ASP.NET AJAX 4.0

New AJAX Support For Data-Driven Web Apps

Bertrand Le Roy - October 2008

CODE DOWNLOAD AVAILABLE AT: MSDN Code Gallery **(188 KB)**
Browse the Code Online

This article is based on prerelease versions of ASP.NET. All information herein is subject to change.

THIS ARTICLE DISCUSSES:

- Server-side data manipulation
- UpdatePanel and the client
- Reducing postbacks and payloads
- Client-side template rendering

THIS ARTICLE USES THE FOLLOWING TECHNOLOGIES:
ASP.NET AJAX 4.0

Contents

AJAX is an exciting Web platform for many reasons. Using AJAX, many tasks that were traditionally performed on the server happen in the browser instead, resulting in fewer round-trips to the server, lower bandwidth consumption, and faster, more responsive Web UIs. While these outcomes are the result of offloading a good deal of work to the client, the browser still isn't the environment of choice for many developers who would rather have the full power and flexibility of server apps at their disposal.

The solution employed until now has involved the UpdatePanel control, which has allowed developers to build AJAX applications while still retaining the full array of server tools. But UpdatePanel carries a lot of weight from the traditional postback model—an UpdatePanel request is still a full postback. In fact, using UpdatePanel, the whole form (including ViewState) is posted to the server, almost the entire page lifecycle gets executed there, and the rendering still happens on the server. Obviously, this method defeats one of the main reasons for moving to AJAX. The only real savings here are that XmlHttpRequest is used instead of a regular HTTP POST request and only updated parts of the page and the ViewState are sent back to the client. Thus, the response is much smaller, but the request isn't.

A pure AJAX approach will almost always perform better than the UpdatePanel approach. In a purely AJAX solution, the rendering happens on the client and the server sends back only data, which is usually much smaller than the equivalent HTML. This approach can also substantially reduce the number of network requests: having the data on the client allows much of the application's UI logic to run in the browser.

The main problem with the pure AJAX approach, however, is that the browser lacks the tools to turn data into HTML. Out of the box, it has only two crude facilities for doing so: innerHTML, which replaces the full contents of an element with the HTML string you provide, and the somewhat slower Document Object Model (DOM) APIs that operate on tags and attributes (similar in terms of abstraction level to HtmlTextWriter).

In this article, I'll show three iterations of a page written with classic postback, then with UpdatePanel, and then using pure AJAX to illustrate how techniques employed on the server can sometimes perform better on the client. The first two examples can be built today with the publicly available ASP.NET 3.5 SP1, while the third version will use some of the new client features in ASP.NET 4.0.

A Postback-Based Master-Details Page

The page I'll be building will show products in a list which, when selected, will display the detailed description of that product in a panel to the right of the list. I'll use the AdventureWorks sample database, which you can download from go.microsoft.com/fwlink/?LinkId=124953. I'll create only a rudimentary data layer using LINQ to SQL because data layers are not the focus of this article.

I'll start by adding the AdventureWorks .mdf file into the App_Data folder of the application. Then, I'll simply add a new "LINQ to SQL classes" .dbml file and drop the Product, ProductPhoto, and ProductProductPhoto tables and the vProductModelCatalogDescription view from the server explorer onto the design surface. The resulting data layer can be seen in **Figure 1**.

