Building .NET Components and Controls for ASP+

Richard Anderson

Introduction

ASP+ is a key element of the new Microsoft .NET platform. It provides developers with the scalable rich platform they need to build modern web sites using strongly typed compiled web pages.

Gone are the days of ASP spaghetti code that is interpreted and prone to runtime errors. Everything in ASP+ is compiled down to a .NET component. Does this mean ASP+ is more complex to use than ASP? No. As an ASP+ developer, you can choose whether you build your own .NET components, or whether you let the ASP+ runtime create them for you using the same development model (or lack off) that ASP uses today.

In this session we'll start by looking at how components form the foundations of ASP+, and I'll show you the how ASP+ uses components to generate all page output. After a brief review of how the core components of ASP+ fit together, we'll examine when and how you may choose to build your own components using VB and C#.

I'll show you how to use components in an ASP+ page, and show how cool and easy deployment becomes for your web applications thanks to XCOPY deployment: gone are the days of registering and unregistering components!

I'll wrap up the session by reviewing the components and interfaces used for ASP+ server side control development. ASP+ has a rich server side control model that fits most requirements, but you can easily extend components, or write your own. This session assumes no prior knowledge of component development, but does require basic understanding of programming concepts such as classes and inheritance.

Richard Anderson is an established software developer and author who has actively been working with Microsoft technologies for as long as he can remember. Richard works for Business Management Solutions, www.bms.com, a leading edge e-commerce company who specialize in next generation payroll systems. Richard is a co-author of many books, including the recently released "A Preview of Active Server Pages+", and "Beginning Components for ASP", both published by Wrox Press.

What is an ASP+ Control?

An ASP+ control is best defined as a .NET class that, somewhere in its class hierarchy, derives from the `Control` class. The `Control` class is the fundamental building block through which the ASP+ page framework generates all page output, and through which postback data and events are processed.

Here is a brief outline of the types of controls you can create in ASP+:

- Pages – as we have discussed, these are files that end in `.ASPX` and which you do not have to compile yourself.
- User Controls (pagelets) – these are essentially just ASP pages reused within ASP+ pages. They are declared in a similar fashion to normal controls and they provide a quick and cool way of resuming UI in your applications.
- Composite Controls – these are similar to user controls, except they are code based, and you do have to compile them. They render their output by populating their controls collection with other controls.
- Custom Controls – These are essentially controls you write from scratch. They typically emit HTML, but they can still make use of other controls, by directly calling their `RenderControl` method (this method essentially just calls the `Render` method, assuming the control is marked as being visible).

All of these controls are compiled into .NET components, so performance should be equal. The only difference really is how much control you want. Pages and User Controls do not protect your source code, and therefore do not protect your IPR, but later versions of ASP+ may well provide tools to protect your code (e.g. just let you ship a DLL). Pages and User Controls also suffer a slight performance hit when first accessed, as they have to be compiled.

The only real downside of custom and composite controls are that you have to understand how to use Visual Studio.NET (and therefore must buy it), or you need to make use of the command line compilers. You also have to manually manage dependencies and recompile them.

Because everything is compiled in ASP+, and everything makes use of the Control base class, it isn't surprising that a lot of the techniques involved when writing controls is the same, not matter what type of control you're writing. This is certainly true for accessors (control field accessing), events, and methods. Indeed, such concepts are identical outside .NET, so we'll cover them pretty quickly. When building ASP+ application you are always dealing with the `Control` class.

