Script Callbacks in ASP.NET

http://i.msdn.microsoft.com/Global/Images/clear.gif Contents

[Out-of-Band Calls](http://msdn.microsoft.com/en-us/magazine/cc163941.aspx#S1)   
[ASP.NET 2.0 Script Callbacks](http://msdn.microsoft.com/en-us/magazine/cc163941.aspx#S2)   
[Using Script Callbacks](http://msdn.microsoft.com/en-us/magazine/cc163941.aspx#S3)   
[Refresh the Data, Not the Page](http://msdn.microsoft.com/en-us/magazine/cc163941.aspx#S4)   
[Under the Hood of ASP.NET 2.0](http://msdn.microsoft.com/en-us/magazine/cc163941.aspx#S5)   
[Script Callbacks in ASP.NET 1.x](http://msdn.microsoft.com/en-us/magazine/cc163941.aspx#S6)   
[Conclusion](http://msdn.microsoft.com/en-us/magazine/cc163941.aspx#S7)

If you're involved in Web development you may have faced a problem that you couldn't find a good solution for—making client-to-server calls outside the current page. For example, you might want to validate the content of a textbox against data stored on the server asynchronously, without interrupting the continuity of the work or without forcing a full page refresh, which is particularly heavy for UI-rich pages. How would you code such an out-of-band call from the client to the server and back?

There are a few issues to address to solve this problem. First and foremost, you need to set up a parallel HTTP channel for sending the request and getting the response. The new request should be invisible to the user to avoid any interference with the displayed page. Finally, the response you get from this invisible request must be merged with the current page through dynamic changes to the document object model (DOM) of the page.

The great news about the scripting object model in ASP.NET 2.0 is that it allows calls to server events from client code without causing the page to post back and refresh. This is implemented through a callback mechanism.

In this column, I'll outline the problem and discuss some feasible solutions. Then, I'll focus on the details of the ASP.NET 2.0 script callback implementation and provide a tiny but effective framework that you can use in ASP.NET 1.*x* as well.

**Out-of-Band Calls**

The display of a Web page is the final step in a relatively simple process that begins with an HTTP command sent by a browser to a target URL or IP address. The browser opens a socket to the specified IP address, sends the packets, and waits for the response. When the response text arrives, it is displayed according to its content type. As a result, the previously displayed page is wiped out and fully replaced with the new one.

This pattern remains valid even when a page posts to itself—the common pattern with ASP.NET pages. In this case, the page's output is cleared and redisplayed—an effect that might not be desired by the user. If you can make assumptions about the browser's capabilities, I recommend you take another route: out-of-band calls. It's an alternative postback pattern that proves particularly effective for rich client Web applications.

In this process I have a component in the currently viewed page opening a socket to the specified remote URL and sending a request. If the request is sent asynchronously, the user can continue working with the page in a non-blocking way without experiencing delays or waiting. When the response is fully downloaded on the client, the same component detects the event, retrieves the information from the response, and decides how to update the page. While the request is ongoing, the user continues to see the same page unchanged. When the new data arrives, the component refreshes the current user interface taking full advantage of the browser's capabilities—namely, DHTML support. This generic pattern can find various implementations—the first was Remote Scripting; the latest incarnation is ASP.NET 2.0 script callbacks.

A few years ago, Microsoft released an interesting technology—Remote Scripting—that, for whatever reason, was never embraced by as broad an audience as it deserved. Remote Scripting closely follows the model I just described. It uses some client-side script to trigger a Java-language applet which, in turn, opens the socket to the destination URL. The remote URL must be a classic ASP page that includes a call to the server-side layer of Remote Scripting and exposes an object with a given name—public\_description. The particular naming schema required here serves to define the contract between the caller and the server page. (In classic ASP, there's still no notion of interfaces.)

The return value of the call must be a string. The string is processed on the client and passed to a JavaScript callback function for merging the content with the rest of the page. Using Remote Scripting is effective because it works with a fair number of browsers and doesn't require a recent version of Microsoft® Internet Explorer. However, the mechanism of out-of-band calls—regardless of how you code them—is really powerful only if the browser allows for dynamic changes to the page. Otherwise, you can do little more than display a pop-up message. For more information about Remote Scripting, check out the article at [Using Remote Scripting](http://library/en-us/rmscpt/Html/rmscpt.asp).

