# Eventing Within Composite Web Custom Controls

## Introduction

This is the WccCoolButton; it is a multifunction composition based composite control. It has three operation modes: text, image and button. Each mode conceals an underlying LinkButton, Button or ImageButton control respectively. This composite control was created to be used inside future composite controls, and to give the end users of those future controls additional flexibility over the look and feel of the way the future control will operate. This will be the first article in a series of articles in which I will create a double pane picklist, which will use the WccCoolButton control instead of normal buttons. To start with, I had a lot of problems determining the order in which the controls are rendered in a composite control. The books that I was working off of had light examples, and didn't really cover the eventing of a composite control. As there are many examples of simple composite controls, I want to focus on the eventing and control "packing order" that the framework follows when rendering a Composite style control. For some examples of simple web custom controls, see This MSDN Article.

## Composition vs. Rendering

This control example uses Composition style, which means that this control doesn't need to handle the IPostBackEvent interface, or Post Back data. Rendering style, while giving greater flexibility over the control output also forces you to handle all the nuts and bolts of event handling and data post back. Better to let the framework handle that chore, and use Composition when you can. Composition type controls do have extra performance overhead, check out This MSDN Article to learn more.

## Custom Control Packing Order

One of the most painful lessons I had to learn was that Composition based controls like this one render the controls first, and then set the properties later. This caused many problems, as I attempted to set the properties first, and then call CreateChildControls() to render the control. This produced many random rendering errors, and ViewState troubles. So, the best tip I can pass on is to declare all your controls as private members in your Custom Control Class, and then allocate ALL your controls inside CreateChildControls(). After that, put your custom actions inside the property Get/Set sections. Your goal is to actually affect the created controls that are already in position in your Controls collection from CreateChildControls(). All your properties will then work as expected when you used them from the designer and at runtime.

Here is the order that your custom control is created (This is a conceptual order, as I understand it, the actual internal mechanics may be different):

1. The control object is created, the control collection is blank.
2. Your properties are set to the values that you have specified in the designer or are contained in the ViewState by the framework.

3. If your properties are all guarded with EnsureChildControls(), then CreateChildControls() is called by the framework when the framework tries to set your FIRST property. Use CreateChildControls() to set all the defaults for your properties. Make sure that you don't try to set your default properties for your private controls until after you have allocated all your controls!

   Important: Because you have guarded all your properties with EnsureChildControls(), your control is actually created, and set to defaults before your first property gets a chance to get/set anything. You should only set properties to default values inside CreateChildControls(), and leave the customizing inside your properties set{} section.
4. The framework sets the rest of your properties now that your controls are all created.
5. You set any additional properties at runtime or in the designer depending on how you are using the control.
6. The framework renders the control's HTML to the output stream.

## Eventing and Child Controls

When you create your child controls inside CreateChildControls(), you can wire them up to event handlers. In WccCoolButton, I wire three Button types to three different event handlers, and then have each event handler call the same exposed Click event if a delegate has been assigned to it. This way, I can change the WccCoolButton to any of the three button types, and have one exposed click event. To do this, you must have each of your controls that will be wired to events created inside CreateChildControls() and wired to their respective event handlers. If your controls are not present in the Controls collection, then the framework will not fire the events. The best way I have found to handle this is to create all the controls I will need, and then set the ones I will not use right away to .Visible = false. This way, the frame work handles tracking the view state information, and ensures that the events stay hooked up to the controls. Then, I change the visibility of the controls with exposed properties in the set {} section.

