# Control Extensibility

Web server controls are the essence of ASP.NET, as most aspx pages contain both static HTML and dynamic web server controls. Although ASP.NET includes a comprehensive collection of controls, third party developers have also built up an impressive array of controls, both free and commercial, for almost any task you can imagine. Furthermore, you can create your own controls by implementing both design-time and run-time behavior.

Purchasing controls may not be the best solution if embedded ones will fit your needs with minor modification. A third party control is usually a sophisticated component designed for a specific purpose. For basic functionality in a lightweight package, it’s probably not optimal.

Another option is to extend a standard ASP.NET control and modify the way it renders. ASP.NET’s extensibility concepts are not limited to providers; controls are extensible and customizable if you use adapters. In this chapter, I’ll cover adapter Table 9-4 and explain:

\* How to implement and activate a control adapter

\* How to make the adapter work depending on the current browser

\* How to create adapters at a page level

After reading this chapter, you’ll able to “adapt” an existing control so that it renders according to your needs. This could include different ways of creating HTML, adding JavaScript, or using cascading style sheets to format the output.

# Adaptive Control Behavior

Adaptive control behavior is architecture that changes the way controls render in order to suit the needs of specific clients. Adapters intercept states of the life cycle to alter rendering behavior. By default, each control has a designated adapter in a one-to-one relationship. It’s possible to change this relationship under certain circumstances—when, for example, you encounter a browser that requires a different kind of markup.

To control the adaptive behavior, choose from the following techniques:

\* Configure the application to support different markups

\* Choose the XhtmlTextWriter or ChtmlTextWriter class to create specific markup

\* Create a TextWriter class that writes the markup to the output stream

\* Use a filter to recognize a client device and decide what kind of markup you need

\* Create an adapter and assign the adapter to a control

The markup is not the only aspect you may want to change. Depending on the capabilities of the target device you may need to think about:

\* How to work with postback data

\* Managing the view state

\* Preventing changes to the control

## The Default Behavior

The render process of a page uses the HtmlTextWriter class by default. The RenderControl method is called recursively with an instance of the HtmlTextWriter. Each control adds its markup to the writer. By the end of the process, the writer holds the complete markup of the page and writes it to the output stream.

ASP.NET includes several writer types compatible with specific output devices. For HTML 3.2, the Html32TextWriter is used. Which writer to use depends on the TagWriter of the System.Web.HttpRequest.Browser. If a browser supports HTML 4.0, the page framework should compose XHTML. Using *web.config*, you can configure this behavior and replace the Html32TextWriter with an XHtmlTextWriter. The Browser property returns an HttpBrowserCapabilities object.

The underlying code is relatively straightforward. If a custom TextWriter is defined, the CreateHtmlTextWriterFromType is used. If not, the default Html32TextWriter is used. The following code snippets are decompiled from the System.Web.HttpRequest class and the System.Web.UI.Page class, respectively.

internal HtmlTextWriter CreateHtmlTextWriterInternal(TextWriter tw)

{

Type tagWriter = this.TagWriter;

if (tagWriter != null)

{

return Page.CreateHtmlTextWriterFromType(tw, tagWriter);

}

return new Html32TextWriter(tw);

}

public Type TagWriter

{

get

{

try

{

if (!this.\_havetagwriter)

{

string str = this["tagwriter"];

if (string.IsNullOrEmpty(str))

{

this.\_tagwriter = null;

}

else if (string.Compare(str, typeof(HtmlTextWriter).FullName, ⮰

StringComparison.Ordinal) == 0)

{

this.\_tagwriter = typeof(HtmlTextWriter);

}

else

{

this.\_tagwriter = BuildManager.GetType(str, true);

}

this.\_havetagwriter = true;

}

}

catch (Exception exception)

{

throw this.BuildParseError(exception, "tagwriter");

}

return this.\_tagwriter;

}

}

public static HtmlTextWriter CreateHtmlTextWriterFromType(TextWriter tw, ⮰

Type writerType)

{

HtmlTextWriter writer;

if (writerType == typeof(HtmlTextWriter))

{

return new HtmlTextWriter(tw);

}

if (writerType == typeof(Html32TextWriter))

{

return new Html32TextWriter(tw);

}

try

{

Util.CheckAssignableType(typeof(HtmlTextWriter), writerType);

writer = (HtmlTextWriter) HttpRuntime.CreateNonPublicInstance(⮰

writerType, new object[] { tw });

}

catch

{

throw new HttpException(SR.GetString("Invalid\_HtmlTextWriter", ⮰

new object[] { writerType.FullName }));

}

return writer;

}

As you can see, the instance of the abstract TextWriter class is responsible for the rendering process. Because ASP.NET produces HTML, the HtmlTextWriter class is the optimum point for beginning to implement a custom writer class. The XhtmlTextWriter and ChtmlTextWriter types mentioned earlier derive from the HtmlTextWriter and Html32TextWriter respectively. They are concrete implementations for specific rendering behavior.

