Introduction

Microsoft's ASP.NET technology brings an object-oriented and event-driven programming model and unites it with the benefits of compiled code. However, its server-side processing model has several drawbacks inherent in the technology:

- Page updates require a round-trip to the server, which requires a page refresh.
- Round-trips do not persist any effects generated by Javascript or other client-side technology (such as Adobe Flash)
- During postback, browsers other than Microsoft Internet Explorer do not support automatically restoring the scroll position. And even in Internet Explorer, there is still a flicker as the page is refreshed.
- Postbacks may involve a high amount of bandwidth as the __VIEWSTATE form field may grow, especially when dealing with controls such as the GridView control or repeaters.
- There is no unified model for accessing Web Services through JavaScript or other client-side technology.

Enter Microsoft's ASP.NET AJAX extensions. AJAX, which stands for **A**synchronous **J**avaScript **A**nd **X**ML, is an integrated framework for providing incremental page updates via cross-platform JavaScript, composed of server-side code comprising the Microsoft AJAX Framework, and a script component called the Microsoft AJAX Script Library. The ASP.NET AJAX extensions also provide cross-platform support for accessing ASP.NET Web Services via JavaScript.

This whitepaper examines the partial page updates functionality of the ASP.NET AJAX Extensions, which includes the ScriptManager component, the UpdatePanel control, and the UpdateProgress control, and considers scenarios in which they should or should not be utilized.

This whitepaper is based on the Beta 2 release of the Visual Studio 2008 and the .NET Framework 3.5, which integrates the ASP.NET AJAX Extensions into the Base Class Library (where it was previously an add-on component available for ASP.NET 2.0). This whitepaper also assumes that you are using Visual Studio 2008 and not Visual Web Developer Express Edition; some project templates that are referenced may not be available to Visual Web Developer Express users.

Partial Page Updates

Perhaps the most visible feature of the ASP.NET AJAX Extensions is the ability to do a partial or incremental page updates without doing a full postback to the server, with no code changes and minimal markup changes. The advantages are extensive – the state of your multimedia (such as Adobe Flash or Windows Media) is unchanged, bandwidth costs are reduced, and the client does not experience the flicker usually associated with a postback.

The ability to integrate partial page rendering is integrated into ASP.NET with minimal changes into your project.

Walkthrough: Integrating Partial Rendering into an Existing Project

Please note: this walkthrough creates a Web Site project in Visual C#. Code files are presented in both Visual C# and Visual Basic; however, markup files will only be listed with their C# code-behind references. You may need to adjust @Page directives accordingly.

1.) In Microsoft Visual Studio 2008, create a new ASP.NET Web Site project by going to *File → New → Web Site…* and selecting "ASP.NET Web Site" from the dialog. You can name it whatever you like, and you may install it either to the file system or into Internet Information Services (IIS).

2.) You will be presented with the blank default page with basic ASP.NET markup (a server-side form and an `@Page` directive). Drop a Label called `Label1` and a Button called `Button1` onto the page within the form element. You may set their text properties to whatever you like.

3.) In Design view, double-click `Button1` to generate a code-behind event handler. Within this event handler, set `Label1.Text` to "You clicked the button!".

Listing 1: Markup for default.aspx before partial rendering is enabled

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC
    "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
```

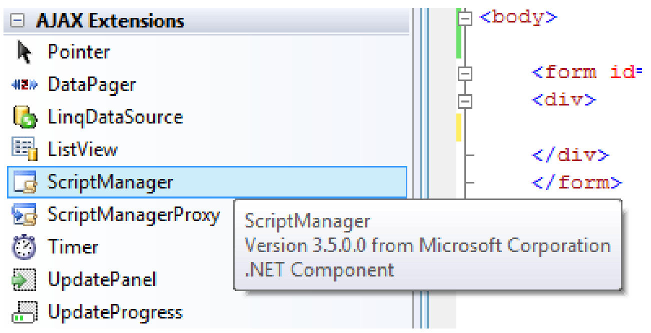
```
    <form id="form1" runat="server">
    <div>
        <asp:Label ID="Label1" runat="server"
            Text="This is a label!"></asp:Label>
        <asp:Button ID="Button1" runat="server"
            Text="Click Me" OnClick="Button1_Click" />
    </div>
    </form>
</body>
</html>
```

Listing 2: Codebehind (trimmed) in default.aspx.cs

