Developing Web Controls With The Client API

Developers Guide To Developing Custom WebControls Utilizing The DotNetNuke Client API

Jon Henning

Revision 1.0.0
Last Updated: March 22, 2006
Applies to: DotNetNuke v. 3.3/4.1

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 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.

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.

Abstract

The purpose of this guide to assist the developer in extending DotNetNuke (DNN) by developing webcontrols that can be utilized inside the DotNetNuke Core, DotNetNuke modules, or any ASP.NET application.

Contents

Chapter 1: Getting Started

Introduction

The purpose of this guide to assist the developer in the development of webcontrols that can be used in any web application, including DotNetNuke.

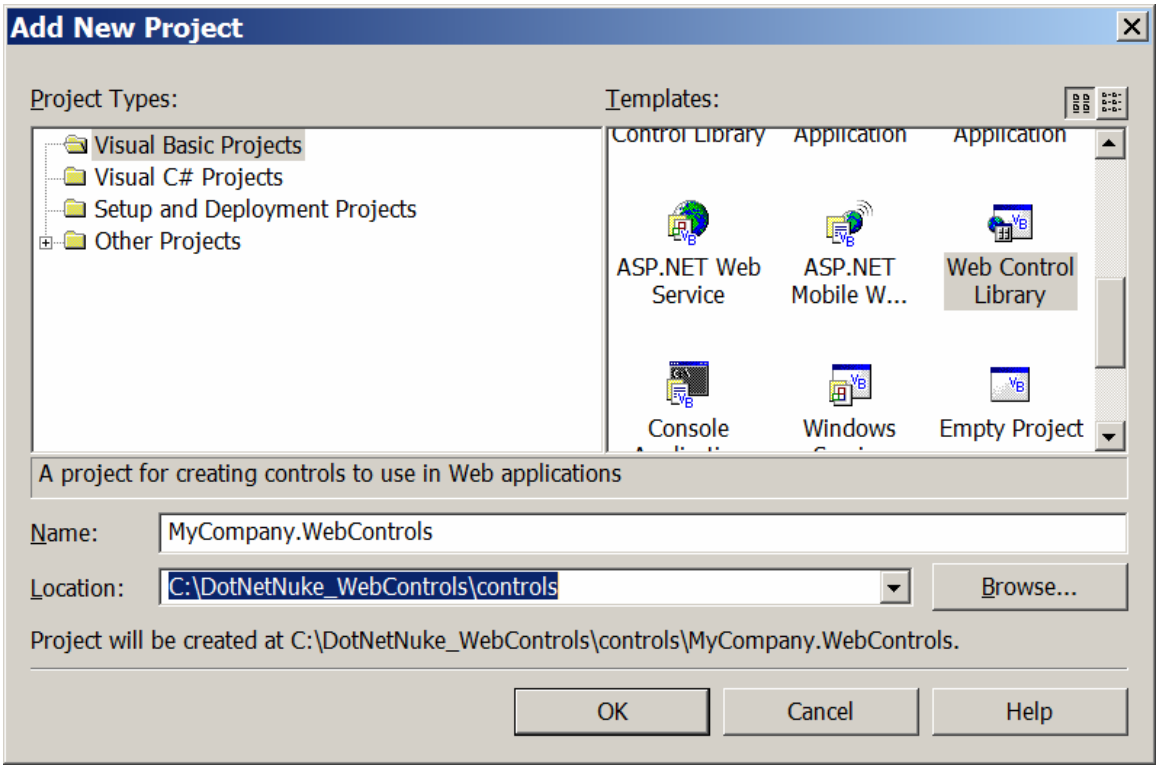
In this guide we will cover every stage of developing webcontrols, from creating your Visual Studio.NET project, to creating a control that "plays nice" with the design-time environment. The goal of this guide is to get you started developing webcontrols in a short amount of time. To accomplish this we will discuss how some of the currently existing webcontrols are written, giving code-snippets where applicable. Additionally, we will walk through the creation of a new control completely outside the DotNetNuke WebControls assembly.

Throughout this guide we will refer to actual examples that are included as parts of the main DotNetNuke WebControl distribution file that can be downloaded from http://www.dotnetnuke.com. The examples we will focus on for the majority of the guide will be the DNNTextSuggest control. We will also pull code from other controls included within the DotNetNuke WebControls, this is so all code listed within this guide is easily accessible to you.

