From: www.luckstar.com.tw

Date: 2012.04.30

Subject: ASP.NET and jQuery to the max – Part1

**Die Update Panel Die**

**Or how we can use ASP.NET in a real AJAX flavor and live happily.**

The UpdatePanel represents Microsoft's main interpretation of AJAX techniques. The UpdatePanel's indubitable merit is that it brings the magic of AJAX in every day's programming work for any ASP.NET programmer. But now, three years after it was born, the UpdatePanel shows all its limits even to me, a trusty and loyal Microsoft programmer.

Let's see why the UpdatePanel is starting to stink.

The UpdatePanel is an ASP.NET container control.

When certain events occur, it communicates with the server without a full page postback. Communication consists in sending the event name, control name, arguments, page control values, cookies, and the entire viewstate to the server.

The server replies with a full set of markup that has to be injected into the client web page.

The disadvantages of this approach are:

* We have no control of the bandwidth occupation since the viewstate and the replied markup can be very heavy.
* It's very difficult to manage more than one UpdatePanel in a single page in a deterministic way because of the mastermind approach due to triggers and the cascade effects when UpdatePanels are nested.
* Some ASP.NET controls don't work if they are put inside an UpdatePanel (i.e., validators), and we are forced to use the ASP.NET AJAX Control Toolkit that is a sort of black box control set thought to work with MS AJAX. Using this toolkit is a real pain, because making its components work inside a complex page is sort of a trial and error job. very often, we are forced to make a component work the way it wants and not the way we want...

**So How Do I Work with AJAX in ASP.NET?**

It's a matter of bravery, fantasy, and no fear of the unknown.

Since several years, AJAX has been available from JavaScript. JavaScript is able to modify the page DOM, manage CSS dynamically, and use the [XMLHttpRequest](http://en.wikipedia.org/wiki/XMLHttpRequest) object. The problem has always been that it is hard to write JavaScript, because different browser engines interpret it differently.

Furthermore, JavaScript is hard to debug, and since is not a statically typed language, it requires a high level of programming discipline and control over the source code.

Some of these limits today are only bad memories, thanks to [jQuery](http://jquery.com/).

jQuery is a great JavaScript framework that guarantees compatibility across a large range of modern browsers. And, jQuery is also our reply to the question how it is possible to work with AJAX in ASP.NET.

Subsequently, I describe my personal interpretation of the problem.

To reach a complete and organic method for organizing web projects and abandoning Microsoft AJAX, but continuing to work with ASP.NET, I was inspired by suggestions and tricks found on the web and personal work experiences.

This article is a full report of what I understand as modern web development.

You can download a simple sample project to better understand the following points.

**Project organization**

**Let's sharpen our weapons.**

In the demo project, you can see that every ASPX page contains the HTML markup, some Web Service methods, and the onLoad server method to initialize controls. In no other case can an ASPX page contain other methods than these.

Every page has some JavaScript files to accomplish the required tasks: from server communication to page rendering.

**Register Events on Client Page Loading**

The main page of the demo project is *StuffSelection.aspx*.

*StuffSelection.aspx* needs two JavaScript pages: to manage events, to expose functions, and to apply dynamic CSS styles.

I decided to put a *js* file depending on the ASPX page in the same folder as the page. To preserve the order inside the project, any *js* file must have the same name as the related ASPX page:

* *StuffSelection.aspx*
* *StuffSelection.js*
* *StuffSelection\_Proxy.js*

In *StuffSelection.js*, we firstly need to link to the onchange event on the category selector (a DropDownList control), because we have to register the event on the client side to avoid the server events approach.

Since ddlCategory is a server ID, first of all, we have to locate the object on the client side.

We can't use <%= ddlCategory.ClientID %> to resolve the client ID because our JavaScript code doesn't reside on the ASPX page, but we can wrap the object with a client tag and use an appropriate jQuery selector to reach it (in our code, it is a simple span).

In this way, we can load the handler to the change event in the StuffSelection document.ready function.

