DotNetNuke Client API Client Callback

Jon Henning

Version 1.1.0
Last Updated: June 20, 2006
Category: Client API

Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, places, or events is intended or should be inferred.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Perpetual Motion Interactive Systems, Inc. Perpetual Motion Interactive Systems may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Perpetual Motion, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Copyright © 2005, Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

DotNetNuke® and the DotNetNuke logo are either registered trademarks or trademarks of Perpetual Motion Interactive Systems, Inc. in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

DotNetNuke Client API Client Callback

Abstract

In order to clarify the intellectual property license granted with contributions of software from any person or entity (the "Contributor"), Perpetual Motion Interactive Systems Inc. must have a Contributor License Agreement on file that has been signed by the Contributor.

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

Contents

Introduction 1

Goals 1

Implementing Tomorrow's Technology Today 1

What a Developer Needs To Do 2

Accessing FORM Values In A Callback 5

How It Works 6

LifeCycle of a Client Callback 7

Closing Thoughts 16

Additional Information 18

Appendix A: Document History 19

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

DotNetNuke Client API-ClientCallback

Introduction

The DotNetNuke Client API is composed of both server-side and client-side code that works together to enable a simple and reliable interface for the developer to provide a rich client-side experience. The ability to have client-side code communicate with the server without requiring a postback improves on this experience.

Goals

Provide a consistent means of communicating information between the client-side script and server-side code.

Allow a broader range of browsers to support Client Callbacks than ASP.NET 2.0

Enable Client Callbacks to work in ASP.NET 1.x

Implementing Tomorrow's Technology Today

As evidenced by our implementation of the Provider model and Membership Provider, the DotNetNuke team demonstrated its commitment to continually move towards the latest standards that are pushed forth in the ASP.NET platform.

It is a common practice to have client side script need to send data to the server. Within a web browser this is typically accomplished by either a POST or a GET. Both of these commands usually require a repainting of the page, since it is usually the form within the page that does the post, or the document's location being changed to do the GET. Over the years, there have been many attempts to improve the interaction between the client side code and the web server. Technologies like Remote Data Services (RDS), Remote Scripting, Javascript Remote Scripting (JSRS), and the Webservice Behavior were developed to address this interaction. RDS is outdated since it relies on COM and ActiveX. Remote Scripting was introduced and was quite successful since it utilized a Java Applet to initiate the request to the server. Unfortunately, Netscape decided to not support unsigned Java Applets starting with version 6, thus making it about as useful as an ActiveX control. Additionally, the JVM is no longer being included with the Windows platform. The Webservice behavior is a very cool way to have the client-side code interact with the server, however, since it relies on a behavior and the MSXML parser, it is an Internet Explorer only solution. That leaves JSRS, which works by creating a hidden IFRAME object on the page with a FORM embedded in it that can be posted to the server. This makes it the most widely accepted standard for the interaction, since it only requires the browser to support the IFRAME object and FORM elements.

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

Another enhancement that is to be offered by the next version of ASP.NET is Script Callbacks. While there have been many ways to accomplish this in the past (see sidebar), there really was not a standard way to implement the callback. For a detailed explanation of this new feature as it relates to ASP.NET 2.0 read the following articles.

http://www.devx.com/dotnet/Article/20239/0/
http://bitarray.co.uk/ben/articles/234.aspx

The DotNetNuke Team decided to enhance the ClientAPI to support client Callbacks in ASP.NET 1.x. An additional requirement was to make this implementation support a broader range of browsers. In order to accomplish this, the reliance on the XmlHttp object needed to be abstracted. This is now handled in the new dnn.xmlhttp namespace on the client.

A complete listing of the methods within this namespace along with their descriptions can be found in the DotNetNuke ClientAPI Namespace Browser.

This document will outline the steps a developer needs to take to implement the Client Callback functionality, along with give a detailed look at what the ClientAPI is doing "under the covers".

What a Developer Needs To Do

If you read one of the links listed above the following steps should familiar to you.