```
public partial class _Default : System.Web.UI.Page
{
    protected void Button1_Click(object sender, EventArgs e)
    {
        Label1.Text = "You clicked the button!";
    }
}
```

4.) Press F5 to launch your web site. Visual Studio will prompt you to add a web.config file to enable debugging; do so. When you click the button, notice that the page refreshes to change the text in the label, and there is a brief flicker as the page is redrawn.

5.) After closing your browser window, return to Visual Studio and to the markup page. Scroll down in the Visual Studio toolbox, and find the tab labeled "AJAX Extensions." (If you do not have this tab because you are using an older version of AJAX or Atlas extensions, refer to the walkthrough for registering the AJAX Extensions toolbox items later in this whitepaper, or install the current version with the Windows Installer downloadable from the website).



```
<body>
    <form id=
    <div>
    </div>
    </form>
```

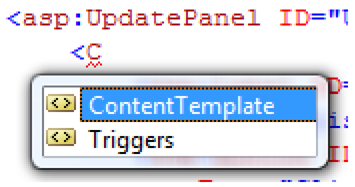
ScriptManager
Version 3.5.0.0 from Microsoft Corporation
.NET Component

a. *Known Issue:* If you install Visual Studio 2008 Beta 2 onto a computer that already has Visual Studio 2005 installed with the ASP.NET 2.0 AJAX Extensions, Visual Studio 2008 will import the "AJAX Extensions" toolbox items. You can determine whether this is the case by examining the tooltip of the components; they should say “Version 3.5.0.0”. If they say “Version 2.0.0.0,” then you have imported your old toolbox items, and will need to manually import them by using the “Choose Toolbox Items” dialog in Visual Studio. You will be unable to add Version 2 controls via the designer.

6.) Before the `<asp:Label>` tag begins, create a line of whitespace, and double-click on the UpdatePanel control in the toolbox. Note that a new `@Register` directive is included at the top of the page, indicating that controls within the System.Web.UI namespace should be imported using the `asp:` prefix.

7.) Drag the closing `</asp:UpdatePanel>` tag past the end of the Button element, so that the element is well-formed with the Label and Button controls wrapped.

8.) After the opening `<asp:UpdatePanel>` tag, begin opening a new tag. Note that IntelliSense prompts you with two options. In this case, create a `<ContentTemplate>` tag. Be sure to wrap this tag around your Label and Button so that the markup is well-formed.



9.) Anywhere within the `<form>` element, include a `ScriptManager` control by double-clicking on the ScriptManager item in the toolbox.

10.) Edit the `<asp:ScriptManager>` tag so that it includes the attribute `EnablePartialRendering=”true”`.

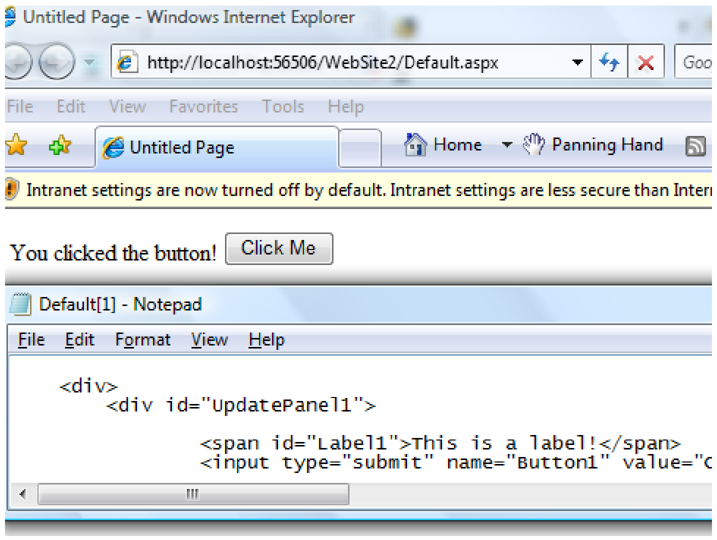
Listing 3: Markup for default.aspx with partial rendering enabled

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
<%@ Register Assembly="System.Web.Extensions, Version=1.0.61025.0,
   Culture=neutral, PublicKeyToken=31bf3856ad364e35"
    Namespace="System.Web.UI" TagPrefix="asp" %>

<!DOCTYPE html PUBLIC
    "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
```

```
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <asp:ScriptManager EnablePartialRendering="true"
    ID="ScriptManager1" runat="server"></asp:ScriptManager>
    <div>
        <asp:UpdatePanel ID="UpdatePanel1" runat="server">
            <ContentTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text="This is a label!"></asp:Label>
                <asp:Button ID="Button1" runat="server"
                    Text="Click Me" OnClick="Button1_Click" />
            </ContentTemplate>
        </asp:UpdatePanel>
    </div>
    </form>
</body>
</html>
```