*//File: StuffSelection.aspx*

<span id="spanDdlCategory">

 Select Stuff category: <asp:DropDownList ID="ddlCategory" runat="server" />

</span>

The JavaScript code:

*//File: StuffSelection.js*

$(document).ready(function () {

 **var ddlCategory = $('#spanDdlCategory select');**

 **ddlCategory.change(function () {**

 **var categoryValue = $(this).val();**

 **GetStuffList(categoryValue);**

 **});**

});

function GetStuffList(categoryValue) {

 if (categoryValue == 0) {

 $('#divStuffList').html('');

 }

 else {

 *// TODO: Get Stuff from server*

 }

}

**jQuery Server Communication**

**The lightness of smart communication**

The onChange client event of our DropDownList calls the GetStuffList function. GetStuffList has the task of calling the server to obtain the list of items for that category and - if possible - without making a complete postback. I usually create another JavaScript file to call the server.

It's a good thing to physically separate the different functionality areas of the JavaScript logic. Don't hesitate to do that if you use clear naming rules. Within some chapters, we will see how to optimize JavaScript files proliferation.

Our new JavaScript file will be *StuffSelection\_Proxy.js*.

For this kind of helper files, I usually mimic a static class with static methods. It's very easy to obtain this effect with JavaScript.

*//File: StuffSelection\_Proxy.js*

**function StuffSelection\_Proxy()** { }

**StuffSelection\_Proxy.GetStuffListHttpGet** =

 function (category, successCallback, failureCallback) {

 $.ajax({

 type: "GET",

 contentType: "application/json; charset=utf-8",

 url: "StuffSelection.aspx/GetStuffListServiceHttpGet?category=" + category,

 success: function (data) { successCallback(data); },

 error: function (data) { failureCallback(data); }

 });

}

Consequentially, our calling function in the *StuffSelection.js* file will change in this way:

*//File: StuffSelection.js*

function GetStuffList(categoryValue) {

 if (categoryValue == 0) {

 $('#divStuffList').html('');

 }

 else {

 **StuffSelection\_Proxy.GetStuffListHttpGet(categoryValue, null, null);**

 }

}

In this way, we will start to request something from a web method called GetStuffListService.

Later, we will see how to receive data from the web method, but now let's describe how GetStuffList works.

This 'static' method receives as a parameter the value of the category DropDownList and two callback functions. For now, only the first parameter is useful to communicate to our server method. To do that, we use a kind of data representation which is called [JSON](http://www.json.org/).

JSON represents data as a value-key pair collection. Since our web method expects a single parameter called "category", we build our JSON parameter from the pair "category" as key and a category value as value.

The result will be something like {"category":"1"}, depending on the category selection.

We use the JSON.stringify method that guarantees a syntactically correct result from the various grouped elements.

JSON is a static library available in almost all browsers, except for older IE (Internet Explorer natively implements JSON since version 8). To work around this problem, I recommend the json2 library that checks the browser for native JSON ability; otherwise, it supplies the application with JSON method support (<http://www.json.org/json2.js>).

$.ajax is a jQuery method that allows to post a request to the server. By default, the communication is posted in asynchronous mode so that users don't experience page freezing and can post multiple requests to the server without having to wait for any response. $.ajax can notify of a successful response, and a failure one in case of server errors.

The Url option identifies the endpoint for the client request. An endpoint could be a method of a Web Service (*.asmx*), a web handler (*.ashx*), or a web method inside a page (*.aspx*).

$.ajax supports the GET and POST methods, and in the demo project, I included both to show the two different syntaxes. I'll show only the GET call here that is more correct in this case.

*//File: StuffSelection.aspx*

[WebMethod]

[ScriptMethod(UseHttpGet = true,

 ResponseFormat = ResponseFormat.Json, XmlSerializeString = false)]

public static IList<Stuff> GetStuffListServiceHttpGet(int category)

{

 return StuffHelper.GetStuffList(category);

}

Let's analyze the result of this communication (a simple post to our web method could be analyzed through Firebug; see chapter "Debug with Firebug").

The complete post weighs less than 1 KBytes because it contains only our simple request with the category parameter, a JSON string as response, and eventually, cookies and the message header. In no way will we receive more data than that, no more markup, no more viewstate. Hooray!

**What Are We Losing With this 'Smart' Communication?**

**Not so much, don't worry.**

Well... we are losing control state.

