Evolving Custom Controls

Steven Smith
ASPAlliance.com

December 2003

Applies to:
Microsoft® ASP.NET

Summary: ASP.NET's support for Web controls provides an excellent way to package up commonly used behaviors and deploy them to other developers. But when should you build a control, and when should you just add the behavior to a Web form directly? This article examines this issue, and steps through the evolution of a control from behavior on a Web form to full-fledged control. (18 printed pages)

Download EvolvingControlsSample.msi.

Contents

Introduction

Controls in Microsoft® ASP.NET exist to eliminate duplicate, redundant code. After a control has been properly built, tested and deployed, developers can easily use it on many Web forms by dragging it onto the form and modifying its properties. Unfortunately, it takes a fair amount of effort to get a control to this stage. Progress would slow to a crawl if everything in an application that might ever be duplicated were written as a deployable control. Thus, the decision to convert some functionality into a control should not be taken lightly, or be made too early in the life of an application. It is much easier to build, maintain, and debug functionality in a Web form than in a control, so usually it is best to start in this form and move toward more reusable techniques only when it becomes necessary.

A particularly common task in Web applications is data validation. ASP.NET provides a suite of validation controls which can greatly simplify this task. However, these controls are generic, so for common pieces of data, the configuration of these controls may be repeated on many pages in a site. One of the most common pieces of data entered into most Web application is an email address. Validating email addresses usually involves the use of regular expressions at a minimum, perhaps combined with a RequiredFieldValidator and sometimes a custom validator to check if an MX record exists for the domain specified. All of these controls and their properties, then, may be repeated on many different pages within an application or organization. This repetition makes this behavior a prime candidate for evolution into a control, and it is this process that I will demonstrate in this article.

Using the scenario described in the next section, I will evolve a piece of functionality through the following stages of development:

1. Web Form
2. Web User Control
3. Composite Control

a. Simple
b. Including Templates
c. Including Designer Support
d. Including Configuration Support

Scenario

So, our task is to accept an email address in a textbox, and to validate it using a regular expression. We should use the standard validation controls so that any error messages are tabulated in a ValidationSummary control, and we should provide an easy way to separate the input textbox from the validation controls, so that they could be placed in a separate table cell if desired.

Step 1: Web Form

The simplest way to meet the requirements specified in the scenario is to place a TextBox on a Web form along with a couple of validation controls. This approach takes only a few minutes to build, with the end result looking something like Figure 1.

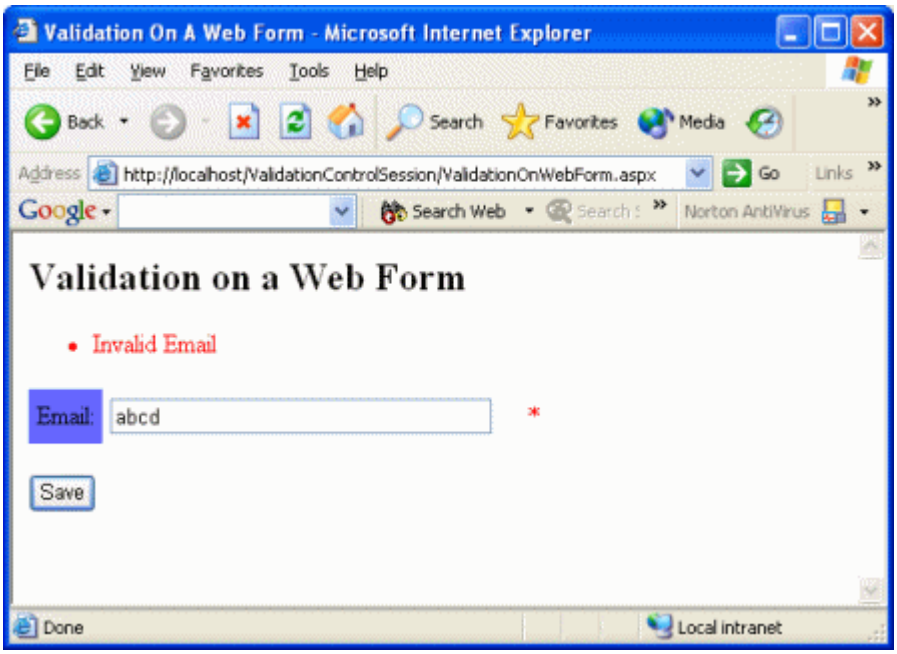

Figure 1. Validating an email address textbox on a Web form using ASP.NET validation controls