Step 1 - Implement the IClientAPICallbackEventHandler

In order to handle a client callback your page or a control on the page must implement the `IClientAPICallbackEventHandler` interface.

```
Public Class WebForm1
Inherits System.Web.UI.Page : Implements IClientAPICallbackEventHandler
```

Step 2 - Obtain Javascript to Invoke CallBack

This is done by calling the `GetCallbackEventReference` passing in the following parameters

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

Argument	Description
objControl	The control that is responsible for handling the callback. This control must implement the `IClientAPICallbackEventHandler` interface.
strArgument	This string is evaluated on the client-side and is passed to the server-side callback handler.
strClientCallback	The pointer to the client-side function that is to be invoked when callback is successful and complete.
strContext	This string is evaluated on the client-side and will be passed to the client side callback methods. For example, if you have multiple controls that can issue the same callback, when the method returns you need to distinguish which control caused the callback. This context variable allows you to pass the control reference to the `strClientCallBack/strClientErrorCallBack` function
strErrorClientCallback	The pointer to the client-side function that is to be invoked when callback errors out.
strPostChildrenOf	If specified the callback will include all child elements (including self) of passed in ID. This means that the controls that normally post on a submit will now be included and accessible on the server-side through Request(MyControl.ClientID). Additionally, you will have complete access to variables set with setVar.

Step 3 - Assign Callback Script to Javascript Event Handler
Assign the resulting javascript from the GetCallbackEventReference method to the client-side event handler. For example, if you wish to have the onclick event of a button do a callback

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

```
mybutton.attributes.add("onclick", GetCallbackEventReference(Page, "'test'",
"successFunc", "this", "errorFunc")
```

Step 3b – Register Javascript

It has been found that the ClientAPI does not automatically register the correct scripts when obtaining the GetCallbackEventReference method. Until the next version after 3.2/4.0 releases you will also need to add the following code just before the association of the onclick method with the callback

```
If ClientAPI.BrowserSupportsFunctionality(ClientAPI.ClientFunctionality.XMLHTTP) _
  AndAlso ClientAPI.BrowserSupportsFunctionality(ClientAPI.ClientFunctionality.XML) Then
   ClientAPI.RegisterClientReference(Me.Page, ClientAPI.ClientNamespaceReferences.dnn_xml)
   ClientAPI.RegisterClientReference(Me.Page, ClientAPI.ClientNamespaceReferences.dnn_xmlhttp)
```

Step 4 - Handle Potential Callback in `Page_Init`

You will need to add this code to the Page_Init of any page that is supposed to handle callbacks. This call is responsible for checking each request to see if it is a callback. It must be done early in the page's lifecycle so that no response is written back other than the result of the callback.

```
Protected Overrides Sub OnInit(ByVal e As System.EventArgs)
   ClientAPI.HandleClientAPICallbackEvent(Me)
End Sub
```

Note: If you are simply trying to enable callbacks within DotNetNuke you do not need this step, for it is already done in the page. Since the ClientAPI can be used on any project this step is documented.

Step 5 - Write Server-Side Event Handler

In order to implement the `IClientAPICallbackEventHandler` interface you need to implement the methods within the interface. In this case there is a single method, `RaiseClientAPICallbackEvent`.

```
Public Function RaiseClientAPICallbackEvent(ByVal eventArgument As String) As String _
  Implements UI.Utilities.IClientAPICallbackEventHandler.RaiseClientAPICallbackEvent
   Return "HELLO: " & eventArgument
End Function
```

Step 6 - Write Callback Javascript Function and Error Javascript Function

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

```
function successFunc(result, ctx)
{
   alert('received: ' + result + ' (' + ctx.id + ')');
}
function errorFunc(result, ctx)
{
   alert('failed: ' + result + ' (' + ctx.id + ')');
}
```

Accessing FORM Values In A Callback

Starting with version 1.2 of the ClientAPI (included in DNN 3.3/4.1), you will now be able to specify that your callback should contain the values of one or more FORM elements. For example, if you wish to have the entire up-to-date form values pushed to the server you would register your client callback script like this.

```
mybutton.attributes.add("onclick", GetCallbackEventReference(Page, "'test'",
"successFunc", "this", "errorFunc", "Form")
```

Note: The last parameter (Form) specifies the element ID whose children we wish to post the values for.

You would then be able to access the form values during a callback through code like this.

```
Dim sVal As String = Request(Me.txt.ClientID)
```

Note: You cannot use the Me.txt.Value property since a callback will be processed before the control values are populated

You should note that you only want to post back the values necessary for your callback in order to keep your payload as small as possible. The Register function allows you to pass in any client-side id and it will loop its child controls and only post those values. For example, you could have a DIV tag containing 3 textbox controls, by specifying the DIV tag's client-side ID as the last parameter, those three textbox control values will be accessible in the callback.

