (url.indexOf("?") == -1)
            url += "?";
        else
            url += "&";
        url += query;
    }
    this.request.open(this.method, url, this.async);
    for (var i = 0; i < this.headers.length; i++)
        this.request.setRequestHeader(
                this.headers[i].name, this.headers[i].value);
    if (this.async)
        this.request.onreadystatechange = callback;
    var body = null;
    if (this.method == "POST") {
        body = query;
        this.request.setRequestHeader("Content-Type",
            "application/x-www-form-urlencoded");
    }
    this.sent = true;
    this.request.send(body);
    if (!this.async)
        callback();
```


© Copyright IBM Corporation 2009

```
}
```

In your Ajax callback, you may use `isCompleted()` (shown in Listing 10) to verify the request's status.

Listing 10. The isCompleted() function of xhr.js

```
XHR.prototype.isCompleted = function() {
    if (this.request && this.sent)
        if (this.request.readyState == 4)
            if (this.request.status == 200)
                return true;
            else
                this.showRequestInfo();
    return false;
}
```

Reporting errors

If an error occurs on the server side, `isCompleted()` calls `showRequestInfo()`, whose code is included in Listing 11. This function opens a window and prints the request's information: HTTP method, URL, parameters, headers, and response.

Listing 11. The showRequestInfo() function of xhr.js

```
var xhrErrorWindow = null;

XHR.prototype.showRequestInfo = function() {
    if (xhrErrorWindow && (xhrErrorWindow.closed || xhrErrorWindow._freeze))
        return;
    xhrErrorWindow = window.open("", "XHRError", "menubar=no, resizable=yes, "
            + "scrollbars=yes, width=600, height=600");
    var doc = xhrErrorWindow.document;
    doc.writeln("<p align='right'>");
    doc.writeln("<button onclick='window._freeze = true'>Freeze</button>")
    doc.writeln("</p>");
    doc.writeln(htmlEncode(this.method + " " + this.url));
    doc.writeln("<pre>" + this.request.status + "</pre>");
    doc.writeln("Parameters:");
    doc.writeln("<pre>");
    for (var i = 0; i < this.params.length; i++) {
        doc.writeln(htmlEncode(this.params[i].name
                + "=" + this.params[i].value));
    }
    doc.writeln("</pre>");
    doc.writeln("Headers:");
    doc.writeln("<pre>");
    for (var i = 0; i < this.headers.length; i++) {
        doc.writeln(htmlEncode(this.headers[i].name
                + "=" + this.headers[i].value));
    }
    doc.writeln("</pre>");
    doc.writeln("Response:");
    var response = this.request.responseText;
    doc.writeln(response);
    doc.close();
    xhrErrorWindow.focus();
}
```


© Copyright IBM Corporation 2009

Enabling window.focus() in Firefox
Select **Tools > Options > Content**, click the **Advanced** button next to **Enable JavaScript**, and make sure **Rise or lower windows** is checked in **Advanced JavaScript Settings**.

If a previous HTTP error has already opened the error window, the `focus()` call makes it the current window, which can cause usability issues if the HTTP error occurs again and again. In this case, it is also difficult to analyze the error because the window's contents would be refreshed every second, making the scrolling impossible.

To fix these usability problems, the `showRequestInfo()` function adds a button that sets a variable named `_freeze` when clicked. The request's information is not refreshed if `_freeze` is `true`. In addition, the error window is not reopened if the user closes it. After making changes in your code, you can just refresh the application's page to verify whether the error still occurs or is fixed.

The `htmlEncode()` function (see Listing 12) takes a string parameter and replaces the `&`, `<` and `>` characters with `&`, `<`, and `>`, respectively.

Listing 12. The htmlEncode() function of xhr.js