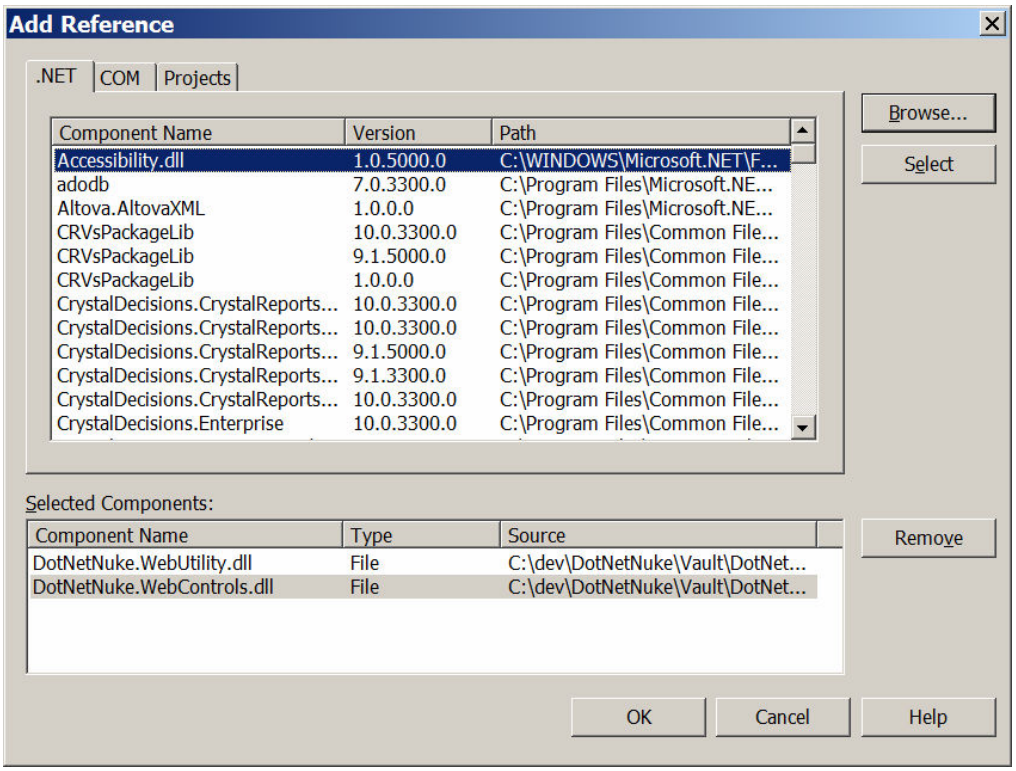
Chapter 2: Setting Up A WebControl Project

Creating A WebControl Library

To get started, open up Visual Studio.NET and Add A New Web Control Library Project.



Next you will need to add a reference to the DotNetNuke.WebUtility.dll and possibly the DotNetNuke.WebControls.dll if you plan on utilizing the DNNNode / DNNNodeCollection objects.



This is all that is necessary to create a separate webcontrol library. From this point on, we will be talking about the code as if it were in the main webcontrol library, but you could have just as easily substituted the code from that library into your own.

Chapter 3: Creating The WebControl Class

Choosing Your Base Class

Depending on the control you are developing you may simply wish to extend an existing control's functionality. This is the case with both the DNNTextSuggest, which inherits from TextBox and DNNLabelEdit, which inherits from Label. There are cases where you develop a control that has no pre-existing control you wish to extend, or you don't like a lot of aspects of a control and do not want to extend it. Instead, you want to write everything from scratch. In this case you typically will inherit from the .NET Framework's WebControl class. Examples of these include the DNNTree and DNNMenu controls.

One thing that you always need to keep in mind when making design decisions for your controls is that you must account for the scenario that a particular browser may not support your functionality, and therefore you will need to have an alternate mode of rendering. This is probably easier to understand with a concrete example. Lets take the DNNTextSuggest, which inherits from the TextBox control. Our assumption has to be that Microsoft knew what they were doing when creating the TextBox control and no matter what browser is viewing this control it functions in a similar manner. When we are developing our own controls we must keep to this principal. In the DNNTextSuggest control's case a "downlevel" browser will simply render the TextBox to the client, which in turn allows us access to whatever the client input through its properties (.Text in this case). While the "uplevel" browsers GUI experience is different (i.e. as the user types a list of suggested text choices is presented), the underlying result for the developer programming against our control is the same (i.e. he gets his results through the .Text property). Hopefully this simplistic view of the principal I am encouraging you to follow is helpful. If not, keep reading for I will be discussing more complex scenarios throughout this document.