Insert 19835f0903.tif

Figure 9-1. The Control Adapter Architecture

### Using Control Adapters

To modify the behavior described earlier, you’ll need an implementation of the System.Web.UI.Adapters.ControlAdapter base class. In each phase of the lifecycle, the control checks whether or not an adapter is available. If there is one present, the adapter provides an alternative method that replaces the default method that would otherwise be called in that step in the lifecycle. If the adapter only modifies portions of the behavior, it can call the control’s default method instead. Adapters that modify the state persistence behavior differ in that they don’t modify, but completely replace, the default behavior. When the adapter intercepts the life cycle, the following actions (at a minimum) must occur:

\* Override the OnInit method of the control adapter to modify the initializing phase

\* Override the Render or RenderChildren method to add custom markup

The abstract base class has the following structure:

public abstract class ControlAdapter

{

protected ControlAdapter();

protected HttpBrowserCapabilities Browser { get; }

protected Control Control { get; }

protected Page Page { get; }

protected PageAdapter PageAdapter { get; }

protected internal virtual void BeginRender(HtmlTextWriter writer);

protected internal virtual void EndRender(HtmlTextWriter writer);

protected internal virtual void LoadAdapterControlState(object state);

protected internal virtual void LoadAdapterViewState(object state);

protected internal virtual void OnInit(EventArgs e);

protected internal virtual void OnLoad(EventArgs e);

protected internal virtual void OnPreRender(EventArgs e);

protected internal virtual void OnUnload(EventArgs e);

protected internal virtual void Render(HtmlTextWriter writer);

protected virtual void RenderChildren(HtmlTextWriter writer);

protected internal virtual object SaveAdapterControlState();

protected internal virtual object SaveAdapterViewState();

}

Before you start implementing adapters to change a control’s behavior, you’ll need to learn the purpose for each property and method.

Table 9-1. The ControlAdapter base class

|  |  |
| --- | --- |
| Class Member | Description |
| Browser | The browser capabilities (HttpBrowserCapabilities) of the client making the current HTTP request. |
| Control | The control to which this control adapter is attached. |
| Page | The page containing the control associated with this adapter. |
| PageAdapter | The page adapter (System.Web.UI.Adapters.PageAdapter) for the page defined by Page, above. |
| BeginRender | Called prior to the rendering of a control. The method generates opening tags required by a specific target. It takes a System.Web.UI.HtmlTextWriter object to render the target-specific output. |
| EndRender | Called after the rendering of a control. The method generates closing tags that are required by a specific target. It takes a System.Web.UI.HtmlTextWriter object to render the target-specific output. |
| CreateChildControls | Creates the target-specific child controls for a composite control. |
| LoadAdapterControlState | Loads adapter control state information saved by the System.Web.UI.Adapters.ControlAdapter.SaveAdapterControlState method during a previous request to the Page. Receives the state as a StateBag object. |
| LoadAdapterViewState | Loads adapter view state information saved by the System.Web.UI.Adapters.ControlAdapter.SaveAdapterViewState method during a previous request to the Page. Receives the state as a StateBag object. |
| OnInit | Overrides the System.Web.UI.Control.OnInit method for the associated control in order to hook into the initializing phase. |
| OnLoad phase. | Overrides the System.Web.UI.Control.OnLoad method for the associated control in order to hook into the load |
| OnPreRender | Overrides the System.Web.UI.Control.OnPreRender method for the associated control. |
| OnUnload | Overrides the System.Web.UI.Control.OnUnload method for the associated control. |
| Render | Generates target-specific markup for the control to which the adapter is attached. Takes a System.Web.UI.HtmlTextWriter object to render the target-specific output. |
| RenderChildren | Generates the target-specific markup for the child controls in a composite control to which the adapter is attached. Takes a System.Web.UI.HtmlTextWriter object to render the target-specific output. |
| SaveAdapterControlState | Saves control state information for the control adapter into a StateBag object. |
| SaveAdapterViewState | Saves view state information for the control adapter into a StateBag object. |

You now have all the information you need in order to implement a custom control adapter. However, before the adapter can be used, it must be configured. Configuration depends on a device filter.

The System.Web.UI.WebControls.Adapters.WebControlAdapter base class is the preferred way to change the behavior of built-in web controls. This class adds virtual methods RenderBeginTag, RenderEndTag and RenderContents, to more closely mirror the render behavior of WebControls.

public class WebControlAdapter : ControlAdapter

{

protected WebControl Control { get; }

protected bool IsEnabled { get; }

protected virtual void RenderBeginTag(HtmlTextWriter writer);

protected virtual void RenderContents(HtmlTextWriter writer);

protected virtual void RenderEndTag(HtmlTextWriter writer);

}

The following table explains its properties and methods.