```
function htmlEncode(value) {
    return value ? value.replace(/&/g, "&amp;")
        .replace(/</g, "&lt;").replace(/>/g, "&gt;") : "";
}
```

Implementing the JavaScript-RPC mechanism

This section presents the JSP tag file that generates the JavaScript functions implementing the client-side portion of the RPC mechanism. You'll also see how their server-side counterparts are invoked and how the results are returned. First of all, though, a few notes on security.

Authorizing function calls

Server-side scripts can be treated like regular resources and you may restrict access to them using the standard security procedures of Java EE. You would typically define one or more roles whose users would have the right the access the scripts depending on the security constraints specified in the web.xml file.

Whether or not you restrict the access to your scripts, Web clients should not be able to invoke any function of a server-side script. A simple way to control which JavaScript functions may be called through the RPC mechanism is to maintain them

© Copyright IBM Corporation 2009

in a `Set` collection. The `AuthorizedCalls` bean (shown in Listing 13) provides thread-safe methods for managing the set of authorized calls.

Listing 13. The AuthorizedCalls class

```
package jsee.rpc;

import java.util.*;

public class AuthorizedCalls implements java.io.Serializable {
    private Set<String> calls;

    public AuthorizedCalls() {
        calls = new HashSet<String>();
    }

    protected String encode(String scriptURI, String functionName) {
        return scriptURI + '#' + functionName;
    }

    public synchronized void add(String scriptURI, String functionName) {
        calls.add(encode(scriptURI, functionName));
    }

    public synchronized void remove(String scriptURI, String functionName) {
        calls.remove(encode(scriptURI, functionName));
    }

    public synchronized boolean contains(
            String scriptURI, String functionName) {
        return calls.contains(encode(scriptURI, functionName));
    }

    public String toString() {
        return calls.toString();
    }

}
```

The sample application of this article needs to authorize the calls from a JSP page. The authorize.tag file (see Listing 14) has two attributes (named `function` and `script`) whose values are passed to the `add()` method of `AuthorizedCalls`. In addition, any relative script URI is converted to an absolute URI to ensure that each script is uniquely identified by the URI. Because the `AuthorizedCalls` instance is stored in the `session` scope, the server-side functions will be executed only on behalf of the authorized users (in case you restrict the access to the script for some of your users).

Listing 14. The authorize.tag file

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

<jsp:useBean id="authorizedCalls"
    class="jsee.rpc.AuthorizedCalls" scope="session"/>
<%
    String functionName = (String) jspContext.getAttribute("function");
    String scriptURI = (String) jspContext.getAttribute("script");
```


© Copyright IBM Corporation 2009

```
    if (!scriptURI.startsWith("/")) {
        String base = request.getRequestURI();
        base = base.substring(0, base.lastIndexOf("/"));
        scriptURI = base + "/" + scriptURI;
    }
    authorizedCalls.add(scriptURI, functionName);
%>
```

Another security-related aspect that is very important to analyze is how the parameters of a remotely called function are treated on the server side. It might be tempting to encode JavaScript objects as JSON strings in the Web browser and send them to the server where they could be decoded very easily with `eval()`. This would be a big mistake; it would allow malicious users to inject code that would be executed on your server.

The sample code in this article allows only primitive types (such as strings and numbers) for the parameters that are submitted with Ajax. On the server side, they are treated as strings, letting the JavaScript engine automatically convert them to numbers when required. If you need more complex types, you should not rely on `eval()` for decoding the parameters on the server. Use your own encoding/decoding methods.

Using a tag file to produce JavaScript code

The MonitorPage.jsp file, which was presented in the first section of this article, uses the `<js:rpc>` tag (see Listing 15) to generate the JavaScript routines that call the server-side functions.

Listing 15. Using <js:rpc> in MonitorPage.jsp

```
<js:rpc function="getInfo()" script="MonitorScript.jss"
        method="GET" async="true" jsonVar="json">
    showInfo(json, "info");
</js:rpc>

<js:rpc function="allocMem(size, seconds)" script="MonitorScript.jss"
        validator="valid('Size', size) && valid('Seconds', seconds)"
        method="POST" async="true"/>

<js:rpc function="gc()" script="MonitorScript.jss"
        method="POST" async="false">
    alert("Garbage Collected");
</js:rpc>
```

Listing 16 contains the rpc.tag file, which implements the custom tag. The tag file declares its attributes and the used JSP libraries, invokes the authorize.tag file with `<js:authorize>`, sets two JSP variables named `xhrVar` and `paramList`, and generates the client-side JavaScript function with the given name and parameters.