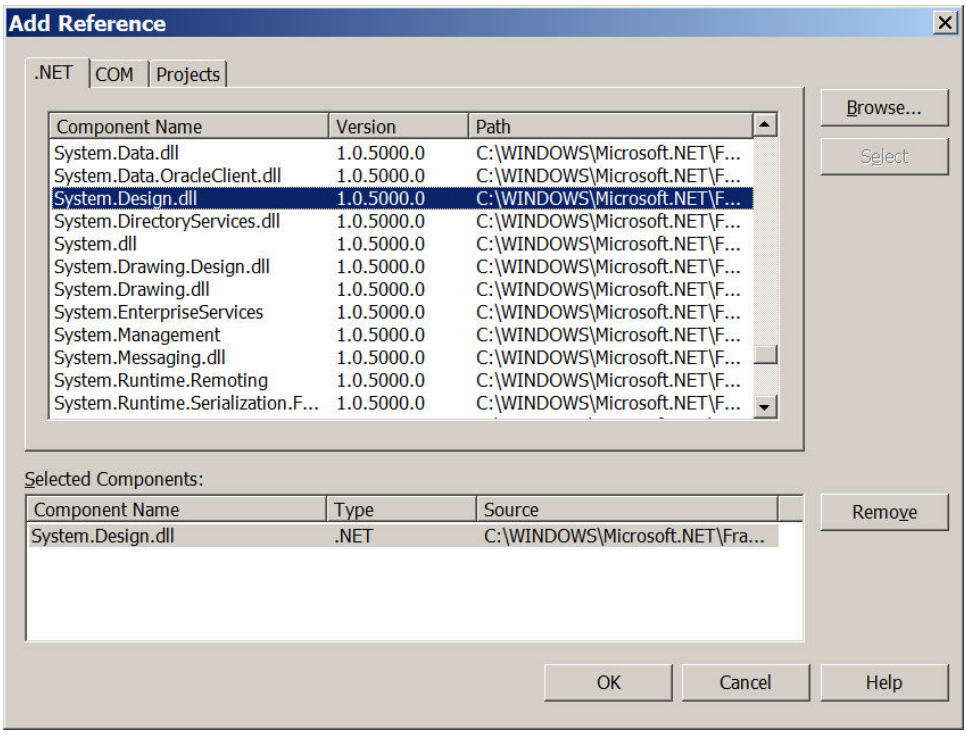
Class Declaration

Now that we have determined what base class to use we are ready to create our class.

```
<Designer(GetType(Design.WebControls.DNNTextSuggestDesigner)), _
  ToolboxData("<{0}:DNNTextSuggest runat=server></{0}:DNNTextSuggest>")> _
  Public Class DNNTextSuggest
      Inherits TextBox
      Implements IPostBackEventHandler
      Implements IClientAPICallbackEventHandler
```

Notice that we chose some attributes that will determine how the aspx/ascx designer will render our control. The Designer attribute points to another class that will provide a design-time view of our control, whereas the ToolboxData specifies the default tag to use when placing the control on the page. Creating a nice design time look to your control goes beyond the scope of this document, there are many articles out there that cover this, including one that I wrote for asp.netPRO magazine back in March 2003.

Note: In order to support the Designer you need to Import the System.Design namespace (Add Reference).



You should also note that we implement a few interfaces here to handle both PostBacks and Callbacks into the control. For more information on Callbacks see the DotNetNuke Client API Client Callback document.

Events

Depending on what type of control you are writing, you may need to expose one or more events. Two common events exposed by the DotNetNuke WebControls are raised when a PostBack occurs and when a Client CallBack is invoked. The DNNTextSuggest control exposes two such events.

```
Public Delegate Sub DNNTextSuggestEventHandler(ByVal source As Object, _
  ByVal e As DNNTextSuggestEventArgs)
Public Delegate Sub DNNDNNNodeClickHandler(ByVal source As Object, ByVal e As DNNTextSuggestEventArgs)

Public Event NodeClick As DNNDNNNodeClickHandler
Public Event PopulateOnDemand As DNNTextSuggestEventHandler
```

Properties

Defining Properties is just like any other class, only instead of persisting the values to a member variable we are utilizing ViewState to remember our settings between page requests. Personally, I don't like using ViewState at all and would rather make the developer who programmatically assigns the properties to have to re-assign them with each request, but this pattern of doing things was not started with the first DotNetNuke webcontrol and I did not deviate from it.

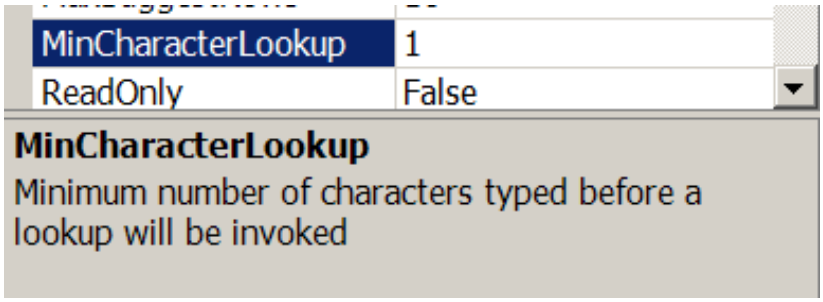
```
Public Property MinCharacterLookup() As Integer
   Get
      If Len(ViewState("MinCharacterLookup")) = 0 Then
         Return 1
      Else
         Return (CType(ViewState("MinCharacterLookup"), Integer))
      End If
   End Get
   Set(ByVal Value As Integer)
      ViewState("MinCharacterLookup") = Value
   End Set
End Property
```

Additionally you are recommended to decorate your properties with attributes that make sense.

```
<DefaultValue(1), Category("Behavior"), _
  Description("Minimum number of characters typed before a lookup will be invoked")> _
      Public Property MinCharacterLookup() As Integer
```

In this case we are specifying that the DefaultValue of the property is 1, the property should be grouped under the Behavior category, and the Description to show in the property browser is specified. Of these properties the most important is to specify a DefaultValue, for this will determine when the property will be persisted to the aspx/ascx page, and in turn when it gets stored in ViewState.

Here is how the property will look in the Property Browser, notice the Description.