You may recall, that the way the ClientAPI passes variables back and forth through its setVar/getVar functionality is with a HIDDEN FORM element. Due to this, it is easy to only post this information for your callback.

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

```
mybutton.attributes.add("onclick", GetCallbackEventReference(Page, "'test'",
"successFunc", "this", "errorFunc",
DotNetNuke.UI.Utilities.ClientAPI.DNNVARIABLE_CONTROLID)
```

Note: A constant has been defined within the ClientAPI namespace that represents this hidden variable (__dnnVariable).

If you have registered your callback script specifying the dnnVariable or the entire form you will be able to access your variables through the normal method

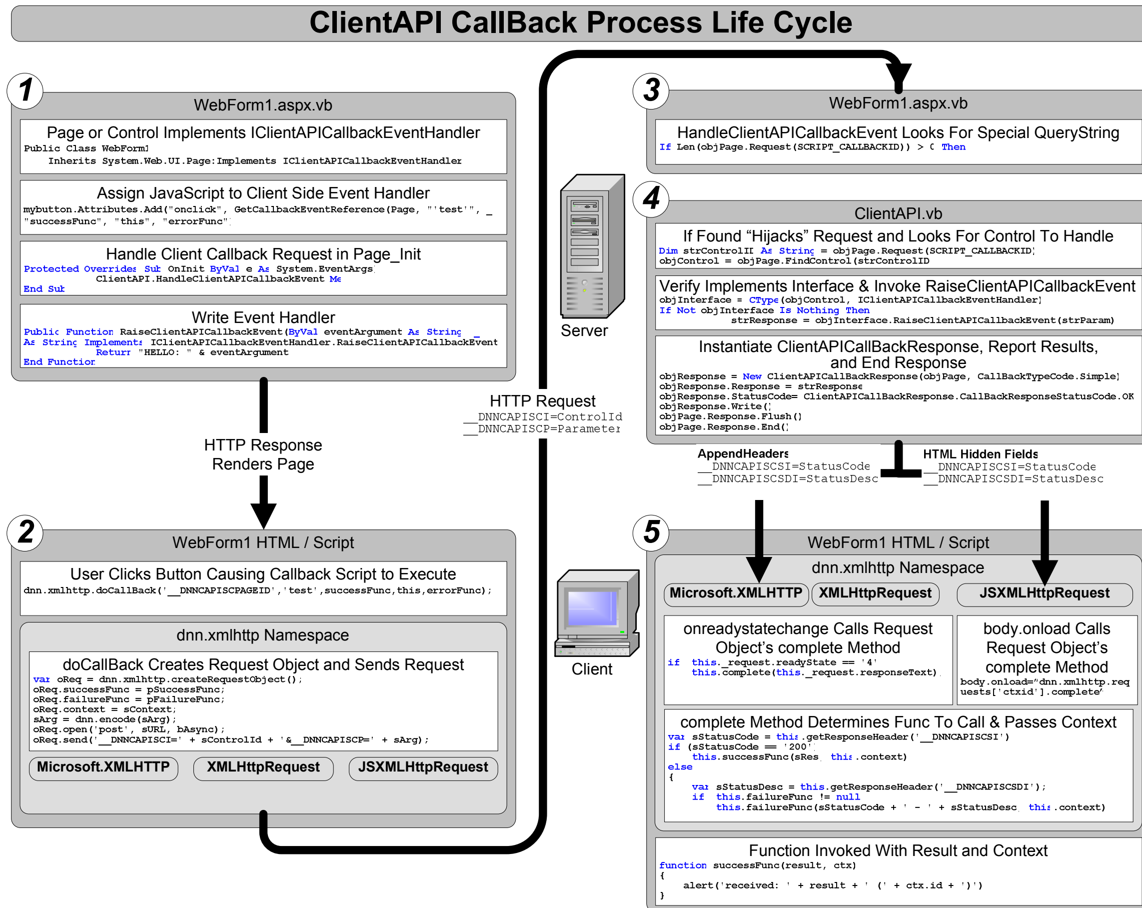
```
DotNetNuke.UI.Utilities.ClientAPI.GetClientVariable(Me.Page, "myvar")
```

Note: You will not be able to set any values and have them reflected on the client.

How It Works

If you are only interested in using the Client Callback, the preceding steps is all you need to know. The following diagram and detailed description is for those of you interested in how this feature works “behind the scenes”.

