Your First ASP.NET Custom Control

<http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control>

By **[Refky Wahib](http://www.codeproject.com/script/Membership/View.aspx?mid=2574282)**, 23 Aug 2008

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | http://www.codeproject.com/script/Ratings/Images/stars-fill-lg.png  http://www.codeproject.com/script/Ratings/Images/stars-empty-lg.png | 4.42 (35 votes) | |  |

Top of Form

* [**Download demo project - 26 KB**](http://www.codeproject.com/KB/custom-controls/FirstCustomControl/ServerCustomerControl.zip)

CustomEmail for Asp.NET 

Introduction

This article is mainly for developers who have not much experience with ASP.NET custom controls. In this article, I will explain the concept behind custom controls, with some useful guidelines. In the source code, there are a few useful controls like Email link control that generate an email link that displays the name, email subject, and body of an email. There is another useful custom control, a custom edit box that has a bound label, so once you click the label, it will jump to the associated edit box and it is accessible as well. I will also try to focus on the benefits of creating custom controls for programmers who have not used custom controls at all, or for those who rely only on using user controls.

How to use the code

All the code is in one project library, so you can compile and reuse it, but the website is a simple tutorial that has links for testing, with a full description of all the features for each test. This solution was made with Visual Studio 2008 and C#.

Background

In my experience as a .NET programmer, I realised that after ASP.NET 2 was released, it minimized the need for creating custom controls. And, user controls have more great features compared with what they had in ASP.NET 1.0. For example, it was not easy to reference a user control and manipulate its state via code (there was a workaround), or to invoke the intellisense in code without casting (there was a workaround too). But more than that, the design view was very bad, you just see a gray box with the user control ID. I have seen a few programmers at work doing simple custom controls in ASP.NET 1.0, but most of the controls that I saw in ASP.NET 2.0 were user controls, not custom controls. One day, I made a quick technical session about data-bindable custom controls, and after the session, there was a lot of argument about why we should create complex code if user controls can do all that we need. And, in my work, I realised that it is essential to know how to create your own custom controls to solve many problems that you can’t solve using other methods. When I was learning to develop some AJAX enabled controls, I found myself dragged again to custom controls more than user controls, so I hope that this article will be useful as I expect, for non-custom-control developers.

Content

* What is a user control, and what is a custom control?
* Reasons for favouring custom controls over user controls.
* Toolbox, icon, and tag registration.
* ViewState round trip management.
* Simple custom controls vs. composite custom controls.
* Restore posted data .
* Managing simple events.
* Extending custom controls.
* Custom controls and Web Parts.
* Other features.
* Step by step summary for creating your first custom control.

What is user control, and what is a custom control?

User control, is a container for markup (HTML and Server controls) that you may use in any web page as a unit of code; you can add properties, events, and methods to make it fully customisable. User control creation is very simple compared with custom control creation, and normally, you build it visually, via drag and drop of controls from the Toolbox. Custom control, is a class extending the Control or WebControl class. You have to register the control before you consume it just as any .NET component in the Toolbox.

Reasons for favouring custom controls over user controls

Sure, we don’t have to create a custom control in all scenarios, but I will list a few scenarios that will make us think about custom controls as the first option.

* If you want to share your control across different applications, websites, and even external applications based on .NET, use custom controls, which are compiled components that are very easy to register and use.
* For security purposes, if you are developing a control that will be used in other organisations or sections, or the control has sensitive data, or does sensitive functionality, use custom controls, so you can compile and deploy them as an assembly, not like a user control that does not not have a compiled version.
* If performance and scalability are issues, always go for custom controls, simply because you can control the rendering process, and you can change the implementation even after release to give the same results with better performance.
* Rich design support: if you want to support design mode with rich tools, go ahead with custom controls. Uou can control the icon, tag prefix registration, and the generated markup in design time. You will be able to see a lot of built-in properties and events in the Properties window, and all custom properties and events as well. On the other hand, a user control has very poor design support, and you will see very few built-in properties, and no Events tab in the Properties window.
* To build your own library or organisation reusable library, we should go for custom control solutions.
* If you like to make your control interact with other controls and be fully supported in design mode, you should think of custom controls. For instance, custom controls can be contained by other controls, and you can drag page controls inside a custom control in design mode, but you can’t drag controls inside a user control in the main page. If you want to create a non-visual component and want it to be fully supported in design time, choose a custom control.
* A custom control is portable, not just for ASP.NET, but for SharePoint (if it extends the WebPart class). If you like to create an extendable web application that has a place holder for plug-ins, you will accept a custom control in the assemblies, so your website can be extended without touching the code.