Overridden Methods

AddAttributesToRender

Probably the most important method in your entire server-side control development is AddAttributesToRender. This overridden method is responsible for serializing the settings that were set against your control to the client.

```
Protected Overrides Sub AddAttributesToRender(ByVal writer As Web.UI.HtmlTextWriter)
   MyBase.AddAttributesToRender(writer)

   If MinCharacterLookup > 1 Then writer.AddAttribute("minchar", MinCharacterLookup)
```

As you can see we are sending down the properties on our control as an attribute on our HTML tag. These properties are typically abbreviated since this will decrease our overall payload. The reading of these attributes into our client-side control will be covered later in this document.

Note: I am a big supporter of handling the rendering of HTML on the client instead of the server. I cover my reasoning in depth in the DotNetNuke Navigation WebControls document.

OnInit

Page 5 of the DotNetNuke Client API document discusses the ability to pass information back and forth from the client and server through the use of getVar/setVar. The Client API accomplishes this with a known HIDDEN HTML control named __dnnVariable. If you plan on using your control outside of DotNetNuke and your control relies on the usage of GetClientVariable/RegisterClientVariable you must make sure that this control gets registered. This is not as easy as it sounds since we need to guarantee that it gets added to a control that will not prefix its name and ASP.NET does not allow us to add a control to a parent control's collection during the rendering of a control. If you have control over the pages the webcontrol will be added, I suggest simply adding this control to the page.

```
<input id="__dnnVariable" type="hidden" name="__dnnVariable" runat="server">
```

However I realize this is not practical for most control developers, therefore I have come up with a way to successfully insert the control to the page, though I consider it a bit of a hack.

```
Protected Overloads Overrides Sub OnInit(ByVal e As EventArgs)
   MyBase.OnInit(e)
   If ClientAPI.NeedsDNNVariable(Me) Then
      'This is to allow us to add a control to our parent control collection-kindof hack
      AddHandler Me.Page.Load, AddressOf ParentOnLoad
   End If
End Sub

Protected Sub ParentOnLoad(ByVal Sender As Object, ByVal e As System.EventArgs)
   ClientAPI.RegisterDNNVariableControl(Me)
…
```

Currently both the DNNMenu and DNNTree utilize this logic.

Event Handlers

PreRender

This event is typically used to register our client-side scripts and register our postback handler.

```
Private Sub DNNTextSuggest_PreRender(ByVal sender As Object, ByVal e As EventArgs) _
  Handles MyBase.PreRender
    RegisterClientScript()
    Page.RegisterRequiresPostBack(Me)
End Sub

Public Sub RegisterClientScript()
  If IsDownLevel = False Then
```

```
    ClientAPI.RegisterClientReference(Me.Page, ClientAPI.ClientNamespaceReferences.dnn_dom)
    ClientAPI.RegisterClientReference(Me.Page, ClientAPI.ClientNamespaceReferences.dnn_xml)
    ClientAPI.RegisterClientReference(Me.Page, ClientAPI.ClientNamespaceReferences.dnn_dom_positioning)
    ClientAPI.RegisterClientReference(Me.Page, ClientAPI.ClientNamespaceReferences.dnn_xmlhttp)

    If Not ClientAPI.IsClientScriptBlockRegistered(Me.Page, "dnn.controls.DNNTextSuggest.js") Then
       ClientAPI.RegisterClientScriptBlock(Me.Page, "dnn.controls.dnnTextSuggest.js",
         "<script src=""" & TextSuggestScriptPath & "dnn.controls.dnnTextSuggest.js""></script>")
    End If
    ClientAPI.RegisterStartUpScript(Page, Me.ClientID & "_startup", _
         "<script>dnn.controls.initTextSuggest(dnn.dom.getById('" & Me.ClientID & "'));</script>")
  End If
End Sub
```

In the case of the DNNTextSuggest control we will be utilizing the dnn, dnn.dom, dnn.xml, dnn.dom.positioning, and dnn.xmlhttp namespaces on the client. In addition to these, we are also registering our own script file (dnn.controls.dnntextsuggest) and registering the function to call on startup (initTextSuggest).

Handling PostBacks - RaisePostBackEvent

Whenever a control needs to support a client-side event that causes a post back, as is the case with the DNNMenu, DNNTree, and DNNTextSuggest when a node is clicked with its ClickAction = PostBack, ASP.NET needs to be notified that the control will subscribe to its postback handling. This is accomplished through the invocation of the RegisterRequiresPostback method (typically in PreRender). In addition to this method the control must implement the IPostBackEventHandler in the class declaration (noted above) and implement that interface's sole method RaisePostBackEvent.

```
Public Overridable Sub RaisePostBackEvent(ByVal eventArgument As String) _
  Implements IPostBackEventHandler.RaisePostBackEvent
    Dim args() As String = Split(eventArgument, ClientAPI.COLUMN_DELIMITER)

    If args.Length > 1 Then
      Select Case args(1)
        Case "Click"
           Dim oArg As DNNTextSuggestEventArgs = New DNNTextSuggestEventArgs(Me.DNNNodes, args(0))
           OnNodeClick(oArg)
      End Select
    End If
End Sub
Public Overridable Sub OnNodeClick(ByVal e As DNNTextSuggestEventArgs)
    RaiseEvent NodeClick(Me, e)
End Sub
```