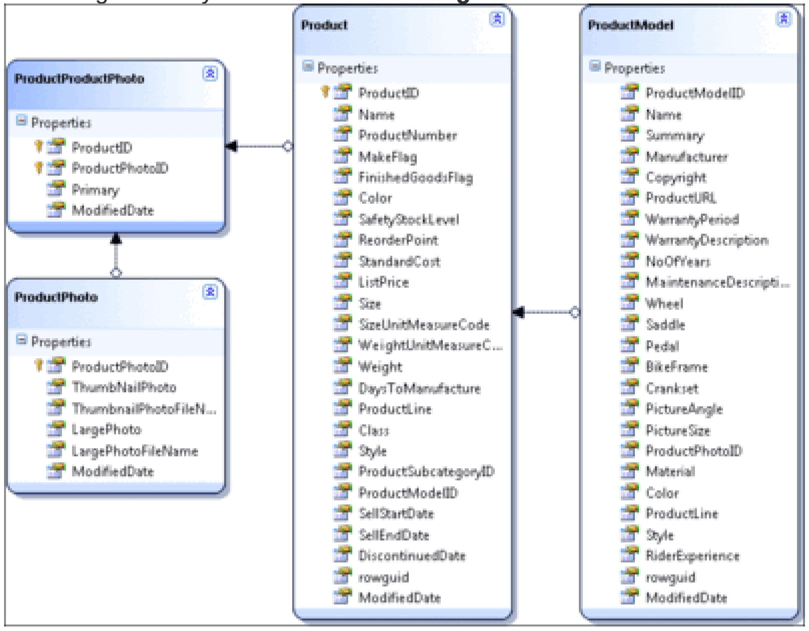
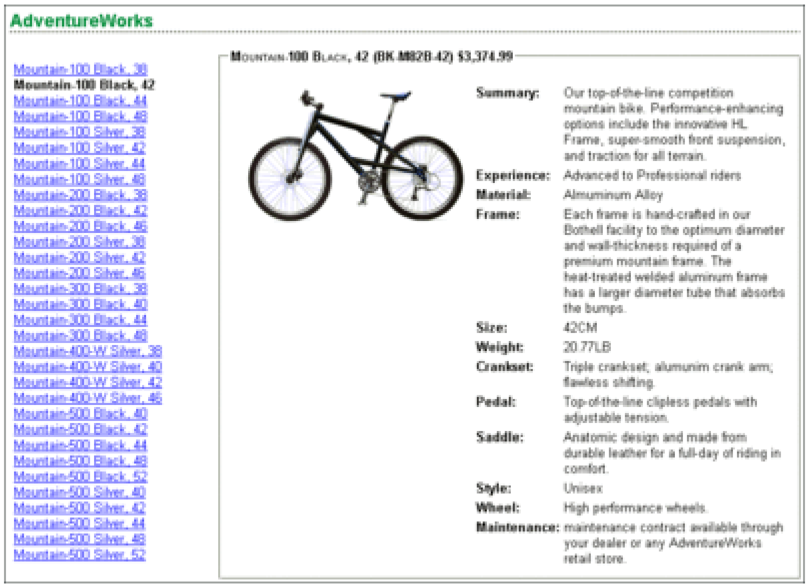


Figure 1 **The Data Layer Architecture** (Click the image for a larger view)

The page will consist of two view panes, one for the list of products and the other one for the product details. **Figure 2** shows the rendered page. On the first request to the page, the product list is bound to data using this code:

Copy Code

```
private void BindProductList() {
    ProductList.DataSource = from p in AdventureWorksContext.Products
                             where p.ProductSubcategoryID == 1
                             //Mountain bikes
                             orderby p.Name
                             select p;
    ProductList.DataBind();
}
```


AdventureWorks

- Mountain-100 Black, 38
- Mountain-100 Black, 42
- Mountain-100 Black, 44
- Mountain-100 Black, 48
- Mountain-100 Silver, 38
- Mountain-100 Silver, 42
- Mountain-100 Silver, 44
- Mountain-100 Silver, 48
- Mountain-200 Black, 38
- Mountain-200 Black, 42
- Mountain-200 Black, 46
- Mountain-200 Silver, 38
- Mountain-200 Silver, 42
- Mountain-200 Silver, 46
- Mountain-300 Black, 38
- Mountain-300 Black, 40
- Mountain-300 Black, 44
- Mountain-300 Black, 48
- Mountain-400-W Silver, 38
- Mountain-400-W Silver, 40
- Mountain-400-W Silver, 42
- Mountain-400-W Silver, 46
- Mountain-500 Black, 40
- Mountain-500 Black, 42
- Mountain-500 Black, 44
- Mountain-500 Black, 48
- Mountain-500 Black, 52
- Mountain-500 Silver, 40
- Mountain-500 Silver, 42
- Mountain-500 Silver, 44
- Mountain-500 Silver, 48
- Mountain-500 Silver, 52

MOUNTAIN-100 BLACK, 42 (BK-M82B-42) $3,374.99

Summary:	Our top-of-the-line competition mountain bike. Performance-enhancing options include the innovative HL Frame, super-smooth front suspension, and traction for all terrain.
Experience:	Advanced to Professional riders
Material:	Almuminum Alloy
Frame:	Each frame is hand-crafted in our Bothell facility to the optimum diameter and wall-thickness required of a premium mountain frame. The heat-treated welded aluminum frame has a larger diameter tube that absorbs the bumps.
Size:	42CM
Weight:	20.77LB
Crankset:	Triple crankset; alumunim crank arm; flawless shifting.
Pedal:	Top-of-the-line clipless pedals with adjustable tension.
Saddle:	Anatomic design and made from durable leather for a full-day of riding in comfort.
Style:	Unisex
Wheel:	High performance wheels.
Maintenance:	maintenance contract available through your dealer or any AdventureWorks retail store.

Figure 2 **he List of Bikes and Details** (Click the image for a larger view)

The list itself and its template can be seen in **Figure 3**. The code that queries the database can be found in the downloadable project and is fairly straightforward; it just queries the database for products in the Mountain Bike category and binds the ListView control to it. Of course, I could have used a data source control to do that same job but I find the code approach more flexible and predictable. Your results may vary if you are more of a design-view person.