Table 9-2. WebControlAdapter

|  |  |
| --- | --- |
| Member | Description |
| Control | The Web control to which this adapter is attached. |
| IsEnabled | Indicates whether the Web control and all its parent controls are enabled. |
| Render | Generates the target-specific markup for the control to which the adapter is attached. |
| RenderBeginTag | Creates the beginning tag in the markup for the Web control. |
| RenderContents | Generates the target-specific inner markup for the Web control to which the adapter is attached. |
| RenderEndTag | Creates the closing tag in the markup for the Web control. |

Note that the WebControlAdapter is not an abstract class but the basic class in the hierarchy of adapters responsible for regular WebControls. It’s in the System.Web.UI.WebControls.Adapters namespace. The WebControlAdapter class is the base class for the following implementations:

\* HierarchicalDataBoundControlAdapter

\* DataBoundControlAdapter

\* HideDisabledControlAdapter

\* MenuAdapter

The MenuAdapter is a concrete implementation used to render Menu controls. The HierarchicalDataBoundControlAdapter provides a virtual method, PerformDataBinding, which calls Control.PerformDataBinding. Overriding this method in a derived class allows you to change the binding behavior. (This is also the case for the DataBoundControlAdapter class.) The HideDisabledControlAdapter can be applied to any control. Using this adapter removes the control from pages in which the control is disabled. It overrides the adapter’s Render method and, if its Enabled property is false, the control’s Render method is not called.

Insert 19835f0902.tif

Figure 9-2. The class diagram for control adapter classes

The base classes can be used to perform basic tasks without you having to implement your own adapter. With them you can also vary a single behavior, such as data binding, while benefiting from all the other default adapter functionality.

In the class diagram, you’ll also find a PageAdapter class that derives from ControlAdapter. Thus, pages can have adapters, too.

### Using Page Adapters

As a Page is essentially a specialized Control, the control adapter also supports pages. The PageAdapter base class exists to make writing a custom page adapter easy. This base class inherits from the ControlAdapter described earlier and extends the adapter with page-specific features.

public abstract class PageAdapter : ControlAdapter

{

public virtual StringCollection CacheVaryByHeaders { get; }

public virtual StringCollection CacheVaryByParams { get; }

protected string ClientState { get; }

public virtual NameValueCollection DeterminePostBackMode();

protected internal virtual string GetPostBackFormReference(string formId);

public virtual ICollection GetRadioButtonsByGroup(string groupName);

public virtual PageStatePersister GetStatePersister();

public virtual void RegisterRadioButton(RadioButton radioButton);

public virtual void RenderBeginHyperlink(HtmlTextWriter writer, ⮰

string targetUrl, ⮰

bool encodeUrl, ⮰

string softkeyLabel);

public virtual void RenderBeginHyperlink(HtmlTextWriter writer, ⮰

string targetUrl, ⮰

bool encodeUrl, ⮰

string softkeyLabel, ⮰

string accessKey);

public virtual void RenderEndHyperlink(HtmlTextWriter writer);

public virtual void RenderPostBackEvent(HtmlTextWriter writer, ⮰

string target, ⮰

string argument, ⮰

string softkeyLabel, ⮰

string text);

public virtual void RenderPostBackEvent(HtmlTextWriter writer, ⮰

string target, ⮰

string argument, ⮰

string softkeyLabel, ⮰

string text, ⮰

string postUrl, ⮰

string accessKey);

protected void RenderPostBackEvent(HtmlTextWriter writer, ⮰

string target, ⮰

string argument, ⮰

string softkeyLabel, ⮰

string text, ⮰

string postUrl, ⮰

string accessKey, ⮰

bool encode);

public virtual string TransformText(string text);

}

The following table explains the properties and methods provided by the PageAdapter. (The PageAdapter attaches to a web page. This is the “page” referred to in the table descriptions.)

Table 9-3. Properties and methods in the PageAdapter, additional to those provided by the ControlAdapter

|  |  |
| --- | --- |
| Member | Description |
| CacheVaryByHeaders | A list (of type IList) of additional HTTP headers by which caching is varied for the Web page. |
| CacheVaryByParams | A list (of type IList) of additional parameters from HTTP GET and POST requests by which caching is varied for the Web page. |
| ClientState | An encoded string containing the view and control state data of the Web page. |
| DeterminePostBackMode | Evaluates whether the Web page is in postback mode and returns a name/value collection of type System.Collections.Specialized.NameValueCollection of the postback variables. |
| GetPostBackFormReference | Returns a DHTML code fragment that the client browser can use to reference the form on the page that was posted. |
| GetRadioButtonsByGroup | Retrieves a collection of radio button controls specified by groupName. Takes the name of a System.Web.UI.WebControls.RadioButton group and returns an ICollection object with RadioButton instances. |
| GetStatePersister | Returns a PageStatePersister object used by the Web page to maintain the control and view states. |
| RegisterRadioButton | Adds a radio button control to the collection for a specified radio button group. |
| RenderBeginHyperlink | Renders an opening hyperlink tag, including the target URL, to the response stream. It takes the current TextWriter instance to write the data. This method is overloaded in order to support different sets of parameters. |
| RenderEndHyperlink | Renders a closing hyperlink tag to the response stream. |
| RenderPostBackEvent | Renders a PostBack event to the response stream as a hyperlink, including the encoded and possibly encrypted view state, and the event target and argument. This method is overloaded to support different sets of parameters. |
| TransformText | Transforms text for the target browser. |