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.



LifeCycle of a Client Callback

1 – Setup and Registration

This step is already covered in the “What a Developer Needs to Do” section.

2 – Client-Side Handling of the Callback

The script that is rendered to the client will look something like this

```
dnn.xmlhttp.doCallBack('__DNNCAPISCPAGEID','test',successFunc,this,errorFunc);
```

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

This statement will call the doCallBack method in the dnn.xmlhttp namespace. The first parameter represents the control's id that is going to handle the callback. Since a page does not typically have an id assigned, a pre-defined constant (__DNNCAPISCPAGEID) is used. There will be a check for this control id on the server-side, and if found no FindControl will be issued, since we will know it is the page itself handling the request.

The next parameter ('test') is evaluated on the client side. While the hard-coded string is something that is easily understood, you should take note that we needed to enclose it in quotes. This is necessary to allow for any javascript to be evaluated on the client-side, so things like new Date(), myfunction(), or myobject.myproperty can be used.

The third and the fifth parameters contain a pointer to the methods that will be invoked when the asynchronous request is complete. Note that they are pointers, they do **not** contain parenthesis.

Probably the most commonly misunderstood parameter is the fourth one (context). This is because it is **not** passed to the server-side request, instead it lives on the client. In the ClientAPI's implementation of the callback functionality, this parameter is attached to the request object as you will see in the following section. The ability to attach context to a request is important because when the asynchronous request finally returns there would be no other easy way to determine which event/control initiated the request.

```
doCallBack and the XmlHttpRequestObject
var oReq = dnn.xmlhttp.createRequestObject();
oReq.successFunc = pSuccessFunc;
oReq.failureFunc = pFailureFunc;
oReq.context = oContext;
sURL += '__DNNCAPISCI=' + sControlId + '&__DNNCAPISCP=' + escape(sArg);
oReq.open('post', sURL, bAsync);
oReq.send();
```

The first thing the doCallBack method does is create a new XmlHttpRequestObject. This object is responsible for abstracting the various methods each browser can make client callbacks to the server. As of this writing, there are three implementations: ActiveX, Native, and Javascript IFrames. ActiveX is supported by Internet Explorer, Native is supported by most Gecko implementations, and the fallback is Javascript IFrames, which is theoretically supported by any browser that can create an IFrame.

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

Like other aspects of the ClientAPI, the XmlHttp namespace can be configured to work with the various browsers through the ClientAPICaps.config file. It is worth noting how this file is setup in relation to each browser and its functionality. You will notice by the first section that pretty much all browsers support the XMLHTTP namespace.

```
<functionality nm="XMLHTTP" desc="Client Side HTTP Requests">
   <supports>
      <browser nm="IE" minversion="4" />
      <browser nm="Netscape" minversion="5" />
      <browser nm="Opera" minversion="7" />
      <browser contains="Konqueror" />
   </supports>
</functionality>
```

However, the functionality only works in some browsers when they specify that they need the Javascript helper object/file (dnn.xmlhttp.jsxmlhttprequest.js). This allows the ClientAPI to only send down the JS helper namespace when necessary, thus decreasing the initial payload when the user is browsing with a feature-rich browser.

```
<functionality nm="XMLHTTPJS" desc="Requires Javascript HTTP Requests">
   <supports>
      <browser contains="Opera" />
      <browser contains="Konqueror" />
   </supports>
</functionality>
```

Note: This same type of configuration is used in the XML namespace section.

The constructor to the `XmlHttpRequest` object accepts the underlying request object. This object is expected to implement two methods: open and send. The open method accepts the request type (GET/POST), the URL, and whether or not the call should be asynchronous. The send method simply sends off the request and has no parameters.

IMPORTANT: Because the IFRAME implementation of the Request object currently only supports asynchronous mode, it is highly recommended that unless you know your end-user's browser configuration, you always pass in true for this parameter.

Argument	Description
_request	Holds reference to underlying request object
successFunc	Stores pointer to function to be invoked upon successful request

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

Argument	Description
failureFunc	Stores pointer to function to be invoked upon failed request
Context	Stores context of the request to be passed along to the success/failure function

After the request object is created, the `doCallBack` method assigns the pointers to the functions for success and failure. Then it attaches the context object to the request. Next, it assigns two parameters (`__DNNCAPISCI`, `__DNNCAPISCP`). These are important since it is by them that the server-side `HandleClientAPICallbackEvent` is able to determine if the incoming request is a callback or a normal request. The __DNNCAPISCI parameter contains the ID of the server-side control that implements the `IClientAPICallbackEventHandler` interface. The `__DNNCAPISCP` parameter contains the argument the `RaiseClientAPICallbackEvent` is to receive.

