Four Helpful Features to Add to Your Base Page Class

Published: 18 Dec 2008
By: Scott Mitchell
Download Sample Code

Scott Mitchell explains how to create an ASP.NET base Page class that includes extra functionality.

Contents [hide]

Introduction

All ASP.NET pages extend the Page class in the System.Web.UI namespace, which defines the base functionality, events, methods, and properties common to all ASP.NET pages. This includes common objects like `Request`, `Response`, and `Session`, along with the mechanisms for serializing view state, injecting client-side script, and managing the ASP.NET page's lifecycle. You can provide additional functionality to all your ASP.NET pages in your application by:

1. Creating your own class that includes the extra functionality desired,
2. Have this class derive from the .NET Framework's `Page` class, and
3. Configure your ASP.NET pages to extend the custom class created in step 1, rather than the .NET Framework's `Page` class.

Such custom classes are referred to as *base page classes*. For more information on creating and using base page classes check out Using a Custom Base Class for your ASP.NET Pages' Code-Behind Classes.

Base page classes usually contain customizations that are specific to the

application for which they were created, but can also contain more common functionality that, for one reason or another, was not provided by Microsoft in their `Page` class. This article shares four helpful features you can add to your base page class.

Note:

The complete code for the following features is available for download from this article, along with a demo page. The code snippets below show just the code relevant to the particular feature being examined.

Display a JavaScript Alert

The JavaScript `alert(message)` function displays the specified message in a modal dialog and is commonly used to display status information. For example, you can have any user input validation errors displayed in a client-side dialog box by adding a ValidationSummary control to your page and setting its `ShowMessageBox` property to `true`. Doing so causes the ValidationSummary control to inject JavaScript on the page that uses the `alert` function to display a summary of validation errors.

Much like how the ValidationSummary works, you can display your own client-side dialog boxes by injecting JavaScript from the page's server-side code. This involves using the `ClientScript` class and entering the JavaScript syntax to emit, and is rather cumbersome. A better approach is to encapsulate this logic into a method in the base page class. After doing this, any ASP.NET page that extends the base page class can display a client-side dialog box by calling this method.

The base page class in the download includes a method named `DisplayAlert` that takes a string as input and emits the client-side script to display the passed-in string in a dialog box.

```
01. protected virtual void DisplayAlert(string message)
02. {
03.     ClientScript.RegisterStartupScript(
04.                         this.GetType(),
05.                         Guid.NewGuid().ToString(),
06.                         string.Format("alert('{0}');",
                               message.Replace("'", @"\'")),
07.                         true
08.                     );
09. }
```

The `DisplayAlert` method can be called from any ASP.NET page that extends the custom base page class with the following code:

```
1. protected void Page_Load(object sender, EventArgs e)
2. {
3.     base.DisplayAlert("Hello, world!");
4. }
```

When the page is visited the necessary JavaScript code is sent back with the page's HTML and the browser, upon receiving and interpreting the JavaScript, displays a message box with the greeting, "Hello, world!"

Recursively Search the Control Hierarchy

All ASP.NET controls include a `FindControl(id)` method that searches the children controls for a one whose `ID` property is equal to the passed-in id string. Unfortunately, `FindControl` does not penetrate through naming containers, which are commonly used in templated controls. As a result, it can be a pain to get a programmatic reference to those controls within templated controls. The good news is that it's not at all hard to write a recursive `FindControl` method, and such a method is a good fit for a base page class.

My implementation below has two overloads: the first accepts just an *id* input parameter and starts the search at the root of the page's control hierarchy; the second one allows for a starting node in the hierarchy. If you are searching within a particular control, use the second method so as to narrow the search space.

```
01. protected virtual Control FindControlRecursive(string id)
02. {
03.     return FindControlRecursive(id, this);
04. }
05. protected virtual Control FindControlRecursive(string id, Control
    parent)
06. {
07.     // If parent is the control we're looking for, return it
08.     if (string.Compare(parent.ID, id, true) == 0)
09.         return parent;
10.     // Search through children
11.     foreach (Control child in parent.Controls)
12.     {
13.         Control match = FindControlRecursive(id, child);
14.         if (match != null)
15.             return match;
16.     }
17.     // If we reach here then no control with id was found
18.     return null;
19. }
```

The demo available for download shows this method in action with a LoginView control. The LoginView control is composed of templates and renders one of its templates based on the user visiting the page. For example, if the visitor is anonymous then the LoginView's AnonymousTemplate is rendered. Because the LoginView control is composed of templates and because these templates act as naming containers, you have to use the `FindControl` method to programmatically work with any of the Web controls within its templates. However, you have to be certain to use the `FindControl` method from the parent container because it does not penetrate naming containers.