Handling CallBacks - RaiseClientAPICallbackEvent

Registering for callback is covered in depth in the Client API Client Callback document, so I will not go into many details here. What I will cover is how we let the client-side control know how to invoke a callback. In the AddAttributesToRender overload we first verify that the client browser supports a callback, then just like any other attribute, we send down the script necessary to invoke the callback.

```
If ClientAPI.BrowserSupportsFunctionality(ClientAPI.ClientFunctionality.XMLHTTP) Then
    writer.AddAttribute("callback", ClientAPI.GetCallbackEventReference(Me, "'[TEXT]'",
      "this.callBackSuccess", "this", "this.callBackFail", "this.callBackStatus"))
End If
```

You should note the use of the token [TEXT] as the argument. This allows us to perform some client-side logic on the actual value before we replace the token out with the actual text. If no logic is needed, it is recommended to not use a token and use the actual value instead. For the DNNTextSuggest this could be this.getText().

The event handler for the callback is pretty simple, though quite intriguing if you look closely.

```
Public Function RaiseClientAPICallbackEvent(ByVal eventArgument As String) As String _
  Implements IClientAPICallbackEventHandler.RaiseClientAPICallbackEvent
    RaiseEvent PopulateOnDemand(Me, New DNNTextSuggestEventArgs(Me.DNNNodes, eventArgument))
    Return Me.DNNNodes.ToXml
End Function
```

We simply raise an event passing in the appropriate argument. It is assumed that the code that is handling the event will populate the control's DNNNode Collection inside it, and we simply return the Xml from that collection to the client.

Final Note For Server Side

One final note before we shift our focus towards the client-side script. For the controls that support Populate On Demand it is worth noting how they gracefully handle browsers that support a callback and ones that only support a postback. For the developer using your control both types of browsers will be handled the same. That is to say that the only event that he/she needs to respond to is the PopulateOnDemand event. Inside that event the current Node's NodeCollection is populated accordingly.

Chapter 4: Creating The WebControl Script

Fear Of Javascript

It continues to amaze me that there is such fear (for lack of a better word) in the developer community when it comes writing javascript. In fact, there are even some creative ways in which product developers are offering to completely avoid its use and still offer a rich client-side experience. Personally, I only understand part of the fear. That part is the problems associated with writing code that works great on one browser and doesn't function at all on another. The goal of the ClientAPI (along with many other javascript frameworks such as Atlas and prototype) is to take this problem out of the equation. It is my hope that this guide and the other documentation of the ClientAPI I am providing will assist you in getting comfortable with javascript and see just what is possible when used.

Javascript Objects

As noted in the DotNetNuke Client API document, javascript is actually a powerful language with support for objects and methods. In a scenario where you allow for more than a single control on a page, the ability to encapsulate your control's specific information and behavior inside an object is very useful. To start out, lets look at two things each control we develop will require.

Initializing Our Control

```
dnn_control.prototype.initTextSuggest = function (oCtl)
{
   dnn.controls.controls[oCtl.id] = new dnn.controls.DNNTextSuggest(oCtl);
   return dnn.controls.controls[oCtl.id];
}
```

This script is responsible for creating our control object. You should note that the calling of the script was registered as a startup script in the PreRender event on the server. The control that is passed in as an argument is the same control we rendered on the server, as referenced by dnn.dom.getById('" & Me.ClientID & "')

Constructor

```
//------- Constructor -------//
dnn_control.prototype.DNNTextSuggest = function (o)
{
   this.ns = o.id;               //stores namespace for menu
   this.container = o;                    //stores container
   //--- Appearance Properties ---//
   this.css = dnn.dom.getAttr(o, 'css', '');
…
   this.minChar = new Number(dnn.dom.getAttr(o, 'minChar', '1'));

   this.postBack = dnn.dom.getAttr(o, 'postback', '');
   this.callBack = dnn.dom.getAttr(o, 'callback', '');
…
   dnn.dom.addSafeHandler(o, 'onkeyup', this, 'keyUp');
…
```

In javascript, a constructor is nothing more than a function. The unique thing about the function is it utilizes the *this* keyword. This allows us to define properties of our control. The first property (ns) we store is very important in cases where we are going to be creating new controls on the client side, for we need to guarantee that the new controls get assigned a unique id.

The next property assigned is a reference to the HTML container for our control. Typically your controls will have many properties that will affect the way they render (like css) or the way the function (minChar). Since we are always conscious about our payload being sent back and forth, the values are only sent when they are not the default. This default value assignment is handled with the getAttr method exposed by the dnn namespace.

The constructor is also the place that we typically register the events that our object cares about. We accomplish this by associating an event exposed by our container (o.onkeyup) with method defined on our object (this.keyUp).

Before we move onto the declaring of our methods I wanted to note that the script for both our postBack and callBack are treated just like any other property assignment. Later I will show you how we invoke them.

Method Declaration

I want to take a quick moment here to let you know that I plan on reorganizing how I am declaring the methods that a custom object exposes. This has recently been done with the DNNTextSuggest contol, but will eventually make its way into all of the ClientAPI script files.