Note that in order to meet the last requirement in the scenario, this example places the validation controls in a separate column of the HTML table from the TextBox. We'll see that while this is simple to do now, it will be a bit more complicated to allow for this functionality as we begin encapsulating the control logic. In this case, all of the relevant code for this example is encapsulated in the first and only row of the HTML table, shown here:

```
<tr>
   <td>Email:</td>
   <td bgcolor="#ffffff"><asp:TextBox id="EmailTextBox" runat="server"
     Width="238px"></asp:TextBox></td>
   <td bgcolor="#ffffff">
      <asp:RequiredFieldValidator id="RequiredFieldValidator1"
        runat="server" ErrorMessage="Email required." Text="*"
        ControlToValidate="EmailTextBox" />
      <asp:RegularExpressionValidator id="RegularExpressionValidator1"
```

```
        runat="server" ErrorMessage="Invalid Email" Text="*"
        ControlToValidate="EmailTextBox" ValidationExpression="[\w-
        ]+(?:\.[\w-]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}" />
   </td>
</tr>
```

> **Note** There are many regular expressions that can be used to validate email addresses. Some are better than others, but none can really tell whether an email address belongs to the person who is entering it. You can find a library of regular expressions for email addresses and many other purposes at http://RegExLib.com/.

As you can see, there isn't much required to meet the scenario requirements in this simple case. Consider now, that you are the developer who has just created this Web form. You're asked to build another form that will also have an email address input on it that must be validated. For some, this will be enough duplication of code for you to want to package up the duplicate logic into some kind of a reusable format. Others will simply whip out their good old cut-and-paste technique and be done with the second page in no time. To bring these folks along with the rest of us, let's continue this fantasy by suggesting that the next day, another ten pages need building with the same validation logic. And by the way, the regular expression needs replacing with one that works better. With this turn of events, only the most hardcore cut-and-paster will resist the urge to eliminate duplicate code, and move on to Step 2 in the evolution of controls.

Step 2: Web User Control

When you have some repeating UI behavior that you'd like to encapsulate so you can use it across multiple Web forms, the easiest way to achieve this is with a Web user control. By doing nothing more than removing the contents of the `<tr>...</tr>` tag in Step 1 and placing it in a user control (ValidationInUserControl.ascx), we can achieve this reuse. The resulting user control looks like this:

```
<%@ Control Language="c#" AutoEventWireup="false"
  Codebehind="ValidationInUserControl.ascx.cs"
  Inherits="ValidationControlSession.ValidationInUserControl1"
  TargetSchema="http://schemas.microsoft.com/intellisense/ie5"%>
<asp:TextBox id="EmailTextBox" runat="server"
  Width="238px"></asp:TextBox>
</td>
<td bgcolor="#ffffff">
   <asp:RequiredFieldValidator id="RequiredFieldValidator1"
     runat="server" ErrorMessage="Email required." Text="*"
     ControlToValidate="EmailTextBox" />
   <asp:RegularExpressionValidator id="RegularExpressionValidator1"
     runat="server" ErrorMessage="Invalid Email" Text="*"
     ControlToValidate="EmailTextBox" ValidationExpression="[\w-
     ]+(?:\.[\w-]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}" />
```

The table row from Step 1 changes like so:

```
<tr>
   <td>Email:</td>
   <td bgcolor="#ffffff">
      <uc1:ValidationInUserControl id="ValidationInUserControl1"
        runat="server"></uc1:ValidationInUserControl>
   </td>
</tr>
```

Finally, the page needs to reference the user control using the Register page directive:

```
<%@ Register TagPrefix="uc1" TagName="ValidationInUserControl"
  Src="ValidationInUserControl.ascx" %>
```

In the situation where only a couple of developers will ever need the functionality, a user control is a great solution. It's very quick to implement, requiring virtually no changes in syntax from the Web form technique. By default, user controls cannot be used across application domains, which can greatly limit their usefulness as generic reuse containers. However, this limitation is easily overcome by following the steps laid out in this Knowledge Base article titled, HOW TO: Share ASP.NET Pages and User Controls Between Applications by Using Visual Basic .NET. Eventually, though, the need to share this functionality with other developers, or the need for design time support, will push the functionality from a user control.