To better understand this, consider the following LoginView, which has a TextBox named `UserNameInTextBox` in its LoggedInTemplate and a Label named `CurrentTime` in its AnonymousTemplate.

```
01. <asp:LoginView ID="lgnView" runat="server">
02.     <LoggedInTemplate>
03.         You are logged in. Welcome back,
04.         <asp:TextBox ID="UserNameInTextBox" runat="server">
```

```
            </asp:TextBox>
    </LoggedInTemplate>
    <AnonymousTemplate>
        You are not logged in. The current time is:
        <asp:Label ID="CurrentTime" runat="server"></asp:Label>
    </AnonymousTemplate>
</asp:LoginView>
```

To set the `CurrentTime` Label control's `Text` property you need to first use the `FindControl` method to get a reference to it. But you need to be careful here because the `FindControl` method's search does not penetrate naming containers and the LoginView's template constitutes a naming container. Consequently, the following call to `FindControl` will return `null`:

```
Label CurrentTime = Page.FindControl("CurrentTime") as Label;
```

Instead, you have to start at the parent container, LoginView:

```
Label CurrentTime = LoginView.FindControl("CurrentTime") as Label;
```

Note the difference in the two code snippets above: one calls the `Page` class's `FindControl` method, the other calls the LoginView's `FindControl` method. The first returns null because the search does not drill into the LoginView's templates to find `CurrentTime`, but the second one succeeds because it starts its search from that same naming container that the Label exists in.

A simpler approach is to use our new recursive method, `FindControlRecursive`. Because this method plows through naming containers you can search starting at the top of the page's control hierarchy (the default behavior) and it will still find the control nestled within the LoginView's template (assuming that the page is being requested by an anonymous visitor).

```
Label CurrentTime = base.FindControlRecursive("CurrentTime") as
   Label;
if (CurrentTime != null)
    CurrentTime.Text = DateTime.Now.ToString();
```

It may seem a bit superfluous to create a `FindControlRecursive` method for this particular example; after all, we can reference the Label by calling the LoginView control's `FindControl` method. But imagine that the LoginView control was itself within a naming container. Then in order to reference the Label you would have to first reference the LoginView using `FindControl` from its parent naming container and then use `FindControl` again to get the Label. Our recursive `FindControl` implementation does not require this extra `FindControl` call.

A Generics-based implementation of a recursive `FindControl` method can be found at Steve Smith's blog: Recursive FindControl.

Record Page Execution Times

It's helpful at times to know how long it took a particular ASP.NET page to be processed on the server, or how long a particular operation took. While ASP.NET's tracing feature provides this information and more, sometimes that blob of information is much more than is needed. A simpler approach is to have the base page optionally record and display the page's execution time.

Because we might not want to measure and record the execution time on all pages, it's a good idea to add a property to the base page class that indicates whether these measurements are to be made. The base page class available for download has a Boolean property named `MeasureExecutionTime` that does just that; it defaults to false.

```
protected virtual bool MeasureExecutionTime
{
    get

```

```
        {
            object o = ViewState["MeasureExecutionTime"];
            if (o == null)
                return false;
            else
                return (bool)o;
        }
        set
        {
            ViewState["MeasureExecutionTime"] = value;
        }
    }
```

A Stopwatch class instance named `watch` is used to measure the execution time. This object is instantiated and started in the base page's `OnInit` method, which executes during the Init stage of the page lifecycle.

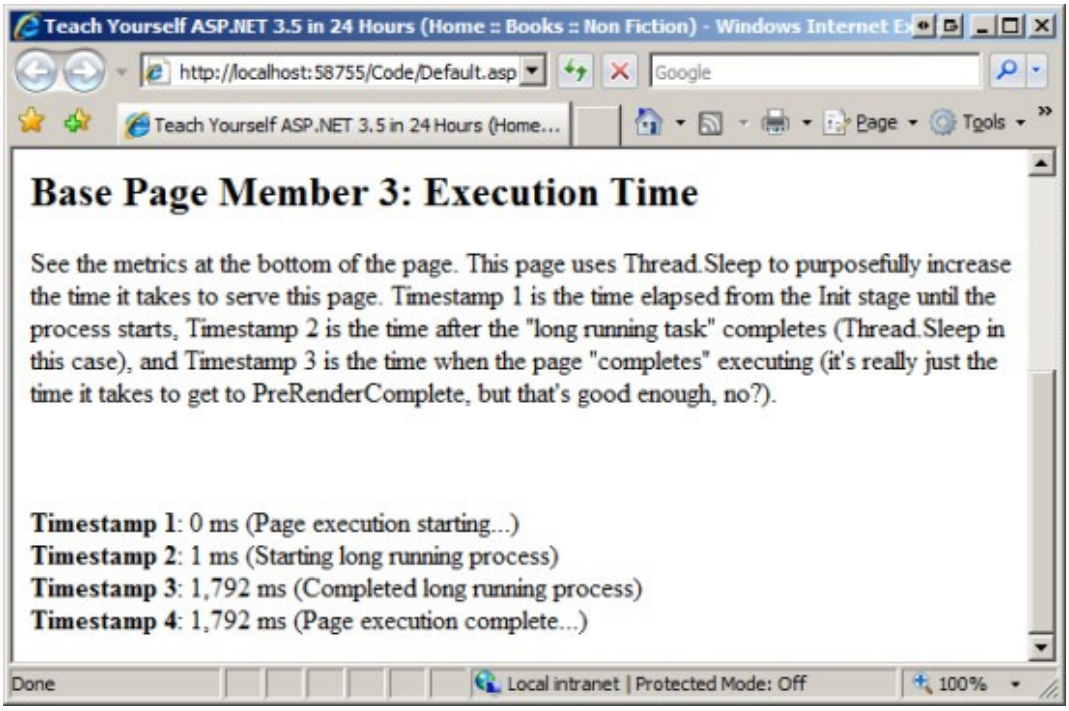
```
protected override void OnInit(EventArgs e)
{
    base.OnInit(e);
    // Start measuring page execution time
    if (this.MeasureExecutionTime)
    {
        watch = new Stopwatch();
        watch.Start();
        this.RecordTimestamp("Page execution starting...");
    }
}
```