The PageAdapter supports the persistence layer and the render behavior of controls that group multiple tags. RadioButtons need special treatment when handling groups. You can define a group of radio buttons in HTML by giving them the same name. However, ASP.NET requires different IDs for each radio button, and by default the name and ID are the same. The PageAdapter will thus render each radio button group appropriately to keep the groups separate.

The GetStatePersister property can be overloaded in order to change the view state persister globally. Using the abstract base class, you can write a custom page adapter to change the behaviour of all pages.

But why does the rendering of a page depend on the client device? Remember that the view state has a strong influence on the client. The view state could grow and consume a significant portion of the page’s rendered content. (Refer to chapter 1 for a refresher about view state.) The user experience on mobile devices and smart phones generally deteriorates with large pages. To mitigate this, you could write a page adapter that creates a regular view state for PC based browsers and server-side view state for mobile devices.

That implies that, as for ControlAdapter implementations, the PageAdapter must be configured and associated with a set of clients.

### Device-Specific Filter for Adaptive Behavior

A device filter recognizes a client device and assigns a device-specific adapter to its controls. The filter can also identify control properties, user defined attributes and templates, and can be controlled with @Page and @Control directives. Recall that each control has a one-to-one relationship with an adapter, so if the filter returns several adapters, the most specific one is used.

Device filters are based on browser definition files. You can find the default definition files in the folder:

%SystemRoot%\Microsoft.NET\Framework\v2.0.50727\CONFIG\Browsers

Browser files are used to create the HttpBrowserCapabilities object that exposes the device’s properties. The definitions form a hierarchy, preventing the files from redefining basic definitions. For instance, the *ie.browser* file contains several IE-related definitions. The following snippet shows settings specific to Internet Explorer:

<browsers>

<browser id="IE" parentID="Mozilla">

<identification>

<userAgent match="^Mozilla[^(]\*\([C|c]ompatible;\s\*MSIE (?'version'(?'major'\d+)(?'minor'\.\d+)(?'letters'\w\*))(?'extra'[^)]\*)" />

<userAgent nonMatch="Opera|Go\.Web|Windows CE|EudoraWeb" />

</identification>

<capture>

</capture>

<capabilities>

<capability name="browser" value="IE" />

<capability name="extra" value="${extra}" />

<capability name="isColor" value="true" />

<capability name="letters" value="${letters}" />

<capability name="majorversion" value="${major}" />

<capability name="minorversion" value="${minor}" />

<capability name="screenBitDepth" value="8" />

<capability name="type" value="IE${major}" />

<capability name="version" value="${version}" />

</capabilities>

</browser>

<browser id="IE5to9" parentID="IE">

<identification>

<capability name="majorversion" match="^[5-9]" />

</identification>

<capture>

</capture>

<capabilities>

<capability name="activexcontrols" value="true" />

<capability name="backgroundsounds" value="true" />

<capability name="cookies" value="true" />

<capability name="css1" value="true" />

<capability name="css2" value="true" />

<capability name="ecmascriptversion" value="1.2" />

<capability name="frames" value="true" />

<capability name="javaapplets" value="true" />

<capability name="javascript" value="true" />

<capability name="jscriptversion" value="5.0" />

<capability name="msdomversion" ⮰

value="${majorversion}${minorversion}" />

<capability name="supportsCallback" value="true" />

<capability name="supportsFileUpload" value="true" />

<capability name="supportsMultilineTextBoxDisplay" value="true" />

<capability name="supportsMaintainScrollPositionOnPostback" ⮰

value="true" />

<capability name="supportsVCard" value="true" />

<capability name="supportsXmlHttp" value="true" />

<capability name="tables" value="true" />

<capability name="tagwriter" ⮰

value="System.Web.UI.HtmlTextWriter" />

<capability name="vbscript" value="true" />

<capability name="w3cdomversion" value="1.0" />

<capability name="xml" value="true" />

</capabilities>

</browser>

…

</browsers>

The filter consists of two parts. Firstly, the userAgent tag provides a match attribute which contains a regular expression. This expression is matched against the user-agent HTTP header sent by each client device.

The capabilities section contains a setting for the preferred tagwriter, which you can use to assign a particular behavior when creating tags. This is much less common than using a full blown adapter, but is expedient if you simply want to write XHTML instead of HTML.