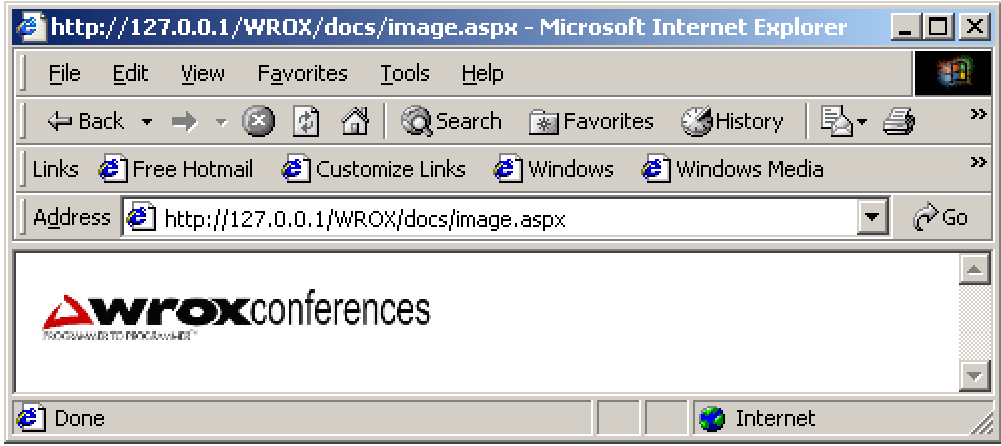
Whether you ever resort to using the VS.NET compilers to create a .NET component or not, the ASP+ page compiler always does. To understand this, lets look in detail at what happens when you create a simple ASP+ page like this:

```
<asp:image imageURL="wrox.jpg" width=200 runat=server />
```

First off, we'll see that the ASP+ renders the following HTML to your browser:

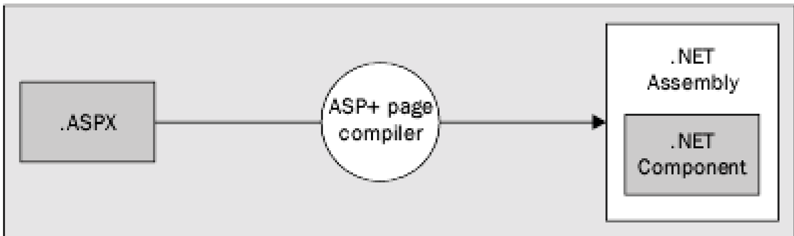
```
<img src="wrox.jpg" border="0" style="width:200px;">
```

Which unsurprisingly looks something like this:



So how do ASP+ pages generate their output?

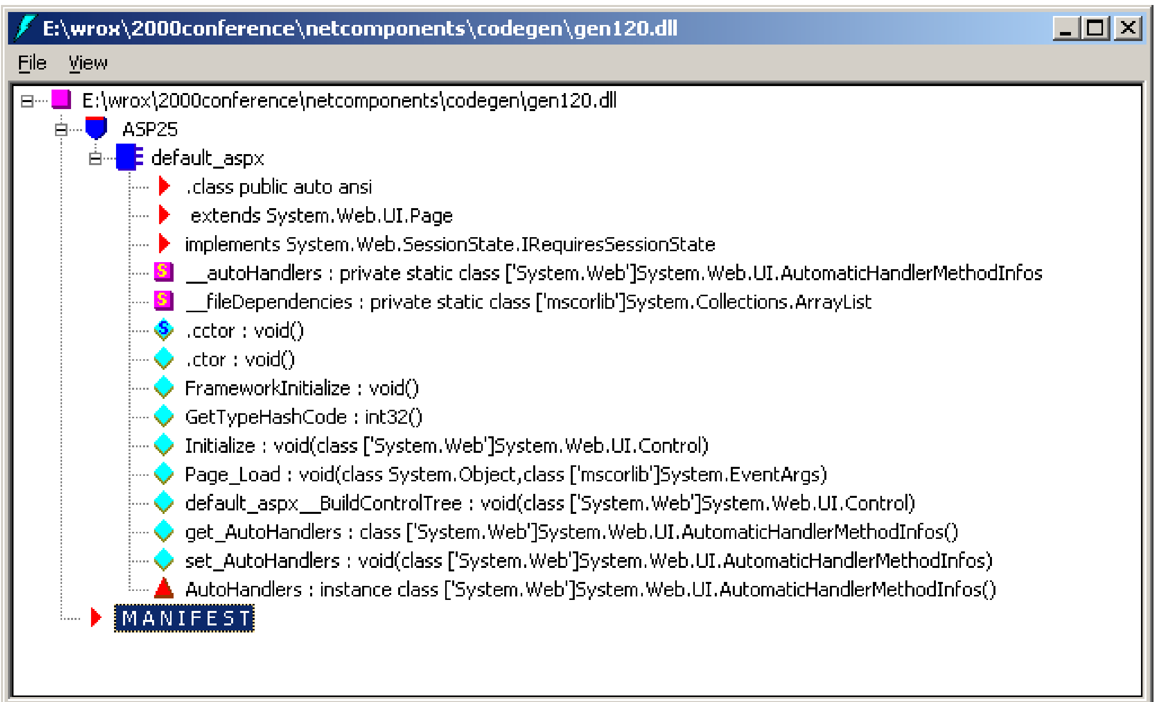
The transformation from the original ASP+ page to the rendered HTML is performed by the ASP+ page framework, also commonly referred to as **Web Forms**. In a nutshell, each ASP+ page you create is converted into a .NET component the first time that the page is requested by a client:



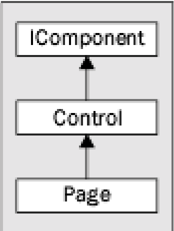
The ASP+ page compiler performs this compilation, and the output as shown is a .NET assembly (a DLL) that is created and placed into the `codegen` directory:

```
Command Prompt

E:\wrox\2000conference\netcomponents\codegen>dir
 Volume in drive E has no label.
 Volume Serial Number is E861-E234

 Directory of E:\wrox\2000conference\netcomponents\codegen

08/23/2000  08:01p      <DIR>          .
08/23/2000  08:01p      <DIR>          ..
08/23/2000  08:01p               5,120 gen1e7.dll
08/23/2000  08:01p                   0 gen1e7.err
08/23/2000  08:01p                 990 gen1e7.out
08/23/2000  08:01p              17,920 gen1e7.pdb
08/23/2000  08:01p              11,606 gen1e7.simple_aspx.vb
08/23/2000  08:01p                   0 gen1E7.tmp
               6 File(s)         35,636 bytes
               2 Dir(s)   4,440,879,104 bytes free

E:\wrox\2000conference\netcomponents\codegen>_
```

If you open the assembly up using the ILDASM tool, you can see that the DLL created contains a single .NET component that derives from the `Page` class:



The assembly contains a namespace called `ASP25 (just a fairly random name)`, which contains a .NET component called `default_aspx`. The component naming simply reflects the page that was requested, with the any dots replaced with underscores. The `default_aspx` class shown derives from the **System.Web.UI.Page**. This class in turn derives from the `Control` class:



> **All components must implement `IComponent`. This interface provides the commonly used `Dispose()` method, and also exposes an `ISite` property that facilities the logical containment of components (e.g. a component uses the `ISite` interface to talk to its container).**

At runtime, this compiled page component is actually rendered by code that looks something like this:

```
default_aspx page;

page = new default_aspx;
page. RenderControl(writer);
```

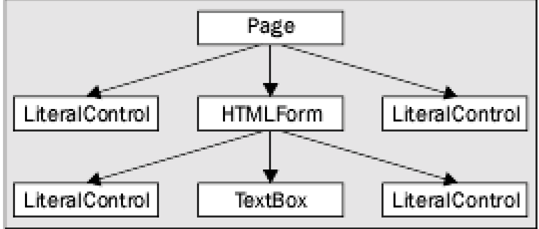
where `writer` is defined as the class `HtmlTextWriter`. As we'll see later on, this class provides methods for creating HTML output and manages tag starts/ends, tag stacking, text indenting, attributes creation, etc.

An ASP+ page itself uses a number of ASP+ controls to generate its output. In the sample earlier, the ASP+ page would use a single component to generate the page output. The prefix `asp:` indicates to the ASP+ page compiler that a .NET component is being declared (assuming the `runat=server` attribute is specified). To determine the actual class it has to create, ASP+ removes the `asp:` prefix and searches for a match in the namespaces `System.Web.Htmlcontrols` or `System.Web.UI.WebControls`. We'll talk more about user-registered namespaces later, where essentially the same searching happens, except private namespaces (also called user namespaces) are searched.

