PRINT THIS PAGE NOW

Saturday | 04 April 2009 | 18:01 PDT

Visual Studio Magazine Online

On VB

XML Literals, WCF and LINQ

Learn how to create powerful templates that can be called from both client- and server-side code.

April 1, 2009 · by Steele Price

One of the great things about .NET Framework is that there is so much included to make your life as a developer easier. Over the past few years we've been flooded with new capabilities. Staying current with all of these tools is nearly a full-time job. So what's a developer to do? I choose to pick the tools that apply directly to a problem I need to solve and use those new capabilities to enhance my productivity. Even more power and flexibility comes when we can take several of these capabilities and combine them for a unique solution to a problem.

We're going to look specifically at three pieces of the framework that can be combined to provide a new technique. This technique helps in writing more responsive ASP.NET pages, while at the same time making the code more readable. The three technologies are: XML literals, Windows Communication Foundation (WCF) Factory Services and LINQ. XML literals and LINQ are new in Visual Basic 9 (VB9). LINQ gives us a common syntax for querying just about any data, be it SQL, XML or objects. Even though WCF has been here for a while, the out-of-the-box readiness for building factory services is little-known. Here we'll show you how to create WCF services without changes to config files for endpoints, behaviors and bindings.

XML Literals

VB9 includes XML literals, an incredibly useful new tool. With XML literals, what used to be an archaic, difficult process of reading and writing raw XML or XHTML has become straight-forward and simple. Type raw XML into the Visual Studio Editor and it understands that you want an XElement. An XML literal on its own is a remarkable piece of technology that can help VB developers in many, many ways.

What used to make code difficult to read becomes transparent when using XML literals. For example, using a StringBuilder to write long strings makes writing joined strings easier and more performant:

```
Dim MyText = String.Empty
Dim sb As New StringBuilder
With sb
  .Append("This is a String.")
  .Append(vbCrLf)
  .Append(vbTab)
  .Append("You will notice that the whitespace ")
  .Append("is retained when we use the string.")
  .Append(vbCrLf)
  .Append(vbTab)
  .Append(vbCrLf)
  .Append(vbTab)
  .Append("This is just the beginning and is a bit ")
  .Append("easier to read than StringBuilder, right?")
  MyText = .ToString
End With
```

Reading and writing these in the editor is less productive than if you could just write the whole string and retain the whitespace as follows:

```
<MyString>
This is a String.
  You will notice that the whitespace is retained when
  we use the string.

  This is just the beginning and is a bit easier to read
  than StringBuilder, right?
</MyString>
```

Compared to StringBuilder, XML literals are far easier to read and much less to type. Use MyString.Value to get the text inside without the <MyString> tags. The goal here is not to maximize performance in the runtime, but to be more productive. You can actually read the string and don't worry; performance will still be very good.

String handling with XML literals is just a side note. For structured elements that really are XML, such as XHTML, there is no comparison. The XML literals approach gives us a highly readable, productivity enhanced solution to writing structured XML in code behind. Consider the following:

```
Dim MyTable = _
<table>
  <tr>
    <td>
      First Cell Contents
    </td>
    <td>
      Second Cell Contents
    </td>
  </tr>
</table>
```

Reading XHTML in code behind is now remarkably easier. You also get true IntelliSense for the elements that are in a known namespace, which of course means even less typing. XML namespaces are recognized by using Imports statements such as the following for XAML:

```
Imports <xmlns=
         "http://schemas.microsoft.com/
          winfx/2006/xaml/presentation">
Imports <xmlns:x=
         "http://schemas.microsoft.com/
          winfx/2006/xaml">
Imports System.Windows.Markup
```

The result of all this is that you can do things the same way for code behind as you are used to doing in ASPX source view. Productivity increases by continuing to write markup this way. You do not have to learn another language or syntax to achieve the same in code behind.

Embedded expressions also look very similar to what you are used to seeing in markup:

```
<td><%= item.@id %></td>
```

An embedded expression lets you access something outside the XML literal and embed it into the result. In this case; item.@id is an attribute from an item element in another XElement. Additionally, variables aren't limited here; you can use lambdas, functions or anything else available to VB.