The actual work of creating HTML markup from the dataset is all taken care of by the ListView control, which is very convenient and effortless. All I have to do is provide the template for that HTML in the LayoutTemplate and ItemTemplate properties (see **Figure 3**). This template will render the list of products as links inside an unordered list (UL and LI tags).

Figure 3 Unordered List of Products

Copy Code

```
<asp:ListView ID="ProductList" runat="server"
    DataKeyNames="ProductId"
    OnSelectedIndexChanging="ProductList_SelectedIndexChanging"
    OnSelectedIndexChanged="ProductList_SelectedIndexChanged">
    <LayoutTemplate>
        <ul ID="itemPlaceholderContainer" runat="server">
            <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
        </ul>
    </LayoutTemplate>
    <ItemTemplate>
        <li><asp:LinkButton runat="server" ID="Select" CommandName="Select"
                                          Text='<%# Eval("Name") %>' /></li>
    </ItemTemplate>
</asp:ListView>
```

Note that the links themselves are not plain links but rather LinkButton controls, which means that they will post back to the page rather than navigate to a different one. They are effectively links that have the semantics of buttons. Of course, one could easily improve the accessibility of the page in the absence of JavaScript by replacing those LinkButton controls with appropriately styled regular Button controls.

The key feature of the LinkButtons that I'm using here is that instead of attaching an event handler to each of the buttons, I'm setting the CommandName property to "Select." The result will be that when clicked, the button will bubble the command up the control tree until it is handled by a control that understands it. This is a very powerful feature that enables an arbitrary UI element to send commands to its parent controls without having to know much more than what command and arguments it may expect. That's what allows for powerful data controls such as ListView to still enable complete developer control over the markup. You will see how this translates to similar browser concepts when I build the pure AJAX version of this same page.

At this stage, I have a product list that supports selection without having written any code. One could continue down this no-code path using DataSource controls and ControlParameters to tie the selected data key of the list to the selected data key of the details view, but I chose to do it in code. Next, I'll handle the SelectedIndexChanged event of the list and call the BindProductDetails method with the relevant product ID:

Copy Code

```
protected void ProductList_SelectedIndexChanged(object sender,
                                                 EventArgs e) {
    var productId = (int)ProductList.SelectedDataKey.Value;
    BindProductDetails(productId);
}
```

BindProductDetails queries the database for the product information and photos and then binds these to the corresponding controls in the details view.

The photos are being served by a simple handler that queries the database for the image bytes and copies them onto the response's binary stream (see **Figure 4**). This handler will be used by each of the three versions of the page. I now have a data-driven master-details view of my products, written entirely with a mix of imperative and declarative server code, but there are several things that could be improved.

Figure 4 Get Product Photo

Copy Code

```
public void ProcessRequest (HttpContext context) {
    int id;
    if (int.TryParse(context.Request.QueryString["id"], out id)) {
        context.Response.ContentType = "image/gif";
        AdventureWorksDataContext dc = new AdventureWorksDataContext();
        var bytes = dc.ProductPhotos
            .Where(p => p.ProductPhotoID == id)
            .Single().LargePhoto;
        context.Response.OutputStream.Write(bytes.ToArray(), 0, bytes.Length);
    }
    else {
        throw new HttpException(404, "Image not found");
    }
}
```

This is a very typical Web Forms page in that it is stateful in terms of volume of the state. A quick check on the rendered source in the browser shows about 4KB of ViewState due to the fact that the different views remember all of their internal state and carry it around with every postback.

At the same time, the page does not present clues about what its current state is; no matter what you do on the page, the URL in the navigation bar of the browser remains "1_WebForm.aspx." If a user bookmarks the page, he will always initially see the page without any product detail.

This could be fixed in this simple example by changing the links in the product list to be regular links instead of LinkButton controls and replacing the selection postbacks with plain navigation to a details page (I could even turn ViewState off and save about 4KB twice per round-trip). If you've followed the recent development of the ASP.NET Model View Controller (MVC) library (see msdn.microsoft.com/magazine/cc337884), you've probably guessed that this is a typical case where the MVC approach makes a lot of sense.

Navigation-based approaches also greatly improve the searchability of the site (which could be the subject of an entirely different article). That said, typical data-driven applications will be much more complex than this simple example and plain navigation often isn't the right way to build the UI flow. Therefore, even with this simple application, I will spend time illustrating how postback and Web Forms concepts translate and are improved with AJAX.

Both postback and link navigation might also be more disruptive to the user experience than you would like: during both postback and navigation, the UI is frozen and no other user interaction is possible until the server responds with new content, which then needs to be rendered to replace the whole document, sometimes losing subtle pieces of state, such as the scroll position.

Another problem is that users have very specific expectations about how the Back button and the history should work. Unfortunately, in the postback model, you have little or no control over what gets into the history or what happens when the user presses Back, Forward, or Refresh. Ideally, what constitutes a change of state and creates a history point should be in the hands of the developer, but in a postback application, almost any user interaction will create an entry in browser history.