To understand how controls are used to output a page, lets take a slightly more complex page as shown here and examine the control hierarchy (see the `docs/simple1.aspx` file):

```
<html>
 <body>
   <form method="post" runat=server>
    Name: <asp:textbox runat=server/>
   </form>
 </body>
</html>
```

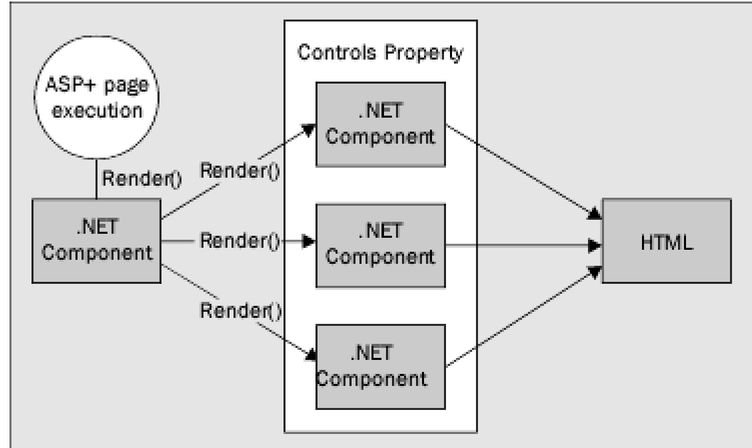
The ASP+ page compiler will output an ASP+ page component that will use six child components (controls) to create the page when rendered:



The `LiteralControl` control is created by the ASP+ page compiler for any text (including tags) that it finds between tags that are not marked with the `runat=server` attribute. The `HtmlForm` control represents the `<form>` element and so:

> **All classes in the `system.web.ui.htmlcontrols` namespace are prefixed with `Html`.**

Although somewhat simplified, this diagram gives you an idea of how this recursive nesting of child controls produces HTML output at runtime:



Each ASP+ control can have one or more child components which are held in the `Controls` collection, a property of the `control` class. The default rendering behavior for a control is to enumerate the `Controls` collection, and ask each contained control to render itself by calling its `RenderControl` method. This process is then recursively applied. The `controls` collection effectively provides a way to navigate down the control hierarchy. The `Parent` property of the `Control` class allows you to move up the control hierarchy.

Control Properties / Attributes on Elements

When an HTML element is marked with the `runtat=server` attribute, there is effectively a one to one mapping with a .NET component. Furthermore, each attribute specified on the tag is assumed to be a property of component. So, if we have:

```
<asp:somecontrol attrib1="a" attrib2="b" runat=server />
```

Code is effectively generated like this:

```
somecontrol ctl = new somecontrol;
ctl.attrib1 = "a";
ctl.attrib2 = "b";
```

If you specify an attribute for a control that does not map to a class property, the default controls in the `system.web.ui.HtmlControls` and `system.web.ui.WebControls` namespaces will just emit what you typed into the outputted HTML (see the `docs\badattrib.aspx` file). In your control you can choose to adopt the same behavior or you can write your control such that attributes must map to properties.

ASP+ Custom Server Controls

Writing an ASP+ server control is relatively simple, thanks to the `WebControl` and `Control` base classes. These provide most of the plumbing code required to interact with the ASP+ framework to render output.

Sample6

Here is a simple C# control that outputs the date and time:

```
using System;
using System.Web;
using System.Web.UI;

namespace WroxControls
```

```
{

public class SystemDateCS : Control
 {

// Draw the UI

 protected override void Render(HtmlTextWriter objWriter)
  {

  objWriter.Write(System.DateTime.Now);
  }
 }

}
```

This class derives from the `Control` class and overrides the `Render` method to output the current date and time. For ASP+ to use our class we have to compile it. You can do this with VS.NET or, as I've being doing to date, you can create a simple batch file to build the control for you:

```
set compiler=csc
set outdir=..\bin\datecs.dll
set inc=System.Web.dll

%compiler% /target:library /out:%outdir% /reference:%inc% date.cs
```