Step 3: Composite Control

Returning to our role-playing session, let's now envision that we have our functionality encapsulated in a user control, and it's working great for us. However, now fifty other developers, including some designers, both in your company and in some of your partners' organizations, need this same functionality. Naturally you don't want to just send them all an .ascx file, especially since the designers will immediately complain that it doesn't work in the designer (I realize it's a stretch that designers are using Microsoft Visual Studio® .NET, but we're pretending). Thus, a solution that retains the functionality but is easier to deploy and renders well in a WYSIWIG designer is necessary. This leads us to the third and final big step in the evolution of custom controls: the custom control itself.

There are several different kinds of custom controls. A custom control is really anything that derives from System.Web.UI.Control, whether directly or indirectly (by inheriting from something that inherits from System.Web.UI.Control). However, within this broad definition, a subclass of custom controls is the composite control. A composite control is a control that holds several other controls. It is basically a way of gluing together a few controls into one larger-grained control, and is in my opinion the simplest way to create non-trivial behavior in a custom control. For our scenario, the composite control makes the most sense, since we are moving from a user control that includes three Web controls already. Since this solution works, we'd rather continue with the same basic design rather than start from scratch and re-create or extend the TextBox control, for instance.

> **Note** Before going any further into the depths of building custom controls, I must tell you that if you're going to be building controls yourself, you should strongly consider buying "Developing Microsoft ASP.NET Server Controls and Components" by Nikhil Kothari and Vandana Datye, from Microsoft Press. This book is the bible on control development and its lead author, Nikhil Kothari, is a member of the ASP.NET team and wrote many of the controls that ship with ASP.NET.

Simplest Case

Before jumping into the Composite Control code, we need a base class to handle some plumbing required of all composite controls. This base class is taken from Nikhil Kothari's book, referenced in the note above.

```
public abstract class CompositeControl : WebControl, INamingContainer
{
   public override ControlCollection Controls
   {
      get
      {
         EnsureChildControls();
         return base.Controls;
      }
   }
}
```

With composite controls, all of the heavy lifting of the control is done in the **CreateChildControls()** method. This method must be called before any of the child controls are referenced, but it is not guaranteed to be called at any particular point in the control's life cycle prior to **Render()**. Thus, before doing anything that might reference a child control, the **EnsureChildControls()** method should be called. This method simply checks to see if **CreateChildControls()** has been called yet, and if not, it calls it. Otherwise, it does nothing. Since any access to the control's Controls collection would require the child controls to have been created, the base class is inserting

the **EnsureChildControls()** check into the property get for the Controls collection. Also, all composite controls should be marked with the INamingContainer interface. This interface has no implementation but is required to ensure that the subcontrols are given proper names when the page is parsed. By implementing this interface here in the base class, none of the classes that derive from this base class will need to do so.

Now, with the base class complete, we can create our own composite control. Most of the work of the control will be done in its **CreateChildControls()** method. However, another important piece of the control is its properties, and how they are implemented. For this control, we are only going to implement a few simple properties, all of which will correspond to properties of child controls. We will show how these are implemented after examining the **CreateChildControls()** method.

```
protected override void CreateChildControls()
{
   Controls.Clear();

   _emailTextBox = new TextBox();
   _emailTextBox.ID = "emailTextBox";

   _separatorLiteral = new Literal();

   _emailRequiredValidator = new RequiredFieldValidator();
   _emailRequiredValidator.ID = "emailRequiredValidator";
   _emailRequiredValidator.ControlToValidate = _emailTextBox.UniqueID;
   _emailRequiredValidator.Text = "*";
   _emailRequiredValidator.ErrorMessage = "(Default Message) - Email
     Required.";
   _emailRequiredValidator.Display = ValidatorDisplay.Dynamic;

   _emailFormatValidator = new RegularExpressionValidator();
   _emailFormatValidator.ID = "emailFormatValidator";
   _emailFormatValidator.ControlToValidate = _emailTextBox.ID;
   _emailFormatValidator.ValidationExpression = @"[\w-]+(?:\.[\w-
     ]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}";
   _emailFormatValidator.Text = "*";
   _emailFormatValidator.ErrorMessage = "(Default Message) - Email
   Invalid.";
   _emailFormatValidator.Display = ValidatorDisplay.Dynamic;

   this.Controls.Add(_emailTextBox);
   this.Controls.Add(_separatorLiteral);
   this.Controls.Add(_emailRequiredValidator);
   this.Controls.Add(_emailFormatValidator);
}
```

Basically, this method is taking the exact same controls used in the Web form and Web user control, and programmatically instantiating them, setting their properties, and adding them to the control's Controls collection. The only control described here that wasn't used in the previous examples is the _separatorLiteral, which is used to hold the HTML that is rendered between the TextBox and the validation controls. The order in which the controls are added at the end of the method is important, since this is the order in which they will be rendered.