Insert 19835f0900.tif

Figure 9-3. User Agent Settings for Internet Explorer

Caution: As the User Agent string can be fake, don’t trust the settings for security related activities.

Adding client device filters can be accomplished on a per application basis using the *App\_Browsers* folder. This is a special folder which can be added to any Web application project. Place a custom browser file into this folder and define device specific settings there.

Insert 19835f0901.tif

Figure 9-4. Adding a \*.browser file to the current project’s App\_Browsers folder

The browser file has a similar structure to the default ones. The inheritance behavior is crucial. By using the right parentID attribute, you can inherit all common settings from an existing device and add or change a few values to match your current device.

A typical file contains the following tags, at a minimum:

<browsers>

<browser id="NewBrowser" parentID="Mozilla">

<identification>

<userAgent match="Unique User Agent Regular Expression" />

</identification>

<capture>

<userAgent match="NewBrowser (?'version'\d+\.\d+)" />

</capture>

<capabilities>

<capability name="browser" value="My New Browser" />

<capability name="version" value="${version}" />

</capabilities>

<controlAdapters markupTextWriterType="">

<adapter adapterType=""

controlType="" />

</controlAdapters>

</browser>

<browser refID="Mozilla">

<capabilities>

<capability name="xml" value="true" />

</capabilities>

</browser>

</browsers>

The types defined in the controlAdapters tag build the relationship between controls and adapters. They are specific to the current device. However, by using a common matching pattern, you can assign the filter to any device.

While the adapterType is a type derived from the System.Web.UI.Adapters.ControlAdapter, the controlType must derive from System.Web.UI.Control. The markupTextWriterType attribute defines the writer to render the content. This type is instantiated and the object passed to the Render and RenderChildren methods of the control adapter. For an embedded control, this looks like:

<controlAdapters markupTextWriterType="System.Web.UI.XhtmlTextWriter" >

<adapter controlType="System.Web.UI.WebControls.Menu"

adapterType="System.Web.UI.WebControls.Adapters.MenuAdapter">

</adapter>

</controlAdapters>

Note: The browser capabilities file does not function for certain browser settings. The file matches the user agent string against a predefined set of properties. For instance, Internet Explorer supports JavaScript, hence the HttpBrowserCapabilities object returns true for that setting. If, however, a user disables JavaScript, this action does not change the property. You should read the object’s name literally—as a **capability** rather than as actual behavior.

## Using Control Adapters

There are several usage scenarios for adapters. In fact, adapters are one of the most powerful tools for extending control behavior.

### Device Friendly Adapters

The main purpose of control adapters is to support specific client devices that do not function well with the standard HTML created by built-in controls. The intended behavior is defined within the .browser configuration file.

### CSS Friendly Adapters

When you read about adapters, you’ll often see the term “CSS friendly control adapters”. This wrongly implies that the adapter supports CSS (cascading style sheets). An adapter is simply an extensibility technique that allows page developers to alter the render behavior for all or particular client devices. Replacing pure HTML with CSS support is a common scenario, but it is not the only one supported by adapters. The reason for the name is the lack of CSS support in ASP.NET, and the development of adapters to overcome this limitation. For a sophisticated example of the usage of adapters to support CSS, refer to the website *www.asp.net/CSSAdapters*. This site offers a complete set of adapters using specific style definitions to format controls. The MenuAdapter shows how the replacement of <table> tags with nested <ul> tags and CSS styles, saves a lot of space and results in a freely configurable menu control. Other examples in the package support complex controls such as FormView and TreeView.

### Other Adapter Ideas

An adapter can output any kind of markup. It’s not limited to HTML or XHTML, and you can even create XAML to support WPF applications or Silverlight. (As this does sound strange, I’ll reiterate it so that you know it’s not a mistake.) It is possible to write a regular ASP.NET page consisting entirely of ASP.NET controls and render them in XAML—or anything else. The TextWriter class, used to add content to the output stream, writes text, whether or not it consists of markup. You can also use adapters to expose the source of a page by replacing the tags with &lt; and &gt; entities. I’ll demonstrate this in one of the following examples.

# Writing a Custom Control Adapter

The following example shows how a custom control derived from a built-in control can be modified with an adapter.

## Steps for Creating the Example

The example consists of the following parts:

\* A custom control that adds two properties to the built-in CheckBoxList

\* An adapter that changes the render behavior using these properties

\* A page that uses the custom control

\* A .browser file that assigns the adapter to the control

### The Example Code

The next code listing shows the custom control. The two properties store the names of the image resources. The purpose of the control is to display an image list. The control takes ListItem members, and instead of rendering CheckBox elements, two images are used—an OnImage if the item is selected (checked) and an OffImage otherwise.