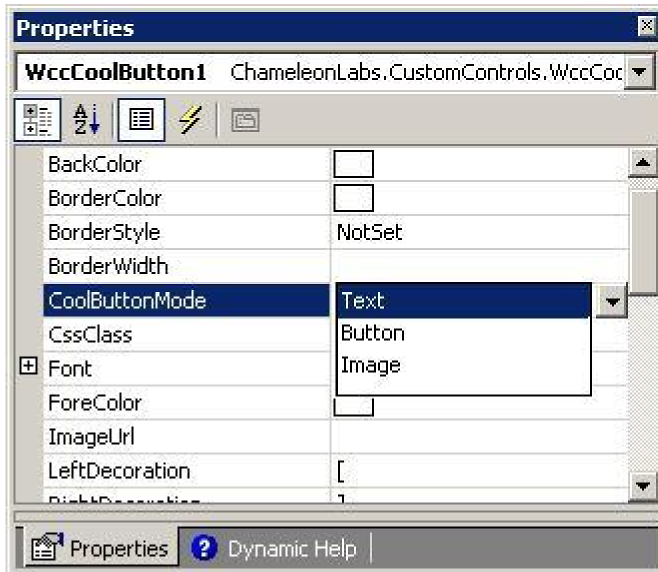
## On With the Code

Starting at the top, one thing I noticed that people have been asking about on usenet was "How do I get a pull down to appear in the designer for my custom property?" I think it's worth a quick note here; just define a public enum for all the values that will appear in the pulldown for your control. Then, make a property that will return that enum type, and the designer takes care of the rest.

```
namespace ChameleonLabs.CustomControls.WccCoolButton
{
    public enum CoolButtonMode
    {
        Text = 0,
        Button = 1,
        Image = 2
    }

    [Bindable(false), Category("Appearance"),
    Description("The mode of the button, use Text for a decorated
                 text button, Button for a normal submit button style,
                 or Image for an Image button style.")]
```

```
    public CoolButtonMode CoolButtonMode
    {
        get
        {
        ...
```




Here is the top of the WccCoolButton class. Notice that I am declaring all the controls, but NOT allocating them. Allocation of the controls must take place in CreateChildControls(). Don't define a constructor and set the defaults in there, do that in CreateChildControls() as well. Also notice that I am exposing one event that can be delegated: Click. If you register an event handler with the designer, or at runtime, then it will be called when any of the three buttons is clicked.

**Collapse**

```
/// <summary>
/// The CoolButton is a multi-function composite control
/// that is to be used inside
/// composite controls to allow more flexibility
/// in the composite control's look and
/// feel. This control can also be used as a stand alone control,
/// when you need to change a button's look on the fly.
/// </summary>
[DefaultProperty("Text"),
ToolboxData("<{0}:WccCoolButton runat=server></{0}:WCCCOOLBUTTON>"),
Designer(typeof(ChameleonLabs.CustomControls.WccCoolButton.
                Design.WccCoolButtonDesigner))]
public class WccCoolButton :
       System.Web.UI.WebControls.WebControl, INamingContainer
{
     // The sole event hook that the button exposes.
     public event EventHandler Click;

     private LinkButton btnLink;
     private ImageButton btnImage;
     private Button btnButton;

     private Label lblLeftDecoration;
     private Label lblRightDecoration;
```

Here is a simple property example that most of the data properties follow. Of note is the way that this property is "guarded" with EnsureChildControls(). EnsureChildControls makes sure that the CreateChildControls() method has been called before proceeding. This also ensures that all those private variables at the top of the object are created in CreateChildControls(). I try to use the built in base controls (like this Label control) as much as possible to capitalize on their existing view state control features.

```
        [Bindable(true), Category("Appearance"), DefaultValue("["),
        Description("The text decoration that is displayed on
          the left side of the button when the CoolButtonMode
          is set to 'Text'.")]
        public string LeftDecoration
        {
            get
            {
                this.EnsureChildControls();
                return lblLeftDecoration.Text;
            }

            set
            {
                this.EnsureChildControls();
                lblLeftDecoration.Text = value;
            }
        }
```

Here is an example of a more complex property where we affect the WccCoolButton control's look by setting the button mode. When the user changes the button mode in the designer or runtime, then the control is updated by hiding the child controls that we don't want shown, and setting the proper child control to visible to cause the framework to draw it. Note that just because a control is not drawn doesn't keep it from being included in the host page's ViewState collection. I am storing the CoolButtonMode in the ViewState in this example, so that it will persist across button clicks. Remember, each button click causes a page postback to occur, so we must keep the state of all the variables that will persist across postback in ViewState. Note that the base controls like Button, Label and the rest manage their own internal view state, so we only need worry about non-web control variables like this eCoolButtonMode state variable.