11.) Open your web.config file. Notice that Visual Studio has automatically added a compilation reference to System.Web.Extensions.dll.

a. What's New in Visual Studio 2008: The web.config that comes with the ASP.NET Web Site project templates automatically includes all necessary references to the ASP.NET AJAX Extensions, and includes commented sections of configuration information that can be un-commented to enable additional functionality. Visual Studio 2005 had similar templates when ASP.NET 2.0 AJAX Extensions were installed. However, in Visual Studio 2008, the AJAX Extensions are opt-out by default (that is, they are referenced by default, but can be removed as references).


Untitled Page - Windows Internet Explorer
http://localhost:56506/WebSite2/Default.aspx
File Edit View Favorites Tools Help
Untitled Page
Home
Panning Hand
Intranet settings are now turned off by default. Intranet settings are less secure than Inter
Click Me
Default[1] - Notepad
File Edit Format View Help

You clicked the button!

```
    <div>
        <div id="UpdatePanel1">

                <span id="Label1">This is a label!</span>
                <input type="submit" name="Button1" value="C
```

12.) Press F5 to launch your website. Note how no source code changes were required to support partial rendering – only markup was changed.

When you launch your website, you should see that partial rendering is now enabled, because when you click on the button there will be no flicker, nor will there be any change in the page scroll position (this example does not demonstrate that). If you were to look at the rendered source of the page after clicking the button, it will confirm that in fact a post-back has not occurred – the original label text is still part of the source markup, and the label has changed through JavaScript.

Visual Studio 2008 Beta 2 does not appear to come with a pre-defined template for an ASP.NET AJAX-Enabled web site. However, such a template was available within Visual Studio 2005 if the Visual Studio 2005 and ASP.NET 2.0 AJAX Extensions were installed. Consequently, configuring a web site and starting with the AJAX-Enabled Web Site template will likely be even easier, as the template should include a fully-configured web.config file (supporting all of the ASP.NET AJAX Extensions, including Web Services access and JSON serialization – JavaScript Object Notation) and includes an UpdatePanel and ContentTemplate within the main Web Forms page by default. Enabling partial rendering with this default page is as simple as revisiting Step 10 of this walkthrough and dropping controls onto the page.

The ScriptManager Control

ScriptManager Control Reference

Markup-Enabled Properties:

Property Name	**Type**	**Description**
AllowCustomErrors-Redirect	Bool	Specifies whether to use the custom error section of the web.config file to handle errors.
AsyncPostBackError-Message	String	Gets or sets the error message sent to the client if an error is raised.
AsyncPostBack-Timeout	Int32	Gets or sets the default amount of a time a client should wait for the asynchronous request to complete.
EnableScript-	Bool	Gets or sets whether script globalization

Globalization		is enabled.
EnableScript-Localization	Bool	Gets or sets whether script localization is enabled.
ScriptLoadTimeout	Int32	Determines the number of seconds allowed for loading scripts into the client
ScriptMode	Enum (Auto, Debug, Release, Inherit)	Gets or sets whether to render release versions of scripts
ScriptPath	String	Gets or sets the root path to the location of script files to be sent to the client.