This control exposes four properties, which can be used to override the default error messages, HTML separator code, and TextBox text. The properties are **Text**, **SeparatorHtml**, **RequiredErrorMessage**, and **InvalidErrorMessage**, all of String type. Their implementation is identical, simply acting as a facade to the child controls. The **Text** property is shown here; the others are implemented similarly.

```
[
Bindable(true),
Category("Appearance"),
DefaultValue(""),
Description("The text to display in the Email textbox.")
]
public string Text
{
```

```
   get
   {
      EnsureChildControls();
      return _emailTextBox.Text;
   }
   set
   {
      EnsureChildControls();
      _emailTextBox.Text = value;
   }
}
```

Instead of storing the value in a private local variable or ViewState, as is typical, these values are stored directly within the child controls. Naturally, this requires the child controls exist at the time their properties are read or written, and so **EnsureChildControls()** is called before each such reference.

Using this control in a Web form is as straightforward as using the Web user control, but with a bit better design-time support and a reference to an assembly instead of to a user control file. This control could be placed in the global assembly cache (GAC), if desired, in which case it would be accessible by every application on a machine.

```
<tr>
   <td>Email:</td>
   <td bgColor="#ffffff">
      <cc1:emailboxsimple id="EmailBoxSimple1" runat="server"
        Width="320px" Height="27px" SeparatorHtml='</td><td
        bgcolor="#FFFFFF">'
         Font-Bold="True" ForeColor="Blue"></cc1:emailboxsimple>
   </td>
</tr>
```

The new page directive to register this control would look like this:

```
<%@ Register TagPrefix="cc1"
  Namespace="ValidationControlSession_Controls"
  Assembly="ValidationControlSession_Controls" %>
```

Notice that because our control, EmailBoxSimple, inherits from WebControl (via the base class, CompositeControl), we can specify height, width, and style attributes for the control. This is free functionality we pick up through the beauty of inheritance.

Now the simple control is done. It works, but it's not perfect. For instance, it seems pretty clumsy to be passing in raw HTML as a string parameter to the control, and the validation expression it is using is hard coded, so it's not very easy to change.

Composite Control with Template Support

For this example, it would be pretty easy to argue that it doesn't merit a template for the separator HTML property. However, it does make the control a bit more powerful, and it's a great way to demonstrate how to enhance controls by adding templates, so let's do it even if it's not merited.

Templates, if you're not familiar with them, allow developers to add dynamic content to a control declaratively. The Repeater, DataList, and DataGrid are the most common template-driven controls in ASP.NET 1.x. The controls use templates to describe Items, Headers, Separators, and so on. For the EmailBox control we're building, we need a SeparatorTemplate.

There are several pieces involved in getting templates to work. Each template must be exposed as a property of type ITemplate. The templates need to be instantiated in a container during the **CreateChildControls()** method. Finally, the container itself must be defined, and perhaps written to provide some base level of functionality. The property for the SeparatorTemplate (which will replace the SeparatorHtml property from the previous code) would look like this:

```
private ITemplate _separatorTemplate;
[
Browsable(false),
DefaultValue(null),
TemplateContainer(typeof(EmailBox))
]
public virtual ITemplate SeparatorTemplate
{
   get
   {
      return _separatorTemplate;
   }
   set
   {
      _separatorTemplate = value;
   }
}
```

Apart from the new type and the addition of the **TemplateContainer()** attribute, this property is no different from the previous implementation. The **TemplateContainer()** attribute must specify the type of the class in which the template will be instantiated. This can either be the custom control itself (EmailBox in this case) or a separate class devoted to this purpose (for an example of this method, see Templated Control Sample. Note that templates should not normally show up in the property browser, so the **Browsable()** attribute is set to false.