I know what you're thinking, so I'll go ahead and say it. "If you're loading from XML and returning XML, can't you just use XSLT?" Of course we could, but what fun is that? In fact, I came up with this technique in a project overhaul that was using XSLT in its previous version. During development, it became quite clear that working in XSLT was not the way to go for us. XSLT is a completely different mindset for most of us and has a fairly steep learning curve. In addition, the old system was forcing SQL to return data as XML, which is not exactly optimal. This approach eliminated that issue while dramatically improving performance. Ultimately, using XML literals means you can be productive faster and have more tools at your disposal for debugging, without learning a new language.

WCF Factory Services

I have a Web site that is currently in production. I need to add some performance features to it and I don't want to retool to accomplish what I need. I could use AJAX and update panels, but then I would need to account for the entire page lifecycle, which in some cases may be quite large.

For example, I have a page that contains aggregate information and I wish to display that in a table format. I could put all my aggregates into GridView objects and then place those inside update panels, then … no, no, no. I'm sure you see the trouble here already. If I have more than one update panel on the page, I'm really loading the entire page multiple times, just to get my aggregated information to run asynchronously.

How can you achieve this without creating a new project, using PageMethods or something else with a large lifecycle? Enter Windows Communication Foundation (WCF). WCF has always had a lesser-known feature called a factory service. With factory services, you don't have to worry about things like security, as it inherits the same security as the site. As for config settings, there are none. This is a perfect solution when you need to access some information in an AJAX way from client-side code.

The factory services are quite similar to WebMethod(), but they're completely WCF compliant. To create a factory service, you simply create a new WCF service. But wait, you said factory service; I don't see any factory service in my Add Items templates.

That is correct, to create a configuration-less factory service you need to do it by hand; editing the .SVC file of an AJAX-enabled WCF service. To do this, open the .SVC file with the XML editor and add the following:

```
Factory="System.ServiceModel.Activation.
WebScriptServiceHostFactory"
```

Then you have to go and delete the config information. Because this isn't an intuitive way to proceed, I created a template that you can install to do the same thing. Install the template by copying WCF_FactoryService.zip to C:\Users\<yourusername>\Documents\Visual Studio 2008\ Templates\ ItemTemplates\Visual Basic

With this template, you can create services that your Web application can use in a very easy way with no bother to the current configuration. If you need to move to a full service with different bindings, it's very easy to change this from a factory service. Simply delete the factory attribute from the .SVC file and add your binding, behavior, service and endpoint configurations to web.config.

Something you might be thinking: Why use WCF services and not a regular page that just returns the small amount of data we need?

While this approach makes sense in some instances, you still have a full-page lifecycle to deal with. Furthermore, the results aren't as easy to work with when you want to insert them into a portion of an already rendered page with JavaScript. If you used a page you would have to do quite a bit of parsing. In addition to the speed of execution for WCF, there are supporting features that will make life easier. Throwing FaultExceptions is one thing you cannot do with page results. You can throw HttpExceptions, but they aren't handled the same way on the client.

Using LINQ

LINQ provides the magic you need to glue all this together. Writing markup in code behind is easier to read, while accessing the same code from both the client side and the server side gives you a lot of power. Combining LINQ and embedded expressions in XML literals gives you a better way to handle looping through data.

Instead of a table, let's shift to something more demonstrative, the <ul> element. Say you just need to iterate through a group of items and return an unordered list. Unordered lists are great for things like menus and navigation. How is this accomplished with our new toolset? Let's look into that now. In the past you had to do something like the following:

```
Dim ul as New HtmlGenericControl("ul")
Dim li As HtmlGenericControl
For Each i In items
  li = New HtmlGenericControl("li")
  li.InnerHtml = i.Value
  ul.Controls.Add(li)
Next
```

Now you can do this:

```
Dim MyMenu = <ul><%=
                   From i In items Select <li><%=
                   i.Value %></li>%></ul>
```

The embedded expressions let you insert external data into your elements with much less code. Given the same data, the output is identical.

Let's dissect what's happening here. MyMenu is an XElement cast by Option Infer, which is a new feature in VB9 that lets you declare variables without explicitly stating a data type. The compiler infers the data type of a variable from the type of its initialization expression. The <ul> element is outside the LINQ query so it's not repeated. The LINQ query iterates through the items and returns a group of <li> elements inside the <ul> element, with the item's value inserted into the list item.

All of this can now be done in one line of easy to read and simple to type code. Anything used for lists of data can be done this way: tables, unordered lists, ordered lists, select inputs (dropdowns), and the like. Have you ever done a view source on a page only to see a dropdown with all the states listed as options in the HTML? Using WCF services with AJAX will keep people from being able to see this with view source. You'll probably get your page to load faster if that was an AJAX call.