**Collapse**

```
[Bindable(false), Category("Appearance"),
  Description("The mode of the button, use Text for a decorated text button,
     Button for a normal submit button style,
     or Image for an Image button style.")]
        public CoolButtonMode CoolButtonMode
        {
            get
            {
                CoolButtonMode retVal;

                if(ViewState["eCoolButtonMode"] == null)
                    retVal = CoolButtonMode.Text;
                else
                    retVal = (CoolButtonMode) ViewState["eCoolButtonMode"];

                return(retVal);
```

```
            }

            set
            {
                ViewState["eCoolButtonMode"] = value;

                this.EnsureChildControls();

                HtmlTable t = (HtmlTable) Controls[0];

                switch(CoolButtonMode)
                {
                    case CoolButtonMode.Button:
                        this.btnLink.Visible = false;
                        this.btnButton.Visible = true;
                        this.btnImage.Visible = false;

                        t.Rows[0].Cells[0].Visible = false;
                        t.Rows[0].Cells[2].Visible = false;
                        break;

                    case CoolButtonMode.Image:
                        this.btnLink.Visible = false;
                        this.btnButton.Visible = false;
                        this.btnImage.Visible = true;

                        t.Rows[0].Cells[0].Visible = false;
                        t.Rows[0].Cells[2].Visible = false;
                        break;

                    default: // The control is in CoolButtonMode.Text mode.
                        this.btnLink.Visible = true;
                        this.btnButton.Visible = false;
                        this.btnImage.Visible = false;

                        t.Rows[0].Cells[0].Visible = true;
                        t.Rows[0].Cells[2].Visible = true;
                        break;
                }
            }
        }
```

These next few slides all deal with my overridden CreateChildControls(). First, I am allocating all the controls.

```
        /// <summary>
        /// Add the child controls to the container, sizing
                /// it to the User's specifications.
        /// </summary>
        protected override void CreateChildControls()
        {
            // Setup the controls on the page.
            btnLink = new LinkButton();
            btnImage = new ImageButton();
            btnButton = new Button();
            lblLeftDecoration = new Label();
            lblRightDecoration = new Label();
            ...
```

Next, we wire up the event handlers so that all the controls will be registered with the framework for PostBack event handling.

```
            ...
            // Setup the events on the page.
            btnLink.Click += new EventHandler(this.OnBtnLink_Click);
            btnButton.Click += new EventHandler(this.OnBtnButton_Click);
            btnImage.Click += new
ImageClickEventHandler(this.OnBtnImage_Click);
            ...
```

Now, we create the control's table to layout the child controls, and put the child controls into the table. This is pretty straight forward table building. One thing to note is the use of the styles on the table. This is done to get the table to sit inline with surrounding html elements, otherwise the table will be kicked down a line.

Collapse

```
            ...
            HtmlTable table = new HtmlTable();
            HtmlTableRow newRow;
            HtmlTableCell newCell;

            // Make sure that the composite control flows with
                        // the surrounding text properly.
            table.Border = 0;
            table.Style.Add("DISPLAY", "inline");
            table.Style.Add("VERTICAL-ALIGN", "middle");

            newRow = new HtmlTableRow();

            newCell = new HtmlTableCell();
            newCell.Controls.Add(lblLeftDecoration);
            newRow.Cells.Add(newCell);

            newCell = new HtmlTableCell();
            newCell.Align = "center";

            // Add all the buttons to the control, so that if they are switched
            // programatically, the event handlers will stay linked. If the
controls
            // are not included in the Controls collection, then the event
handling
            // doesn't persist. We will use the visibility to determine which
one is
            // actually rendered for the user to see.
            newCell.Controls.Add(btnLink);
            newCell.Controls.Add(btnButton);
            newCell.Controls.Add(btnImage);

            newRow.Cells.Add(newCell);

            newCell = new HtmlTableCell();
            newCell.Controls.Add(lblRightDecoration);
            newRow.Cells.Add(newCell);

            if(newRow.Cells.Count > 0)
                table.Rows.Add(newRow);

            Controls.Add(table);
            ...
```

Now that the table is allocated, and all the controls are in place, it's time to set the defaults. You must set the defaults for your controls, simply specifying the DefaultValue attribute will not set them. The DefaultValue attribute will only cause

the designer to put your property in bold if you change the property value from the value specified in the DefaultValue attribute. Setting the CoolButtonMode property also sets the visibility on the child controls so that only the child controls that make up the Text mode buttons will be shown.

```
            ...
            // Setup the defaults for the controls.
            this.LeftDecoration = "[";
            this.RightDecoration = "]";
            this.CoolButtonMode = CoolButtonMode.Text;
            ...
```

Once the table is built, and the controls are allocated and wired to events, we need to define the event handlers. The only thing special about these is that they check to see if there is a registered event handler to kick the event up to. If there is, then they call the delegate's click method to send the event up the line. This is how the control will expose its Click event to the designer so that you can wire it into an OnClick event on a hosting web page, or other composite control. All three event handlers call the same Click delegate, which means that no matter which button control is clicked, the same event is sent to the registered delegate. I could have used the same delegate for both the OnBtnLink_Click and OnBtnButton_Click events but for clarity's sake I wanted to use individual delegates for all three controls.