The control behaves as any standard CheckBoxList would. You can add ListItems and fill in the appropriate data. However, the adapter changes the render behavior to just display images. There is no treatment of content because the list is read-only, and the user can’t change anything or post back the data.

Imagine that the adapter is used to render the control for devices that don’t allow interaction, such as a kiosk application. In that situation, a disabled CheckBox would look unprofessional.

Listing 9-1. A custom control which modifies a built-in one

public class MyCheckBoxList : CheckBoxList

{

[Browsable(true)]

public string OffImage

{

get

{

if (ViewState["OffImage"] == null)

{

OffImage = "";

}

return (string)ViewState["OffImage"];

}

set

{

ViewState["OffImage"] = value;

}

}

[Browsable(true)]

public string OnImage

{

get

{

if (ViewState["OnImage"] == null)

{

OnImage = "";

}

return (string)ViewState["OnImage"];

}

set

{

ViewState["OnImage"] = value;

}

}

}

This control has two additional properties. The images the properties refer to can be defined as embedded resources. To achieve this, use the WebResourceAttribute as follows:

[assembly: WebResourceAttribute( ⮰

"Apress.Extensibility.Adapters.Resources.OnImage.png", "image/jpg")]

[assembly: WebResourceAttribute( ⮰

"Apress.Extensibility.Adapters.Resources.OffImage.png", "image/jpg")]

Furthermore, we’ll need a simple test scenario in order to get the control working. The code shown below is a snippet from an .aspx page. The attribute goes anywhere but inside a class, specifically the *AssemblyInfo.cs* file is a great place.

Listing 9-2. A simple usage scenario

<form id="form1" runat="server">

<div>

<cc1:MyCheckBoxList ID="MyCheckBoxList1" runat="server" ControlOrientation="Horizontal" BackColor="Red" ForeColor="Blue">

</cc1:MyCheckBoxList>

</div>

</form>

The code behind file contains only the data source. To see the control in action, add a few ListItems, and choose images via the Selected property. If Selected equals true, OnImage is shown—otherwise, it’s OffImage.

Listing 9-3. The code behind file adds a few items for testing purposes

public partial class \_Default : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

var lic = new List<ListItem>();

lic.Add(new ListItem("Value 1", "1") { Selected = true });

lic.Add(new ListItem("Value 2", "2") { Selected = false });

lic.Add(new ListItem("V-alue 3", "3") { Selected = true });

lic.Add(new ListItem("Value 4", "4") { Selected = false });

MyCheckBoxList1.Items.AddRange(lic.ToArray());

}

}

}

Last, but not least, the adapter is required. This renders the control through a reference to the control, using the appropriate Render methods.

Listing 9-4. The adapter creates images and labels instead of checkbox controls

public class MyCheckBoxListAdapter : ⮰

System.Web.UI.WebControls.Adapters.WebControlAdapter

{

private MyCheckBoxList CheckBoxListControl

{

get

{

return ((MyCheckBoxList)Control);

}

}

protected override void RenderBeginTag(System.Web.UI.HtmlTextWriter writer)

{

writer.WriteLine();

writer.WriteBeginTag("table");

writer.Write(HtmlTextWriter.TagRightChar);

writer.Indent++;

}

protected override void RenderEndTag(System.Web.UI.HtmlTextWriter writer)

{

writer.WriteEndTag("table");

writer.WriteLine();

writer.Indent--;

}

protected override void RenderContents(System.Web.UI.HtmlTextWriter writer)

{

switch (CheckBoxListControl.RepeatDirection)

{

case RepeatDirection.Horizontal:

writer.WriteBeginTag("tr");

writer.Indent++;

writer.Write(HtmlTextWriter.TagRightChar);

for (int i = 0; i < CheckBoxListControl.Items.Count; i++)

{

writer.WriteBeginTag("td");

writer.Write(HtmlTextWriter.TagRightChar);

RenderCheckbox(writer, i);

writer.WriteEndTag("td");

}

writer.WriteEndTag("tr");

writer.Indent--;

break;

case RepeatDirection.Vertical:

for (int i = 0; i < CheckBoxListControl.Items.Count; i++)

{

writer.WriteBeginTag("tr");

writer.Write(HtmlTextWriter.TagRightChar);

writer.WriteBeginTag("td");

writer.Write(HtmlTextWriter.TagRightChar);

RenderCheckbox(writer, i);

writer.WriteEndTag("td");

writer.WriteEndTag("tr");

}

break;

}

}

private void RenderCheckbox(HtmlTextWriter writer, int i)

{

Image img = new Image();

Label l = new Label();

if (CheckBoxListControl.Items[i].Selected)

{

img.ImageUrl = Page.ClientScript.GetWebResourceUrl(this.GetType(),

String.Format("Apress.Extensibility.Adapters.Resources.{0}.png",

CheckBoxListControl.OnImage));

l.Text = String.Format("{0} (on) ",

CheckBoxListControl.Items[i].Text);