Instantiating the template and adding it to the controls tree is done in CreateChildControls(), as you might expect, but it uses a slightly different syntax than the controls we have seen thus far. Instead of just adding the literal control to the collection, the template's **InstantiateIn()** method is called. This method takes the container as a parameter, so in this case the parameter to pass would be `this`. The final code used to add all the controls to the Controls collection at the end of CreateChildControls looks like this:

```
this.Controls.Add(_emailTextBox);
ITemplate template = SeparatorTemplate;
if (template == null)
{
   template = new DefaultSeparatorTemplate();
}
template.InstantiateIn(this);
this.Controls.Add(_emailRequiredValidator);
this.Controls.Add(_emailFormatValidator);
```

In this case, if no template is specified at all, a default will be used. This is an easy technique you can use with any templated control, and requires a class for the default template, which can usually be placed in the same file as the control. The DefaultSeparatorTemplate class is shown here:

```
private sealed class DefaultSeparatorTemplate : ITemplate
{
   void ITemplate.InstantiateIn(Control container)
   {
      Literal separatorLiteral = new Literal();
      separatorLiteral.Text = " ";

      container.Controls.Add(separatorLiteral);
   }
}
```

In this case, the default template will render a space between the TextBox control and the validation controls.

Composite Control with Designer Support

Another feature of custom controls that separates them from user controls is their designer support. User controls have no designer support—they display as little grey boxes. Custom controls have good designer support with

minimal effort. In this case, since the whole control can be rendered at design time just as easily as at runtime, the control's designer looks good even without any customization. However, that is not always the case, so the next thing we'll briefly look at is how to improve designer support for custom controls.

One thing we can definitely improve for this control is its toolbox icon. By default, custom controls will use a gear icon when added to the toolbox of Visual Studio of Web Matrix. Changing this default icon requires a few steps to be followed exactly.

1. Add a bitmap to the control project. Name it the same as the control class that it will represent.
2. In the bitmap file's **file** properties, set its Build Action to Embedded Resource.
3. Open the bitmap for editing. Set its height and width to 16 pixels.
4. Edit the bitmap. Note that whatever color is specified in the lower left corner of the bitmap will be treated as transparent.
5. Compile the project and add the control to the toolbox—the new bitmap should be shown there. More information and troubleshooting steps for this process can be found in Creating Custom ASP.NET Controls in VS.NET.
6. Beyond a toolbox icon, the design time behavior of the control on the design surface of the IDE can be customized by adding an attribute to the control's class specifying which class to use for its design time behavior. The class declaration, complete with designer attribute, looks like this:

```
[
DefaultProperty("Text"),
Designer(typeof(ValidationControlSession_Controls.Design.EmailBoxDesign
  er))
]
public class EmailBox : CompositeControl { . . . }
```

Like the **TemplateContainer** attribute, the **Designer** attribute expects the type of a class as its parameter. In this case, that class should be a child of the CompositeControlDesigner base class, which again can be found in Nikhil Kothari's book on custom controls. The base class is shown here:

```
public class CompositeControlDesigner : ControlDesigner
{
   public override string GetDesignTimeHtml()
   {
      // Retrieve the controls to ensure they are created.
      ControlCollection controls = ((Control)Component).Controls;
      return base.GetDesignTimeHtml();
   }

   public override void Initialize(IComponent component)
   {
      if (!(component is Control) &&
         !(component is INamingContainer))
      {
         throw new ArgumentException("Component must be a container
         control.", "component");
      }
      base.Initialize(component);
   }
}
```

This base class, like the CompositeControl base class shown above, ensures that the child controls of the composite control are instantiated before they are needed. In this case, creating a reference to the custom control's Controls collection is sufficient to ensure that all of the child controls are created, since this will invoke the **get()** property specified in the CompositeControl base class. This class also does some type checking to ensure that the control referencing it is of the proper type and has implemented the proper interface.

Armed with a base class, we can then implement the designer itself, which inherits from CompositeControlDesigner. There are three methods that may be overridden in a designer, and these are shown

in the listing for EmailBoxDesigner below:

```
public class EmailBoxDesigner : CompositeControlDesigner
{
   public override string GetDesignTimeHtml()
   {
      // add custom logic here if desired.
      return base.GetDesignTimeHtml();
   }

   protected override string GetEmptyDesignTimeHtml()
   {
      return "[EmailBox]";
   }

   protected override string GetErrorDesignTimeHtml(Exception e)
   {
      return "[EmailBox - With Errors]";
   }
}
```

In this case, since the composite control renders fine out-of-the-box (since its child controls all work fine at design time), there isn't much to do here. However, with this skeleton code, you could easily implement whatever design time support you needed for your control.

