From: www.luckstar.com.tw

Date: 2012.04.30

Subject: ASP.NET and jQuery to the max – Part2

**Introduction**

This is my second article regarding ASP.NET and jQuery. I advise you to read [the first part](http://www.codeproject.com/KB/ajax/AjaxAndAspNET.aspx) before approaching this one.

In this article, I would like to speak about two important arguments that contribute to make our new way of programming more complete and flexible. The first topic regards the creation of reusable components that could interact with the server side. The second one regards the error handling on the client side.

**Reusable components**

**Recycle your artwork**

One interesting subject to discuss is how to create reusable components with the new techniques explained in the first part of the article. To accomplish this task, we rely on the ASP.NET UserControl facility, say *.ascx* components. In effect, we don't need a lot of customization to reuse the native ASP.NET approach, but our interpretation of reusable components implies that any *ascx*:

* must carry with it some JavaScript files: they represent our infrastructure to manage the client logic and to communicate with the server;
* is allowed to access some endpoints that offer services to the hosting pages;
* could be instantiated in the same hosting page more than once: ScriptManager solves this problem transparently for us in ASP.NET (it differentiates IDs and removes duplicate JavaScript references), but in our new world, we don't use it anymore.

**Carry on our UserControl with JavaScript**

**Load the gun**

I like very much having my JavaScript files either near the owner page or owner component. In case of owner pages, we resolved the problem with some strong naming convention rules ([see part 1](http://www.codeproject.com/KB/ajax/AjaxAndAspNET.aspx)). In this way, we reached our goal to group the files linked to each other and to easily merge every linked JavaScript file in one only file during the minimization process.

Naming convention comes to help us again. If we decide to put every user control in only one folder, we could call it *UserControls* as I do, with a great effort of imagination, in the demo project. But we can decide to organize our project with other criteria too. In any way, by setting strong rules for naming conventions and file locations inside the project, it's mandatory to maintain our solution well-ordered and maintainable.

Inside the *UserControls* folder, you can find a user control (*EditorControl.ascx*) that accomplishes a simple task: save and retrieve key-value elements.

We can see this component in action inside the page *HostingPageUserControl.aspx*. To test the component, try to supply an ID on the first textbox and any text on the textarea above. Then press the Save button. Now try to recover the saved item, supplying the right ID and pressing the button *Get text*. OK, I know it's a trivial task, but we have to stay focused on the functionalities and not on the usefulness.

If we open the *HostingPageUserControl.aspx* file, we'll see a simple registration tag for the user control and a simpler inclusion of it inside the page.

File: HostingPageUserControl.aspx

<%@ Register src="UserControls/EditorControl.ascx"

 tagname="Editor" tagprefix="SmartUserControls" %>

...

<SmartUserControls:Editor ID="Editor1" runat="server" />

From this point of view, nothing has changed in comparison to traditional ASP.NET, but now let's see the content of the *UserControls* folder in the Solution Explorer.



We can see the *.ascx* file, two JavaScript files that comply with our usual naming convention rules, and a Web Service, the *.asmx* file, that contains the endpoints of our control. As usual, the JavaScript files are minimized and merged during compilation time and the resulting file (*EditorControl.min.js*) is copied in the folder *js/min*.

Open *EditorControl.ascx*. We can see the references to our JavaScript files in debug mode and in release mode and the markup for the elements of the component.

File EditorControl.ascx

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

 <%= this.**RegisterClientScript**("~/UserControls/EditorControl.js")%>

 <%= this.**RegisterClientScript**("~/UserControls/EditorControl\_Proxy.js")%>

<% } else { %>

 <%= this.**RegisterClientScript**("~/js/min/EditorControl.min.js")%>

<% } %>

<div style="float: left; margin-right: 50px;"

 id="editControlContainer" runat="server">

A question arises: why do we use this strange "RegisterClientScript" method and not our usual "ResolveAndVersionUrl" to introduce JavaScript references inside the control?

**Make user controls reusable more than once in the same hosting page**

**Or... I want the script manager back!**

If we use the ResolveAndVersionUrl method to include the referenced JavaScript files, the hosting page works the same, but if we try to include our control more than once in the same page, things change.

In the demo project, there is another page, *HostingPageUserControlMulti.aspx*, that contains three instances of EditorControl.

If we use ResolveAndVersionUrl, we'll find three references to *EditorControl.min.js*. More than one reference to the same JavaScript file in the same page represents a serious validation error on the client side and reduces performance of downloading the page.

RegisterClientScript is a method that resides on the BasePage class (inside the *Controls* folder) that calls the usual ResolveAndVersionUrl only if that particular JavaScript file has not been included inside the page yet, otherwise the reference is not added. It's one of the services that the ASP.NET ScriptManager offers to developers in the background with its traditional RegisterClientScriptBlock method.

One other problem to solve regards the IDs of controls. If we include EditorControl more than once inside a single page, we risk duplication of the HTML elements' IDs if we use them for the contained element. That way, an HTML tag with a certain ID will become not uniquely identifiable on the client side by jQuery and the page can't be validated. For this reason, we are forced to renounce to use IDs. The JavaScript of the control knows that it can't entrust IDs to select elements but classes, relative position, attribute, and type.