The base page class includes a method named `RecordTimestamp` that, when called, stores the number of elapsed milliseconds and passed-in description in a `List` of `TimestampInfo` objects. The `TimestampInfo` class includes properties that record the elapsed time (in milliseconds) along with a human-friendly description. A page developer can call `RecordTimestamp` whenever he would like to record the elapsed time. During the `PreRenderComplete` event, the `Stopwatch` object is stopped and the times are displayed on the page. Specifically, the times are tacked on immediately before the closing `</form>` tag using a series of `LiteralControl` instances.

Note that the `MeasureExecutionTime` property defaults to `false`. Therefore, to use this functionality you will need to explicitly set this property to true in your ASP.NET pages in the `Page_Init` event handler. The following code shows an ASP.NET page that uses this page execution timing feature to not only track the total page load time, but also to determine how long a particular operation takes. In this case we are timing a call to `Thread.Sleep`; in practice you would be interested in timing potentially long running tasks, like Web Service calls or database queries.

```
protected void Page_Init(object sender, EventArgs e)
{
    base.MeasureExecutionTime = true;
}
protected void Page_Load(object sender, EventArgs e)
{
    // Record timestamp
    base.RecordTimestamp("Starting long running process");
    // Do long running process
    Random rnd = new Random();
    Thread.Sleep(rnd.Next(1000) + 1000);
    // Record timestamp
    base.RecordTimestamp("Completed long running process");
}
```

As the screen shot in Figure 2 shows, the output isn't pretty, but the idea is that this information is used solely for diagnostic purposes and is not intended to be displayed on a production website.



Setting the Page Title

The `Page` class in the .NET Framework includes a `Title` property that can be used to programmatically set the text in the page's `<title>` element. The browser displays the text within the `<title>` element in the window's title bar, it is the default text used when a page is bookmarked, and search engines typically display this title when listing search results. Consequently, it's a good idea to give your page's meaningful titles. Unfortunately, Visual Studio gives each new ASP.NET page a default title of "Untitled Page," which means if you forget to change this you'll end up with a lot of pages on your site with the same meaningless title.

Note:

Here's a fun bit of trivia: you can use Google's search filter `allintitle:"title"` to search for web pages with a particular title. A search for "Untitled Page" returns over 8.1 million results! You can use the `site:domainName` filter to see how many pages in Google's index are named "Untitled Page" on your site (or anyone else's). As of the time of this writing Google reports that there are 7,920 web pages titled "Untitled Page" on microsoft.com.

An easy fix is to beef up your base page class so that it sets the page's title automatically (if it was not already set). The base page class available for download has a method called `SetPageTitle` that is automatically called during the `Load` event. If the base page class property `AutoSetPageTitle` is true (the default) and the page's `Title` property has not yet been set or is set to "Untitled Page", then the page's `Title` property is set based on the site map or the page's file name.

```
01. protected virtual void SetPageTitle()
02. {
03.     if (this.AutoSetPageTitle && (string.IsNullOrEmpty(this.Title) ||
04.        string.Compare(this.Title, "Untitled Page", true) == 0))
05.     {
06.         if (SiteMap.Enabled)
07.         {
08.             // See if this page is in the SiteMap
09.
```