It seems terrible at first look... how can we work without stateful controls?

In ASP.NET, the control state resides in the same controls and in the viewstate (for non-visible controls). When an UpdatePanel does a partial postback, we get access on the server side to every page control with its consistent state.

This is the reason why we can access the content of textboxes, the selected item of a combobox, and so on even during a partial postback.

But ASP.NET is not a smart guy, because it can't predict what control states we need on the server side, so it posts everything: the visible control values and the entire viewstate that contains a lot of useless information even for a partial postback. Too heavy for us!

Trust me and don't worry. The stateful controls approach was a great comfort in the old style web form architecture, but it is not necessary... no more.

**Thus Spoke Our Server**

Now we have to collect the server reply.

To do that, we need to implement a success callback function to pass as parameter to our proxy method. Let's see how our *js* file changes again.

*//File: StuffSelection.js*

function GetStuffList(categoryValue) {

 if (categoryValue == 0) {

 $('#divStuffList').html('');

 }

 else {

 StuffSelection\_Proxy.GetStuffListHttpGet(categoryValue,

 **successCallback, failureCallback**);

 }

}

**var successCallback = function (data) {**

 **var response = eval(data.d);**

 **$('#divStuffList').html(response[0].Name + ' ' +**

 **response[0].Description + ' (€ ' + response[0].Price + ')');**

 **}**

**var failureCallback = function (data) {**

 **alert('Request failure');**

**}**

Firebug helps us again. Put a breakpoint on the first line of the successCallback function to see how the server replies to the request.

The "data" parameter contains a string in JSON format that can be evaluated and converted into a JavaScript object.

I evaluate data.d because "d" is a container object that boxes any JSON serialization done by ASP.NET. In that way, I have a variable response that contains an object returned by the web method, i.e., List<Stuff> of the requested category.

Naturally, a List<T> has to be serialized into objects that JavaScript can understand. ASP.NET serializer converts it into an array of T, and this is enough for our goal.

**Client Templating**

**...Or how I can completely (and finally) control my grids.**

Now we have an array of objects and we need to print it on our web page.

The former method:

$('#divStuffList').html(response[0].Name + ' ' + response[0].Description ...

is naturally a fast way to show the result, but it is not the best way to render objects.

We better use templating.

JavaScript templating is available through independent frameworks or through jQuery plug-ins.

jQuery plug-ins are the hidden treasury of this fantastic framework, because they supply the core framework with tons of graphical effects, services, components, and so on. Think about anything you need in your web page and surely a jQuery plug-in exists to accomplish this task.

Lately, I have been using a jQuery template plug-in called [jBind](http://arashkarimzadeh.com/index.php/jquery/11-jbind-jquery-bind-template.html). It's a very good plug-in, but it seems that it hasn't been supported for a long time. So I won't describe this plug-in in detail but the concepts behind templating in JavaScript (waiting for the [Microsoft client templating plug-in](http://weblogs.asp.net/scottgu/archive/2010/05/07/jquery-templates-and-data-linking-and-microsoft-contributing-to-jquery.aspx)).

Templating on the client side is exactly the same as templating on the server side. Instead of having Repeaters, GridViews, or other super complex ASP.NET controls, you only have a piece of HTML with some placeholders inside. Usually, template plug-ins link data by binding key names to labels with the same name.

If our data source is an object array, template plug-ins repeat a group of HTML tags enough times to render all objects.

We can decide to put our template inside the same page in which it'll be rendered (making that piece of code invisible), or create tags dynamically from JavaScript.

I prefer the first approach because I can easily view my template for design purposes by unhiding it.

For the same reason, I can draw my template with an editor, using styles and classes, and immediately viewing the resulting graphical rendering.

This is a simple example of templating:

//File: StuffSelection.aspx

<div style="display: none;">

 <%-- Templates --%>

 <%-- Template for stuff list --%>

 <div id="divStuffListTemplate">

 <div>

 <table class="tableList">

 <tr>

 <td>Name</td>

 <td>Code</td>

 <td>Description</td>

 <td>Price</td>

 <td>Is available</td>

 </tr>

 *<!--data*-->

 <tr>

 **<td>{Name}</td>**

 **<td>{Code}</td>**

 **<td>{Description}</td>**

 **<td>€{Price}</td>**

 **<td>{IsAvailable}</td>**

 </tr>

 *<!--data*-->

 </table>

 </div>

 </div>

</div>

The external div is useful to hide/unhide the entire block. The most interesting lines are the ones that contain placeholders that have to be linked to the array object.

