DotNetNuke Client API

Jon Henning

Version 1.0.0
Last Updated: June 20, 2006
Category: Client API

DotNetNuke 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.

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

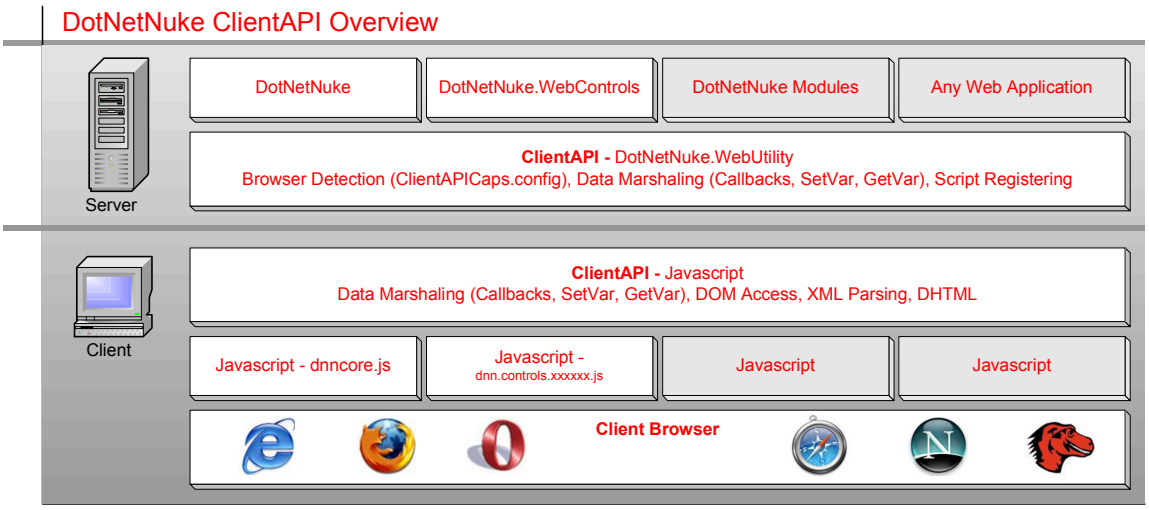
In order to clarify the intellectual property license granted with contributions of software from any person or entity (the "Contributor"), Perpetual Motion Interactive Systems Inc. must have a Contributor License Agreement on file that has been signed by the Contributor.

Contents

DotNetNuke Client API Guide

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 following diagram outlines the current consumers utilizing the DotNetNuke ClientAPI, along with the intended consumers of this functionality in the future (grayed out).



Goals

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

Allow for functionality to easily be turned off and resort to the less-efficient way of responding to client-side events- post-backs.

Provide a uniform, cross-browser API that the developer can program against to provide a rich UI. This includes but is not limited to accessing the Document Object Model (DOM), Dynamic HTML (DHTML), eXtensible Markup Language (XML).

Allow for the API to be enhanced and extended by both the DNN Core Team members and Third Party developers.

Provide ability to easily create Unit Tests to assist in development, testing, and troubleshooting of the API. The tests should be easy to copy to any platform (windows, linux, Mac, etc.) and run, providing output that can then be sent to the developer for troubleshooting.