XML literals give you the power to create data presentation dynamically, in ways that are more productive than many other alternatives. Any LINQ-enabled data, SQL, entities, objects, files, WMI, XML, RSS feeds and the like can be accessed this way.

Now that you have all three pieces in place, your Web application is wired up to both client and server sides. Accessing the service from the server is as simple as calling a method. The service is already there and calling it is as simple as Services. GetPresidentsList().

On the client you can use ASP.NET AJAX or jQuery to call the service. Referencing services in ASP.NET AJAX is incredibly easy -- just add a ScriptManager that points to the service as follows:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
 <Services>
  <asp:ServiceReference Path=
                   "~/Services/MyFactoryService.svc" />
 </Services>
</asp:ScriptManager>
```

Calling the service in JavaScript is now very easy and completely wired up for us:

```
function getPresidents(){
  var ws = new Services.MyService();
  ws.GetPresidentsList(getPresidentsComplete);
}

function getPresidentsComplete(result, eventArgs){
  if (result.d !== null){
    $find("<%= PresidentsArea.ClientID %>").InnerHtml =
 result.d;}
}
```

This makes an asynchronous call to the service and the result is returned when the service performs a callback to getPresidentsComplete (). You have the ability to call the service both synchronously and asynchronously, depending on what you need to accomplish. Here's the same idea using jQuery:

```
$.ajax({
  type: 'POST',
  url: '/Services/MyFactoryService.svc/'
     + 'GetPresidentsList',
  data: '{}',
  contentType: 'application/json; charset=utf-8',
  dataType: 'json',
  success: function(result, eventArgs) {
    if (result.d !== null){
      $("#<%=  PresidentsArea.ClientID %>")
         [0].innerHTML =     result.d;}
  },
  error: onServiceError
});
```

The json notation is assigned for the dataType because WCF is returning JSON for the result object. The syntax is a little different, but is still pretty easy to use. jQuery's nice ajax() method does all the work for you, and what you get returned from the service ends up in result.d. I used an anonymous function here to retrieve the result, which we expect to be XHTML into the InnerHtml of our div element target.

We can inspect for errors, but you can see that if we threw any FaultExceptions in the WCF service, then another function called onServiceError runs. The Error Handling function is passed to the result object for us to inspect for errors and react accordingly.

My normal process for using this in a production application is to define a single function in my master page for calling the service. If you provide accessibility through several shortcuts, getting results from the services and assigning those results become even easier.

Here's a sample that I've used:

```
function execMyFactoryService(method, target)
{
  var rval = '';
  $.ajax({
    type: 'POST',
    url: '/Services/MyFactoryService.svc/' + method,
    data: '{}',
    async: true,
    contentType: 'application/json; charset=utf-8',
    dataType: 'json',
    success: function(result, eventArgs) {
           if (result.d !== null){
             target.innerHTML = result.d;
           }
         },
    error: function(result) {
        var msg = '';
        if (result.get_message) {
          msg = 'Error: ' + result.get_message();
        } else {
          msg = result.responseText;
          if (msg == '') {
            msg = 'Error: Unknown... missing Service?'; }
        }
        alert(msg);
        target.innerHTML = '';
      }
  });
}
```

Assigning our result to a position on our page is simple. Place a <div> element anywhere on your page and set the innerHTML to the result. Anywhere I'm using that master page; I will have a function that I can use to directly assign the results:

```
execMyFactoryService('GetPresidentsList',
           $("#<%= PresidentsArea.ClientID %>")[0]);
```

All the error handling and result inspection is handled centrally and when we're assigning results as the response to some action on the Page, this makes understanding what's happening much easier. If we were to use the full $.ajax syntax everywhere we needed to use it, it might make the scripting too verbose to be readable at first glance.

jQuery provides a great productivity boost through a superb set of tools for navigating and manipulating DOM objects in JavaScript. Some of the same features are in ASP.NET AJAX, but the two models can be used together to provide even more power for your client-side activities.

The Whole Picture

Now that I've explained the parts and pieces, how does it all work together? Let's explore this a little further. A complete template looks like this inside the service:

```
<OperationContract()> _
Public Function GetPresidentsTable() As XElement
  Dim items = XElement.Load(ApplicationPhysicalPath & _
     "App_Data/SampleData.xml")
  Try
    Dim result = _
      <table>
        <thead>
          <th>Position</th>
          <th>Name</th>
          <th>Began Term</th>
          <th>Finished Term</th>
        </thead>
        <%= From i In items.Elements Select _
          <tr>
            <td><%= i.@id %></td>
            <td><%= i.@Name %></td>
            <td><%= i.@Start %></td>
            <td><%= i.@End %></td>
          </tr> _
        %>
      </table>
    Return result

  Catch ex As Exception
    ' This is not returning a StackTrace,
    ' it's a shortcut to get the current Method Name
    Throw New FaultException(New StackTrace() _
       .GetFrame(0).GetMethod() _
       .Name & ": " & ex.Message)
    Return Nothing
  End Try
End Function
```

The service is a WCF factory service. These are very simple to create, with the Visual Studio Template included here.

I use a Services Folder in my Application for organization and security. Add the Services Folder, and then add a web.config for the folder. This lets us assign different security rights to the services in the folder if we need to limit them by Roles. Adding a New Item to your Web Application looks like Figure 1.

The new Template is in the My Templates area at the bottom of the dialog. If you're like me, you probably have many installed Templates you need to scroll through to get to My Templates.

When the template creates the service, we can inspect the configuration data using Open With.

Choose XML Editor from the list. This will open the .SVC file for us, rather than the code behind that gets opened when you double-click on the file itself. You may see some IntelliSense confusion from the editor, but you can safely ignore it.

The generated service is:

```
<%@ ServiceHost
  Language="VB"
  Debug="true"
  Service="Services.MyFactoryService"
  Factory="System.ServiceModel.Activation.
     WebScriptServiceHostFactory"
  CodeBehind="MyService.svc.vb" %>
```

The code behind is:

```
Imports System.ServiceModel
Imports System.ServiceModel.Activation
Imports System.Web.Script.Serialization
Imports System.Runtime.Serialization

Namespace Services

  ''' <summary>
  ''' This Service provides access to various
  ''' MyFactoryService procedures through WCF
  ''' </summary>
  ''' <remarks></remarks>
  <ServiceBehavior( _
     IncludeExceptionDetailInFaults:=True)> _
  <ServiceContract(Namespace:="Services", Name:= _
     "MyFactoryService")> _
  <AspNetCompatibilityRequirements( _
     RequirementsMode:= _
     AspNetCompatibilityRequirementsMode.Allowed)> _
  Public Class MyService

    <OperationContract()> _
       Public Sub DoWork()
    End Sub

  End Class

End Namespace
```

All that's left for us to do is enter the code into DoWork(). Usually you'll want to rename this, which is fine, because what's presented here is just a stub to get us started.

For debugging we simply add a Try/Catch block to handle anything bad that may cause the template to break. For this example we are just returning Nothing, but we could add very rich error handling here by throwing FaultExceptions. FaultExceptions are handled quite nicely by ASP.NET AJAX and jQuery on the client. You can decide how to handle problems in the service and the client can respond using a combination of results and error handling.

Let's talk briefly about what you can do with this in your toolbox. I want to display a bunch of images dynamically. Maybe I want to get the results from a search out on the Web. I could configure something in the client, or I could stay consistent and use our own services. This lets us control everything and not expose possibly sensitive information in the client-side code, such as a password. We won't build that one right now, but think about how useful it could be to control going out to your external service accounts to grab dynamic status or images from social networking services.

In our Presidents sample we could go look on an external service for the images of the Presidents. Configuring this is really quite easy now that we have a set of tools to work with that makes this simple. I could use a WebClient to call the external API and work with the results in the service prior to returning them.

What if I already have the images? How do I get them from the file system easily? I would create a service method similar to this:

```
<OperationContract()> _
Public Function GetPortrait( _
      ByVal value As String) As XElement
     Dim image = (From FileName In _
        My.Computer.FileSystem.GetFiles( _
           HttpContext.Current.Server.MapPath( _
           "~/Images"))
        Where FileName.StartsWith(value) Let File = _
        My.Computer.FileSystem.GetFileInfo(FileName)
        Select <img src=<%=
           HttpContext.Current.Server.MapPath( _
           File.FullName) %>
           title=<%= File.Name %>></img>).First()
   Return image
End Function
```

I have provided a dynamic function to return an image from our file system -- certainly not a daunting task for us now. We can expand this to handle getting the image path any way that suits our needs and we can surround all the proper error handling in the function. I can return either an appropriate fault exception or an empty element to just ignore missing images, so the client doesn't see errors.

When designing a template consider what's static and what requires looping. Looping should be done with either LINQ or lambdas in this case. A Do Loop or For Each works just as well, but I find myself replacing them with LINQ since I had no other alternative before. It also makes the code much easier to work with in templates if you use LINQ.

Something to consider here is the use of lambda expressions. Lambdas are a new feature in VB9 and can save you a lot of hassle when you need a quick function. LINQ itself is actually a framework built on lambda expressions and are an integral part of how it works. Lambdas in Visual Basic look like this:

```
Function(u) If(u.Age < 18, _
   u.ParentPermissionGranted, u.PermissionGranted)
```

This function will evaluate the condition, and then return a different field from the user object based on the condition. The interesting thing here is that we didn't have to tell the lambda what u is, inference will do that for us most of the time. Occasionally inference gets confused and you may need to qualify what u is which you can do with:

```
Function(u As User)
```

I don't want to go on too much about lambdas, you will use them all the time with LINQ. Just about any time you need a Where clause you'll be using lambdas. LINQ hides some of the verbosity for you when you're using the full query syntax, but it's still a lambda. When you use the extension syntax you almost always use lambdas:

```
items.Elements.Where(Function(i) i.@id = 1)
```

Another reason I bring this up now is to answer the question: What if we need to return complex conditioning to our data when we build the return? This is much easier with lambdas than creating a bunch of functions that may only be used for this specific template. When building our template, maybe we want to change the elements returned based on some criteria in the data. Extended syntax like lambdas make very powerful tools for building templates.

Easy Formatting Based on Conditions
Because we have a true ternary If() command available to us we can make templates that do this:

```
<%= If(items.Count = 0, _
  <tr id="norecords" class="GridRow">
    <td style="text-align: left" colSpan="4">
      <div>No records to display.</div>
    </td>
  </tr>, _
  CType(Nothing, XElement)) _
%>
<%=  -- - continue normal processing  -- - %>
```

This template lets us return Nothing when we have real data to work with, or an element describing the lack of data to the user. I can go on and on about the wealth of possibilities you have at your disposal when using all these techniques together.

The Bottom Line
Currently, nothing else is this flexible and this easy with the out-of-the-box tools we get with Visual Studio 2008. You have complete access to all the features of .NET Framework for building your templates. Calling them from both the client and server is extremely easy, and because you're using WCF, ther's no added page lifecycle overhead. A similar service can be created to return XAML instead of XHTML. This makes migrating or extending your templates extremely easy if you need access from Silverlight or WPF in the future.

T4 templates and other code generator-based utilities are great when you can generate everything in advance. This technique goes beyond that to generate during runtime, and we can generate runtime-ready code in any XML derivative such as XAML and XHTML.

ASP.NET 4.0 is expected to provide a new template system, but that doesn't change the usefulness of this technique. ASP.NET 4.0 will be using a completely different system, which may or may not be as easy to use and as flexible in what it returns. From what I've seen so far, it looks good, but it still doesn't solve the immediate issues that are addressed by using this technique.

XML literals, WCF factory services and LINQ, taken by themselves, are very useful. When these are combined, you get a completely different picture for solving real-world problems. All while maintaining a productive, easy-to-implement style.

XML literals provide high productivity and readability when writing structured XML in code behind. It can also help format long or complex strings to make them more readable. Embedded expressions give you the power to insert external data into the XML in a very easy-to-read and -maintain way.

WCF factory services allow you to extend the power of the framework to your client-side code without the hassle of all the configuration knobs and buttons that are available to WCF

services. Starting with a factory service does not limit you in any way from upgrading to the full configuration format later. A simple change enables a progressive upgrade path when you need this ability.

Finally, using LINQ in embedded expressions enables you to insert external data into XML in a way that is easy to read, with much less code to accomplish the task. LINQ allows you to start thinking differently about how you process loops through any data utilizing the same syntax. Now go see what you can do with all this new power and productivity.

About the Author

Steele Price has been professionally designing and developing data-driven applications for more than 25 years. He has been a Microsoft MVP for Visual Basic since 2005 and frequently works with the Microsoft VB team to help improve the language. Price is currently chief technology officer at Digital Dreamshop, a micro ISV developing enterprise applications with VB.NET, Silverlight, Windows Presentation Foundation and LINQ.

PRINT THIS PAGE NOW | SEND | SHARE | COMMENT | MORE | TOP

PRINT THIS PAGE NOW

<··· back to previous page