Another implementation of the out-of-band pattern that has been around for a while is based on Web services. The webservice.htc DHTML behavior is a simple, lightweight component that encapsulates the invocation of remote methods using the SOAP protocol. This behavior enables pages viewed with Internet Explorer 5.0 and later to communicate directly with Web services regardless of the platform. In this case, you use XmlHTTP instead of a Java-language applet, but the idea behind the mechanism is similar. The bad news is that you are limited to an uplevel family of browsers; the good news is that you can access server-side code on virtually any platform precisely because a Web service is involved. More information and the source code of the webservice.htc behavior is available for download at [WebService Behavior](http://archive/en-us/samples/internet/behaviors/library/webservice/default.asp).

All in all, out-of-band calls are an old problem that developers (including developers at Microsoft) made several attempts to solve. However, until ASP.NET 2.0 no solution was fully integrated with the hosting framework. In ASP.NET 2.0 the scripting model of the Page object is enriched with callback abilities that provide an ASP.NET-specific implementation of a kind of remote scripting. I'll take a look at that next, then I'll move on to provide an implementation that works with ASP.NET 1.*x*.

**ASP.NET 2.0 Script Callbacks**

In ASP.NET 2.0, the overall scripting model has been significantly enhanced. In addition to script callbacks, you have full access to the contents of the page's <head> tag, can programmatically assign the input focus to a particular control, read and write the title of the page, and make buttons and other controls post to any page in the application. Check out the Beta 1 documentation for quick examples and references.

To use ASP.NET 2.0 script callbacks, you define a trigger element in the page (not a Submit button) and bind it to some JavaScript code. This code will retrieve input data from the current page input fields and prepare a call to a system-provided script function named WebForm\_DoCallback in Beta 1. This function is expected (in the final release) to open the HTTP connection to the specified remote ASP.NET page. The ASP.NET runtime detects a callback invocation and executes a particular method on the page. The return value of the server-side method is passed back to the client as the response to the previous request. On the client, the response gets passed to a user-defined JavaScript callback function that can then update the user interface via DHTML. The bottom line is that a round-trip still occurs, but the page is not fully refreshed. More importantly, the user can continue working with the controls in the page while the parallel request is served. **Figure 1** shows the architecture of script callbacks in ASP.NET 2.0.

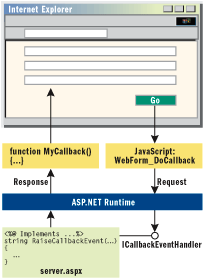


Figure 1**Script Callbacks**

You can use callbacks to update individual elements of a page, such as a chart or a panel, provide different views of the same data, download additional information on demand, or auto-fill one or more fields. In particular, the ASP.NET 2.0 TreeView control uses script callbacks extensively to implement its expand/collapse features and the GridView control uses callbacks to page and sort without explicit postbacks.

**Using Script Callbacks**

Developing an ASP.NET 2.0 page that makes use of script callbacks is a two-step procedure. First, you write the server-side code that will be invoked from the client. In doing so, you basically define the data model for the call. You decide what information must be exchanged and in which format. The actual type being exchanged must be a string, but the contents of the string can be quite different—JavaScript, XML, a comma-separated string of values, and so on.

The second step requires you to define the client-side code to trigger the out-of-band call. The remote invocation begins when a call is made to a built-in JavaScript function named WebForm\_DoCallback. You don't necessarily need to know the name and signature of this function, as you can get it from a new member of the Page class—the GetCallbackEventReference method:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl04');" \o "Copy Code)

string js = GetCallbackEventReference(this, arg, func, "null", "null");

A call to this JavaScript function is bound to a clickable element in the page—for example, a <button> tag. It is essential that the clickable element not be a Submit button. For this reason, you can't render this button using the <asp:button> control because such a server control outputs the submit button markup:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl05');" \o "Copy Code)

<input type="submit" id="button" ...>

You need to attach some client-side code (the onclick handler) to an HTML element that can fire an event if the user interacts with it. Note that if you associate a client's onclick handler with a Submit button, the client code is executed properly, but after that the page posts back and cancels all the dynamic changes brought about by the remote out-of-band call.