Composite Control with Configuration Support

Returning one last time to our fantasy, let us imagine that the fifty developers (and designers) across several organizations who are using your control would now like to be able to customize more of the behavior of the control. "No problem," you think. "I can just add more properties."

"Well," they say. "We'd really rather not have to set a bunch of properties. We want to change the default behavior of the control to suit us."

"Oh," you think. While considering giving each of them the full source to the control so they can recompile it whenever they want to change its behavior within their organization, you come up with another idea—using a configuration file to manage the default behavior of the control.

Certainly you could build configuration support into everything you write from day one, but again this would slow your development down considerably and add a great deal more functionality to test and debug that is not being used. However, once there is demand for such support, as in the above scenario, it can be a great way to increase the flexibility of your control so that it can be made to work in a great many situations without recompiling it.

ASP.NET and the .NET Framework leverage configuration files quite a bit, and the .NET Framework's System.Configuration library provides great tools for adding configuration support to your own controls. Assuming we're going to use the web.config file for ASP.NET custom controls, there are basically three ways you can store your control's configuration data:

- <appSettings />—You can store your variables as name-value pairs in the appSettings collection of the web.config file.
- Custom configSection using NameValueFileSectionHandler—This uses the same format as the appSettings collection, but with a name you provide.
- Custom configSection with custom SectionHandler—This is a complete roll-your-own solution.

In the first case, you risk using keys that collide with keys used by the application for its own configuration settings. Although this is the simplest technique, it should usually be avoided for this reason. If you do decide to use it, I recommend you name your keys with a prefix identifying your component, like EmailBox.RequiredErrorMessage.

The second example provides a good mix of ease-of-use and flexibility. It takes just a couple of minutes to set up, and does not have the name collision risk that using the appSettings collection carries with it. This is the technique I will show for this article.

The third example does not require a huge amount of work, assuming your configuration data is not very complex. You can find more information on how to write your own configuration section handler at Creating New Configuration Sections.

For the EmailBox control, there are seven properties that I'm going to provide configuration support for. These include the **Text**, **ErrorMessage**, and **Display** of the RequiredFieldValidator and the RegularExpressionValidator, plus the **ValidationExpression** of the RegularExpressionValidator. In the web.config file, inside the <configuration> node, we'll need to add the following:

```
<configSections>
   <section name="EmailBox"
     type="System.Configuration.NameValueFileSectionHandler, System,
     Version=1.0.3300.0, Culture=neutral,
     PublicKeyToken=b77a5c561934e089" />
</configSections>
<EmailBox>
   <add key="RequiredValidator_Text" value="+" />
   <add key="RequiredValidator_ErrorMessage" value="(Config) Email
     Required." />
   <add key="RequiredValidator_Display" value="Static" />
   <add key="FormatValidator_Text" value="+" />
   <add key="FormatValidator_ErrorMessage" value="(Config) Email
     Invalid." />
   <add key="FormatValidator_ValidationExpression" value="Bob" />
</EmailBox>
```

The System.Configuration.NameValueFileSectionHandler is the class that the appSettings section is handled by, and so provides very familiar syntax (not to mention not requiring us to write our own section handler). Within the section specified ("EmailBox"), each configuration data element is added by using the <add key='" value="" /> syntax, just like with appSettings.