**Collapse**

```
        /// <summary>
        /// This delegate is called when the CoolButtonMode is set to Text.
        /// It's only job is to forward the event to any registered handelers
that
        /// are encapsulating this control, including parent composite
controls, or
        /// the page itself.
        /// </summary>
        /// The sender of the event
        /// An EventArgs object.
        protected virtual void OnBtnLink_Click(object sender, EventArgs e)
        {
            if (Click != null)
            {
                Click(this, e);
            }
        }

        protected virtual void OnBtnButton_Click(object sender, EventArgs e)
        {
            if (Click != null)
            {
                Click(this, e);
            }
        }

        protected virtual void OnBtnImage_Click(object sender,
                               ImageClickEventArgs e)
        {
            if (Click != null)
            {
                Click(this, e);
            }
        }
```

Finally, there is the Designer class. The Designer class is covered in This MSDN Article, so I won't bother here. One thing noteworthy is the setting of the WccCoolButton.Text member to the controls UniqueID. This provides the same "default naming" functionality that you get when you first place the label control on a WebForm inside the designer. The Label control receives a default name like Label1. In this case, if there are no controls in the Controls collection then the Text property is set to the UniqueId. Another important aspect is that setting the Text property has the side effect of calling EnsureChildControls(), and creating the rest of the controls. If this didn't happen, the control would be created AFTER the designer renders it when it is first dropped on the page which would result in the control being created fine, but appearing to be empty. Setting a property with EnsureChildControls() solves this issue. If you are making a Composite control that doesn't have a Text property to set, make a public member that calls EnsureChildControls() which you can call from the GetDesignTimeHtml() override.

**Collapse**

```
/*********************************************************************
 *
 * The control designer.
 *
 *********************************************************************/
namespace ChameleonLabs.CustomControls.WccCoolButton.Design
{
    public class WccCoolButtonDesigner : ControlDesigner
    {
        /// <summary>
        /// Returns a design view of the control as rendered by the control
itself.
        /// </summary>
        /// <returns>The HTML of the design time control.</returns>
        public override string GetDesignTimeHtml()
        {
            WccCoolButton cb = (WccCoolButton) Component;

            // If there are no controls, then it's the first time through the
            // designer, so set the text to the unique id. This will also
            // cause EnsureChildControls() to be called in Text(), which will
            // build out the rest of the control.
            if(cb.Controls.Count == 0)
                cb.Text = cb.UniqueID;

            StringWriter sw = new StringWriter();
            HtmlTextWriter tw = new HtmlTextWriter(sw);

            cb.RenderBeginTag(tw);
            cb.RenderControl(tw);
            cb.RenderEndTag(tw);

            return(sw.ToString());
        }
    }
}
```

## Assign a Custom Icon (and make it stick!)

One thing that I had issues with was getting the icon to be associated with the control in the designer toolbox. Here is a link to The MSDN Article that describes how to assign a custom icon to your control. They left out one crucial item, which is what

to do if you change the namespace of your control. You must also change your project's default namespace to reflect your control's namespace. Note that this means that if you are developing two or more controls under one project, then they must use the same namespace, or their icons will not associate properly. Here is why: when you compile your project, VS.NET helps you out by changing the name of your embedded icon to include the default namespace of your project. If your namespace of your project is not the same as your class, then the framework can't find the icon to tie to your control in the toolbox, and gives up. So, change that default namespace in your project settings, and it will work!

## Conclusion

My goal for this example was to show you how to create a Web Custom Control that could do the following:

- Create a composite control that has three distinct views.
- Enable the control's views to be controlled at both runtime and design time.
- Expose a single event that may be fired from any of the child control's click events.

This was a very basic example, stay tuned for the PickList example where I will incorporate this WccCoolButton control into a multi-paned, data-bindable pick list control complete with sorting, filtering, and more fun design time support!

I would like to say thanks to everyone who asks/answers questions on UseNet, it was an invaluable stomping ground for research into Web Custom Control design. I would also like to thank The Code Project, where I found some of the best examples of Custom Control code (especially those awesome articles by Shawn Wilde).