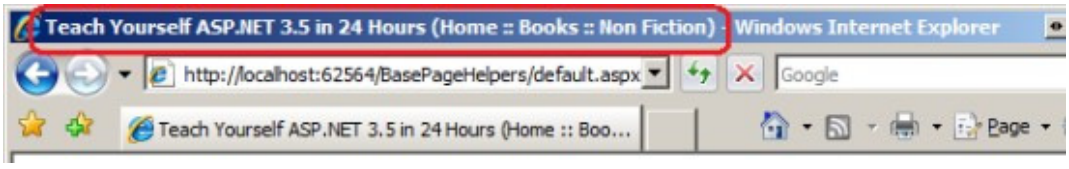
```
            SiteMapNode current = null;
            if (string.IsNullOrEmpty(this.SiteMapProvider))
                current = SiteMap.CurrentNode;
            else
                current =
                   SiteMap.Providers[this.SiteMapProvider].CurrentNode;
            if (current != null)
            {
                // Build up the title
                SetPageTitle(current);
                return;
            }
        }
        // If we reach here we still do not have a title set... use
           the filename
        this.Title =
           Path.GetFileNameWithoutExtension(Request.PhysicalPath);
    }
}
protected virtual void SetPageTitle(SiteMapNode current)
{
    if (this.AutoSetPageTitle)
    {
        StringBuilder titleBuilder = new StringBuilder(200);
        titleBuilder.Append(current.Title);
        current = current.ParentNode;
        if (current != null)
        {
            string parentPathReverse = current.Title;
            while (current.ParentNode != null)
            {
                current = current.ParentNode;
                parentPathReverse = string.Concat(current.Title,
                   this.SiteMapNodeSeparator, parentPathReverse);
            }
            titleBuilder.Append("
               (").Append(parentPathReverse).Append(")");
        }
        this.Title = titleBuilder.ToString();
    }
}
```

If your site is configured to use ASP.NET's site map feature then those pages that are referenced in the site map will have their title set to the value: **Title (RootTitle :: GrandParentTitle :: ... :: ParentTitle)**

Where Title is the title of the current page in the site map, *RootTitle* is the title of the root node in the site map, and *GrandParentTitle* down to *ParentTitle* are the titles of the current node's ancestors. The demo web page is located in the site map in the following hierarchy:

- Home
- Books
- Non Fiction
- Demo Page

The title of the demo page in the site map is: "Teach Yourself ASP.NET 3.5 in 24 Hours." As a result, the base page sets this page's title to: **Teach Yourself ASP.NET 3.5 in 24 Hours (Home :: Books :: Non Fiction)**



If the page is not in the site map, or if the site map feature is disabled, then the `Title` property is set to the file name of the page (less the extension). In

other words, if the page is titled `About.aspx` the title will be set to About.

Happy Programming!

Further Reading

- Displaying a Breadcrumb in the Page's Title
- Dynamically Setting the Page's Title in ASP.NET
- HttpModule for Timing Requests
- Recursive FindControl
- Using a Custom Base Page Class for your ASP.NET Pages' Code-Behind Classes

<< Previous Article | Continue reading and see our next or previous articles | Next Article >>

About Scott Mitchell

Scott Mitchell, author of eight ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, tra...

This author has published **11** articles on DotNetSlackers. View other articles or the complete profile **here**.

Other articles in this category

JavaScript with ASP.NET 2.0 Pages - Part 1
ASP.NET 2.0 has made quite a few enhancements over ASP.NET 1.x in terms of handling common client-si...
ASP.NET ComboBox
The ASP.NET ComboBox is an attempt to try and enhance some of the features of the Normal ASP.NET Dro...
Upload multiple files using the HtmlInputFile control
In this article, Haissam Abdul Malak will explain how to upload multiple files using several file up...
JavaScript with ASP.NET 2.0 Pages - Part 2
ASP.NET provides a number of ways of working with client-side script. This article explores the usag...
Using WebParts in ASP.Net 2.0
This article describes various aspects of using webparts in asp.net 2.0.

You might also be interested in the following related blog posts

Self-reference hierarchy with Telerik TreeView for Silverlight read more
Mapping references and collections in

Telerik OpenAccess ORM (Part 1) read more
New Entity Framework Feature CTP for VS2010 Beta 2 read more
Telerik announces second Beta release of RadControls for Silverlight/WPF Q3 2009 read more
Four Little Known, Helpful Methods, Properties, and Features for ASP.NET Developers read more
WebAii Testing Framework Support for Extended Silverlight RadControls read more
Getting and Setting Query String Values in JavaScript read more
Entirely unobtrusive and imperative templates with Microsoft Ajax Library Preview 6 read more
Next version of EF Code Only Design laid out by MS read more
Comments on my recent benchmarks. read more

Top

Discussion

Subject	Author	Date
Recursive Find Control	Julio Vides	7/29/2009 6:25 AM
????	Scott Lee	12/25/2008 2:49 AM
Nice one!	Shahriar Hyder	4/6/2009 12:56 AM
FXCOP and StyleCop	Clive Chinery	4/6/2009 9:17 AM

Please login to rate or to leave a comment.