Code-Only Properties:

Property Name	**Type**	**Description**
AuthenticationService	AuthenticationService-Manager	Gets details about the ASP.NET Authentication Service proxy that will be sent to the client.
IsDebuggingEnabled	Bool	Gets whether script and code debugging is enabled.
IsInAsyncPostback	Bool	Gets whether the page is currently in an asynchronous post back request.
ProfileService	ProfileService-Manager	Gets details about the ASP.NET Profiling Service proxy that will be sent to the client.
Scripts	Collection<Script-Reference>	Gets a collection of script references that will be sent to the client.
Services	Collection<Service-Reference>	Gets a collection of Web Service proxy references that will be sent to the client.
SupportsPartialRendering	Bool	Gets whether the current client supports partial rendering. If this property returns **false**, then all page requests will be standard postbacks.

Public Code Methods:

Method Name	Type	Description
SetFocus(string)	Void	Sets the focus of the client to a particular control when the request has completed.

Markup Descendants:

Tag	Description
<AuthenticationService>	Provides details about the proxy to the ASP.NET authentication service.
<ProfileService>	Provides details about the proxy to the ASP.NET profiling service.
<Scripts>	Provides additional script references.
<asp:ScriptReference>	Denotes a specific script reference.
<Service>	Provides additional Web Service references which will have proxy classes generated.
<asp:ServiceReference>	Denotes a specific Web Service reference.

The ScriptManager control is the essential core for the ASP.NET AJAX Extensions. It provides access to the script library (including the extensive client-side script type system), supports partial rendering, and provides extensive support for additional ASP.NET services (such as authentication and profiling, but also other Web Services). The ScriptManager control also provides globalization and localization support for the client scripts.

Providing Alterative and Supplemental Scripts

While the Microsoft ASP.NET 2.0 AJAX Extensions include the entire script code in both debug and release editions as resources embedded in the referenced assemblies, developers are free to redirect the ScriptManager to customized script files, as well as register additional necessary scripts.

To override the default binding for the typically-included scripts (such as those which support the Sys.WebForms namespace and the custom typing system), you can register for the `ResolveScriptReference` event of the ScriptManager class. When this method is called, the event handler has the opportunity to change the path to the script file in question; the script manager will then send a different or customized copy of the scripts to the client.

Additionally, script references (represented by the `ScriptReference` class) can be included programmatically or via markup. To do so, either programmatically modify the

`ScriptManager.Scripts` collection, or include `<asp:ScriptReference>` tags under the `<Scripts>` tag, which is a first-level child of the ScriptManager control.

Custom Error Handling for UpdatePanels

Although updates are handled by triggers specified by UpdatePanel controls, the support for error handling and custom error messages is handled by a page's ScriptManager control instance. This is done by exposing an event, `AsyncPostBackError`, to the page which can then provide custom exception-handling logic.

By consuming the AsyncPostBackError event, you may specify the `AsyncPostBackErrorMessage` property, which then causes an alert box to be raised upon completion of the callback.

Client-side customization is also possible instead of using the default alert box; for instance, you may want to display a customized `<div>` element rather than the default browser modal dialog. In this case, you can handle the error in client script:

Listing 5: Client-side script to display custom errors

```
<script type=”text/javascript”>
<!--
Sys.WebForms.PageRequestManager.getInstance().add_EndRequest(
   Request_End);
function Request_End(sender, args)
{
   if (args.get_error() != undefined)
   {
         var errorMessage = “”;
         if (args.get_response().get_statusCode() == “200”)
         {
               errorMessage = args.get_error().message;
         }
         else
         {
               // the server wasn’t the problem...
               errorMessage = “An unknown error occurred...”;
         }
         // do something with the errorMessage here.

         // now make sure the system knows we handled the error.
         args.set_errorHandled(true);
   }
}
// -->
</script>
```

Quite simply, the above script registers a callback with the client-side AJAX runtime for when the asynchronous request has been completed. It then checks to see whether an error was reported, and if so, processes the details of it, finally indicating to the runtime that the error was handled in custom script.

Globalization and Localization Support

The ScriptManager control provides extensive support for localization of script strings and user interface components; however, that topic is outside of the scope of this whitepaper. For more information, see the whitepaper, "Globalization Support in ASP.NET AJAX Extensions."