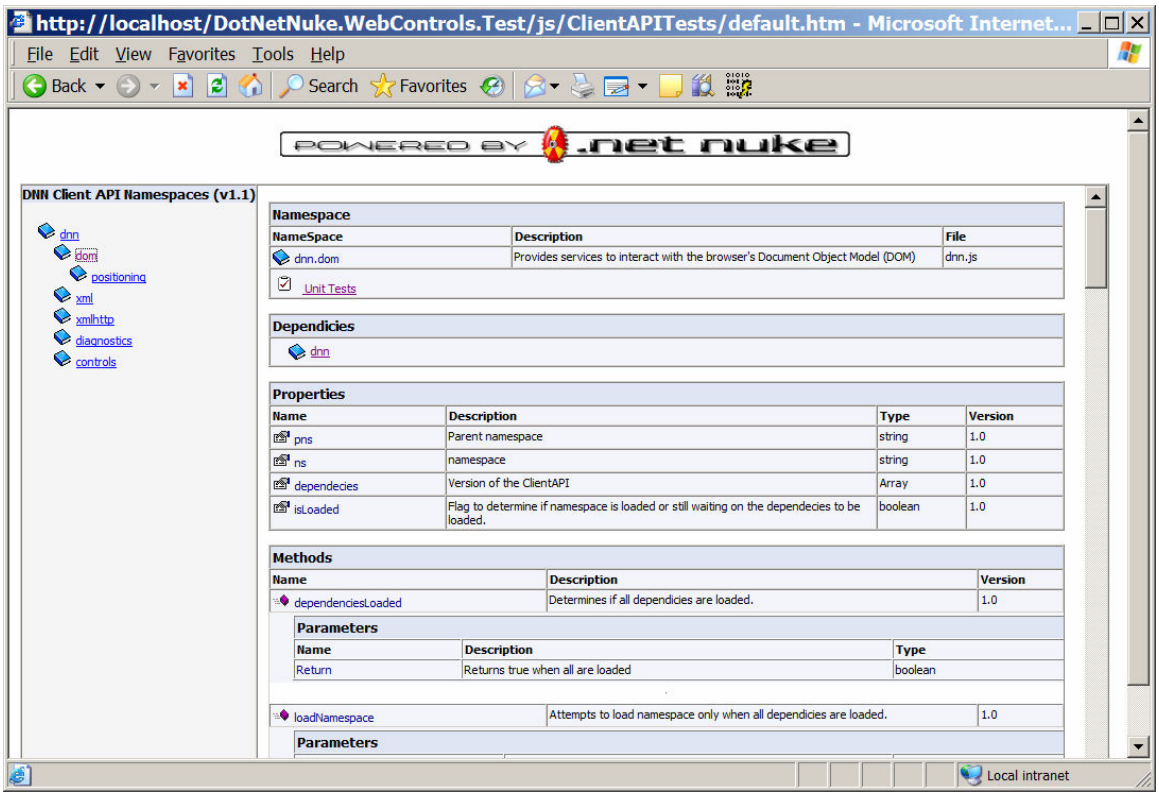
DNN Client API Namespaces and Javascript Objects

It has been decided by members within the core team that the DNN Client API will be written in a .NET Framework like fashion. What this means, is that instead of calling a set of functions in a flat model like dnn_getById() and dnn_getBrowserVersion(), you will be using methods like dnn.dom.getById() and dnn.dom.browser.version. As you can see there are what appears to be namespaces just like in .NET. And in order to access these namespaces a "reference" will need to be added on the server-side to allow the client-side code access to them. This type of functionality is made possible through the creation of our own custom Javascript Objects. The internet is full of examples explaining how to create Javascript Objects, complete with properties, methods, and events (to some extent), so if your new to them I suggest doing some research. Here are some examples.

http://www.webdevelopersjournal.com/articles/jsintro3/js_begin3.html
http://www.sitepoint.com/article/oriented-programming-1

DNN Client API Namespaces - Reference

Instead of maintaining the documentation for the namespace hierarchy in two places, we have decided to document all the methods, objects, etc. in the Unit Test htm pages. Open up the **dnnclientapi.htm** file found in the js\ClientAPITests\ folder to browse the namespace hierarchy.



Registering Namespace Scripts

In order for a namespace to be available for the client to use it has to be registered on the client. Before it is registered, it the developer first needs to check to see if the browser supports its functionality. This is accomplished through the following code.

```
If ClientAPI.BrowserSupportsFunctionality(ClientAPI.ClientFunctionality.DHTML)
```

If this function returns true then the script can be registered.

```
ClientAPI.RegisterClientReference(objPage, ClientAPI.ClientNamespaceReferences.dnn)
```

This will in turn register all namespaces required for the requested namespace.

It should be noted that due to a bug in ASP.NET 1.x the order in which the scripts get registered on the client are not guaranteed. Take the following code for example.

```
Dim i As Integer
For i = 0 To 10
```

```
    Page.RegisterClientScriptBlock(i.ToString, "<script src=""" & i.ToString & ".js" &
"""></script>")
Next
```

You would expect the resulting source on the page to resemble this.

```
    <script src="0.js"></script>
    <script src="1.js"></script>
    <script src="2.js"></script>
    <script src="3.js"></script>
    <script src="4.js"></script>
    <script src="5.js"></script>
    <script src="6.js"></script>
    <script src="7.js"></script>
    <script src="8.js"></script>
    <script src="9.js"></script>
    <script src="10.js"></script>
```

What actually is output is this.

```
    <script src="2.js"></script>
    <script src="3.js"></script>
    <script src="0.js"></script>
    <script src="1.js"></script>
    <script src="6.js"></script>
    <script src="7.js"></script>
    <script src="4.js"></script>
    <script src="5.js"></script>
    <script src="8.js"></script>
    <script src="9.js"></script>
    <script src="10.js"></script>
```

This obviously leads to problems when you have scripts that rely on parent namespaces to already be present. Due to this limitation quite a bit of complexity needed to be added to the client side scripts to make sure the namespaces got loaded in order. If you scan the javascript code you will see a lot of code relating to this (i.e. `loadNamespace, dependenciesLoaded`).