The UpdatePanel Version

An easy way to improve this page is to use an UpdatePanel. An UpdatePanel will allow you to delimit the parts of the page that change when user actions would normally cause a postback. In this very simple case, the region of the page that I'll want to update is the details view. To enable UpdatePanel partial updates, I just need to add a ScriptManager control to the page, right after the form tag, like so:

Copy Code

```
<asp:ScriptManager ID="ScriptManager1" runat="server"/>
```

I also must add the UpdatePanel itself, around the details view:

Copy Code

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server" RenderMode="Inline">
    <Triggers>
        <asp:AsyncPostBackTrigger ControlID="ProductList"
            EventName="SelectedIndexChanged" />
    </Triggers>
    <ContentTemplate>
        <div class="float" id="productDetails">
            <fieldset>
            ...
            </fieldset>
        </div>
    </ContentTemplate>
</asp:UpdatePanel>
```

Notice that this UpdatePanel has a trigger that watches the SelectedIndexChanged event of the product list. This isn't necessary when all controls that might trigger a partial update are themselves inside the UpdatePanel, but here the product list should remain outside of the UpdatePanel because its rendering doesn't need to be updated when its SelectedIndexChanged event occurs. If I didn't provide the trigger, a regular postback would occur instead of the partial update. Also, when using UpdatePanel remember to always ID all postback controls that may trigger a partial update. Forgetting to do this can lead the page to use regular postbacks for no apparent reason.

That's all there is to transforming a classic postback Web Form that has the appearance of AJAX. But it doesn't seem like I really have all the features I want. One problem is that I just lost what little support there was for the Back button. Now if the user goes back after browsing half a dozen bikes, he's going to go back to whatever site he visited before yours. Pressing the Forward button when he realizes this will take him back to the default state of the application (in this case, a list of products where none is selected).

Fortunately, ASP.NET 3.5 SP1 provides a simple way to return Back button support to the page. ScriptManager now has a very convenient EnableHistory property, an AddHistoryPoint method, and a Navigate event that together enable the application developer to control browser history far beyond anything regular postbacks allowed. This feature not only brings back what was lost by using UpdatePanel, but does so in a much more powerful form.

The big difference from regular postbacks is that I can now decide exactly what constitutes a change in the state of the application and filter out any user interaction I deem less important. I can also gain "bookmarkability" and ensure readable and meaningful entries in the browser's history dropdown.

In order to add history management to the page, I need to determine what information one would expect to be preserved when using a bookmark. Here, there's only one relevant piece of information that should enter state: the currently selected product's ID.

I'll need to intercept any event that is going to change that state. Actually, the only event I need to handle is the same one I used as a trigger earlier: the SelectedItemChanged event on the product list. I'm already handling that event to re-bind the details view, so I'll just add some code to create a new history point every time the event is raised:

Copy Code