The UpdatePanel Control

UpdatePanel Control Reference

Markup-Enabled Properties:

Property Name	**Type**	**Description**
ChildrenAsTriggers	bool	Specifies whether child controls automatically invoke refresh on postback.
RenderMode	enum (Block, Inline)	Specifies the way the content will be visually presented.
UpdateMode	enum (Always, Conditional)	Specifies whether the UpdatePanel is always refreshed during a partial render or if it is only refreshed when a trigger is hit.

Code-Only Properties:

Property Name	**Type**	**Description**
IsInPartialRendering	bool	Gets whether the UpdatePanel is supporting partial rendering for the current request.
ContentTemplate	ITemplate	Gets the markup template for the update request.
ContentTemplateContainer	Control	Gets the programmatic template for the update request.
Triggers	UpdatePanel-TriggerCollection	Gets the list of triggers associated with the current UpdatePanel.

Public Code Methods:

Method Name	**Type**	**Description**
Update()	Void	Updates the specified UpdatePanel programmatically. Allows a server request to trigger a partial render of an otherwise-untriggered UpdatePanel.

Markup Descendants:

Tag	**Description**
<ContentTemplate>	Specifies the markup to be used to render the partial rendering result. Child of <asp:UpdatePanel>.
<Triggers>	Specifies a collection of *n* controls associated with updating this UpdatePanel. Child of <asp:UpdatePanel>.
<asp:AsyncPostBackTrigger>	Specifies a trigger that invokes a partial page render for the given UpdatePanel. This may or may not be a control as a descendant of the UpdatePanel in question. Granular to the event name. Child of <Triggers>.
<asp:PostBackTrigger>	Specifies a control that causes the entire page to refresh. This may or may not be a control as a descendant of the UpdatePanel in question. Granular to the object. Child of <Triggers>.

The `UpdatePanel` control is the control that delimits the server-side content that will take part in the partial rendering functionality of the AJAX Extensions. There is no limit to the number of UpdatePanel controls that can be on a page, and they can be nested. Each UpdatePanel is isolated, so that each can work independently (you can have two UpdatePanels running at the same time, rendering different parts of the page, independent of the page's postback).

The UpdatePanel control primarily deals with control triggers – by default, any control contained within an UpdatePanel's `ContentTemplate` that creates a postback is registered as a trigger for the UpdatePanel. This means that the UpdatePanel can work with the default data-bound controls (such as the GridView), with user controls, and they can be programmed in script.

By default, when a partial page render is triggered, all UpdatePanel controls on the page will be refreshed, whether or not the UpdatePanel controls defined triggers for such action. For example, if one UpdatePanel defines a Button control, and that Button control is clicked, all UpdatePanel controls on that page will be refreshed by default. This is because, by default, the `UpdateMode` property of the UpdatePanel is set to `Always`. Alternatively, you may set the UpdateMode property to `Conditional`, which means that the UpdatePanel will only be refreshed if a specific trigger is hit.

Custom Control Notes

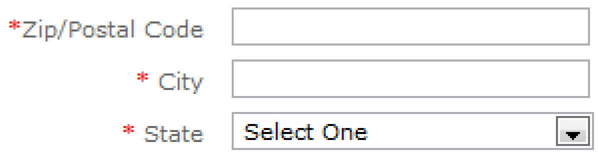
An UpdatePanel can be added to any user control or custom control; however, the page on which these controls are included must also include a ScriptManager control with the property EnablePartialRendering set to **true**.

One way in which you might account for this when using Web Custom Controls is to override the protected `CreateChildControls()` method of the `CompositeControl` class. By doing so, you can inject an UpdatePanel between the control's children and the outside world if you determine the page supports partial rendering; otherwise, you can simply layer the child controls into a container `Control` instance.