The good news is that Microsoft has fixed this issue in ASP.NET 2.0 by introducing a new class called `ClientScriptManager`. So once we migrate to ASP.NET 2.0, this rather messy client logic can be removed.

Passing Data Between Server and Client

Overview

One of the challenges in developing any web application is to pass information between the server-side code and client-side code. Typically this happens in one of two ways. 1) Setting the value of a hidden form field on the server side and reading the value on the client using the DOM (Document Object Model). 2) Set the javascript variable directly by writing out a string on the server side that does something like

```
var m_sName = '<%=User.Name%>';
```

While both of these methods are acceptable in most applications, a more formalized way is being defined for the DotNetNuke framework. This should allow for consistency across modules, yield cleaner code, and solve a few problems with cross-browser compatibility.

Those of you familiar with the .NET Framework probably heard of ViewState before. ViewState works by storing information in a hidden form field that is sent down to the client in an encrypted format and gets posted back to the server, where it is decrypted and used. While this works fine for what it was intended to be used for (maintaining the state (properties) of the controls on the page), it is not useful at all to the client side code, since it cannot easily be deciphered there.

The DNN ClientAPI offers a way to read and write data that can be used on both the client and server side code. On the server side a variable is "registered" for use via the RegisterClientVariable method.

```
Public Shared Sub RegisterClientVariable(ByVal objPage As Page, ByVal strVar As String,
ByVal strValue As String, ByVal blnOverwrite As Boolean)

objPage     Page object being rendered
strVar      Variable name to set
strValue    Value to set
blnOverwrite   Flag determines if variable is appended or overwritten
```

To read the variable in the GetClientVariable method is used.

```
Public Shared Function GetClientVariable(ByVal objPage As Page, ByVal strVar As String)
As String

objPage     Page object being rendered
strVar      Variable name to get

Similarly, the client can access the variable through the getVar function found in the
dnn namespace.

dnn.getVar(sKey)

sKey - Variable name to get
```

To write the variable use the setVar function also found in the dnn namespace.

```
dnn.setVar(sKey, sVal)

sKey - Variable name to set
sVal - Value to set
```

Where Its Used

An example of where this functionality is used is in the maximize/minimize client script. Since in an UpLevel rendering of a module with the maximize/minimize functionality a round trip to the server is not done and the icon representing the plus and minus sign can be customized, the client will need to know what icons to display when clicked. This information is set on the server side and passed to the client by the following code.

```
ClientAPI.RegisterClientVariable(Page, "min icon " & objPortalModule.ModuleId.ToString,
sModulePath & [MinIcon], True)
```

Then it is accessed on the client side by the following code.

```
sMinIcon = dnn.getVar('min_icon_' + sModuleID);
```

Another example of this functionality in DotNetNuke is found in how it allows for multiple client side functions to subscribe to the body.onload event. When DNN wants to set focus to a control when the body loads it will do so by utilizing the AddBodyOnloadEventHandler method.

```
Public Shared Sub AddBodyOnloadEventHandler(ByVal objPage As Page, ByVal strJSFunction As
String)

objPage     Page object being rendered
strJSFunction  Javascript function to call
```

This method is used in the SetFormFocus method found in the Globals.vb file.

```
ClientAPI.AddBodyOnloadEventHandler(control.Page, "  dnn SetInitialFocus('" &
control.ClientID & "');")
```

Under the covers the AddBodyOnloadEventHandler method is actually calling the following code.

```
ClientAPI.RegisterClientVariable(objPage, "__dnn_pageload", strJSFunction, False)
```

Notice how the last parameter is set to False. This allows for multiple calls to this method and the functions will be executed on the client side in the order they were registered. Note: You should always end your function call with a semi-colon (;).