From a JavaScript perspective, we only have to activate our plug-in according to its syntax. In my case, the code is this:

*//File: StuffSelection.js*

var successCallback = function(data){

 var response = eval(data.d);

 *//$('#divStuffList').html(response[0].Name + ' ' +*

 *// response[0].Description + ' (€ ' + response[0].Price + ')');*

 $('#divStuffList').html('');

 var template = $('#divStuffListTemplate').html();

 **$(template).bindTo(response, { fill: true, appendTo: '#divStuffList' });**

}

Since client templating could be directly applied to the HTML code. It is super easy to add classes, decorations, styles, numbering, alternating background colors, and so on. And without doubt, theming HTML is easier and more flexible than skinning a GridView or whatever server control.

**And If I Need to Send a Complex Type to the Server?**

**Well... it's difficult in ASP.NET too.**

Indeed it's not difficult with a bit of JavaScript, and it makes possible to happily forget that our server approach is stateless (as promised before).

We have a page to register a new user (*SignupUser.aspx*), and we need to pass a lot of data to our web method: user name, first name, surname, password, and whatever else.

It is possible to implement a web method with several arguments, but our intention is to build a clear and maintainable structure. In other words, we want to pass a complex object to the server.

With the ASP.NET serialization process (and deserialization too), this is done automatically.

In our JavaScript, we define a simple pseudo class named UserJs.

*//File: SignupUser\_User.js*

function UserJs() {

 this.FirstName = '';

 this.Surname = '';

 this.Username = '';

 this.Password = '';

}

We can do the same on the server:

*//File Ecommerce.Business.UserJs*

public class UserJs

{

 public string FirstName { get; set; }

 public string Surname { get; set; }

 public string Username { get; set; }

 public string Password { get; set; }

}

This is all you need to exchange data between the client and server.

If our web method RegisterUserService in the *SignupUser.aspx* page expects an Ecommerce.Business.UserJs parameter, we have to pass a serialized object that has the same name, same fields (same field names), and same type (if automatic conversion is not possible) from JavaScript. That's all.

*//File: SignupUser.js*

var user = new UserJs();

user.FirstName = $('#firstName').val();

user.Surname = $('#surname').val();

user.Username = $('#userName').val();

user.Password = $('#password').val();

SignupUser\_Proxy.RegisterUser(**user**, successCallback, failureCallback);

In the sample project, let's try to register a user with Visual Studio in Debug mode to check the deserialized UserJs instance in the web method.

**And if I Need to Receive a Complex Type from the Server at Load Time?**

**Same question but different direction**

We can't tie ourselves to call specific web methods during page load to supply our client controls with their initial values. Imagine that you have several controls to feed and you don't want to massively use ASP.NET server controls.

There is a simple but very smart workaround to supply the client with values available on the server side.

Don't think of injecting JavScript variables from the server with the RegisterClientScriptBlock or RegisterStartupScript functions. We are trying to be clean and elegant in our new programming way.

Instead, have a look at this web page: <http://west-wind.net/Weblog/posts/259442.aspx>.

This Hawaiian guy has written a beautiful class that allows to load a collection of key - value objects on the server and make it available on the client side in a sort of hashtable.

Let's look at *GetListFromServerVars.aspx* in our demo project.

This page loads the entire Category collection (category name - category value) through the AddClientVariables method supplied by BasePage.

Then, *GetListFromServerVars.js* reads all the values and prints them on the page.

*//File: GetListFromServerVars.aspx*

foreach(Category category in categoryList)

{

 base.AddClientVariable(category.Key, category.Value);

}

*//File: GetListFromServerVars.js*

var html = 'Computer: ' + serverVars["Computer"];

html += '<br/>Monitor: ' + serverVars["Monitor"];

html += '<br/>Keyboard: ' + serverVars["Keyboard"];

**Minimize JavaScript**

**The zen garden of optimization**

Until now, we haven't taken care of the proliferation of JavaScript files. I appreciate a development method that splits the source code in several logic units and therefore in several files. It's a matter of organization, order, and maintainability.