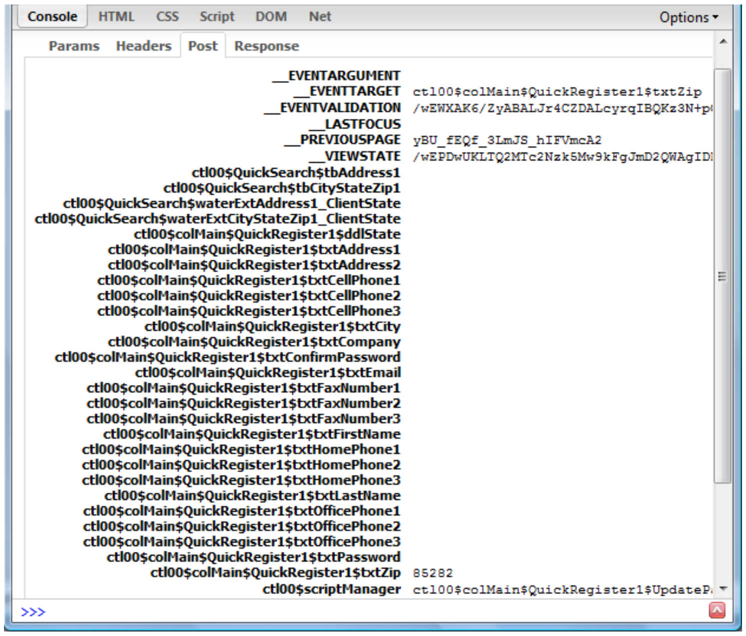
UpdatePanel Considerations

The UpdatePanel operates as something of a black-box, wrapping ASP.NET postbacks within the context of a JavaScript XMLHttpRequest. However, there are significant performance considerations to bear in mind, both in terms of behavior and speed. To understand how the UpdatePanel works, so that you can best decide when its use is appropriate, you should examine the AJAX exchange. The following example uses an existing site and, Mozilla Firefox with the Firebug extension (Firebug captures XMLHttpRequest data).

Consider a form that, among other things, has a postal code textbox which is supposed to populate a city and state field on a form or control. This form ultimately collects membership information, including a user's name, address, and contact information. There are many design considerations to take into account, based on the requirements of a specific project.



*Zip/Postal Code	
* City	
* State	Select One



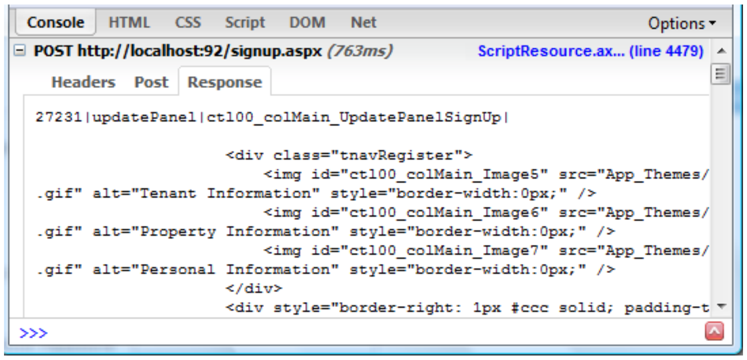
__EVENTARGUMENT	
__EVENTTARGET	ctl00$colMain$QuickRegister1$txtZip
__EVENTVALIDATION	/wEWXAK6/ZyABALJr4CZDALcyrqIBQKz3N+p
__LASTFOCUS	
__PREVIOUSPAGE	yBU_fEQf_3LmJS_hIFVmcA2
__VIEWSTATE	/wEPDwUKLTQ2MTc2Nzk5Mw9kFgJmD2QWAgID
ctl00$QuickSearch$tbAddress1	
ctl00$QuickSearch$tbCityStateZip1	
ctl00$QuickSearch$waterExtAddress1_ClientState	
ctl00$QuickSearch$waterExtCityStateZip1_ClientState	
ctl00$colMain$QuickRegister1$ddlState	
ctl00$colMain$QuickRegister1$txtAddress1	
ctl00$colMain$QuickRegister1$txtAddress2	
ctl00$colMain$QuickRegister1$txtCellPhone1	
ctl00$colMain$QuickRegister1$txtCellPhone2	
ctl00$colMain$QuickRegister1$txtCellPhone3	
ctl00$colMain$QuickRegister1$txtCity	
ctl00$colMain$QuickRegister1$txtCompany	
ctl00$colMain$QuickRegister1$txtConfirmPassword	
ctl00$colMain$QuickRegister1$txtEmail	
ctl00$colMain$QuickRegister1$txtFaxNumber1	
ctl00$colMain$QuickRegister1$txtFaxNumber2	
ctl00$colMain$QuickRegister1$txtFaxNumber3	
ctl00$colMain$QuickRegister1$txtFirstName	
ctl00$colMain$QuickRegister1$txtHomePhone1	
ctl00$colMain$QuickRegister1$txtHomePhone2	
ctl00$colMain$QuickRegister1$txtHomePhone3	
ctl00$colMain$QuickRegister1$txtLastName	
ctl00$colMain$QuickRegister1$txtOfficePhone1	
ctl00$colMain$QuickRegister1$txtOfficePhone2	
ctl00$colMain$QuickRegister1$txtOfficePhone3	
ctl00$colMain$QuickRegister1$txtPassword	
ctl00$colMain$QuickRegister1$txtZip	85282
ctl00$scriptManager	ctl00$colMain$QuickRegister1$UpdateP