At this point it is probably worth discussing how the Javascript IFRAME request works. The other two implementations are pretty much a "black-box" to the developer so I will not go into more detail with them until we get to the section on returning results.

The JsXmlHttpRequest Object

The constructor of this object is as follows

```
dnn.xmlhttp.contextId += 1;
this.contextId = dnn.xmlhttp.contextId;
this.method = null;
this.url = null;
this.async = true;

this.iframe = document.createElement('IFRAME');
this.iframe.name = 'dnniframe' + this.contextId;
this.iframe.id = 'dnniframe' + this.contextId;
this.iframe.height = 0;
this.iframe.width = 0;
this.iframe.style.visibility = 'hidden';
document.body.appendChild(this.iframe);
```

Due to the asynchronous nature of this methodology, there is a need to uniquely identify a request, since more than one can be made at a time. Therefore the xmlhttp namespace

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

contains a shared/static counter (`contextId`) that will be incremented with each new request. This id is then assigned to the request object. Next, the method, url, and async properties are all initialized. Finally the creation of the IFRAME is handled, appending the `contextId` to the suffix of the control. You will also note that the display of the IFRAME is set such that it will not be visible to the user.

The open method on the `JsXmlHttpRequest` object simply assigns the `method`, `url`, and `async` properties. Whereas, the send method is a little more complicated.

```
this.assignIFrameDoc();
if (this.doc == null)  //opera does not allow access to iframe right away
{
   window.setTimeout(dnn.dom.getObjMethRef(this, 'send'), 1000);
   return;
}
this.doc.open();
this.doc.write('<html><body>');
this.doc.write('<form name="TheForm" method="post" target="" ');
var sSep = '?';
if (this.url.indexOf('?') > -1)
   sSep = '&';

this.doc.write(' action="' + this.url + sSep + '__U=' + this.getUnique() + '">aaa');
this.doc.write('<input type="hidden" name="ctx" value="' + this.contextId + '">');
this.doc.write('</form></body></html>');
this.doc.close();

this.assignIFrameDoc();       //opera needs this reassigned after we wrote to it

this.doc.forms[0].submit();
```

The `assignIFrameDoc` method is responsible for assigning the `JsXmlHttpRequest`'s doc property to the document object of the IFRAME. Due to variations in the way each browser accomplishes this, the implementation has been moved to its own method.

The next line verifies that the doc property has been set successfully. I have found that Opera does not allow instant access to a newly created IFRAME's document. Therefore we will wait a second and try calling ourself again.

Note: Discussion of the `getObjMethRef` *method is beyond the scope of this document. For now, just assume it is a magic function that allows us to obtain the pointer to the current request's send method.*

The last couple lines should be self-explanatory. We are writing out a new HTML FORM that we intend on submitting to the server. Take note of the FORM's ACTION being assigned the same parameters as the other transports, along with a new one that makes

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

sure each call to the server gets processed due to it's uniqueness. You should also note that we are sending the `contextId` of the request to the server. You will see why this is necessary in the next couple sections.

3 – Page_Init Handling of CallBack Requests

It is important that the page that is to handle the request look for a potential callback in the init method. This cannot be done any sooner since the controls on the page are not loaded yet (thus the `FindControl` would not work), and it should not be done any later since there is a chance that some unwanted response text may be generated.

4 – HandleClientAPICallbackEvent

The first thing the function does is check to see if the current request is a callback. It does this by looking for the presence of a particular parameter (`__DNNCAPISCI`).

```
If Len(objPage.Request(SCRIPT_CALLBACKID)) > 0 Then
```

If found we know this is a callback request, therefore we take over the request. A `ClientAPICallBackResponse` object is instantiated and will be responsible for reporting back the status codes and response from the server.

```
objResponse=New ClientAPICallBackResponse(objPage, CallBackTypeCode.Simple)
```

Note: The second parameter in the constructor allows for different Callback types to be handled. For right now the Simple type is the only one supported, but the vision is to allow for more complicated requests/responses to be handled (i.e. xmlrpc like serialization of parameters).