Incase you're a VB developer, here is the VB source for the same control:

```
imports System
imports System.Web
imports System.Web.UI

namespace WroxControls

  public class SystemDateVB

    inherits Control

'   Draw the UI

    Overrides protected Sub Render(objWriter as HtmlTextWriter)

    objWriter.Write(System.DateTime.Now)
    End Sub

  End Class

end namespace
```

and this is the make file:

```
set compiler=vbc
set outdir=..\bin\datevb.dll
set inc=System.Web.dll

%compiler% /target:library /out:%outdir% /reference:%inc% date.vb
```

Sample7

Let's take a brief look at some of the functions of the HTML text writer that help us generate HTML output (or any other XML based markup language) in our controls:

```
using System;
using System.Web;
using System.Web.UI;

namespace WroxControls
{

  public class SimpleControl : Control
  {

  // Draw the UI

    protected override void Render( HtmlTextWriter     objWriter)
    {

    objWriter.AddStyleAttribute("color", "white" );
    objWriter.AddStyleAttribute("background-color", "red" );

    objWriter.AddStyleAttribute("height", "100" );
    objWriter.RenderBeginTag("div");

    objWriter.RenderBeginTag("p");

    objWriter.Write("Hello World!");

    objWriter.RenderEndTag();

    objWriter.RenderEndTag();

    // Blue div

    objWriter.AddStyleAttribute("font-family", "arial");

    objWriter.AddStyleAttribute("font-size", "26");
    objWriter.AddStyleAttribute("background-color", "blue");

    objWriter.RenderBeginTag("div");

    objWriter.AddAttribute("hello","world");
    objWriter.RenderBeginTag("p");

    objWriter.Write("Hello World!");

    objWriter.RenderEndTag();

    objWriter.RenderEndTag();

    }
  }

}
```

This renders the following (tidy!) HTML:

```
<html>
<body>

<div style="color:white;background-color:red;height:100;">
  <p>Hello World!</p>
</div>

<div style="font-family:arial;font-size:26;background-color:blue;">
  <p hello="world">Hello World!</p>
</div>

</body>
</html>
```

The `HtmlTextWriter` control effectively manages the stack of elements being rendered, takes care of outputing the correct closing tag names, adding attributes, and even building the style attribute. The attributes and style collections are reset after each call to `RenderBeginTag()`.

Page Generation /Life Cycle

Before going much further into the world of ASP+ control development, we need to take a closer look at the complete life cycle of an HTML page, from the initial rendering, through form submits, etc.

The basic execution sequence in ASP+ for a page is as follows:

- OnInit**LoadState**
- **Process posted data**
- **Validation controls checked**
- OnLoad
- **Process posted data (second chance)Postback data change notification**
- **Postback event firing**
- PreRender
- SaveState
- Render
- Dispose

When you first see this list it can be a bit overwhelming. The items shown in **bold** only occur when a postback occurs, that is, when an HTTP post has been submitted automatically by ASP+ or because the user has clicked a submit button.

Managing State

If our controls need to maintain state, they can achieve this in a number of different ways. Like ASP, controls can use the Application or Session services, but these consume server resources and, of course, require server affinity in some scenarios. ASP+ provides a page scope persistence mechanism that is just as easy to use, but which uses hidden fields to persist state.

Using the `StateBag` class (that is, exposed as the `State` property of the control class), a control can save state. This is shown in this property accessor:

```
Public Property Count as integer
```

```
      get
        dim obj as Object = State("Count")

        if obj is nothing then
          return 1
        end if

        Count = CInt(obj)
          end get

      set
          if ( value < 0 ) then
           throw new ArgumentOutOfRangeException()
        end if

        State("Count") = value
      end set

End Property
```

ASP+ automatically roundtrips control state using view state, a specially encoded field that is placed into a hidden form field. The golden rule to remember with state management is that all state is saved before the render method is called.