In the original iteration of this application, a control was built that incorporated the entirety of the user registration data, including the postal code, city, and state. The entire control was wrapped within an UpdatePanel and dropped onto a Web Form. When the postal code is entered by the user, the UpdatePanel detects the event (the corresponding TextChanged event in the back-end, either by specifying triggers or by using the ChildrenAsTriggers property set to true). AJAX posts all of the fields within the UpdatePanel, as captured by FireBug (see the diagram on the right).

As the screen capture indicates, values from every control within the UpdatePanel are delivered (in this case, they are all empty), as well as the ViewState field. All told, over 9kb of data is sent, when in fact only five bytes of data were needed to make this particular request. The response is even more bloated: in total, 57kb is sent to the client, simply to update a text field and a drop-down field.

It may also be of interest to see how ASP.NET AJAX updates the presentation. The response portion of the UpdatePanel's update request is shown in the Firebug console display on the left; it is a specially-formulated pipe-delimited string that is broken up by the client script and then reassembled on the page. Specifically, ASP.NET AJAX sets the *innerHTML* property of the HTML element on the client that represents your UpdatePanel. As the browser re-generates the DOM, there is a slight delay, depending on the amount of information that needs to be processed.

The regeneration of the DOM triggers a number of additional issues:



```
POST http://localhost:92/signup.aspx (763ms)        ScriptResource.ax... (line 4479)

27231|updatePanel|ctl00_colMain_UpdatePanelSignUp|

                    <div class="tnavRegister">
                        <img id="ctl00_colMain_Image5" src="App_Themes/
.gif" alt="Tenant Information" style="border-width:0px;" />
                        <img id="ctl00_colMain_Image6" src="App_Themes/
.gif" alt="Property Information" style="border-width:0px;" />
                        <img id="ctl00_colMain_Image7" src="App_Themes/
.gif" alt="Personal Information" style="border-width:0px;" />
                    </div>
                    <div style="border-right: 1px #ccc solid; padding-t
>>>
```

- If the focused HTML element is within the UpdatePanel, it will lose focus. So, for users who pressed the Tab key to exit the postal code text box, their next destination would have been the City text box. However, once the UpdatePanel refreshed the display, the form would no longer have had focus, and pressing Tab would have started highlighting the focus elements (such as links).
- If any type of custom client-side script is in use that accesses DOM elements, references persisted by functions may become defunct after a partial postback.

UpdatePanels are not intended to be catch-all solutions. Rather, they provide a quick solution for certain situations, including prototyping, small control updates, and provide a familiar interface to ASP.NET developers who might be familiar with the .NET object model but less-so with the DOM. There are a number of alternatives that may result in better performance, depending on the application scenario:

- Consider using PageMethods and JSON (JavaScript Object Notation) allows the developer to invoke static methods on a page as if a web service call was being invoked. Because the methods are static, no state is necessary; the script caller supplies the parameters, and the result is returned asynchronously.
- Consider using a Web Service and JSON if a single control needs to be used in several places across an application. This again requires very little special work, and works asynchronously.