The first thing done is to find the control that is meant to handle the callback. The only trick here is that the page itself could be responsible for handling the callback. If that was the case a special ID is used since Page object's typically do not have IDs assigned.

```
Dim strControlID As String = objPage.Request(SCRIPT_CALLBACKID)
If strControlID = SCRIPT_CALLBACKPAGEID Then
   objControl = objPage
Else
objControl = objPage.FindControl(strControlID)
End If
```

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

If no control was found a status code (404) that states the control was not found will be sent.

```
objResponse.StatusCode =
ClientAPICallBackResponse.CallBackResponseStatusCode.ControlNotFound
objResponse.StatusDesc = "Control Not Found"
```

The next step is to verify the control implements the `IClientAPICallbackEventHandler` interface.

```
objInterface = CType(objControl, IClientAPICallbackEventHandler)
If Not objInterface Is Nothing Then
```

If the interface is found to be Nothing/null, then a different status code is sent.

```
objResponse.StatusCode = CallBackResponseStatusCode.InterfaceNotSupported
objResponse.StatusDesc = "Interface Not Supported"
```

If the interface is supported, then the `RaiseClientAPICallbackEvent` method is called passing in the parameter obtained from the request.

```
objResponse.Response = objInterface.RaiseClientAPICallbackEvent(strParam)
```

If no exception is raised the status code sent down to the client will be OK.

```
objResponse.StatusCode = ClientAPICallBackResponse.CallBackResponseStatusCode.OK
```

Otherwise a generic status code is sent.

```
objResponse.StatusCode = CallBackResponseStatusCode.GenericFailure
objResponse.StatusDesc = ex.Message
```

Finally the results are written and the response ended.

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

```
objResponse.Write()
objPage.Response.Flush()
objPage.Response.End()
```

The idea behind the `ClientAPICallBackResponse` object was to abstract all of the implementation details for the different transport protocols away from the code that processes the requests. Hopefully this will allow for additional transport protocols to be handled gracefully. Lets have a look at how the object handles the two different types of requests.

```
Public ReadOnly Property TransportType() As TransportTypeCode
   Get
      If Len(m_objPage.Request.Form("ctx")) > 0 Then
         Return TransportTypeCode.IFRAMEPost
      Else
         Return TransportTypeCode.XMLHTTP
      End If
   End Get
End Property
```

The object contains a property that returns the type of request it is handling. It detects the request type by the presence of the "`ctx`" value posted. While this is not the real reason for passing the request's context Id to the server, it serves as an easy way to identify the XMLHTTP request from the IFRAMEPost.

The writing of the response is where the transport protocol varies. Lets start out by looking at the easier one of the two.

```
Public Sub Write()
    Select Case Me.TransportType
        Case TransportTypeCode.XMLHTTP
            m_objPage.Response.AppendHeader(ClientAPI.SCRIPT_CALLBACKSTATUSID, CInt(Me.StatusCode).ToString)
            m_objPage.Response.AppendHeader(ClientAPI.SCRIPT_CALLBACKSTATUSDESCID, Me.StatusDesc)
            m_objPage.Response.Write(Response)
```

The status code and description can be sent back to the client via a response header. This is because the client-side object is capable of reading the headers. This allows for the response text to simply be placed in the output stream.

Unfortunately, the IFRAMEPost type is not quite as simple. This is due to the fact that there is no way for the client-side code to read the headers. Therefore, the status code and description will have to be embedded in the output stream, along with the response text. Additionally, there is no consistent way for the client-side to detect when the

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

request is finished. (`onreadystatechange` is not supported very well by all browsers) This causes us to use a more reliable way to handle the event. We chose to handle it by utilizing the BODY tag's `onload` event. In order to have the browser invoke the event we need to write out a valid HTML document. And since we have a document we might as well embed the response text, status code, and status description in html elements for easy access.

```
'if context passed in then we are using IFRAME Implementation
Dim strContextID As String = m_objPage.Request.Form("ctx")
m_objPage.Response.Write("<html><head></head>" _
   "<body onload=""window.parent.dnn.xmlhttp.requests['" & strContextID & _
   "'].complete(window.parent.dnn.dom.getById('txt', document).value);""><form>")
m_objPage.Response.Write("<input type=""hidden"" id=""" & SCRIPT_CALLBACKSTATUSID & _
   """ value=""" & CInt(Me.StatusCode).ToString & """>")
m_objPage.Response.Write("<input type=""hidden"" id=""" & SCRIPT_CALLBACKSTATUSDESCID & _
   """ value=""" & Me.StatusDesc & """>")
m_objPage.Response.Write("<textarea id=""txt"">")
m_objPage.Response.Write(Response)
m_objPage.Response.Write("</textarea></body></html>")
```