In case you were wondering, on the client the dnncore.js file syncs the body.onload method and calls any events registered by the following code.

```
function   dnn Page OnLoad()
{
   var sLoadHandlers = dnn.getVar('__dnn_pageload');
   if (sLoadHandlers != null)
      eval(sLoadHandlers);
}
```

Client Side Calls To The Server

Overview

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.

Custom PostBacks

For the v1.0 release of the ClientAPI, there was not enough time to implement any of the above mentioned technologies. However, what it does support is custom

PostBacks which utilize the ASP.NET page's event model. To have your code create a custom PostBack you will need to register the event handler on the server.

```
ClientAPI.RegisterPostBackEventHandler(Me, "MoveToPane", AddressOf
ModuleMoveToPanePostBack, False)
```

The first parameter is the current control being rendered, if there is no control being rendered then simply pass the page object. The second parameter is the event name you wish to register. This name will also be used in the client code. The next parameter is the pointer to the function you wish to be called when the event is raised. Finally, the last Boolean flag is used to determine if multiple event handlers can be specified for this event.

The event handler needs to implement the following interface.

```
Private Sub ModuleMoveToPanePostBack(ByVal args As ClientAPIPostBackEventArgs)
```

The args parameter provides access to the parameters collection through the EventArguments property. For example.

```
args.EventArguments("moduleid")
```

Now that you know how to register the server-side event handler we will discuss the client-side code. The dnn namespace has a method called callPostBack.

```
dnn.callPostBack('MoveToPane', 'moduleid=' + sModuleID, 'pane=' + oPane.paneName,
'order=' + iIndex * 2);
```

It accepts 1 or more parameters. The first parameter is the Event name we have specified for the server. The function then allows for an unlimited number of parameters, the only requirement is that you pass the parameters in as name-value-pairs, delimited by the equal (=) sign. This is to make the server side code will be more readable, since the parameters can be accessed by name instead of offset.

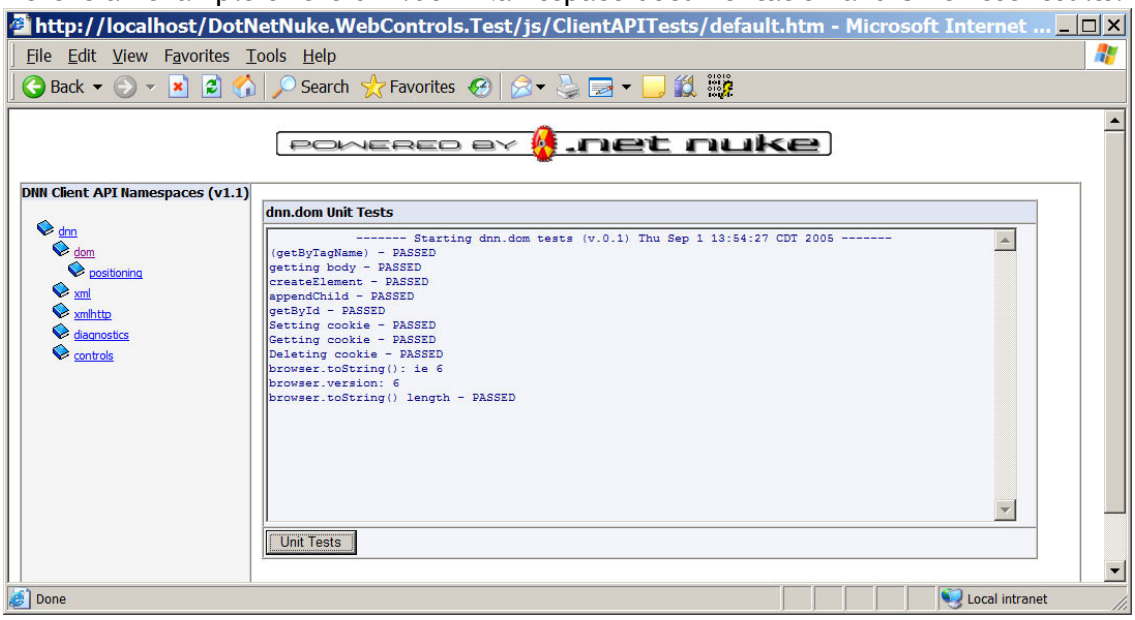
Where Its Used

Currently a custom PostBack is used to notify the server that a Drag-N-Dropped module has been placed in a new location. The server-side code to register the event handler is found in the admin\skins\skin.vb file's InjectModule routine. The client-side code to call the postback is found in the dnncore.js file's __dnn_dragComplete routine.