Handling PostBack

Once a page has been rendered, it is very likely that a user is going to interact with the page, and at some point data is going to be submitted back to the server. When this occurs, ASP+ magically creates the page object to handle the request, and takes care of a lot of basic plumbing tasks for you. For example, all posted data within a form will be automatically pushed into the server side controls that caused the originating HTML tags to be generated.

If your control needs to interact with the data posted back, and maybe raise events if data has changed, your control must implement the `IPostBackDataHandler` Interface. This interface has two methods:

```
bool LoadPostData(string postDataKey, _
                  NameValueCollection postCollection);
```

and

```
void RaisePostDataChangedEvent();
```

`LoadPostData` is called to allow your control to examine its own postback data, and then update its state. To understand this, look at this simple implementation:

```
public virtual bool LoadPostData(string postDataKey, NameValueCollection
postCollection) {
  string current = Text;
  string postData = postCollection[postDataKey];

  if (!current.Equals(postData))
  {
   Text = postData;
  return true;
  }
  return false;
  }
```

A `NameValue` collection class is passed which contains all posted data, along with a key that allows the control to retrieve its postback value. The control can examine this data, and then decide whether to update its state based upon that data. If the state is updated, the control can then decide to return `True` – to indicate that it wants to raise an event – once *all* controls have processed their postback data.

`RaisePostDataChangedEvent` is called for a control once all postback data for all controls has been processed. This is required since the actions taken in an event handler for one control may be dependant upon the state of another control. If the event handler was invoked before all controls had initially loaded their postback data, inconsistencies would arise – hence ASP+ has a two stage process for handling postback.

The second time that PostBack is executed in ASP+, only postback data not processed the first time is processed. This is because controls can be created in the `OnLoad` method of controls that occur after the original postback.

Controls can output HTML that causes a postback by using the `Page.GetPostBackEventReference` method as shown here:

```
writer.AddAttribute( HtmlTextWriter.ATTR_ONCHANGE,"javascript:" +
Page.GetPostBackEventReference(this));
```

This particular piece of code will cause a postback whenever the value of a text box changes.

Handling Events (sample11)

Events allow a control to be notified when the user interacts with an element of UI it generated. For example, imagine a control that rendered a simple database navigation UI. The control needs to know when the user has pressed the forward, back, stop, play buttons, etc. To achieve this, ASP+ allows you to emit JavaScript that automatically causes a postback and details the event. This is sent to the control via the `RaisePostBackEvent` method of the `IPostBackEventHandler` interface. The prototype for the method is shown here:

```
void RaisePostBackEvent(string eventArgument);
```

Events are generated using the `Page.GetPostBackEventReference` method. For example:

```
writer.AddAttribute(HtmlTextWriter.ATTR_HREF,"javascript:" +
Page.GetPostBackEventReference(this));
```

Events in ASP+ are also driven from postback data. However, the postback data uses reserved names (`__EVENTTARGET` and `__EVENTARGUMENT`) so ASP+ knows to generate an event.

Here is the client side HTML emitted by ASP+ for our simple postback button:

```
<html>
<body>

<FORM name="ctrl1" method="post" action="cs.aspx" id="ctrl1">
<INPUT type="hidden" name="__EVENTTARGET" value="">
<INPUT type="hidden" name="__EVENTARGUMENT" value="">
<INPUT type="hidden" name="__VIEWSTATE" value="a0z924111719__x">

<script language="javascript">
  <!--
```

```
    function __doPostBack(eventTarget, eventArgument) {
      var theform = document.ctrl1
      theform.__EVENTTARGET.value = eventTarget
      theform.__EVENTARGUMENT.value = eventArgument
      theform.submit()
    }
  // -->
</script>

  <a href="javascript:__doPostBack('ctrl3', '')">Go to MS.COM</a></strong>
</FORM>

</body>
</html>
```

Summary

ASP+ gives you a powerful model for creating re-usable UI elements for web applications. Whether you choose to develop ASP+ controls or not will really depend on the types of application you write.

Further Resources

"A Preview of ASP+", Wrox Press (ISBN: 1861004753).
ASP+ Quickstart – distributed with .NET and available online at www.aspng.com.