But there is a disadvantage in that. If our web server has to distribute a lot of small files instead of a bigger one, we are losing bandwidth and performance.

But there is a simple workaround: the minimization and merge process.

**Scenario**

Our project has a lot of JavaScript files per each ASPX page. In the JavaScript folder, we have some jQuery plug-in.

**Solution**

In the sample project, there is a folder "*lib*" that contains a program called *jsmin.exe* (see <http://www.crockford.com/javascript/jsmin.html>). Jsmin minimizes and merges JavaScript files.

Now go to the EcommerceWebSite project and ask for its properties. In the Build Events tab, I inserted a post-build event command.

type "$(ProjectDir)js\jquery.\*.js" | "$(ProjectDir)..\..\lib\jsmin" >

 "$(ProjectDir)js\min\Plugins.min.js" %copyright%

ECHO Minify and merge aspx js

type "$(ProjectDir)SignupUser\*.js" | "$(ProjectDir)..\..\lib\jsmin" >

 "$(ProjectDir)js\min\SignupUser.min.js" %copyright%

type "$(ProjectDir)StuffSelection\*.js" | "$(ProjectDir)..\..\lib\jsmin" >

 "$(ProjectDir)js\min\StuffSelection.min.js" %copyright%

type "$(ProjectDir)GetListFromServerVars\*.js" | "$(ProjectDir)..\..\lib\jsmin" >

 "$(ProjectDir)js\min\GetListFromServerVars.min.js"

The post-build command takes care of supplying all jQuery plug-ins found in the JavaScript folder to *jsmin.exe*. Jsmin creates a single JavaScript file that is copied into the folder *js/min*.

In the same way, post-build commands must be configured for any ASPX page that needs JavaScript files.

Every ASPX page will produce a minimized JavaScript file. For example, for the *StuffSelection.aspx* page, a single JavaScript file called *StuffSelection.min.js* will be produced.

At this point, a big problem is born: how can we use the minimized JavaScript files preserving the ability to debug client script? Minimized and merged code is not understandable, even by a good programmer. So we need to maintain the ability to have both: minimized and not minimized files.

The solution is a simple and brilliant idea found on various web sites. JavaScript files linked from ASPX depend on a parameter in our configuration.

When in the *web.config* file, we set Debug mode (<compilation debug="true"/>), we link to the original JavaScript files. And when we set Release mode (<compilation debug="false"/>), we link to the minimized files.

Consequentially, our ASPX files change in this way:

//File: StuffSelection.aspx

<% **if (HttpContext.Current.IsDebuggingEnabled)** **{** %>

 <script src="StuffSelection.js" type="text/javascript"></script>

 <script src="StuffSelection\_Proxy.js" type="text/javascript"></script>

<% **} else {** %>

 <script src="js/min/StuffSelection.min.js" type="text/javascript"></script>

<% **}** %>

In that way, we can have source or minimized files by changing a simple property in the *web.config*.

Naturally, if we want to have the ability to debug client scripts even in production environment, we have to publish both the source JavaScript and the minimized JavaScript.

**Cache Them All**

**When a good feature could cause trouble**

The approach to web programming that I'm presenting here has various consequences. One of them is that the entire UI logic and some pieces of business logic are moved from the server side to the client side. In other words, it means that you have to write tons of JavaScript code.

I consider this fact as an excellent pro with some little cons. One disadvantage is that we are forced to deeply know the various browsers that become important tools of our developing techniques.

jQuery helps a lot in hiding most peculiarities of different browser engines, but the game is not over.

Let's see a simple example.

Imagine that during a quiet and boring Friday afternoon, your boss comes into your office with the usual request: "Can you change the message that appears when I push the button in that web application...". Bosses are special for making this kind of requests on Friday afternoons.

The message is naturally cabled into a JavaScript file.

"Ok" - you think - "the task is very easy this time and I shouldn't stay here until late".