```
protected void ProductList_SelectedIndexChanged(object sender,
                                                       EventArgs e) {
    var productId = (int)ProductList.SelectedDataKey.Value;
    var product = BindProductDetails(productId);
    if (ScriptManager1.IsInAsyncPostBack
                                    && !ScriptManager1.IsNavigating) {
         ScriptManager1.AddHistoryPoint("product",
           productId.ToString(), "AdventureWorks - " + product.Name);
    }
}
```

To ensure that the event wasn't raised as a result of the user navigating back to a previous state of the application, the code checks that the request is a part of an asynchronous postback and that it is not part of a navigation operation. If I didn't perform this check, I would create a new history point that would overwrite any forward history that may exist on the browser.

Once this check has been done, it is safe to call the AddHistoryPoint method, passing in the single piece of state that I care about, the product ID, with the parameter name "product." This name is what's going to be used in the modified URL, as you'll see. The value itself needs to be transformed into a string. Think of the history state as another form of query string. The last piece of information I'm giving the method is the document title. This is a great opportunity to make the user experience better as the user will be able to see meaningful information in the history navigation dropdown that will help him navigate the application (see **Figure 5**).

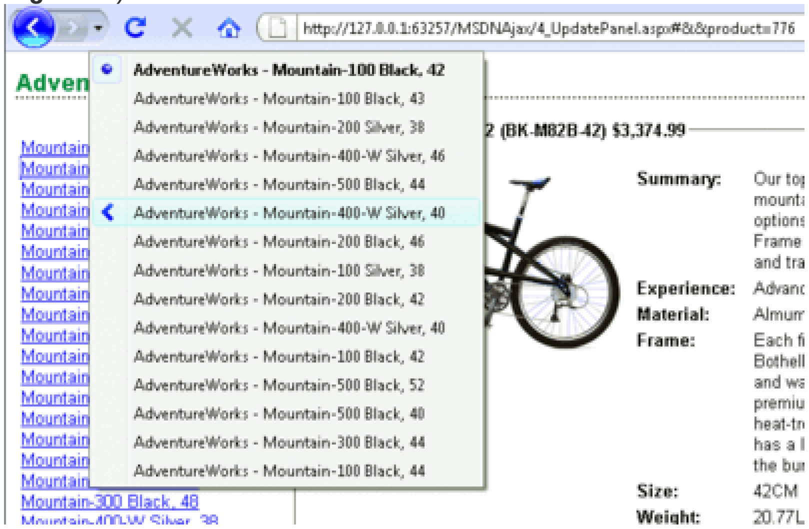

http://127.0.0.1:63257/MSDNAjax/4_UpdatePanel.aspx#&&product=776
Adven
AdventureWorks - Mountain-100 Black, 42
AdventureWorks - Mountain-100 Black, 43
AdventureWorks - Mountain-200 Silver, 38
AdventureWorks - Mountain-400-W Silver, 46
AdventureWorks - Mountain-500 Black, 44
AdventureWorks - Mountain-400-W Silver, 40
AdventureWorks - Mountain-200 Black, 46
AdventureWorks - Mountain-100 Silver, 38
AdventureWorks - Mountain-200 Black, 42
AdventureWorks - Mountain-400-W Silver, 40
AdventureWorks - Mountain-100 Black, 42
AdventureWorks - Mountain-500 Black, 52
AdventureWorks - Mountain-500 Black, 40
AdventureWorks - Mountain-300 Black, 44
AdventureWorks - Mountain-100 Black, 44
Mountain-300 Black, 48
2 (BK-M82B-42) $3,374.99
Summary:
Experience:
Material:
Frame:
Size:
42CM
Weight:
20.77L

Figure 5 **The History Dropdown** (Click the image for a larger view)

The way this state gets persisted is through the browser URL's hash (the part after the # sign that was originally designed to enable intra-document navigation). The reason for using that as the storage medium is that it's the only thing that enables adding a history entry without actually navigating away from the page and its JavaScript and DOM states (because the browser will only allow the addition of another history item if the URL changed). With that comes the constraint that you're storing state in the URL, which has limited space available. Some browsers may reject URLs larger than 1KB. If you need more than that, it may mean that you didn't select the relevant pieces of information as the state and you'll need to do some refactoring. Notice how I use the ID of the product, which is a relatively small piece of data, and not its full name, which may be more friendly but will usually be larger. Insufficient space in the URL may also indicate that regular Web Forms and ViewState may be more appropriate for your design than AJAX and history would be.

The second half of the puzzle is that the state that I just saved needs to be restored when history is navigated. I'll do that by handling the Navigate event on ScriptManager (see **Figure 6**). In the code, I first handle the case where there is no state. This is to return to the default state of the page when going all the way back to the GET request. This may seem a little unusual if you don't know that the state is actually restored by doing a new postback. In this case, the "before" state of the postback, the one that gets restored by the framework automatically, is the "after" state in the browser history chronology, so I have to erase that restored state and substitute it with the default state.

Figure 6 Handling the Navigate Event

Copy Code

```
protected void ScriptManager_Navigate(object sender, HistoryEventArgs e) {
    var productIdString = e.State["product"];
    if (productIdString == null) {
        ProductList.SelectedIndex = -1;
        ProductDetails.DataSource = null;
        ProductDetails.DataBind();
        ProductModelDetails.DataSource = null;
        ProductModelDetails.DataBind();
        ProductPhotoList.DataSource = null;
        ProductPhotoList.DataBind();
        Page.Title = "AdventureWorks";
    }
    else {
        var productId = int.Parse(productIdString);
        var product = BindProductDetails(productId);
        ProductList.SelectedIndex = (
            from p in AdventureWorksContext.Products
            where p.ProductSubcategoryID == 1 // Mountain bikes
            orderby p.Name
            select p).ToList().IndexOf(product);
        BindProductList();
        Page.Title = "AdventureWorks - " + product.Name;
    }
}
```

Second, the state itself should be considered user-provided and should thus be validated, which I'm doing by parsing it as an integer. And finally, I'm restoring the title of the page when restoring the state, in addition to resetting the list and details states.

If you use the page after those changes, the URL in the browser will change every time a product is selected and will look something like this:

Copy Code

```
http://MyServer/MSDNAjax/2_UpdatePanel.aspx#&&5YLQHC81D2
OEdJU/9ZBdHUip1qx3ooPKDhCLgKogupQ=
```

It's ugly and not very readable, right? This is because the framework considers user-provided data dangerous by default, so it hashes the state in order to prevent tampering. However, in many cases, the developer will prefer relatively readable, less scary URLs, even if that means validating the state from code and allowing the user to tamper with it. Tamperable URLs are even considered a plus in some situations (the MSDN library is a good example of this; it allows users to build their own URLs, such as msdn.microsoft.com/library/system.web.ui.scriptmanager.aspx, from a predictable scheme because it makes navigation much easier).

In order to enable this type of scenario, ScriptManager exposes the EnableSecureHistoryState Boolean property. Just by setting this to false, my URLs become much friendlier, like this one:

Copy Code