Traditionally I would define my methods like this

```
dnn_control.prototype.DNNTextSuggest.prototype.keyPress = function (e)
{
}

dnn_control.prototype.DNNTextSuggest.prototype.keyUp = function (e)
{
}
```

I have recently grown to favor this way of defining the same methods.

```
dnn control.prototype.DNNTextSuggest.prototype =
{
keyPress: function (e, element)
{
},

keyUp: function (e, element)
{
}
```

I believe the end result is more readable code, not to mention a slightly smaller script file.

Event Handlers

Lets start out by taking a look at one of the DNNTextSuggest's event handler methods

```
keyUp: function (e, element)
{
    dnn.cancelDelay(this.ns + 'kd');
    dnn.doDelay(this.ns + 'kd', this.lookupDelay, dnn.dom.getObjMethRef(this, 'doLookup'));

    this.prevText = this.container.value;
    if (e.keyCode == KEY_UP_ARROW)
        this.setNodeIndex(this.selIndex - 1);
    else if(e.keyCode == KEY_DOWN_ARROW)
        this.setNodeIndex(this.selIndex + 1);
    else if(e.keyCode == KEY_ESCAPE)
        this.clear();
…
```

The first thing to notice is that we are passed in the event object (e) for our method (handled automatically by the addSafeHandler method).

The first line of code above utilizing the ClientAPI's delay methods to cancel an existing delay if it already exists, then create a new delayed function waiting however long our lookupDelay property specifies before calling the doLookup method. From the user's perspective this is how many milliseconds we will wait after they type in a character before we attempt to perform a client callback.

It is worth noting that we still have access to the *this* object. This may not seem like much, but it is only possible through something cool exposed by javascript called closures.

Warning! Tangent: Since we do not want to perform callbacks with every keystroke the user types, we are storing information like the previous text in order to determine if a callback is necessary. For example, the user typed in 'Smi' and is shown a list of ('Smith', 'SmithA', 'Smiz') as a result of a callback. When the user types in the next letter ('Smit') we know that a callback is not necessary since we have at least one item in our list that is greater than it ('Smiz').

The last series of statements are dealing with the various keystrokes the user may use to cause different affects to happen to the list shown.

Methods

Lets analyze the code necessary to allow it to perform a callback. Before we look at the client side lets recall how we registered it on the server.

```
If ClientAPI.BrowserSupportsFunctionality(ClientAPI.ClientFunctionality.XMLHTTP) Then
    writer.AddAttribute("callback", ClientAPI.GetCallbackEventReference(Me, "'[TEXT]'", _
      "this.callBackSuccess", "this", "this.callBackFail", "this.callBackStatus"))
End If
```

Take note that we are referencing the *this* keyword when we are assigning our callback event functions.

```
doLookup: function ()
{
    if (this.getText().length >= this.minChar)
    {
        if (this.needsLookup())
        {
            this.prevLookupOffset = this.getTextOffset();
            this.prevLookupText = this.formatText(this.getText());
            eval(this.callBack.replace('[TEXT]', this.prevLookupText));
        }
        else
            this.renderResults(null);
    }
    else
        this.clearResults();
},
callBackSuccess: function (result, ctx)
{
    var oText = ctx;
    if (oText.callBackStatFunc != null && oText.callBackStatFunc.length > 0)
    {
        var oPointerFunc = eval(oText.callBackStatFunc);
        oPointerFunc(result, ctx);
    }
    oText.renderResults(result);

},

callBackFail: function (result, ctx)
{
    alert(result);
},
```

The doLookup method starts out verifying that the user typed enough characters to perform a lookup. It then calls another function to determine if the text entered even needs a callback (see Tangent above). Once we know we need to perform a callback we

are assigning a couple properties to the current state of the control. Finally we use the eval javascript function to evaluate our script, replacing out the [TEXT] token with our actual text. In case you remember me discussing the potential "client-side logic" we are doing on the client before sending it to the server (RaiseClientAPICallbackEvent), it is shown here. It has to do with the calling of the formatText method, which is used to support the caseSensitivity property.

If we determined we did not need a callback, we simply call renderResults, just as our callBackSuccess function does.

Client Side Rendering

As I have stated in the DotNetNuke Navigation Controls document, I am in favor of sending only the data necessary to create the controls to the client, instead of sending the data plus the markup. I list my reasons for supporting this approach in that document. Let us now investigate how this client-side rendering typically takes place.

```
renderResults: function (sXML)
{
    if (sXML != null)
    {
        this.DOM = new dnn.xml.createDocument();
        this.DOM.loadXml(sXML);
    }
    this.rootNode = this.DOM.rootNode();
    if (this.rootNode != null)
    {
        if (this.resultCtr == null)
            this.renderContainer();

        this.clearResults();
        for (var i=0; i<this.rootNode.childNodeCount(); i++)
            this.renderNode(this.rootNode.childNodes(i), this.resultCtr);
    }
}
```

Depending on the control, an xml representation of the data is given and loaded into the cross-browser wrapper responsible for parsing the xml (dnn.xml.createDocument). Due to the fact that the XML string may contain data that is not a node (i.e. a comment) a generic method to obtain the root node is used (rootNode) and associated to the control's rootNode property.