Let's see how I did it in the demo project.

The GetText and SaveText functions require the ClientId of a div that contains all control elements. This way we specify a search scope. Then, inside the JavaScript code, the elements are found using selectors that refer to their class, attribute, or type.

File: EditorControl.ascx

<div **runat="server" id="editControlContainer"**

 style="float:left; margin-right:50px;">

<div id="notUnique"></div>

Text id: <br />

<input type="text" value="" maxlength="3" style="width:60px;" />

<input type="button" value="Get text"

 onclick="**GetText('<%=editControlContainer.ClientID %>')**" />

<input type="button" value="Save"

 onclick="**SaveText('<%=editControlContainer.ClientID %>')**"

 style="float:right;" />

File: EditorControl.js

function GetText(**idContainer**) {

 var divContainer = $(**'#' + idContainer**);

 var spanResult = $(divContainer).find('**.spanResult**');

 var textboxId = $(divContainer).find('**:text**');

 var textArea = $(divContainer).find('**textarea**');

This is only one of the possible ways to differentiate the single control instance from the client's point of view. If you want to create your own method, remember two things:

* do not use IDs for control elements to avoid duplication;
* make JavaScript able to understand which control instance it is working with using a container uniquely identified.

**Supply our UserControl with endpoints**

**Take aim**

Every UserControl could offer some services on the server side.

We already learned how to consume server side methods through jQuery, but in the first part of this article, we only focused on the web method way.

An *.ascx* file couldn't expose a web method, so we can create the control endpoints inside a special Web Service (*EditorControlService.asmx*). Our WS complies with the usual naming convention and it can be invoked exactly in the same way as a web method.

Let's see the two calls from the JavaScript file:

File: EditorControl\_Proxy.js

EditorControl\_Proxy.**GetText** = function (idText, getTextCallback) {

 $.ajax({

 type: "GET",

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

 **url: "UserControls/EditorControlService.asmx/GetText?id=" + idText,**

 success: getTextCallback,

 error: feedbackFailure

 });

}

EditorControl\_Proxy.**SaveText** = function (idText, text, saveTextCallback) {

 var jsonData = JSON.stringify({ id: idText, content: text });

 $.ajax({

 type: "POST",

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

 **url: "UserControls/EditorControlService.asmx/SaveText",**

 data: jsonData,

 success: saveTextCallback,

 error: feedbackFailure

 });

}

Well, I don't like very much that the *.asmx* file is placed inside a folder called *UserControls*, but in this way, we can group files strictly related from a logical point of view, in the same physical location.

**Check the result**

**And finally our page works**

If you try to load *HostingPageUserControlMulti.aspx*, you'll see that the three control instances work together like a charm. Have a look at the HTML source code to see the single reference of any JavaScript file. Naturally, try to work with the controls in the page to make sure that every control knows its proper scope and works only with its elements.

As a last experiment, try to uncomment this line in *EditorControl.ascx*:

File: EditorControl.ascx

<div id="notUnique"></div>

Now reload the page and validate the resulting markup: you'll see that the duplication of IDs is considered a serious error.

To validate the markup of a page, you could use any suitable tool. I prefer using [Web developer toolbar](https://addons.mozilla.org/en-US/firefox/addon/60/) for Firefox that can validate remote and local pages. It is a must have tool for tens of other useful features.

**Error management on the client side**

**Well, sometimes our app does something wrong**

Often we need to communicate something to end users in reply of certain actions. When an invoked method has to show to the end user a feedback message, we can think in advance of a nice way to communicate it: with JavaScript alert (not exactly a nice way...), with message dialogs, with balloon popups, and so on.

The problem arises when a returning message is not expected, for example when we need to communicate that something has gone wrong because of an error. My solution is to create a common infrastructure to manage error messages on the client side. Indeed a very common methodology is to return pure exceptions from the server side with customized and user-understandable messages.

On the client side, you simply have to capture exceptions and present them to users in a comfortable way (well... I haven't seen a comfortable exception yet but...).

File: ErrorManagement\_Proxy.js

*// -----------------------------------------*

*// Get error from server*

*// -----------------------------------------*

ErrorManagement\_Proxy.GetError = function (successCallback) {

 $.ajax({

 type: "POST",

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

 url: "ErrorManagement.aspx/ThrowServerError",

 success: successCallback,

 **error: feedbackFailure**

 });

}

*// -----------------------------------------*

*// Failure management.*

*// -----------------------------------------*

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

 **var message = JSON.parse(data.responseText);**

 **alert(message.Message);**

**}**

This way you can dissect your exception and show the user the only important thing: the error message.

On the server side, we raise the exception in a trivial way, with an instruction like this:

File: ErrorManagement.aspx

[WebMethod]

public static bool ThrowServerError()

{

 **throw new Exception**("Hello! This is a managed error message from an async call.");

}

Now we can see what really happens during the response.