```
http://MyServer/MSDNAjax/2_UpdatePanel.aspx#&&product=776
```

The result is a much more fluid page that isn't just made to look like an AJAX version of the Web Form, but one that has many handy additional features such as bookmarkability and optimal handling of the Back button. And all this was accomplished without writing a single line of JavaScript.

The Pure AJAX Version

While there is a lot to like in the UpdatePanel version of the page, I'm still dragging around the weight of the ViewState. In order to trim it down, I need to transfer more logic to the client. To do so, I'll need to start writing some fun JavaScript.

You could very well write a pure AJAX version with ASP.NET AJAX 3.5 SP1, but it would be tedious to take the data and format it as HTML. There are two basic ways you can turn data into HTML.
The first, which is what most client template engines do, is to concatenate strings alternating static template contents with the dynamic data contents. This seems relatively straightforward and fast since it uses innerHTML as its sole method of interaction with the DOM. It has a few problems though.
One thing it doesn't do well is protect from injection attacks: if you're going to generate HTML by concatenating strings, you'll need to encode all data before using it. Otherwise, introducing a quote into an attribute or a script tag into a text node, maliciously or inadvertently, might result in arbitrary code execution (which is bad). Encoding is harder than it seems because you might need different encoding algorithms depending whether you're injecting a text attribute, a URL attribute, or a text node.
A template engine also needs an expression language; while it is easy to inject plain data fields without modifications, this is only the simplest scenario. Often, you'll need to apply a format string, combine multiple fields and more generally manipulate the data before displaying it. This could be achieved by transforming the data before feeding it into the template, but it is easier and more efficient if that ability is built into the template engine. Once you start adding features such as formatting, you very quickly find you need the flexibility of a full expression language.
Being able to intersperse code and markup (as you can with <% %> blocks in ASP) enables interesting scenarios such as repeating markup using a loop around an HTML fragment or conditional rendering using a simple if statement. Again, this scenario requires a full language to be really useful.
Finally, HTML is only half of the story. AJAX applications really are about active content, not just client-side updates to the DOM. Once you've generated HTML, you still have to hook events to elements and attach controls and behaviors. You can do that through code after the HTML generation, but this creates an unpleasant dissymmetry between the HTML, which becomes very easy, and the logic, which becomes more difficult and requires knowledge of the template's structure.
In other words, in order to attach behavior, you need to know where to attach it, which in turn means that any change in the structure of the HTML template markup will require changes in the code that activates it. There are ways to make that coupling looser but an even nicer solution would be to make content activation part of the template engine.
The other way you can generate HTML from data is to manipulate DOM APIs directly and create elements, attributes, and text nodes from code. At first, this looks like a bad option for several reasons. Sure, it has the advantage of being agreeable to standards zealots, but for some strange reason it's a lot slower than innerHTML. But the main reason this is not commonly used is that the DOM APIs are not very expressive, and the resulting code is hard to read and harder to maintain—at least without any help and additional abstractions. There are toolkits such as jQuery that provide excellent abstractions and make the whole process a lot more fun, but even with such tools, it's harder than it needs to be (which is why even jQuery has several template plug-ins).
Some of you may know that Microsoft already had a template engine in ASP.NET AJAX Futures, but it was too slow and complex in design and we wanted to do a lot better. The good thing about this failed first attempt is that we learned a lot about what we didn't want the new version to be (that is, slow and complex).
The dev team tested many different designs for a new template engine for ASP.NET AJAX, from string concatenation to full DOM manipulations, and we evaluated them on performance, simplicity, and flexibility. We also compared them in terms of what scenarios they prevented from happening. There is no ideal solution, but we chose the one that seemed like the better compromise.
The principle of the new engine is simple: we take your template code, which contains HTML, data fields, expressions, declarative component instantiation and imperative code, and we turn that auto-magically into JavaScript that creates the equivalent HTML. That looks simple enough and it is (well, until browser quirks appear). We do pay a performance penalty for using the DOM APIs, but if we're careful and build elements outside of the DOM, and add as few as possible as late as possible, the performance hit is not too big and the amazing flexibility it creates is well worth the trade-offs. All the problems in the string concatenation approach seem to go away naturally.
Injection attacks take care of themselves. As I'm using code to create text nodes and set attribute values, there is no need for encoding because the APIs I use are already safe. This is analogous to using SQL

parameters versus building SQL by concatenating strings. Nobody in his right mind does the latter anymore so why would you want to take a similar risk here?
Now who needs a new expression language? We already have one: JavaScript. When you transform the template markup into JavaScript code, what could be easier than injecting JavaScript expressions into that code you're generating?
To enable the most common template development tasks, one-time, one-way injection of data fields (which on the server is expressed by "<%= expression %>" and in our system is "{{ expression }}"), I use a feature of JavaScript that is often despised—the "with" keyword. It saves me from having to resort to expressions such as "{{ dataItem.myField }}" to inject a field of the data item associated with the template instance. Thanks to the "with" keyword, you can surround the generated code for the template with something like "with(dataItem) {…}" so that any member of the data item is promoted to the top scope of the template function, making the injection of expressions as simple as "{{ myField }}".
You can inject behavior into the template in two ways. First, you can write $attachEvent and $create code from the itemCreated event, or inline in the template using a special $element variable that is available from within the template and references the latest created element. Or you can use the declarative syntax we're providing. For example, if you want to add an autocomplete and a watermark behavior to an input tag, you'd write something like this:
Copy Code
```
<body xmlns:sys="javascript:Sys"
 xmlns:autocomplete="javascript:AjaxControlToolkit.AutoCompleteBehavior"
 xmlns:watermark="javascript:AjaxControlToolkit.extBoxWatermarkBehavior">
...
<input id="search" sys:attach="autocomplete,watermark"
 autocomplete:servicepath="SearchAutoComplete.asmx"
 watermark:watermarktext="Type your search terms here" />
```
Here I register the prefix for each declarative behavior on the HTML or body tag (or on the parent tag for the template) using the xmlns XHTML namespace declaration. This will enable me to extend the XHTML markup in a standard way, and it is similar to @Register directives for server code. The part after "xmlns:" is the prefix that is going to be associated with each behavior or control. The URL for the namespace is using the "javascript:" protocol to map the prefix to a specific JavaScript type. The "sys" namespace is a special system namespace that should be mapped to the Sys namespace, which is the root namespace in AJAX.
The instantiation itself is done through the special attribute sys:attach, whose value is a comma-delimited list of the prefixes of the behaviors or controls to instantiate and attach to the element. Then I can set properties for all of these behaviors without risking a conflict with regular HTML attributes or other behaviors on the same element because they are nicely differentiated by namespace.
One of the most elegant features of the engine is that the compilation of the template into JavaScript code really is analogous to a real compilation step. This means that it only has to happen once per template and that provides the opportunity to execute a number of tasks ahead of time instead of doing them each time the template is instantiated. But that's enough theory for now. How does this apply to our master/details page?

The AJAX Version Templates

The template for the list of products is quite simple:
Copy Code
```
<ul id="productListTemplate" class="sys-template">
    <li>
        <a href="{{ String.format('3_Client.aspx?product={0}',
        ProductID) }}">{{ Name }}</a>
    </li>
</ul>
```
The item template is a list item with a simple link in it. The text of the link is just the name of the product ("{{ Name }}") and the href attribute is a formatted string built from the product ID using plain JavaScript:
Copy Code
```
"{{ String.format('5_Client.aspx?product={0}', ProductID) }}"
```

The class "sys-template" is defined in the CSS for the page to hide the template from the initial rendering of the page. The compiled code for this simple template is shown in **Figure 7**. The details view is a little more complex and actually contains some inline code (see **Figure 8**). I could have used a nested template to render the list of photos, but it is a little simpler to use a regular loop over the markup for one photo. A nested template would have been justified if I were dealing with dynamically changing data and automatic reflection of the changes into the markup (which is a supported scenario, but outside of the scope of this article), but seeing that I'm dealing with one-way, one-time bindings here, inline code is fine.

Figure 7 Template-Compiled Code

Copy Code