So far I have discussed the code needed to instruct the client page to open a channel to the server and send an HTTP request to a remote ASP.NET page. What happens next?

As expected, the ASP.NET runtime takes care of the request and begins processing it. First, the page HTTP handler (the ProcessRequest method on the Page class) determines the postback mode by looking at the request headers and body, including the \_\_VIEWSTATE hidden field. Next, it figures out if the page is being invoked on an out-of-band call. If so, it sets the public IsCallback property of the Page to true. To determine the callback mode, the ASP.NET runtime looks for a \_\_CALLBACKID entry in the Request collection. If such an entry is found, the runtime concludes that a callback invocation is being made. So, how is a callback invocation different from a postback invocation?

The runtime checks to see if the referenced control on a requested page implements the ICallbackEventHandler interface (which could be the page itself). A page that implements an interface will have the following directive (this holds true for ASP.NET 1.*x* too):

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl06');" \o "Copy Code)

<%@ Implements Interface="System.Web.UI.ICallbackEventHandler" %>

If the control implements the required interface, the ASP.NET runtime invokes the RaiseCallbackEvent method on the interface and prepares the response from the results of the call.

The ICallbackEventHandler interface features just one method with the following signature:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl07');" \o "Copy Code)

string RaiseCallbackEvent(string eventArgument)

The actual parameter for the method is retrieved from the Request collection of posted values. The string representation of the input data is contained in an entry named \_\_CALLBACKPARAM. Once again, this string can actually contain everything you want and need, including XML data or Base64 data, comma-separated values, dates, numbers, and so forth. Let's consider an example.

**Refresh the Data, Not the Page**

**Figure 2** contains the source of an ASP.NET 2.0 page that takes advantage of script callbacks. It looks like an ordinary page except for a few additional features, the most important of which are the implementation of the ICallbackEventHandler interface and the JavaScript code to refresh the page at the end of the call. The key things to notice are the client-side mechanism to trigger the out-of-band call, the server-side code that serves the call, and the client-side JavaScript code to merge results with the current page.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 2 Using Script Callbacks

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl35_ctl00_ctl00');" \o "Copy Code)

<%@ page language="C#" %>

<%@ import namespace="System.Data" %>

<%@ implements interface="System.Web.UI.ICallbackEventHandler" %>

<script language="javascript">

function UpdateEmployeeViewHandler(result, context)

{

var o = result.split(",");

e\_ID.innerHTML = o[0];

e\_FName.innerHTML = o[1];

e\_LName.innerHTML = o[2];

e\_Title.innerHTML = o[3];

e\_Country.innerHTML = o[4];

e\_Notes.innerHTML = o[5];

}

</script>

<script runat="server">

public virtual string RaiseCallbackEvent (string eventArgument)

{

// Get more info about the specified employee

int empID = Convert.ToInt32 (eventArgument);

EmployeesManager empMan = new EmployeesManager();

EmployeeInfo emp = empMan.GetEmployeeDetails (empID);

string[] buf = new string[6];

buf[0] = emp.ID.ToString ();

buf[1] = emp.FirstName;

buf[2] = emp.LastName;

buf[3] = emp.Title;

buf[4] = emp.Country;

buf[5] = emp.Notes;

return String.Join(",", buf);

}

void Page\_Load (Object sender, EventArgs e)

{

// Populate the drop-down list

EmployeesManager empMan = new EmployeesManager();

DataTable dt = empMan.GetListofNames();

cboEmployees.DataSource = dt;

cboEmployees.DataTextField = "lastname";

cboEmployees.DataValueField = "employeeid";

cboEmployees.DataBind();

// Prepare the Javascript function to call

string callbackRef = GetCallbackEventReference(this,

"document.all['cboEmployees'].value",

"UpdateEmployeeViewHandler", "null", "null");

// Bind it to a button

buttonTrigger.Attributes["onclick"] =

String.Format("javascript:{0}", callbackRef);