You change your local JavaScript file, test it, commit it (you have a source control system haven't you?), and copy it to the test and production environments. Everything in five minutes and without recompiling the entire project.

Now you can proudly go to your boss telling him: "It's ready".

Instead of being amazed about your super-speed execution, your beloved boss goes to the web site, pushes that button, and... discovers that the message has not changed.

"Damn cache" you think.

Anyway the solution is very easy: press CTRL + F5 and the browser refreshes its cache, the JavaScript file is correctly reloaded, and everything goes well.... but we can't force our users to press CTRL + F5 to be sure to have the new file. And your boss is not so happy about that...

The problem is that the browsers are very aggressive in managing their caches.

If you watch what happens during a page request, you could be surprised. For monitoring requests to a server, you can use, as usual, Firebug, or another beautiful free tool called [Fiddler](http://www.fiddler2.com/).

Both tools could monitor every page request and split it into its elementary parts.

You'll see that for every requested resource, the browser can:

* download it (with code 200);
* take it from the cache after checking with the server if it's changed (code 304);
* just take it from the cache (the resource doesn't appear at all).

I don't know what the rules are that every browser follows to choose the right option, but I surely know that, in some cases, some browsers dramatically fail this task.

The possible solution could be playing with headers to instruct the browsers to cache or not some resources, but this method is quite unpredictable too.

So we have two easier and safer methods to solve this situation:

1. Change the name of the resource when the resource content changes.

This is the method used by jQuery developers (jQuery 1.4.1 is a different resource than jQuery 1.4.2, and your browser is forced to download it). But this method is difficult to implement because when you change a resource name, you'll have to change any reference to that resource too. It's a manual task, prone to errors and lapses, and for these reasons, we don't like it very much. It's suitable for a JavaScript component or framework files, but not for application files.

1. You can instruct your browser to ask for a resource with a GET request.

*code.js?version=1* is different from *code.js?version=2*, and your browser surely asks for it even if it has cached the file *code.js*.

In this way, you can centralize the management of the "version" variable, for example, by putting it into the application *web.config*, and change that value after every modification of every JavaScript file.

In the ASPX file you can write something like:

<script src="code.js?version=<%= Config.Version %>" type="text/Javascript">

</script>

to update any reference in all your project in one step.

I liked this second method very much when I found it on Internet, but it was not perfect. When I change a simple message in one single JavaScript file, I force the browsers to download all JavaScript files in all the application web pages because the version value changes for every file.

To avoid this waste, I simply write down a method that I called ResolveAndVersionUrl that resolves the URL and adds to the resolved resource name a value that represents the file hash.

In this way, only the resources effectively changed are the ones that change their querystring.

Well... I'm not crazy! I don't make my application calculate a single hash of any JavaScript resource at every single page request. For this task, the ASP.NET cache becomes very useful. You can see a simplified example of my algorithm in *Controls/BasePage.cs*, and a typical use in the *CacheJs.aspx* page.

In this example, the cache survives for one day, and it depends on a file on the file system (when I change a JavaScript file, I have to remember to delete the depending file to make the application recalculate all the file hashes).

Since any JavaScript resource is requested with GET, the browser is forced to ask for it to the server, and only if the resource has not changed, take it from its cache.

A request that receives a status 304 weights a bit, but since we have merged all page JavaScript resources in one single file, the total traffic weight is acceptable.

**Debug with Firebug**

**...our best friend.**

Firebug is a fantastic Firefox plug in.

One of its best functions is debugging a client script through several tools:

* breakpoints (absolute and conditional)
* automatic and manual watchers
* contextual inspector
* call stack
* evaluation console
* standard shortcut keys to progress in debugging



Since version 8, Internet Explorer has its developer console which is powerful too. But old habits die hard, so my best friend remains Firebug.

Firebug allows to monitor every post to a server, even the ones that originate from the client script.

In this way, you can monitor the parameters passed to web methods and the data obtained as a reply.



I recommend installing Firecookie together with Firebug to monitor, manage, and edit cookies.

With this instrumentation, your tool belt is quite complete for any task regarding client debugging.