```
function(__containerElement, $dataItem, $parentContext, __instanceId) {
   var __context = {}, $component, __app = Sys.Application,
      __creatingComponents = __app.get_isCreatingComponents(),
      __components = [], __componentIndex, __e, __f, __topElements = [],
      __p = [__containerElement], $index = __instanceId,
      $id = Sys.Preview.UI.Template._getIdFunction(__instanceId),
      $element = __containerElement;
   Sys.Preview.UI.Template._contexts.push(__topElements);
   with(__context) { with($dataItem || {}) {
      $element=__p[1]=document.createElement('LI');
      __topElements.push($element);
      $element=__p[2]=document.createElement('A');
      $component = $element;
      __e = document.createAttribute('href');
      __e.nodeValue = String.format('5_Client.aspx?product={0}',
                                                       ProductID);
      $element.setAttributeNode(__e);
      __p[1].appendChild($element);
      __p[2].appendChild(document.createTextNode(Name));
      $element=__p[2];
      __p[1].appendChild(document.createTextNode(" "));
      $element=__p[1];
   }
}
   for (var __i = 0, __l = __topElements.length; __i < __l; __i++) {
      __containerElement.appendChild(__topElements[__i]);
   }
Sys.Preview.UI.Template._contexts.pop();
 return new Sys.Preview.UI.TemplateResult(this, __containerElement, __topElements,
__components);
}
```

Figure 8 Displaying the Details

Copy Code