The `xhrVar` variable is used on the server side to maintain the name of a JavaScript variable that is used throughout the generated JavaScript code, which

© Copyright IBM Corporation 2009

will be executed in the Web browser. The value of the `xhrVar` variable is composed of the function's name and the `XHR` string. For example, if `function` is `getInfo()`, the value of the JSP variable (and the name of the JavaScript variable) will be `getInfoXHR`.

The other JSP variable named `paramList` will keep the list of parameters that are provided through the `function` attribute between `(` and `)`. For example, if `function` is `allocMem(size, seconds)`, the `paramList` variable will store the `size, seconds` list.

Listing 16. The rpc.tag file

```
<%@ attribute name="function" required="true" rtexprvalue="true" %>
<%@ attribute name="script" required="true" rtexprvalue="true" %>
<%@ attribute name="validator" required="false" rtexprvalue="true" %>
<%@ attribute name="jsonVar" required="false" rtexprvalue="true" %>
<%@ attribute name="method" required="false" rtexprvalue="true" %>
<%@ attribute name="async" required="true" rtexprvalue="true"
    type="java.lang.Boolean" %>

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

<js:authorize script="${script}" function="${function}"/>

<c:set var="xhrVar" value="${fn:trim(fn:substringBefore(function, '('))}XHR"/>
<c:set var="paramList"
    value="${fn:substringBefore(fn:substringAfter(function, '('), ')')}"/>

var ${xhrVar} = new XHR("${method}", "${script}", ${async});

function ${function} {
    <c:if test="${!empty validator}">
        if (!(${validator}))
            return;
    </c:if>
    var request = ${xhrVar}.newRequest();
    ${xhrVar}.addHeader("Ajax-Call", "${function}");
    <c:forEach var="paramName" items="${paramList}">
        <c:set var="paramName" value="${fn:trim(paramName)}"/>
        ${xhrVar}.addParam("${paramName}", ${paramName});
    </c:forEach>

    function processResponse() {
        if (${xhrVar}.isCompleted()) {
            <c:if test="${!empty jsonVar}">
                var ${jsonVar} = eval(request.responseText);
            </c:if>
            <jsp:doBody/>
        }
    }

    ${xhrVar}.sendRequest(processResponse);
}
```

The first line of JavaScript code generated by rpc.tag creates the `XHR` object. Then, the tag file produces the JavaScript function that can be used in the Web browser to invoke its server-side counterpart. If the `validator` attribute has a non-empty

© Copyright IBM Corporation 2009

value, the JavaScript expression is included in the generated code to decide whether the remote call can be made or not.

Next, the `newRequest()` function is used to initialize a new `XMLHttpRequest` object, which is stored in a local JavaScript variable named `request`. The generated code will add the `Ajax-Call` header, whose value is the function's signature. Then, the parameters are added to the XHR object. Listing 17 contains the code generated for the `allocMem()` function.

Listing 17. The allocMem() generated function

```
var allocMemXHR = new XHR("POST", "MonitorScript.jss", true);

function allocMem(size, seconds) {
    if (!(valid('Size', size) && valid('Seconds', seconds)))
        return;
    var request = allocMemXHR.newRequest();
    allocMemXHR.addHeader("Ajax-Call", "allocMem(size, seconds)");
    allocMemXHR.addParam("size", size);
    allocMemXHR.addParam("seconds", seconds);
    function processResponse() {
        if (allocMemXHR.isCompleted()) {
        }
    }
    allocMemXHR.sendRequest(processResponse);
}
```

After the initialization of the `XHR` object, the rpc.tag file generates an Ajax callback named `processResponse()`. This function verifies if the Ajax request is completed and evaluates the response if the `jsonVar` attribute is present.

The content placed between `<js:rpc>` and `</js:rpc>` in the JSP page, is included within the Ajax callback, using `<jsp:doBody/>`. For example, the `<js:rpc function="getInfo()" ...>` element of MonitorPage.jsp contains `showInfo(json, "info");` to process the JSON response. Listing 18 shows where this code is placed within the `getInfo()` function, which is generated by rpc.tag.