Toolbox, icon, and tag registration

New Project template

When you create your project using the Visual Studio 8 ASP.NET Server Control template, it will generate this code for you:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

[DefaultProperty("Text")]

[ToolboxData("<{0}:WebCustomControl1 runat="server">")]

public class WebCustomControl1 : WebControl {

[Bindable(true)]

[Category("Appearance")]

[DefaultValue("")]

[Localizable(true)]

public string Text {

get {

String s = (String)ViewState["Text"];

return ((s == null) ? String.Empty : s);

}

set {

ViewState["Text"] = value;

}

}

protected override void RenderContents(HtmlTextWriter output) {

output.Write(Text);

}

}

It is all that you want to create a basic custom control. It defines the ToolboxData attribute that controls the registration tag in the ASPX page as server markup. It extends the WebControl which is the most abstracted visual control. It generates a property with some decorated attributes. You will see that all internal property fields are replaced with the ViewState object, and finally it renders the control as a simple text. We can modify this generated code a little bit to suit our needs:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

[ToolboxData("<{0}:CustomEditBox runat="\""server\" Label=\"[Label]\" Text=\"[Text]\">")]

I replaced the generated tag with a custom element. Notice that I added more attributes to be generated in the container ASPX page. For the component’s icon, all that you need is to create a BMP image with the same name as the custom control (you may have other controls in the same project, so you may have to create more than one icon image). The icon image is optional.

custom control icon

Finally, this icon will not appear in the tool box in the automatic registration tab. You should register the toolbox item through toolbox's Choose Item context menu, browse the assembly, then register the component. Now, you will see the icon in the toolbox.

http://www.codeproject.com/KB/custom-controls/FirstCustomControl/CustomEmailToolbox.jpg

Almost everything is ready except this code:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

[ToolboxData("<{0}:CustomEditBox runat="\""server\" Label=\"[Label]\" Text=\"[Text]\">")]

How does {0} get the value, and from where? Do not worry about it; you can register your own TagPrefixusing this attribute:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

[assembly: TagPrefix("FirstCustomControl", "FC")]

You should add this line in your *AssemblyInfo.cs* in the class library project, and you will see that theTagPrefix will be registered against a namespace, not a class, so all the classes in this namespace will have the same prefix. Then, each time you drag the control from the toolbar to the web page, it will generates this element:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

<FC:CustomEmailLink ID="CustomEmailLink1" runat="server" />

ViewState round trip management

This topic is very important, and I believe it is the most important topic in custom controls. Although there is a lot of documentation about ViewState, I can tell that many programmers do the same mistakes (like me:)) when they try to save child controls or internal controls in the ViewState, and they end up with the error: “this [object] is not serializable”. First, let us understand what the ViewState is. It is an internal ASP.NET server object that maintains the state of the page and its child controls, so it is very distinct from the Session object that saves objects related to a specific session. The session variables are accessible across all pages, but the ViewState is only for a page, and will be rendered as hidden fields to the client. And, that is why the Sessionobject doesn’t have to be serializable (of course, if it is InProc). ViewState is not a property of HttpContext, because it is a unique object for a single page. Now, let us have a quick look at this code:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

[DefaultProperty("Text")]

[ToolboxData("<{0}:CustomEmailLink runat="server">")]

public class CustomEmailLink : WebControl {

*//.... other code*

*//.... other code*

[Bindable(true)]

[Category("Appearance")]

[Localizable(true)]

public string Email {

get {

String s = (String)ViewState["Email"];

return ((s == null) ? "[Email]" : s);

}

set {

ViewState["Email"] = value;

}

}

This Email property does not save the internal state inside a private field, simply because the custom control will be created each time it is posted back, so the object is created from scratch in the server and restored from the posted page that already has the ViewState data as a hidden field. Let us have a look at this code:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

[DefaultProperty("Text")]

[ToolboxData("<{0}:CustomEditBoxPrimitive runat="\""server\"

Label=\"[Label]\" Text=\"[Text]\">")]

public class CustomEditBoxPrimitive : WebControl {

public string Text {

get {

String s = ViewState["Text"] as string;

return ((s == null) ? "[Text]" : s);

}

set {

ViewState["Text"] = value;

}

}

Now, the CustomEditBoxPrimitive. I named it primitive because it will create a few functionalities that we want from scratch. It uses the internal TextBox control to render a TextBox as part ofCustomEditBoxPrimitive. However, we are not saving the TextBox itself in the ViewState, and we should not even if we can. We are saving only the important properties in the ViewState, like the Text in theTextBox, or the SelectedValue of a drop down list. There is no need at all to save any complex object in the ViewState. Another issue about the serialisation mechanism is that it is an expensive process, and it will affect the performance and scalability. So, be carful when you use the ViewState, save only primitive values like simple data types.