```
<div class="sys-template" id="productDetailsTemplate">
    <fieldset>
        <legend>{{ Name }} ({{ ProductNumber }})
            {{ String.format("{0:C}", ListPrice) }}</legend>
        <ul class="photoList">
            <!--* for (var i = 0; i < Photos.length; i++) { *-->
            <li><img src="{{ String.format('productphoto.ashx?id={0}',
                Photos[i]) }}" /></li>
            <!--* } *-->
        </ul>
        <table>
            <tr><td class="label">Summary:</td><td>{{ Summary }}</td></tr>
            <tr><td class="label">Experience:</td>
                <td>{{ RiderExperience }}</td></tr>
```

```
              ...
            <tr><td class="label">Style:</td><td>{{ Style }}</td></tr>
            <tr><td class="label">Wheel:</td><td>{{ Wheel }}</td></tr>
            <tr><td class="label">Maintenance:</td>
                <td>{{ MaintenanceDescription }}</td></tr>
        </table>
    </fieldset>
</div>
```

The templates are lazily compiled the first time they are instantiated but they are prepared by creating a "new Sys.Preview.UI.Template" using the parent element of the template markup as the parameter of the constructor. The templates themselves are instantiated from the callback from the network call that brings the data back from the Web service on the server:

Copy Code

```
AdventureWorks.GetProducts(1 /* Mountain bikes */,
  function(productArray) {
    renderProductList(productArray, productListTemplate);
    selectProduct(initialProductID, true);
});

function renderProductList(productArray) {
    var target = $get("productList");
    target.innerHTML = "";
    for (var i = 0, l = productArray.length; i < l; i++) {
        productListTemplate.createInstance(target, productArray[i]);
    }
}
```

This will not be necessary in the shipping version of ASP.NET 4.0; its DataView component will take care of the template parsing, compiling and instantiating. Most of the code in this app will eventually go away, but it is useful in showing how things happen under the hood. It also shows how a component developer who wishes to include template rendering might use the feature.

Event Bubbling

Where do I put the code that displays the right details view when a user clicks on one of the products? The code, like its server-side equivalent, uses event bubbling, so I was able to write one single- click event handler for all the links in the list (so I could add or remove links to the list if I wanted to without having to worry about creating new handlers or cleaning up the old ones). The following code shows that handler. All click events for the links in the list will bubble up to the list itself and be handled there. "e.target" is a reference to the element that got actually clicked; in other words, it's the link, which enables me to retrieve the product ID from the href attribute and select the relevant product:

Copy Code

```
$addHandler($get("productList"), "click", function(e) {
    var href = e.target.href;
    selectProduct(parseInt(href.substring(href.indexOf('=') + 1), 10));
    e.preventDefault();
    e.stopPropagation();
});
```

Once that is done, the event's default action (the link navigation) is canceled and the event is prevented from bubbling up further. This is done by calling the W3C standard stopPropagation and preventDefault methods on the event object, which the framework makes available on all browsers, including Internet Explorer.

Managing the Back Button

The only feature that remains to be reproduced from the server-side version is history. Enabling history on the ScriptManager control also enables client-side APIs that are exactly analogous to the server-side APIs

I've used before and that can even be used at the same time (enabling mixed client-server state management).
Creating history points is done by calling Sys.Application.addHistoryPoint from the events that correspond to a state change, in this case, clicking a product in the list:
Copy Code
```
Sys.Application.addHistoryPoint({product: productDetails.ProductID},
    "AdventureWorks - " + productDetails.Name);
```
Correspondingly, state is restored from the "navigate" event on Sys.Application. The HistoryEventArgs arguments that the event handler receives has one property—state—that enables you to retrieve the product to restore:
Copy Code
```
Sys.Application.add_navigate(function(sender, e) {
    var ProductID = parseInt(e.get_state()["product"], 10);
    selectProduct(ProductID, true);
});
```

The Finished Product

The resulting page behaves very much like the UpdatePanel version, but there is no comparison in terms of network traffic. When a product is selected, the UpdatePanel version sends more than 4KB of data to the server and receives back about 8KB. The pure AJAX version, on the other hand, sends only "{"productId":771}" plus standard HTTP headers and receives back 2KB of pure JavaScript Object Notation (JSON) data. That's about 10KB saved each time a user clicks on a product.
This is just one of the exciting features planned for ASP.NET 4.0. Share your thoughts at go.microsoft.com/fwlink/?LinkId=126987.

Bertrand Le Roy, Ph.D., is Program Manager in charge of AJAX at Microsoft. He spent five years as a developer in the same area. He also represents Microsoft at the OpenAjax alliance