}

</script>

<html>

<body>

<form id="Form1" runat="server">

<h1>Select a name and click for details</h1>

<asp:dropdownlist id="cboEmployees" runat="server" />

<button runat="server" id="buttonTrigger">More Info</button>

<br />

<table>

<tr><td><b>ID</b></td><td><span id="e\_ID" /></td></tr>

<tr><td><b>Name</b></td><td><span id="e\_FName" /></td></tr>

<tr><td><b>Last Name</b></td><td><span id="e\_LName" /></td></tr>

<tr><td><b>Title</b></td><td><span id="e\_Title" /></td></tr>

<tr><td><b>Country</b></td><td><span id="e\_Country" /></td></tr>

<tr><td><b>Notes</b></td>

<td><i>

<span id="e\_Notes" /></i></td></tr>

</table>

</form>

</body>

</html>

As I mentioned, you need an interactive HTML element that doesn't submit the page. The HTML 4.0 <button> tag is a good choice here. However, because it needs a dynamically generated onclick event handler, you must mark it with the runat attribute:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl10');" \o "Copy Code)

<button runat="server" id="btn">More Info</button>

The onclick handler points to a built-in piece of JavaScript code that initiates and controls the out-of-band call. The GetCallbackEventReference method on the Page class returns the script string that starts the callback:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl11');" \o "Copy Code)

string js = GetCallbackEventReference(this,

"document.all['cboEmployees'].value",

"UpdateEmployeeViewHandler", "null", "null");

btn.Attributes["onclick"] = String.Format("javascript:{0}", js);

The function receives the arguments described in **Figure 3**. In the previous code snippet, I'm making the call to the current page and passing the value of the item currently selected in a page's dropdown list. UpdateEmployeeViewHandler is the name of the JavaScript function that will merge callback results with the current page. Context is any DHTML reference or extra information you want to pass to the callback is useful to the merge process. Finally, the error callback is a JavaScript function that is automatically invoked if an error occurs during the callback. As you can see in **Figure 4**, you select an employee in the dropdown list and click the button to raise a callback event on the server. The handler for the button is shown in the following code snippet:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl12');" \o "Copy Code)

<button id="buttonTrigger"

onclick="javascript:WebForm\_DoCallback('\_\_Page',

document.all['cboEmployees'].value,

UpdateEmployeeViewHandler, null, null)">

More Info

</button>

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 3 GetCallbackEventReference Arguments

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| control | The control or the page which implements RaiseCallbackEvent. |
| argument | Client-side script that will be executed prior to making the callback. The return value of the evaluation of this script on the client is sent to the RaiseCallbackEvent method via the eventArgument parameter. |
| clientCallback | Name of the client-side event handler which will receive the result of a successful server event. |
| context | Client-side script that will be evaluated on the client prior to making the callback. The result of this will be passed back to the client-side event handler via the context parameter. |
| clientErrorCallback | Name of the client-side event handler which will receive the result of the RaiseCallbackEvent method when an error occurs. |

In addition, the markup of the page contains the related instructions shown in the following code:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl14');" \o "Copy Code)

<script src="WebResource.axd?a=s&amp;r=WebForms.js&amp;t={...}"

type="text/javascript" />

<script type="text/javascript">

<!--

var pageUrl='/Intro20/Samples/Ch01/Callback/Invoke.aspx';

WebForm\_InitCallback();

// -->

</script>

WebResource.axd is a new built-in HTTP handler in ASP.NET 2.0 that's used to retrieve script code into pages. This handler guarantees that all the script code necessary to perform the callback is correctly referenced within the page or control. In particular, WebResource.axd ensures that calls to WebForm\_DoCallback and WebForm\_InitCallback are successfully resolved. If you want to take a look at the real source code behind script callbacks in ASP.NET 2.0, open the Temporary Internet Files folder on your Web server machine and look up the file WebResource.axd. The code injected through this HTTP handler represents the callback manager in charge of sending the request and handling the results. The manager registers any JavaScript callback function and calls it back when the server event has been fully processed. The <script> blocks I just showed you are inserted in the page when you call the GetCallbackEventReference function. For this reason, you should never use WebForm\_DoCallback directly; the call will fail if the right <script> blocks aren't inserted and properly initialized. Another good reason for not calling WebForm\_DoCallback directly is that this name is not part of the public page API. As such, it may change without notice in future builds of ASP.NET 2.0 as well as in future minor or major upgrades of the platform.