Unit Tests and Client API Documentation

One of the main, and probably the most important goals of the Client API is cross-browser compatibility. Since it is not enough to just get the API functioning on a single browser and a single platform, we have decided to create a series of self-contained unit test scripts that can be easily copied into any environment and viewed with any browser. These pages will output the results of the tests into a TEXTAREA so they can easily be sent to the interested parties.

Here is an example of the dnn.dom Namespace documentation and Unit Test results.



A Unit Test page is currently a static HTML document that contains a table for the documentation, a TEXTAREA, and a command button for the unit tests. Additionally, it must reference the dnn.diagnostics.js file along with any other script libraries necessary for the tests, in this case that would include dnn.js

```
<script src="dnn.js"></script>
<script src="dnn.diagnostics.js"></script>
```

The only other thing necessary is the unit test(s). This is accomplished by extending the objects you want to test by appending a new method called UnitTests. For example, to add this to the dnn.dom namespace the following code is used.

```
dnn_dom.prototype.UnitTests = function()
```

If you don't understand the prototype keyword please read the Javascript Objects section of this document (especially the link(s)).

The dnn.diagnostics namespace provides many methods for performing your asserts. To view the many asserts you can use simply navigate to the diagnostics namespace in the Client API Namespace Browser.

Consistency and Standards

In the developing of any system, standards should be laid out to make sure the code is easily maintainable. The main difference in developing a client-side API is that you don't have the liberty to be verbose with your naming conventions and comments, for it will increase the payload that is sent over the wire. Note: This is the first time the browser request only, since a js file will be cached on the client from that point on.

While it would be nice to follow the same naming standards as the server-side code, it has been decided that the client-side will use a single letter prefix for datatypes instead of three letters. Even though javascript doesn't allow declaring variables into data types, we still want to denote the intended type.

string = s
object = o
integer = i
date = d

In addition to data type prefixes it is also necessary to define the scope of the variable. Variables with a scope of the entire page (anything outside the function) are considered member variables and should be denoted as such with a m_

For example, a member variable of type string should be declared

```
var m_sName;
```

Namespace objects should simply replace period (.) with an underscore (_) for their definitions. For example, if you wished to create a new namespace called dnn.dom.positioning you would define your object (function) like this.

```
function dnn_dom_positioning()
```

Then any methods defined for this namespace would look like this.

```
dnn_dom_positioning.prototype.mymethod = function (sID)
```

The last piece of the puzzle comes into play with the naming of namespace objects and functions/variables that are not part of the namespace hierarchy. In Javascript, variables can be defined more than once, and the last one defined ends up "winning". Since we are trying to create a generic library, we need to be careful how we name our objects, methods, and variables. It is therefore recommended that we prefix all non-namespace variables and functions with a __dnn.

For example, the string mentioned above should be declared as

```
var __dnn_m_sName;
```

Similarly, a function like those found in the dnncore.js file should also be prefixed.

```
function __dnn_SetInitialFocus(sID)
```

Note: the __dnn should only be used for naming variables and functions in the core. If you are a module developer DO NOT use this prefix.

Disabling ClientAPI

The ClientAPI can be globally disabled by setting the following attribute in the web.config.

```
<appSettings>
   <add key="ClientAPI" value="0" />
</appSettings>
```

Additionally there is a ClientAPICaps.config file found in the js folder that can be edited to include/exclude certain browsers from performing specific functionality. For each functionality element there are two sections: supports and excludes. For each of these elements there is two ways to either include or exclude a browser.

```
<functionality nm="XML" desc="Client Side XML Parsing">
   <supports>
      <browser nm="IE" minversion="4" />
```

```
      <browser nm="Netscape" minversion="6" />
   </supports>
   <excludes>
      <browser nm="Opera" minversion="6" />
      <browser contains="Safari" />
      <browser contains="Konqueror" />
   </excludes>
</functionality>
```

Using the **nm** attribute along with the **minversion** attribute you can specify the name of the browser that the .NET framework supplies through the Request.Browser.Browser property. Along with this you need to specify a minimum version that supports the functionality.

The other way to include or exclude a particular browser is by specifying the **contains** attribute. This will cause the ClientAPI code to look in the browser's UserAgent to determine if a particular string exists.

This file should provide the flexibility to handled any new browsers and upgrades that are released without the need to recompile the core.

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	Aug 16, 2005	Shaun Walker	• Applied new template