Simple custom controls vs. composite custom controls

It is not a bad idea to begin creating all custom controls as composite (extend the CompositeControl class) controls. The benefit of creating a simple custom control (extend WebControl) is the performance. To build your child controls tree in a StringBuilder variable and render it, it will be much quicker than creating an instance from each child control. In the other hand, composite control support automatic rendering at design time, and container behaviour (for implementing INamingContainer) that create a namespace for the control and unique names for all child controls.

Custom Email

The simple control (extend WebControl) already has a “Controls” property, which you can add the child controls to, and already has a CreateChildControls method that you override to create your own child tree. So, it is not a big difference between both models, as long as you understand the concept behind both.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

public class CustomEditBoxPrimitive : WebControl {

*//...... code*

*//..... code*

[Bindable(true)]

protected override void CreateChildControls() {

BuildControl();

base.CreateChildControls();

}

private void RenderDesign(HtmlTextWriter output) {

output.Write("Text:{0} Label:{1}", this.Text, this.Label);

}

private void InitControls() {

\_innerTextControl.EnableViewState = true;

\_innerTextControl.ID = "Inner\_TextBox";

\_innerLabel.EnableViewState = true;

\_innerLabel.ID = "Inner\_Label";

LoadDataState();

}

As you can see, the control is inherit from WebControl (the most abstract visual control). At the same time, it supports or has the foundation to create child tree controls, like the Controls property and theCreateChildControls() virtual function.

Restore posted data

Restoring posted data is the most interesting part in any custom control that has an input data from user, may be text, or selected index, or others. I was thinking how the Framework knows the changes that happen to any text box rendered to the client. I was thinking that maybe there is a JavaScript generated at the client to capture any changes happened to the client controls. But, I did not see anything like this. I realised that any server button in the page already posts the data to the page form, so all data is already there in theHttpRequest, including the new data. So, I wrote a simple code to extract the posted data of the control, and it works fine for me.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

public class CustomEditBoxPrimitive : WebControl {

*//.... code*

*//.... code*

private void LoadDataState() {

if (this.Context != null) {

foreach (string key in this.Page.Request.Form.AllKeys) {

if (key.ToLower().Contains(this.\_innerTextControl.UniqueID.ToLower())) {

this.Text = this.Page.Request.Form[key];

}

}

}

}

This code is for tutorial purposes only, but sure there is other useful code that we need to add like, for raising an event with any change that has happened. Another way to get posted data is to use the custom control APIs that are supported by the Framework, like implementing the IPostBackDataHandler interface. It has a method LoadPostData that prepares all the data in a key value collection, just ready for use.

PostBack

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

[DefaultProperty("Text")]

[ToolboxData("<{0}:CustomEditBox runat="\""server\" Label=\"[Label]\" Text=\"[Text]\">")]

public class CustomEditBox : CompositeControl,IPostBackDataHandler {

*//... code*

*// .. code*

public bool LoadPostData(string postDataKey,

System.Collections.Specialized.NameValueCollection postCollection) {

if (postDataKey == this.UniqueID) {

foreach (string key in postCollection.Keys) {

if (key.ToLower().Contains(this.\_innerTextControl.UniqueID.ToLower())) {

this.Text = postCollection[key];

}

}

}

return false;

}

When I tried to use it, this method never fired. I thought there was something wrong in the debugging configuration, until I realised there is a relation between the object client name and the object UniqueIDvalue. If ASP.NET finds the name value of any control in the page as the UniqueID of the custom control, it will consider that as an indication for firing posted data to the server control. Anyway, you may add a hidden field that has the name with the same unique ID, just to solve this problem:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

private void BuildControl() {

*//.. code*

*// .. code*

hidControl.Text=String.Format("<input type="hidden" name="{0}" />", this.UniqueID);

this.Controls.Add(hidControl);

ClearChildViewState();

}

With this workaround, your event will be fired and the magic will happen. Sure, we can make the custom control name itself the same as the UniqueID, but I tried it in this tutorial to avoid overriding theRenderContents method, to build only the child controls tree through the CreateChildControlsmethod.

Managing simple events

Managing simple events is very simple and straightforward. You can create your own custom event and raise this event. You may use one of two scenarios: either raise the event corresponding to a custom action, like raising the OnChanged event after comparing the text value after and before posting data:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

[Bindable(true)]

[Category("Appearance")]

[DefaultValue("")]

[Localizable(true)]

public string Text {

get {

String s = ViewState["Text"] as string;

return ((s == null) ? "[Text]" : s);

}

set {

if (ViewState["Text"] != null && (ViewState["Text"].ToString() != value))

if (this.Text != this.\_innerTextControl.Text)

OnTextChanged(this, EventArgs.Empty);

ViewState["Text"] = value;

}

}

Or you may in other scenarios hook to an internal event of any internal control, may be OnClicked of an internal LinkButton, or others.