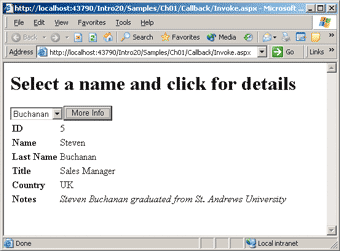


Figure 4**Trying it Out**

The ID of the currently selected employee (shown in **Figure 4**) is ultimately passed to the server-side implementation of the ICallbackEventHandler interface and then it's used to retrieve the information for the client:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl16');" \o "Copy Code)

string RaiseCallbackEvent (string eventArgument)

{

// Get more info about the specified employee

int empID = Convert.ToInt32 (eventArgument);

EmployeesManager empMan = new EmployeesManager();

EmployeeInfo emp = empMan.GetEmployeeDetails(empID);

// Prepare the string for the script callback.

string buf = "";

•••

return buf;

}

The format of the event argument string is arbitrary and left up to you. The same is true for the string you return to the script callback—the aforementioned UpdateEmployeeViewHandler function that you see in **Figure 5**.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 5 UpdateEmployeeViewHandler

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl42_ctl00_ctl00');" \o "Copy Code)

<script language="javascript">

function UpdateEmployeeViewHandler(result, context)

{

// Comma-separated string created in RaiseCallbackEvent

var o = result.split(",");

// Merge data with the existing client page UI

e\_ID.innerHTML = o[0];

e\_FName.innerHTML = o[1];

e\_LName.innerHTML = o[2];

e\_Title.innerHTML = o[3];

e\_Country.innerHTML = o[4];

e\_Notes.innerHTML = o[5];

}

</script>

To merge the server-side generated values with the existing page, you typically use the page's DHTML object model. You give each updateable HTML tag a unique ID and modify its contents scriptwise using the innerHTML property or any other property and method the DOM supplies. **Figure 4** shows the state once a callback operation has completed. More importantly, there's no postback visible to the user even though a round-trip still occurs under the hood.

**Under the Hood of ASP.NET 2.0**

To understand how to implement script callbacks in ASP.NET 1.*x*, you should learn exactly what really happens in ASP.NET 2.0. In particular, you should figure out the contents of the HTTP request sent and what component is used to accomplish that. The ASP.NET 2.0 documentation states that in order to take advantage of script callbacks the user must be running Internet Explorer 5.0 or later. The prototype of WebForm\_DoCallback is shown here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl19');" \o "Copy Code)

function WebForm\_DoCallback(

eventTarget,

eventArgument,

eventCallback,

context,

errorCallback)

{

•••

}

It uses a COM object to issue an HTTP POST or GET command to the specified target URL. The COM object used here is an old acquaintance of many developers:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl20');" \o "Copy Code)

var xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");

The HTTP verb is GET or POST depending on the size of the data to send. If the size exceeds 2KB, a POST command is used. The HTTP request consists of three logical elements: \_\_CALLBACKID, \_\_CALLBACKPARAM, and posted data. The \_\_CALLBACKID value contains the destination URL (the event target parameter), whereas \_\_CALLBACKPARAM carries the input parameter for the server-side stub method. The posted data is collected by the WebForm\_InitCallback method and appended to the HTTP command. The following pseudocode shows how to initiate the out-of-band request:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl21');" \o "Copy Code)

var xml = new ActiveXObject("Microsoft.XMLHTTP");

xml.onreadystatechange = callback;

•••

postData = \_\_theFormPostData +

"\_\_CALLBACKID=" + eventTarget +

"&\_\_CALLBACKPARAM=" + escape(eventArgument);

xml.open("POST", pageUrl, true);

xml.setRequestHeader("Content-Type",

"application/x-www-form-urlencoded");

xml.send(postData);

When the request terminates, the specified callback is invoked to complete the job. As an aside, be aware that the Microsoft.XmlHttp COM object ships with Internet Explorer 5.0 and later. And, of course, these implementation details are subject to change.