If this node is not null and we haven't created a container control to hold our items already we will call the renderContainer method. This element is responsible for holding our list of items.

```
renderContainer: function ()
{
    this.resultCtr = document.createElement('DIV');
    this.container.parentNode.appendChild(this.resultCtr);
```

```
    this.resultCtr.className = this.tscss;
    this.resultCtr.style.position = 'absolute';
    this.positionMenu();
}
```

In this case we are simply using the underlying browser API to create our DIV element and appending it to our controls parent's node collection (i.e. make it a sibling of our textbox already on the page). If we had a need to refer to this element by its ID then we would have needed to be careful in its naming, as you will see when we discuss the rendering of the nodes.

The clearing of any previous results are handled by the clearResults method.

```
clearResults: function ()
{
    if (this.resultCtr != null)
        this.resultCtr.innerHTML = '';
    this.selIndex = -1;
    this.selNode = null;

}
```

Finally the rootNode's collection is looped and a call is made to render each item inside the resultCtr (via the renderNode method).

```
renderNode: function (oNode, oCont)
{
    var oTSNode;
    oTSNode = new dnn.controls.DNNTextSuggestNode(oNode);
    //text must be prefixed by value we are looking for and we must be under the maxRows
    if (this.formatText(oTSNode.text).indexOf(this.formatText(this.getText())) == 0
        && oCont.childNodes.length < this.maxRows)
    {
        var oNewContainer = this.createChildControl('DIV', oTSNode.id, 'ctr'); //container for Node
        oNewContainer.appendChild(this.renderText(oTSNode)); //render text

        if (oTSNode.enabled)
        {
            dnn.dom.addSafeHandler(oNewContainer, 'onclick', this, 'nodeClick');
            dnn.dom.addSafeHandler(oNewContainer, 'onmouseover', this, 'nodeMOver');
            dnn.dom.addSafeHandler(oNewContainer, 'onmouseout', this, 'nodeMOut');
        }

        if (oTSNode.toolTip.length > 0)
            oNewContainer.title = oTSNode.toolTip;

        oCont.appendChild(oNewContainer);
        this.assignCss(oTSNode);
    }
}
```

The first thing to note is that we are taking the passed in xml node and creating a new DNNTextSuggestNode object based off of its values by passing it into our constructor. I will go into more detail about how the DNNTextSuggestNode is coded later. For now, just note that by doing this, we are able to access the node's properties in an easy manner (i.e. oTSNode.text, oTSNode.enabled).

The next check is there to determine if the node should be rendered at all. I mentioned earlier in this document that we only want to do a lookup if necessary. In the cases we determined a lookup was not necessary we will have nodes in our collection that are present, but should not be rendered.

Once we decided that the node should be rendered, we will create its one or more controls. Instead of calling createElement directly, like we did for the resultCtr, we are using a custom method of this control. This is necessary since we will be needing to reference this control by its ID in other portions of code.

```
createChildControl: function (sTag, sNodeID, sPrefix)
{
    var oCtl = document.createElement(sTag);
    oCtl.ns = this.ns;
    oCtl.nodeid = sNodeID;
    oCtl.id = this.ns + sPrefix + sNodeID; //__dm_getControlID(oCtl.ns, oCtl.nodeid, sPrefix);
    return oCtl;
}
```

In order to guarantee the uniqueness of the IDs we assign, the method pulls the main controls namespace (ns) and prefixes it along with its type (i.e. ctr, txt) and nodeid. Additionally, its properties are also added as a custom attribute to the control. We will look at one of the event handlers that are associated to the control to understand why.

This is how event handlers are associated to methods within our custom object.

```
dnn.dom.addSafeHandler(oNewContainer, 'onmouseover', this, 'nodeMOver');
```

When the mouse moves over our container (DIV tag) the nodeMOver method for our *specific instance* of the object is invoked.

```
nodeMOver: function(evt, element)
{
    var oNode = this.DOM.findNode('n', 'id', element.nodeid);
    if (oNode != null)
    {
        var oTSNode = new dnn.controls.DNNTextSuggestNode(oNode);
        oTSNode.hover = true;
        this.assignCss(oTSNode);
    }
}
```

The event handler is passed the element that invoked the event as its second parameter. We will use the custom property we defined (nodeid) to locate the xml node in our DOM. Assuming we find it, we will set the hover property and call a function responsible for assigning any changes in css.

Note: For controls like the DNNTree there are times when we will respond to an event (expand for example), assign a property (expanded = true) and want to update the underlying xml. The underlying DNNNode object that the DNNTreeNode inherits from contains an update method which will update the DOM's underlying xml and

eventually update the variable holding the xml for the control (setVar) so that if a postback occurs, the expanded state of the control is not lost.