}

else

{

img.ImageUrl = Page.ClientScript.GetWebResourceUrl(this.GetType(),

String.Format("Apress.Extensibility.Adapters.Resources.{0}.png",

CheckBoxListControl.OffImage));

l.Text = String.Format("{0} (off) ",

CheckBoxListControl.Items[i].Text);

}

img.ToolTip = CheckBoxListControl.Items[i].Value;

img.RenderControl(writer);

l.RenderControl(writer);

}

}

A CheckBoxList supports a RenderDirection property, which has several more features not fully supported by this example. To obtain the right render direction, a table is used that either grows to the right by adding <td> tags, or downwards using <tr><td> pairs. The private RenderCheckbox method is used to create the content. To simplify this, built-in controls are employed and their internal Render method is used through the public RenderControl method. This ensures that the other adapters responsible for Image and Label controls will work as expected.

In the example, an embedded resource is used, and the properties the custom control exposes are accessed in order to obtain the correct resource. The assembly attributes (WebResourceAttribute) described previously define the embedded resources, which are retrieved by means of the GetWebResourceUrl method.

When building HTML, it’s often necessary to write parts of a tag, empty lines, or other characters. The HtmlTextWriter class has several public constant values to support such characters. The following table explains these:

Table 9-4. Constant characters and strings, as defined in HtmlTextWriter

|  |  |  |
| --- | --- | --- |
| Constant | Character | Description |
| DefaultTabString |  | Represents a single tab character. |
| DoubleQuoteChar | " | Represents the quotation mark (") character. |
| EndTagLeftChars | </ | Represents the left angle bracket and slash mark of the closing tag |
| EqualsChar | = | Represents the equals sign. |
| EqualsDoubleQuoteString | ="" | Represents an equals sign with a double quotation mark |
| SelfClosingChars | / | Represents a space and the self-closing slash mark of a markup tag |
| SelfClosingTagEnd | /> | Represents the closing slash mark and right angle bracket of a self-closing tag. |
| SemicolonChar | ; | Represents a semicolon. |
| SingleQuoteChar | ' | Represents an apostrophe. |
| SlashChar | / | Represents a forward slash. |
| SpaceChar |  | Represents a space character. |
| StyleEqualsChar | : | Represents the style equals character used to set style attributes equal to values. |
| TagLeftChar | < | Represents the opening angle bracket of a markup tag. |
| TagRightChar | > | Represents the closing angle bracket of a markup tag. |

The code should now run as expected. Figure 9-5 illustrates the output of the preceding example.

Figure 9-5. A CheckBoxList that displays images

Although the result may seem unimpressive, my intention was to demonstrate how to modify the render behavior of an existing control. The benefit here is that the other parts of the control that do not affect rendering still function as expected. Developers can choose between using this or another render behavior, depending on their application’s needs or device capabilities. Imagine that a very basic client can’t render checkboxes, but it can render images. Creating a new CheckBoxList control would require a lot more work than simply editing the adapter.

### Why Use HtmlTextWriter?

You may be wondering why the HtmlTextWriter plays such an important role. After all, when building HTML, the StringBuilder is frequently the better and faster alternative. There are several reasons. Firstly, the HtmlTextWriter has a number of useful predefined characters and strings, as shown in Table 9-4. Secondly, it handles indentation well when formatting HTML line by line. This would require additional code with StringBuilder. Thirdly, it is associated with the output stream, which means that the content is written directly to the output (Response.OutputStream). This is the fastest way to transmit the response to the Web server.

However, there are some disadvantages to the HtmlTextWriter. If you create your own HtmlTextWriter, you might associate it with a StringWriter/StringBuilder pair in order to retrieve the content. But using the writer exposed by the adapter classes, you can’t access the stream with anything but write access, and any attempt to read the content will fail.

### Configure the Example

To configure the code, you only need to make an entry in a .browser file. For this example, I define a file called *MyClient.browser* and place it in the default folder for browser definition files, *App\_Browser*.

<browsers>

<browser refID="Default">

<controlAdapters>

<adapter adapterType="Apress.Extensibility.Adapters.MyCheckBoxListAdapter"

controlType="Apress.Extensibility.Adapters.MyCheckBoxList" />

</controlAdapters>

</browser>

</browsers>

This is everything you need. Launch the application and the adapter will start working.

## Writing a Custom Page Adapter

A PageAdapter is like a ControlAdapter in that it’s the preferred way to change the behavior of a page without altering either markup or code. As explained previously, pages are responsible not only for the rendering process but also for saving view state and for several life cycle events. A typical usage scenario involves the view state behavior under specific circumstances. For example, if a client device has a low-bandwidth connection, sending the view state to the client and back during postback will result in disappointed users.

However, to demonstrate creating a custom page adapter I will tackle a simplified scenario.