**Script Callbacks in ASP.NET 1.*x***

At this point, I'd say that implementing script callbacks in ASP.NET 1.*x* shouldn't be that hard. You need a common piece of JavaScript code to inject into all pages that intend to support client callbacks. This code should initiate and control the remote URL invocation. The code in **Figure 6** provides a possible implementation, instantiating the XmlHttp object and preparing a POST command. The function (named DoCallback in the example) extends the query string of the URL with a couple of parameters. The first parameter, Callback, contains a Boolean parameter to indicate whether or not the request is going to be a regular postback or a callback. The second parameter, Param, contains the input parameters for the server-side code. Note that the code in **Figure 6** can be easily extended to work asynchronously:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl22');" \o "Copy Code)

xmlRequest.onreadystatechange = callbackFunction;

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 6 Callback in ASP.NET

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl47_ctl00_ctl00');" \o "Copy Code)

<script language="javascript">

function DoCallback(url, params)

{

// url: URL to invoke

// params: string object to pass to the remote URL

// Add some parameters to the query string

var pageUrl = url + "?callback=true&param=" + params;

// Initialize the XmlHttp object

var xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");

// Prepare for a POST statement and synchronous. (All this is

// arbitrary and can be changed in your own implementation.)

xmlRequest.open("POST", pageUrl, false);

xmlRequest.setRequestHeader("Content-Type",

"application/x-www-form-urlencoded");

xmlRequest.send(null);

// Return the XmlHttp object

return xmlRequest;

}

</script>

The following snippet shows a piece of client code that, attached to a button, fires the client callback and refreshes the user interface:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl25');" \o "Copy Code)

function MoreInfo()

{

var selectedEmpID = document.all["EmployeeList"].value;

var xml = DoCallback("webform1.aspx", selectedEmpID);

// Sync updates

Msg.innerHTML = xml.responseText;

}

How is the server page structured? **Figure 7** shows the full source code of a page just like it. In the Page\_Load event handler, you check to see if the request is a callback using code like this:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl26');" \o "Copy Code)

if (Request.QueryString["callback"] != null)

{

string param;

param = Request.QueryString["param"].ToString();

Response.Write(RaiseCallbackEvent(param));

Response.Flush();

Response.End();

return true;

}

If the query string contains the Callback parameter, you invoke a predefined function—here RaiseCallbackEvent—and pass the return value to Response. Next, you flush the output buffer and end the request. That's it.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 7 The Server Page

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl50_ctl00_ctl00');" \o "Copy Code)

namespace MsdnMag

{

public class CallbackPage : System.Web.UI.Page

{

// References to page controls

protected DropDownList EmployeeList;

protected HtmlButton ButtonGo;

// Initialize the page here

private void Page\_Load(object sender, EventArgs e)

{

if (IsCallback())

return;

if (!IsPostBack)

PopulateList();

// Button-to-callback binding

string callbackRef = "MoreInfo()";

ButtonGo.Attributes["onclick"] = callbackRef;

}

private bool IsCallback()

{

if (Request.QueryString["callback"] != null)

{

string param = Request.QueryString["param"].ToString();

Response.Write(RaiseCallbackEvent(param));

Response.Flush();

Response.End();

return true;

}

return false;

}

// Handle the "Go Get Data" button click

protected void OnGoGetData(object sender, EventArgs e)

{

}

// Populate the drop-down list with employee names

private void PopulateList()

{

SqlDataAdapter adapter = new SqlDataAdapter(

"SELECT employeeid, lastname FROM employees",

"SERVER=(local);DATABASE=

northwind;TRUSTED\_CONNECTION=true;");

DataTable table = new DataTable();

adapter.Fill(table);

EmployeeList.DataTextField = "lastname";

EmployeeList.DataValueField = "employeeid";

EmployeeList.DataSource = table;

EmployeeList.DataBind();

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Implement the callback interface

string RaiseCallbackEvent(string eventArgument)

{

return "You clicked: " + eventArgument;

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

}

}

It is worth noting that in ASP.NET 1.*x* you need to verify yourself if the request is a callback. This check is necessary for any implementation, but in ASP.NET 2.0 it is buried in the Page class.

**Conclusion**

Script callbacks allow you to execute out-of-band calls back to the server. These calls are special postbacks, so a round-trip always occurs; however, unlike classic postbacks, script callbacks don't redraw the whole page and they give users the illusion that everything is taking place on the client. ASP.NET 2.0 provides a built-in mechanism for client callbacks based on the services of a COM object that ships with Internet Explorer 5.0 and later. By using the same object, you can implement a script callback mechanism in ASP.NET 1.*x* as well. The companion code, available from the link at the top of this article, provides a sample implementation.