Object Inheritance and DNNNode

The DotNetNuke Navigation Controls document discusses the need for a generic node collection and object to be utilized across controls (DNNNodeCollection and DNNNode). This collection must be easily serialized to the client. It should be obvious that if the server-side code serializes the object, we will need client-side code to deserialize it. Just like the server-side counterpart the actual object will just be a wrapper for the underlying XmlNode object.

```
dnn_controls.prototype.DNNNode = function (oNode)
{
    if (oNode != null)
    {
        this.node = oNode;
        this.id = oNode.getAttribute('id', '');
        this.key = oNode.getAttribute('key', '');
        this.text = oNode.getAttribute('txt', '');
        this.url = oNode.getAttribute('url', '');
        this.js = oNode.getAttribute('js', '');
        this.target = oNode.getAttribute('tar', '');
        this.toolTip = oNode.getAttribute('tTip', '');
        this.enabled = oNode.getAttribute('enabled', '1') != '0';
        this.css = oNode.getAttribute('css', '');
        this.cssSel = oNode.getAttribute('cssSel', '');
        this.cssHover = oNode.getAttribute('cssHover', '');
        this.cssIcon = oNode.getAttribute('cssIcon', '');
        this.hasNodes = oNode.childNodeCount() > 0;
    }
}
```

Also, the object should support some methods that allow its children to be exposed.

```
dnn_controls.prototype.DNNNode.prototype =
{
childNodeCount: function ()
{
    return this.node.childNodes.length;
}
…
```

Just like the server-side object, each control will have its own specific node object associated to it. And instead of defining all of this basic functionality within each control's script, a generic DNNNode object is defined in the dnn.controls namespace. This object is then inherited from in each control's script.

Javascript supports object inheritance through this syntax. *Note: I removed all the namespace syntax to simplify my point.*

```
function DNNNode()
{
    this.id = '';
}

function DNNTextSuggestNode()
{
    this.hover = false;
}
DNNTextSuggestNode.prototype = new DNNNode;
```

The last line here causes all the defined methods associated to the DNNTextSuggestNode to be replaced by the defined methods of the DNNNode object. Normally, this is fine since our inheritance takes place just after the class declaration. However, in the ClientAPI, thanks to ASP.NET 1.x there is no guarantee on the order the scripts are loaded (see end of document). And since the DNNNode class is defined in a separate file from the DNNTextSuggestNode we cannot write the code directly after the declaration.

Due to this, a new method has been added that instead of overwriting the existing methods, it extends (or overlays) any methods to the existing object. This extending of the object must take place once we know that all script is loaded. Probably the safest location for this code is in the control's init method.

```
dnn_control.prototype.initTextSuggest = function (oCtl)
{
    //Extends is better than inherits cause inherits overwrites any functions we may have
    //defined specific to DNNTextSuggestNode
    dnn.extend(dnn.controls.DNNTextSuggestNode.prototype, new dnn.controls.DNNNode);
…
```

The last piece to the puzzle, before we look at the actual DNNTextSuggestNode declaration has to do with the ability to pass our base class an argument in its constructor. In VB.NET this call would be MyBase.New(SomeArgument). In javascript the way we handle it is by assigning some property (of our choosing) a reference to the object's declaration, then we invoke its constructor through that new property.

```
dnn_control.prototype.DNNTextSuggestNode = function (oNode)
{
    this.base = dnn.controls.DNNNode;
    this.base(oNode);      //invoke base class constructor

    //textsuggest specific attributes
    this.hover = false;
    this.selected = oNode.getAttribute('selected', '0') == '1' ? true : null;
    this.clickAction = oNode.getAttribute('ca', dnn.controls.action.none);
}

//DNNTextSuggestNode specific methods
dnn control.prototype.DNNTextSuggestNode.prototype =
{
childNodes: function (iIndex)
{
    if (this.node.childNodes[iIndex] != null)
        return new dnn.controls.DNNTextSuggestNode(this.node.childNodes[iIndex]);
}
}
```

Take note that we could have called the property anything. For consistency I recommend calling it *base* though. As you can see the DNNTextSuggestNode adds three new properties along with one new method (childNodes).

If you wish to learn more about javascript and how it handles objects, I suggest searching the internet. Here is just one of many articles I found helpful when designing the ClientAPI. http://www.cs.rit.edu/~atk/JavaScript/manuals/jsobj/

Mandatory Script To Handle Random Script Order

As noted in the DotNetNuke Client API document ASP.NET 1.x cannot guarantee the order the scripts will be sent down to the client. It is therefore necessary to include the following script at the beginning of your control's script file.

```
//BEGIN [Needed in case scripts load out of order]
if (typeof(dnn_control) == 'undefined')
   eval('function dnn_control() {}')
//END [Needed in case scripts load out of order]
```

Additionally, you will need this script at the end of your control's script file.

```
//BEGIN [Needed in case scripts load out of order]
if (typeof(dnn_controls) != 'undefined')
{
   dnn.extend(dnn_controls.prototype, dnn_control.prototype);
   dnn.controls = new dnn controls();
}
//END [Needed in case scripts load out of order]
```

I believe going into detail about what this script does along with how the ClientAPI handles the loading of namespaces is outside the scope of this document. Especially, since once we move to the ASP.NET 2.0 platform we will be able to completely remove the namespace loading script from all files.

Final Thoughts

It is my hope that you found this document helpful and possibly inspires you to go out and create your own controls. If you wish to stay up to date on the latest enhancements to the ClientAPI or DotNetNuke WebControls I suggest visiting their respective project pages found on www.dotnetnuke.com. Feel free to post in the forums with any questions you may have.

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.

Appendix A: Document History

Version	Last Update	Author(s)	Changes
1.0.0	March 7, 2006	Jon Henning	• Creation