Listing 18. The getInfo() generated function

```
var getInfoXHR = new XHR("GET", "MonitorScript.jss", true);

function getInfo() {
    var request = getInfoXHR.newRequest();
    getInfoXHR.addHeader("Ajax-Call", "getInfo()");
    function processResponse() {
        if (getInfoXHR.isCompleted()) {
            var json = eval(request.responseText);
            showInfo(json, "info");
        }
    }
    getInfoXHR.sendRequest(processResponse);
}
```


© Copyright IBM Corporation 2009

Invoking the script's function

Every time you call a generated function in the Web browser, the `XHR` object is used to send an Ajax request whose URL ends with .jss. In addition, the signature of the function that must be invoked on the server side is provided as an HTTP header named `Ajax-Call`. The .jss requests are handled by a servlet named `JSServlet`, which was presented in the first part of this series.

When MonitorScript.jss of the sample application is requested, `JSServlet` actually executes three scripts: init.jss, MonitorScript.jss, and finalize.jss. The init.jss script (see Listing 19) gets the request parameters, which are Java strings, converting them to JavaScript strings and storing the parameters as the properties of the `param` object. The init.jss script also provides functions for getting and setting beans.

Listing 19. The init.jss file

```
var debug = true;
var debugStartTime = java.lang.System.nanoTime();

var param = new Object();
var paramValues = new Object();

function initParams() {
    var paramNames = request.getParameterNames();
    while (paramNames.hasMoreElements()) {
        var name = paramNames.nextElement();
        param[name] = String(request.getParameter(name));
        paramValues[name] = new Array();
        var values = request.getParameterValues(name);
        for (var i = 0; i < values.length; i++)
            paramValues[name][i] = String(values[i]);
    }
}

initParams();

function getBean(scope, id) {
    return eval(scope).getAttribute(id);
}

function setBean(scope, id, bean) {
    if (!bean)
        bean = eval(id);
    return eval(scope).setAttribute(id, bean);
}
```

Because all three scripts are executed within the same context, finalize.jss (shown in Listing 20) can use the variables and functions of init.jss and MonitorScript.jss. The finalize.jss script gets the `Ajax-Call` header, verifies if the call is authorized, uses `eval()` to invoke the script's function, and converts the returned object to a JSON string, using `toSource()`. Because the function parameters are transmitted as request parameters, their values are obtained from the `param` object.

Listing 20. The finalize.jss file

© Copyright IBM Corporation 2009

```
var ajaxCall = request.getHeader("Ajax-Call");
if (ajaxCall != null) {
    var authorizedCalls = getBean("session", "authorizedCalls");
    if (authorizedCalls.contains(request.requestURI, ajaxCall)) {
        var ajaxResponse = eval("with(param) " + ajaxCall);
        if (ajaxResponse)
            print(ajaxResponse.toSource());
    }
}

var debugEndTime = java.lang.System.nanoTime();
if (debug)
    println("// Time: " + (debugEndTime - debugStartTime) + " ns");
```

It is safe to use `eval()` for executing the function because the `Ajax-Call` header is verified with `authorizedCalls.contains()`.

Conclusion

In this article, you learned how to use RPC in Ajax and Java applications that rely on JavaScript code on both servers and clients. You've also seen how to implement Java interfaces with JavaScript, how to create Java arrays and start threads in your JavaScript code, and how to manage the life cycle of the Ajax requests when connecting to data feeds.

© Copyright IBM Corporation 2009

Downloads

Description	Name	Size	Download method
Sample application for this article	jsee_part2_src.zip	23KB	HTTP

Information about download methods

© Copyright IBM Corporation 2009

Resources

- "JavaScript EE, Part 1: Run JavaScript files on the server side" 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.
- "Ajax and Java development made simpler, Part 1" (developerWorks, April 2008) is another article that explores the idea of generating JavaScript code dynamically with JSP tag files.
- The developerWorks Web development zone is packed with tools and information for Web 2.0 development.
- The developerWorks Ajax resource center contains a growing library of Ajax content as well as useful resources to get you started developing Ajax applications today.

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.

Trademarks

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

© Copyright IBM Corporation 2009