Extending a custom control

One of the major features of custom controls is the ability to extend them, or to extend an existing rich control. Assume for example, that you have a lookup data in your website and it always is fed via a custom data source, and is initiated with an enumerable type. In this case, you may extend a dropdown list that has a property as an enumerable value, and use it as SourceType. Then, you will use your own business objects that will get the data and populate the list, and this control will be very easy to use. You can create your own library that will be used in your project, or you may have a special date picker or something else. Assume that you want to change the rendering behaviour of a server control, like make the menu render as nested lists, or a GridView as positioned dives (you can do it with CSS adapters). Extending the control can be direct as inheriting from a server control.

http://www.codeproject.com/KB/custom-controls/FirstCustomControl/LinkedEmailExtenderUML.jpg

After rendering:

http://www.codeproject.com/KB/custom-controls/FirstCustomControl/web_email.jpg

Using the Decorator pattern, for more dynamic functionality, which means you inherit from the abstract classWebControl and reference an instance of the Target control from the same WebControl.

http://www.codeproject.com/KB/custom-controls/FirstCustomControl/decorator.gif

For a good documentation of the Decorator pattern, have a look [at this article](http://www.dofactory.com/Patterns/PatternDecorator.aspx).

http://www.codeproject.com/KB/custom-controls/FirstCustomControl/LinkedEmailExtenderDecoratorUML.jpg

Extending the control using the Decorator pattern is out of scope (you will see a sample in the code), but you can create an extender for specific types of controls to add more functionality, or you may create an extender for all visual controls if you inherited and referenced a control of type WebControl. In this case, with one magic class, you can extend all visual controls.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/28783/Your-First-ASP-NET-Custom-Control)

public class LinkedEmailDecoratorExtender:WebControl {

WebControl \_baseControl = null;

public LinkedEmailDecoratorExtender(WebControl baseControl) {

\_baseControl = baseControl;

}

protected override void Render(System.Web.UI.HtmlTextWriter writer) {

writer.Write(" <div {0}="{1}">", "style",

"'border: medium dotted #FF00FF; padding: 5px; margin: 5px;" +

" width: auto; height: auto; font-family: Arial; font-weight: bold;'");

\_baseControl.RenderControl(writer);

writer.Write("</div>");

}

http://www.codeproject.com/KB/custom-controls/FirstCustomControl/web_email_decorator.jpg

Custom controls and Web Parts

Web Parts are amazing features that come in ASP.NET 2.0, and the beauty of ASP.NET 2.0 Web Pats is the compatibility with SharePoint. If you inherit your custom control from a WebPart, you can use it in ASP.NET pages with full support with WebPart zone, and the amazing personalisation behaviour. At the same time, you can reuse it in SharePoint because they inherit from the same base class. All that you need to do is just inherit from WebPart, then you will build up your functionality step by step. It is not a straightforward solution. WebParts are out of the scope of this article. But, I want to highlight the fact that extending WebPart is one of the reasons that lets us consider using custom controls.

Other features

The topic of custom controls is very wide, and I have explained only a very small part. But, we can consider non-visual controls as well, including data sources, AJAX components, and the AJAX Extender toolkit. But, it is not recommended to make a big jump to creating AJAX components, which is very tempting, before learning how to create and use custom controls.

Step by step summary for creating your first custom control

We may summarise building our first custom control (have a look at *CustomEditBox.cs*):

1. Create a new project using the ASP.NET Server Control template.
2. Update the ToolboxData attribute for the appropriate markup that will be created in Design mode.
3. Update TagPrefix in *AssemblyInfo.cs* to control the Prefix registration.
4. Add properties to your class that save and get data from the ViewState object.
5. Create your child controls as class fields.
6. Override OnInit and create instances of child controls, initialise and create your event handlers.
7. Override CreateChildControls and build your child tree.
8. Implement IPostBackDataHandler; if you want to get the posted data, do not forget to make the any rendered control name the same UniqueID.
9. If you want to render the control in design mode as different from runtime, use the DesignModeproperty to switch between the two modes.
10. If you want to extend the control directly to add limited functionality, use inheritance. If you want to add behaviour for a wide range of controls, use the Decorator pattern.

I hope that you enjoyed the article, and the code.

Bottom of Form