Within the control, the trick is to read the configuration data at the right point in the control's life cycle, so that configured values can be properly over-ridden by properties at runtime. Most of this work is done in the **CreateChildControls()** method, because this method is called prior to any property calls that reference child control properties, and that's the only thing we're providing configuration support for in this example. As we're not wasting any trees, the complete **CreateChildControls()** method for the final version of the control is shown below:

```
private const string _configPath = "EmailBox";
private NameValueCollection config;
protected override void CreateChildControls()
{
   Controls.Clear();

   // load config
   NameValueCollection webconfig = (NameValueCollection)
      ConfigurationSettings.GetConfig(_configPath);
   config = new NameValueCollection(10);
   if(webconfig!=null)
   {
      config.Add(webconfig);
   }
   // fill config with programmatic defaults if none specified in
     config file
   if (config["RequiredValidator_Text"] == null)
     config.Add("RequiredValidator_Text","*");
   if (config["RequiredValidator_ErrorMessage"] == null)
     config.Add("RequiredValidator_ErrorMessage","(Default Message) -
     Email Required.");
```

```
    if (config["RequiredValidator_Display"] == null)
      config.Add("RequiredValidator_Display","Dynamic");
    if (config["FormatValidator_Text"] == null)
      config.Add("FormatValidator_Text","*");
    if (config["FormatValidator_ErrorMessage"] == null)
      config.Add("FormatValidator_ErrorMessage","(Default Message) -
      Email Invalid.");
    if (config["FormatValidator_ValidationExpression"] == null)
      config.Add("FormatValidator_ValidationExpression",@"[\w-
      ]+(?:\.[\w-]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}");
    if (config["FormatValidator_Display"] == null)
      config.Add("FormatValidator_Display","Dynamic");

    _emailTextBox = new TextBox();
    _emailTextBox.ID = "emailTextBox";

    _emailRequiredValidator = new RequiredFieldValidator();
    _emailRequiredValidator.ID = "emailRequiredValidator";
    _emailRequiredValidator.ControlToValidate = _emailTextBox.ID;
    _emailRequiredValidator.Text = config["RequiredValidator_Text"];
    _emailRequiredValidator.ErrorMessage =
      config["RequiredValidator_ErrorMessage"];
    _emailRequiredValidator.Display =
      (ValidatorDisplay)Enum.Parse(typeof(ValidatorDisplay),
      config["RequiredValidator_Display"], true);


    _emailFormatValidator = new RegularExpressionValidator();
    _emailFormatValidator.ID = "emailFormatValidator";
    _emailFormatValidator.ControlToValidate = _emailTextBox.ID;
    _emailFormatValidator.ValidationExpression =
      config["FormatValidator_ValidationExpression"];
    _emailFormatValidator.Text = config["FormatValidator_Text"];
    _emailFormatValidator.ErrorMessage =
      config["FormatValidator_ErrorMessage"];
    _emailFormatValidator.Display =
      (ValidatorDisplay)Enum.Parse(typeof(ValidatorDisplay),
      config["FormatValidator_Display"], true);

    this.Controls.Add(_emailTextBox);

    // render the template
    ITemplate template = SeparatorTemplate;
    if (template == null)
    {
       template = new DefaultSeparatorTemplate();
    }

    template.InstantiateIn(this);

    this.Controls.Add(_emailRequiredValidator);
    this.Controls.Add(_emailFormatValidator);
}
```

The config collection is first populated from the web.config file's "EmailBox" section. None of the entries are required, so if the configured values don't exist, we populate the values of those items with hard-coded defaults. Now, when a property is set, the underlying control's property will have been set either to the configured value if there was one, or to the hard-coded value otherwise, but the property **set()** will override this value with its own. Thus, the control will work as expected, with properties trumping configured values, and configured values trumping hard-coded defaults.

Summary

Although custom controls can be written from scratch, it is more often the case that they should evolve from functionality that is first implemented directly on a web form. It is much easier to build, change, debug, and work with functionality in a web form than in a compiled control, so as the behavior is evolving, this is the best place

for the code to live. However, as reuse requirements increase, the code will evolve. In the short term, user controls may provide the needed reuse, but for many highly reusable behaviors, compiled controls are the best end solution. Once the basics are covered, compiled controls can be further enhanced with templates, designer support, and configuration support.

Resources

Walkthrough: Creating a Web Custom Control

Developing a Templated Control

Building Server Controls, Key Concepts (Presentation) and Building Server Controls, Advanced Topics (Presentation) available from http://swarren.net/

About the Author

Steven A. Smith, Microsoft ASP.NET MVP, is president and owner of ASPAlliance.com. He is also the owner and head instructor for ASPSmith Ltd, a .NET-focused training company. He has authored two books, the ASP.NET Developer's Cookbook and ASP.NET By Example, as well as articles in MSDN® and AspNetPRO magazines. Steve speaks at several conferences each year and is a member of the INETA speaker's bureau. Steve has a Master's degree in Business Administration and a Bachelor of Science degree in Computer Science Engineering. Steve can be reached at ssmith@aspalliance.com.

Top of Page