### Steps for Creating the Example

The PageAdapter functions exactly like the ControlAdapter. You’ll need the adapter class and a .browser file in order to configure it. The next listing shows the code which writes the page’s source at the end of the page:

Listing 9-5. Simple PageAdapter to append page source

public class SourcePageAdapter : PageAdapter

{

protected override void EndRender(System.Web.UI.HtmlTextWriter writer)

{

StreamReader sr = File.OpenText( ⮰

this.Page.Server.MapPath(this.Page.Request.Url.LocalPath));

writer.WriteFullBeginTag("pre");

this.Page.Server.HtmlEncode(sr.ReadToEnd(), writer);

sr.Close();

writer.WriteEndTag("pre");

base.EndRender(writer);

}

}

The writer is used to write the page’s content from a StreamReader object. This contains page code read directly from the disk. The example assumes that the page is available.

Note: The example writes the content after all regular content, including the closing <html> tag. This results in invalid HTML. However, all common browsers render this, despite the discrepancy.

Figure 9-6.The page used for the ControlAdapter example and content exposed by the page adapter

The configuration activates this for all clients. Imagine a workstation that you have configured for testing purposes. On it you have set the User-agent string manually in order to send a private key to the server. The .browser file recognizes this private key and activates the adapter only for requests from the testing computer. It will be the only client to receive the content rendered with source. Everyone else will see only the conventional page output. You won’t need to set anything in *web.config*, or in code, or anywhere else.

Caution: Several examples recommend calling the RenderChildren method in order to send the rendered content of a page to a private TextWriter. This would not work with an adapter, as the adapter would be summoned again for the subsequent call, which would lead to an endless loop and eventually to a StackOverflowException.

The next example shows another way to play with HtmlTextWriter. Instead of exposing the server-side source I want expose the HTML send to client below the page.

Listing 9-7. PageAdapter to append rendered Page Source

public class ClientSourcePageAdapter : PageAdapter

{

HtmlTextWriter internalWriter;

HtmlTextWriter newWriter;

StringBuilder sb;

protected override void BeginRender(HtmlTextWriter writer)

{

internalWriter = writer;

sb = new StringBuilder();

newWriter = new HtmlTextWriter(new StringWriter(sb));

base.BeginRender(newWriter);

}

protected override void Render(HtmlTextWriter writer)

{

base.Render(newWriter);

string html = sb.ToString();

writer.Write(html);

writer.WriteBeginTag("pre");

writer.Write(HtmlTextWriter.SpaceChar);

writer.WriteAttribute("style", "border:solid 1px blue");

writer.Write(HtmlTextWriter.TagRightChar);

writer.WriteEncodedText(html);

writer.WriteEndTag("pre");

}

}

The original writer is now replaced by a custom one in the BeginRender method. In Render it allows you to access your own writer instance and write the content to a string using ToString of the StringBuilder object. Firstly, the string is written back to the original writer. The remaining part creates a <pre> tag and a border style to place the encoded content in.

Figure 9-6.The page used for the ControlAdapter example and rendered content exposed by the page adapter

Both examples are very limited in their capabilities. They show the basic principiples as well as the power you get when dealing on a low level with the content. However, keep in mind that changing the render behavior is an option to extend ASP.NET when needed.

Caution Before considering the usage of adapters the common techniques should be tried first. The complete render framework is complex and you might experience weird behavior when not implementing it either completely or wrong.

### Configure the Examples

To configure the code, you’ll just need to make an entry in a .browser file. In the first example, I define a file called *MyClient.browser* and place it in the default folder for browser definition files, *App\_Browser*.

<browsers>

<browser refID="Default">

<controlAdapters>

<adapter adapterType="Apress.Extensibility.Adapters.SourcePageAdapter"

controlType="System.Web.UI.Page" />

</controlAdapters>

</browser>

</browsers>

This is everything you need. Launch the application and the adapter will start working. If you combine the PageAdaper with the ControlAdapter example shown previously, you can put two <adapter> tags within the same <controlAdapters> section. The only reason to split it into several sections or files is to activate it for different kinds of client devices.

For the second example simply replace the class name SourcePageAdapter with ClientSourcePageAdapter to activate this.

# Summary

In this chapter, you looked at a method of extending a core part of ASP.NET—control rendering. Using adapters, you can assign relationships between a control and its adapter. Adapters can change render behavior, persistence of view state and control state, and the treatment of controls during specific states of their life cycle.

Page adapters are specialized control adapters supporting alternative behavior of the page’s render and state persistence processes.

Adapters can be used in conjunction with device filters in order to render controls differently for devices that can’t handle ordinary HTML. Mobile devices especially suffer when it comes to large view states, complex content with several levels of nested tables, or other content originally designed for powerful machines. Using control adapters allows you to replace render behavior with a lean, device-specific version, without changing application code, page markup, or your custom controls.