Go on the *ErrorManagement.aspx* page, turn on Fiddler or Firebug with Net tab activated, and click the first button "Throw async error".

Our simple error management code (feedbackFailure JavaScript function) replies with a message box in which we alert the user that an error has occurred with our custom message.



Now analyze the entire content of the response.



We succeeded to return the custom message, but in the response stack trace, it's possible to read the full path of the ASPX page that contains the endpoint of the client call and this is not a good thing.

The reason ASP.NET returns the entire stack is because we are in debugging mode. Open the application *web.config* file and change:

File: Web.config

<compilation debug="**true**" >

to:

<compilation debug="**false**" >

Now press the Error button again and go to analyze the response.



The result is better now: we are only exposing the name of the endpoint that is already public through the stack trace.

**Go deep with inspection**

**...or how I like to complicate things**

Well, everything seems to work well. And this time without any difficulties. But the enemy is always behind the door and we have not trespassed the threshold yet.

Well, the threshold is the deploying of the web application in its production environment with all the details that make it so professional. One element in particular is very dangerous for our still fragile error infrastructure: the custom error redirector.

Open the *web.config* file again and try to change this line:

<customErrors mode="**Off**" />

to:

<customErrors mode="**On**" defaultRedirect="GenericError.aspx">

 <error statusCode="404" redirect="PageNotFoundError.aspx"/>

</customErrors>

because we assume that in a production environment we need to redirect, at least, the 404 error.

Push the Error button and read the pop up message again:



What's that!? Where has our really explanatory message gone??

CustomErrors is not a smart guy. When it has to manage a full postback error (i.e., a page not found error), it is able to redirect the browser to the correct page.

But when he manages errors in partial postback (say asynchronous requests), it is not able to redirect the page but it intercepts the error the same.

"Why?" I would like to ask. If you are able to do nothing, do nothing please.

**We enter the dark tunnel**

**...and no light is in front of us.**

OK, we are deadlocked.

On one side, we want to use the native .NET exception infrastructure to return managed error messages to the client during asynchronous requests and, on the other side, we want to use CustomErrors to redirect the browser when the application throws an error during full postbacks.

The problem seems to be unsolvable. Indeed, MS UpdatePanel seems to be a little doubtful with this stuff. When CustomErrors is Off and an error occurs inside an UpdatePanel, the server returns a status code 200 (OK) and this strange string: 14|error|500|(custom error text)|.

It seems that MS encoded the real status code in this string (500 is a more correct status code in this case) that UpdatePanel has to read and manage.

If CustomErrors is On but no redirection is set, the server response is something like this: 0|error|500||. We can notice that in this case an eventual custom error message is lost.

Finally, with CustomErrors On and an adequate redirection in case of error 500, the UpdatePanel chooses to redirect you to your custom page even if you are inside an asynchronous postback.

Am I wrong or does it seem a bit like confusing management?

**Break on through to the other side**

**The gate is straight, deep and wide...**

The solution is easy but radical at the same time: an HTTP Module that manages response exceptions and a new section in *Web.config* that I call CustomErrorsAjax that works together with the HTTP Module.

CustomErrorsAjax is a new section that is configurable exactly in the same way as its dumb brother CustomErrors.

The advantage with it is that if you are inside an asynchronous request, it doesn't intercept the error so that our application can use error messages on the client side.

With that solution, our management of errors became linear and clear and we can use the .NET exception to lead messages to any asynchronous request from the client side.

Simply reset ASP.NET CustomErrors to Off.

File: Web.config

<customErrors mode="**Off**" defaultRedirect="GenericError.aspx">

 <error statusCode="404" redirect="PageNotFoundError.aspx"/>

</customErrors>

and activate the CustomErrorsAjax section (I've already included it on the *Web.config* file, so uncomment it):

File: Web.config

<section name="customErrorsAjax"

 type="EcommerceWebSite.CustomErrorAjax.CustomErrorAjaxConfig,

 EcommerceWebSite" requirePermission="false"/>

...

<customErrorsAjax mode="On" defaultRedirect="GenericError.aspx">

 <errors>

 <clear />

 <error statusCode="404" redirect="PageNotFoundError.aspx" />

 <error statusCode="500" redirect="GenericError.aspx?e=500" />

 </errors>

</customErrorsAjax>

Now go to page *ErrorManagement.aspx* and try to push the three buttons.

The first button "Throw async error" correctly replies with our managed error message: "Hello! This is a managed error message from an async call".

The second one "Throw postback error" simply makes a full postback and calls a server method btnError\_Click that returns an error. Our HTTP Module checks if this error is not an async one and redirects our browser on the page specified in *web.config*. In this case, the status code is 500 so the target page is *GenericError.aspx*.

The last button "Throw page not found error" calls for a not existing page and makes our HTTP Module to redirect the 404 error to the right page (*PageNotFoundError.aspx*).

Well, the HTTP Module is newborn and raw. Feel free to improve it according to your needs, the important thing is the core of the solution.

**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. |