Incorporating functionality through Web Services or Page Methods has drawbacks as well. First and foremost, ASP.NET developers typically tend to build small components of functionality into user controls (.ascx files). Page methods cannot be hosted in these files; they must be hosted within the actual .aspx page class. Web services, similarly, must be hosted within the .asmx class. Depending on the application, this architecture may violate the Single Responsibility Principle, in that the functionality for a single component is now spread across two or more physical components which may have little or no cohesive ties.

Finally, if an application requires that UpdatePanels are used, the following guidelines should assist with troubleshooting and maintenance.

- **Nest UpdatePanels as little as possible, not only within-units, but also across units of code.** For example, having an UpdatePanel on a Page that wraps a Control, while that Control also contains an UpdatePanel, which contains another Control that contains an UpdatePanel, is cross-unit nesting. This helps to keep clear which elements should be refreshing, and prevents unexpected refreshes to child UpdatePanels.
- **Keep the *ChildrenAsTriggers* property set to false, and explicitly set triggering events.** Utilizing the `<Triggers>` collection is a much clearer way to handle events, and may prevent unexpected behavior, helping with maintenance tasks and forcing a developer to "opt-in" for an event.
- **Use the smallest possible unit to achieve functionality.** As noted in the discussion of the postal code service, wrapping only the bare minimum reduces time to the server, total processing, and the footprint of the client-server exchange, enhancing performance.

The UpdateProgress Control

UpdateProgress Control Reference

Markup-Enabled Properties:

Property Name	**Type**	**Description**
AssociatedUpdate-PanelID	String	Specifies the ID of the UpdatePanel that this UpdateProgress should report on.
DisplayAfter	Int	Specifies the timeout in milliseconds before this control is displayed after the asynchronous request begins.
DynamicLayout	bool	Specifies whether the progress is rendered dynamically.

Markup Descendants:

Tag	**Description**
<ProgressTemplate>	Contains the control template set for the content that will be displayed with this control.

The UpdateProgress control gives you a measure of feedback to keep your users' interest while doing the necessary work to transport to the server. This can help your users know that you're doing something even though it may not be apparent, especially since most users are used to the page refreshing and seeing the status bar highlight.

As a note, UpdateProgress controls can appear anywhere on a page hierarchy. However, in cases in which a partial postback is initiated from a child UpdatePanel (where an UpdatePanel is nested within another UpdatePanel), postbacks that trigger the child UpdatePanel will cause UpdateProgress templates to be displayed for the child UpdatePanel as well as the parent UpdatePanel. But, if the trigger is a direct child of the parent UpdatePanel, then only the UpdateProgress templates associated with the parent will be displayed.

Summary

The Microsoft ASP.NET AJAX extensions are sophisticated products designed to assist in making web content more accessible and to provide a richer user experience to your web applications. As part of the ASP.NET AJAX Extensions, the partial page rendering controls, including the ScriptManager, the UpdatePanel, and UpdateProgress controls are some of the most visible components of the toolkit.

The ScriptManager component integrates the provision of client JavaScript for the Extensions, as well as enables the various server- and client-side components to work together with minimal development investment.

The UpdatePanel control is the apparent magic box – markup within the UpdatePanel can have server-side Codebehind and not trigger a page refresh. UpdatePanel controls can be nested, and can be dependent on controls in other UpdatePanels. By default, UpdatePanels handle any postbacks invoked by their descendant controls, although this functionality can be finely tuned, either declaratively or programmatically.

When using the UpdatePanel control, developers should be aware of the performance impact that could potentially arise. Potential alternatives include web services and page methods, though the design of the application should be considered.

The UpdateProgress control allows the user to know that she or he is not being ignored, and that the behind-the-scenes request is going on while the page is otherwise not doing anything to respond to the user input. It also includes the ability to abort partial rendering results.

Together, these tools assist creating a rich and seamless user experience by making server work less apparent to the user and interrupting workflow less.

Bio

Scott Cate has been working with Microsoft Web technologies since 1997 and is the President of myKB.com (www.myKB.com) where he specializes in writing ASP.NET based applications focused on Knowledge Base Software solutions. Scott can be contacted via email at scott.cate@myKB.com or his blog at http://weblogs.asp.net/scottcate