5 - Callback Response Handling

You probably are wondering what the javascript after the `onload` part means.

```
window.parent.dnn.xmlhttp.requests['" & strContextID & "'].complete(window.parent.dnn.dom.getById('txt', document).value);
```

I did not cover how the request object gets created on the client-side for a IFRAMEPost earlier because I thought it would introduce more confusion than help. But now we need to have a look in order to understand the above code.

```
var oReq = new dnn.xmlhttp.XmlHttpRequest(new dnn.xmlhttp.JsXmlHttpRequest());
dnn.xmlhttp.requests[oReq._request.contextId] = oReq;
```

As you can see we instantiate a new `JsXmlHttpRequest` object and then assign it to a shared/static array on the `xmlhttp` namespace object. We use its `contextId` as the key to this array. This is necessary to allow the `onload` script to call into this instance of the object. In the above example we reference the object with `window.parent.dnn.xmlhttp.requests['" & strContextID & "']` and invoke its complete method passing it the value contained in the result's textbox (txt).

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

Inside the `complete` method the status code and status description is obtained through the calling of the `XmlHttpRequeset` object's `getResponseHeader` method. This method for the Native and ActiveX objects simply invokes the underlying request object's `getResponseHeader` method. The same holds true for the `JsXmlHttpRequest` object, but we had to write the implementation. It simply obtains a reference to the IFRAME's document and grabs the textbox's value that matches the header's name.

If the status code is successful (200) then we will invoke the function assigned to the `successFunc` property on the request object.

```
if (sStatusCode == '200')
   this.successFunc(sRes, this.context);
```

Note: We are passing the context contained on the request as the second parameter.

If the status code is anything other than successful we invoke the failure function if it exists.

```
var sStatusDesc = this.getResponseHeader('__DNNCAPISCSDI');
if (this.failureFunc != null)
   this.failureFunc(sStatusCode + ' - ' + sStatusDesc, this.context);
```

Closing Thoughts

During the development of the client callbacks I found myself asking the question, *"Should the developer have to be responsible for packing/serializing multiple arguments on the client, then have to unpack them on the server to process?"* For a short time I was writing a `xmlrpc` namespace to handle this type of serialization, but found myself backing away for two reasons.

1. I wanted to implement an existing technology and keep it as close to the ASP.NET 2.0 implementation as possible.

2. A protocol such as XMLRPC or SOAP are responsible for Remote Procedure Calls/Accessing Objects, which doesn't quite fit into what the ASP.NET callback is about. We are not creating and invoking any object on the server. Rather we are invoking a particular method on the server with a defined contract, thus reflection is not necessary.

Those two reasons are not going to prohibit me from implementing a more elegant solution to tackle the serialization problem, but I feel that anything added needs to

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

supplement this “simple” implementation. Hence the `CallBackType=Simple` in the `ClientAPICallBackResponse` object.

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

Additional Information

The DotNetNuke Portal Application Framework is constantly being revised and improved. To ensure that you have the most recent version of the software and this document, please visit the DotNetNuke website at:

http://www.dotnetnuke.com

The following additional websites provide helpful information about technologies and concepts related to DotNetNuke:

DotNetNuke Community Forums
http://www.dotnetnuke.com/tabid/795/Default.aspx

Microsoft® ASP.Net
http://www.asp.net

Open Source
http://www.opensource.org/

W3C Cascading Style Sheets, level 1
http://www.w3.org/TR/CSS1

Errors and Omissions

If you discover any errors or omissions in this document, please email marketing@dotnetnuke.com. Please provide the title of the document, the page number of the error and the corrected content along with any additional information that will help us in correcting the error.

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

Appendix A: Document History

Version	Last Update	Author(s)	Changes
1.0.0	Aug 16, 2005	Shaun Walker	• Applied new template
1.1.0	March 17, 2006	Jon Henning	• Updated for v1.2 of ClientAPI, added documentation for posting form elements

Copyright © 2003-2005 Perpetual Motion Interactive Systems, Inc. All Rights Reserved.