**Regions in JavaScript**

**The icing on the cake**

Yes, it is possible to organize your JavaScript code with regions, now that it is beginning to grow more and more after this article.

You simply have to include the following macro code in Visual Studio:

Option Explicit On

Option Strict On

Imports System

Imports EnvDTE

Imports EnvDTE80

Imports EnvDTE90

Imports System.Diagnostics

Imports System.Collections.Generic

Imports System.Text.RegularExpressions

Public Module JsMacros

 Sub OutlineRegions()

 Dim selection As EnvDTE.TextSelection = \_

 CType(DTE.ActiveDocument.Selection, EnvDTE.TextSelection)

 Const REGION\_START As String = "//\s\*?\#region"

 Const REGION\_END As String = "//\s\*?\#endregion"

 selection.SelectAll()

 Dim text As String = selection.Text

 selection.StartOfDocument(True)

 Dim startIndex As Integer

 Dim endIndex As Integer

 Dim lastIndex As Integer = 0

 Dim startRegions As New Stack(Of Integer)

 Dim rStart As New Regex(REGION\_START, RegexOptions.Compiled)

 Dim rEnd As New Regex(REGION\_END, RegexOptions.Compiled)

 Do

 Dim matchStart As Match = rStart.Match(text, lastIndex)

 Dim matchEnd As Match = rEnd.Match(text, lastIndex)

 startIndex = matchStart.Index

 endIndex = matchEnd.Index

 If startIndex + endIndex = 0 Then

 Return

 End If

 If matchStart.Success AndAlso startIndex < endIndex Then

 startRegions.Push(startIndex)

 lastIndex = startIndex + 1

 Else

 *' Outline region ...*

 Dim tempStartIndex As Integer = CInt(startRegions.Pop())

 selection.MoveToLineAndOffset(CalcLineNumber(text, \_

 tempStartIndex), CalcLineOffset(text, tempStartIndex))

 selection.MoveToLineAndOffset\_

 (CalcLineNumber(text, endIndex) + 1, 1, True)

 selection.OutlineSection()

 lastIndex = endIndex + 1

 End If

 Loop

 selection.StartOfDocument()

 End Sub

 Private Function CalcLineNumber(ByVal text As String, \_

 ByVal index As Integer) As Integer

 Dim lineNumber As Integer = 1

 Dim i As Integer = 0

 While i < index

 If text.Chars(i) = vbLf Then

 lineNumber += 1

 i += 1

 End If

 If text.Chars(i) = vbCr Then

 lineNumber += 1

 i += 1

 If text.Chars(i) = vbLf Then

 i += 1 *'Swallow the next vbLf*

 End If

 End If

 i += 1

 End While

 Return lineNumber

 End Function

 Private Function CalcLineOffset(ByVal text As String, ByVal index As Integer) \_

 As Integer

 Dim offset As Integer = 1

 Dim i As Integer = index - 1

 *'Count backwards from //#region to the previous line counting the white spaces*

 Dim whiteSpaces = 1

 While i >= 0

 Dim chr As Char = text.Chars(i)

 If chr = vbCr Or chr = vbLf Then

 whiteSpaces = offset

 Exit While

 End If

 i -= 1

 offset += 1

 End While

 *'Count forwards from //#region to the end of the region line*

 i = index

 offset = 0

 Do

 Dim chr As Char = text.Chars(i)

 If chr = vbCr Or chr = vbLf Then

 Return whiteSpaces + offset

 End If

 offset += 1

 i += 1

 Loop

 Return whiteSpaces

 End Function

End Module

Now link your macro code with a custom shortcut, and here we are: regions are activated every time your shortcut is pressed.

**Second Part**

You can find the second part of this article [here](http://www.codeproject.com/KB/ajax/AjaxAndAspNETPart2.aspx).

**License**

This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)

**About the Author**

|  |  |
| --- | --- |
| [**Gianluca Negrelli**](http://www.codeproject.com/Members/gianluca-negrelli)Web DeveloperItalyItalyMember | I'm the very proud father of [SyMenu](http://www.ugmfree.it/SyMenu.aspx), a multi awarded portable start menu and [www.ghezee.com](http://www.ghezee.com) an innovative ads web site. |