



6

Creating Consistent Looking Websites

WHAT YOU WILL LEARN IN THIS CHAPTER:

- How to use master and content pages that enable you to define the global look and feel of a web page
- How to work with a centralized base page that enables you to define common behavior for all pages in your site
- How to create themes to define the look and feel of your site with an option for the user to choose a theme at run time
- How to create skins to make site-wide changes to control layout
- What the ASP.NET page life cycle is and why it's important

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter on the Download Code tab at www.wrox.com/remtitle.cgi?isbn=1118311809. The code is in the Chapter 6 download.

When you're building a website you should strive to make the layout and behavior as consistent as possible. Consistency gives your site a professional appearance and it helps your visitors to find their way around the site. Fortunately, ASP.NET 4.5 and Visual Studio 2012 offer a number of great features and tools to implement a consistent design, helping you to create great-looking pages in no time.

In previous chapters you learned how to work with VS, HTML5, CSS, and server controls to create your web pages visually. Chapter 5 introduced you to programming in ASP.NET. This chapter is the first that combines these concepts, by showing you—among many other things—how to use programming code to change the appearance of the site.

The first section shows you how to create a master page that defines the general look and feel of a page. The ASPX pages in your site can then use this master page without the need to repeat the general layout. The remaining sections of this chapter build on top of this master page.

CONSISTENT PAGE LAYOUT WITH MASTER PAGES

With most websites, only part of the page changes when you go from one page to another. The parts that don't change usually include common regions like the header, a menu, and the footer. To create web pages with a consistent layout, you need a way to define these relatively static regions in a single template file. Versions of ASP.NET prior to 2.0 did not have a template solution, so you were forced to duplicate your page layout on every single page in the website, or resort to weird programming tricks. Fortunately, this is no longer the case due to *master pages*. The biggest benefit of master pages is that they enable you to define the look and feel of all the pages in your site in a single location. This means that if you want to change the layout of your site—for instance, if you want to move the menu from the left to the right—you need to modify only the master page, and the pages based on this master pick up the changes automatically.

When master pages were introduced in ASP.NET 2.0, they were quickly embraced by the developer community as *the* template solution for ASP.NET pages because they are very easy to use. Even better, VS has great design-time support, because it enables you to create and view your pages at design time during development, rather than only in the browser at run time.

To some extent, a master page looks like a normal ASPX page. It contains static HTML such as the `<html>`, `<head>`, and `<body>` elements, and it can also contain other HTML and ASP.NET Server Controls. Inside the master page, you set up the markup that you want to repeat on every page, like the general structure of the page and the menu.

However, a master page is not a true ASPX page and cannot be requested in the browser directly; it only serves as the template on which real web pages—called *content pages*—are based.

Instead of the `@ Page` directive that you saw in Chapter 4, a master page uses an `@ Master` directive that identifies the file as a master page:

VB.NET

```
<%@ Master Language="VB" %>
```

C#

```
<%@ Master Language="C#" %>
```

Just like a normal ASPX page, a master page can have a Code Behind file, identified by its `CodeFile` and `Inherits` attributes:

VB.NET

```
<%@ Master Language="VB" CodeFile="Frontend.master.vb"
    Inherits="MasterPages_Frontend" %>
```

C#

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="Frontend.master.cs"
    Inherits="MasterPages_Frontend" %>
```

To create regions that content pages can fill in, you define `ContentPlaceHolder` controls in your page like this:

```
<asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
</asp:ContentPlaceHolder>
```

You can create as many placeholders as you like, although you usually need only a few to create a flexible page layout.

The content pages, which are essentially normal ASPX files, without the code that they're going to take from the master page, are connected to a master page using the `MasterPageFile` attribute of the `Page` directive:

VB.NET

```
<%@ Page Title="" Language="VB" MasterPageFile="~/MasterPages/Frontend.master"
    AutoEventWireup="false" CodeFile="Default.aspx.vb" Inherits="_Default">
```

C#

```
<%@ Page Title="" Language="C#" MasterPageFile="~/MasterPages/Frontend.master"
    AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default">
```

The page-specific content is then put inside a `Content` control that points to the relevant `ContentPlaceHolder`:

```
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
    runat="Server">
</asp:Content>
```

Note that the `ContentPlaceHolderID` attribute of the `Content` control points to the `ContentPlaceHolder` that is defined in the master page. Right now it points to the default name of `ContentPlaceHolder1`, but in a later exercise you see how to change this.

At run time, when the page is requested, the markup from the master page and the content page are merged, processed, and sent to the browser. Figure 6-1 shows a diagram of the master page with just one `ContentPlaceHolder` and the content page that results in the final page that is sent to the browser.

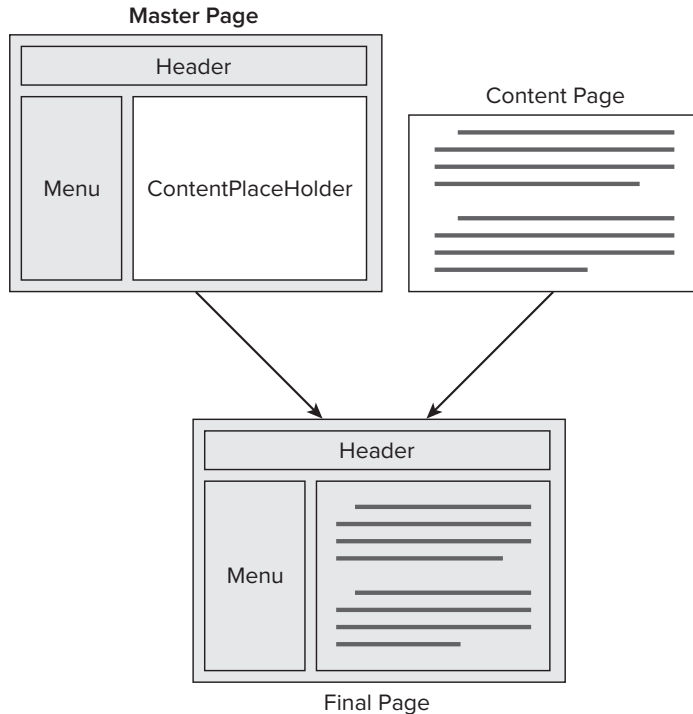


FIGURE 6-1

To see this process in action, the following sections guide you through creating master and content pages.

Creating Master Pages

You can add master pages to the site using the Add New Item dialog box. You can place them in any custom folder in the site, including the root folder, but from an organizational point of view, it's often easier to store them in a separate folder. Just like normal ASPX pages, they support the inline code model as well as the Code Behind model. The master pages used in the Planet Wrox project use the Code Behind model exclusively. In the following exercise, you learn how to create a simple master page and add some HTML to it to define the general structure of the pages in your website.

TRY IT OUT Creating a Master Page

1. Open the Planet Wrox project in Visual Studio if it is not open already.
2. In Chapter 2 you created a folder called `MasterPages` to hold your master pages and then added a single master page to that folder. If you didn't carry out that exercise, add the master page now. To do this, create the `MasterPages` folder in the root of the site, right-click the new folder, choose Add ⇨ Add New Item, and select the Master Page item. Make sure that the master page uses Code Behind and that it is using your preferred programming language. Name the master page `Frontend.master`. Finally, click Add.

3. Add the following highlighted code between the `<form>` tags of the master page, replacing the `<div>` tags and the `ContentPlaceHolder` that VS added for you when you created the master page. Note that this is almost the same code you added to `Default.aspx` in Chapter 3, except for the `<asp:ContentPlaceHolder>` element and the `<a>` element within the Header `<div>`. The `<a>` element takes the user back to the homepage, and will be styled later.

```
<form id="form1" runat="server">
  <div id="PageWrapper">
    <header><a href="/">Header Goes Here</a></header>
    <nav>Menu Goes Here</nav>
    <section id="MainContent">
      <asp:ContentPlaceHolder ID="cpMainContent" runat="server">
      </asp:ContentPlaceHolder>
    </section>
    <aside id="Sidebar">Sidebar Goes Here</aside>
    <footer>Footer Goes Here</footer>
  </div>
</form>
```

Make sure that you have the `ContentPlaceHolder` within the `MainContent <section>` tags. You can drag one from the Toolbox onto the page or enter the code directly, using IntelliSense's helpful hints. In both cases you should give the control an ID of `cpMainContent`.

4. Next, switch the master page into Design View and then drag `Styles.css` from the `Styles` folder in the Solution Explorer onto the master page. As soon as you drop the file, VS updates the Design View and shows the layout for the site that you created in Chapter 3. If the design doesn't change, switch to Markup View and ensure there's a `<link>` tag in the head of the page pointing to your CSS file:

```
<asp:ContentPlaceHolder ID="head" runat="server">
</asp:ContentPlaceHolder>
<link href="../../Styles/Styles.css" rel="stylesheet" type="text/css" />
</head>
```

The page should now look like Figure 6-2 in Design View.

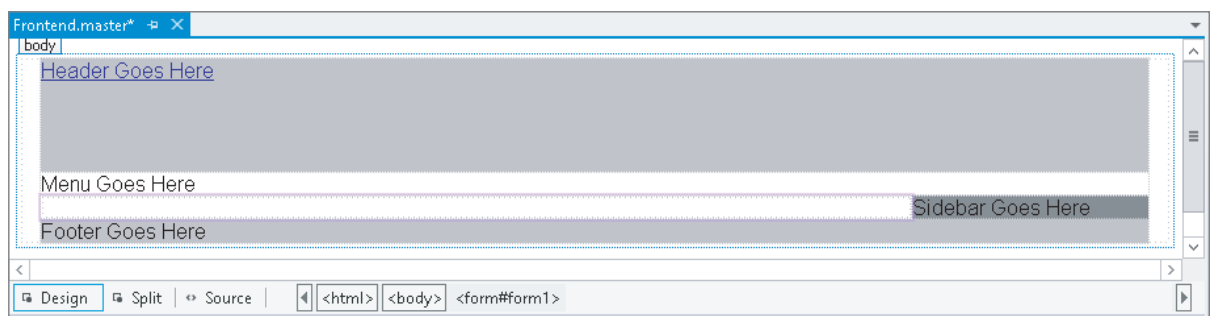


FIGURE 6-2

Note the area with the purple border around it between the menu and the footer region in your Design View. This is the `ContentPlaceHolder` control that is used by the content pages. You see how to use it in the next exercise.

5. Drag the file `modernizr-2.6.2.js` from the `Scripts` folder into the `<head>` section of the Master Page below the CSS file. Then manually replace the two leading periods with a slash, like this:

```
<script src="/Scripts/modernizr-2.6.2.js"></script>
```

6. You can save and close the master page because you're done with it for now.

How It Works

Within VS, master pages behave like normal pages. You can add HTML and server controls to them, and you can manage the page both in Markup and Design View. The big difference is, of course, that a master page isn't a true page itself; it only serves as a template for other pages in your site.

In the next section you see how to use this master page as the template for a content page.

Creating Content Pages

A master page is useless without a content page that uses it. Generally, you'll have only a few master pages, whereas you can have many content pages in your site. To base a content page on a master page, check the `Select Master Page` option at the bottom right of the `Add New Item` dialog box when you add a new `Web Form` to your site. Alternatively, you can set the `MasterPageFile` attribute on the page directly in the Markup View of the page. You saw this `@Page` directive earlier in this chapter when master and content pages were introduced.

Content pages can only directly contain `Content` controls that each map to a `ContentPlaceHolder` control in the master page. These content controls in turn can contain standard markup like HTML and server control declarations. Because the entire markup in a content page needs to be wrapped by `<asp:Content>` tags, it's not easy to turn an existing ASPX page into a content page. Usually the easiest thing to do is copy the content you want to keep to the clipboard, delete the old page, and then add a new page based on the master page to the website. Once the page is added, you can paste the markup within the `<asp:Content>` tags. You see how this works in the following exercise.

TRY IT OUT Adding a Content Page

In this Try It Out you see how to add a content page to the site that is based on the master page you created earlier. Once the page is added, you add content to the `<asp:Content>` regions.

1. In previous exercises you added standard ASPX pages to your project, which should now be "upgraded" to make use of the new master page. If you want to keep the welcome text you added to `Default.aspx` earlier, copy all the HTML between the `MainContent` `<section>` tags to the clipboard (that is, the `<h1>` and the two `<p>` elements that you created earlier) and then delete the `Default.aspx` page from the Solution Explorer. Next, right-click the website in the Solution Explorer and choose `Add` `⇨` `Add New Item`. Select the correct programming language, click `Web`

Form, name the page `Default.aspx`, and then, at the bottom of the dialog box, select the checkboxes for Place Code in Separate File and Select Master Page, as shown in Figure 6-3.

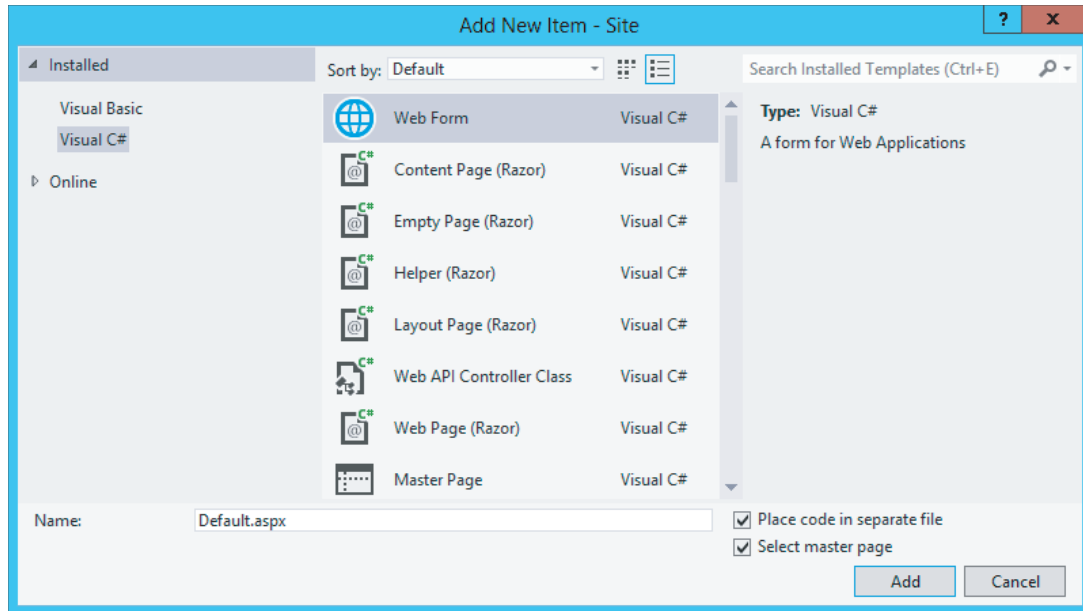


FIGURE 6-3

Finally, click the Add button.

2. In the Select a Master Page dialog box (see Figure 6-4), click the `MasterPages` folder in the left-hand pane, and then in the area at the right, click `Frontend.master`.

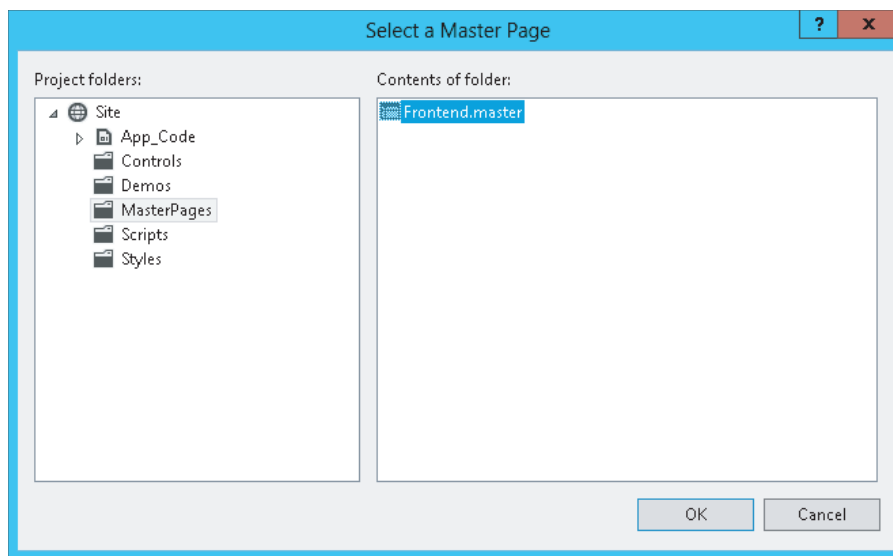


FIGURE 6-4

Click OK to add the page to your website.

Instead of getting a full page with HTML as you got with standard ASPX pages, you now only get two `<asp:Content>` placeholders as shown in this VB.NET example:

```
<%@ Page Title="" Language="VB" MasterPageFile="~/MasterPages/Frontend.master"
    AutoEventWireup="false" CodeFile="Default.aspx.vb" Inherits="_Default" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="cpMainContent" runat="Server">
</asp:Content>
```

3. Switch to Design View and note that everything is grayed out and read-only, except for the `<asp:Content>` region for `cpMainContent`. Figure 6-5 shows you how the page should look.

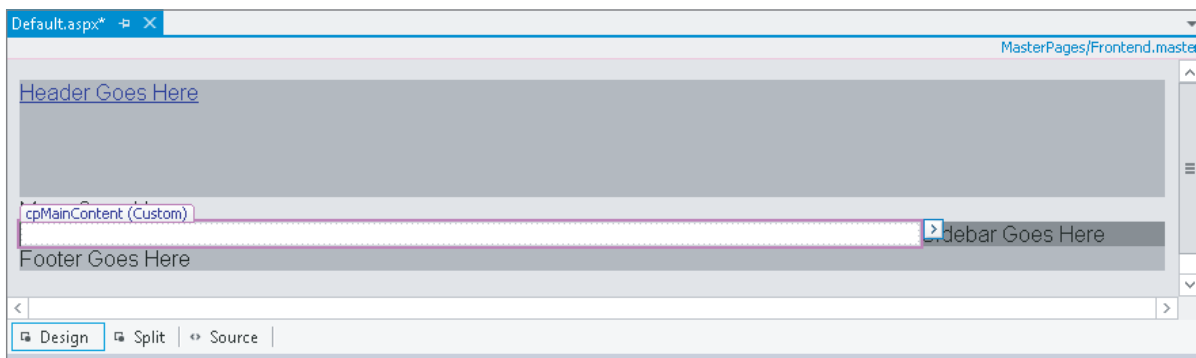


FIGURE 6-5

Also note that VS conveniently lists the master page that this page is based on in the top-right corner of Design View, visible in Figure 6-5. Clicking the name opens the master page in the editor.

4. If you still have the old markup from the `Default.aspx` on the clipboard, click once inside the `cpMainContent` placeholder and press `Ctrl+V`. (Note: you can do this both in Design View and in Markup View). This adds the markup to the page, right between the `<asp:Content>` tags.
5. Save your changes by pressing `Ctrl+S` and press `Ctrl+F5` to open the page in your browser. The browser should display the page very closely to what you saw in Design View (see Figure 6-5).
6. Now take a look at the HTML for the page in the browser. You can do this by right-clicking the page and choosing View Source or View Page Source. Note that the source of the final page in the browser is a combination of the source of the master page and the content page:

```
<div id="PageWrapper">
  <header><a href="/">Header Goes Here</a></header>
  <nav>Menu Goes Here</nav>
  <section id="MainContent">
    <h1>Hi there visitor and welcome to Planet Wrox</h1>
    <p class="Introduction">
      We're glad you're paying a visit to <a
      href="http://www.PlanetWrox.com">
        www.PlanetWrox.com</a>, the coolest music community site on the Internet.
    </p>
    ...
  </section>
</div>
```

The first four lines come from the master page and the bolded lines of HTML code come from the content page.

7. Switch back to VS and create a new page called `Login.aspx` in the root of the site based on the master page. Notice how VS remembered your last settings with regard to the master page and Code Behind (make sure both are checked in case you unchecked them earlier). Switch to Markup View and create an `<h1>` element inside the `cpMainContent` placeholder with the text **Log in to Planet Wrox**. There's no need to add any other controls to this page just yet, but it serves as the basis for the login functionality you create in Chapter 16. Without any content in the `MainContent` element, the `Sidebar` will be moved to the left of the page.
8. Go back to `Default.aspx` and switch to Design View. Beneath the welcome text with the header and two `<p>` elements, create a new paragraph (press Enter in Design View) and type some text (for example, **You can log in here**). Notice how this new paragraph has a `class` attribute called `Introduction` because VS applies the previous class to new paragraphs automatically. Remove this class using the Clear Styles option of the Apply Styles window, or manually remove it from the code in Markup View.
9. Highlight the words "log in" in Design View and choose Format ⇄ Convert to Hyperlink from the main menu. In the dialog box that follows, click the Browse button and select the `Login.aspx` page that you just created. Click OK twice.
10. Save all changes and press Ctrl+F5 again to view `Default.aspx` in the browser. Then click the link you created in the preceding step. You should now be taken to `Login.aspx`. Note that the general layout, like the header and the sidebar, is maintained. The only thing that changes when you go from page to page is the content in the main content area.

How It Works

When a page based on a master page is requested in the browser, the server reads in both the content page and the master page, merges the two, processes them, and then sends the final result to the browser. In step 6 of this exercise you saw that the HTML in the browser for the requested page contained the markup from both files.

Master pages will save you a lot of work when it comes to updating or radically changing the looks of your site. Because the entire design and layout of your site is defined in the master page, you only need to touch that single file when you want to make any changes. All content pages will pick up the changes automatically.

A Closer Look at Master Pages

So far you've seen a master page with a content placeholder for the main content. But if you look at the master page in Markup View, you'll find another content placeholder in the head section of the page:

```
<head runat="server">
  <title></title>
  <asp:ContentPlaceHolder id="head" runat="server">
  </asp:ContentPlaceHolder>
  ...
</head>
```

This placeholder is added for you automatically to each new master page you create. You can use it in content pages to add page-specific content that belongs between the `<head>` tags of a page, such

as CSS (both embedded and external style sheets) and JavaScript. You learn more about JavaScript in Chapters 10 and 11. You need to add content to this placeholder in Markup View, because it's not visible in Design View.

The `ContentPlaceHolder` called `cpMainContent` in the master page currently does not contain any markup itself. However, it doesn't have to be like this. You can easily add your own content there that will serve as the default in your content pages as long as it's not overridden by the content page. For example, you can have the following `ContentPlaceHolder` in a master page:

```
<asp:ContentPlaceHolder ID="cpMainContent" runat="server">
  This is default text that shows up in content pages that don't
  explicitly override it.
</asp:ContentPlaceHolder>
```

When you base a new page on this master page, you won't see this default at first in Markup View. However, you can open the Content control's Smart Tasks panel, shown in Figure 6-6, and choose Default to Master's Content.

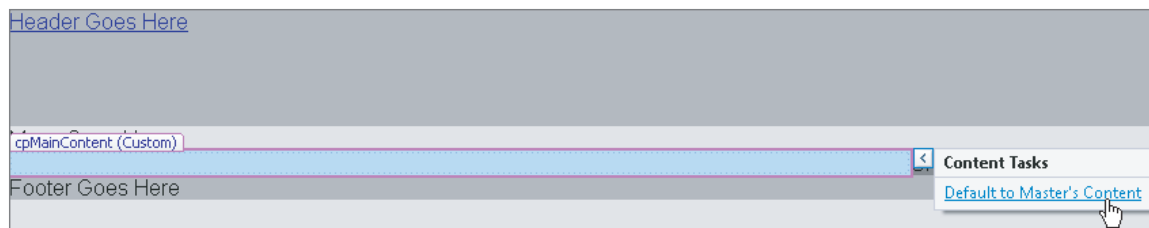


FIGURE 6-6

When you click Yes when asked if you want to default to the master page content, VS removes the entire Content control from the Markup View of the page. However, when you request the page in the browser you will still see the default content from the master page. In Design View, the content is still visible, now presented as a read-only area on the design surface. A master page with default content can be useful if you add a new `ContentPlaceHolder` to the master page at a later stage. Existing pages can simply display the default content, without the need for you to touch all these pages. New pages can define their own content. If you don't have default content in the `ContentPlaceHolder` control in the master page and the content page doesn't have a Content control for the `ContentPlaceHolder`, no output is sent to the browser.

Once you have defaulted to the master page's content, you can create custom content again by opening the Smart Tasks panel and choosing Create Custom Content. This copies the default contents into a new Content control that you can then modify.

Nesting Master Pages

It is also possible to nest master pages. A nested master page is a master that is based on another master page. Content pages can then be based on the nested master page. This is useful if you have a website that targets different areas that still need to share a common look and feel. For example, you can have a corporate website that is separated by departments. The outer master page defines the global look and feel of the site, including corporate logo and other branding elements. You can

then have different nested master pages for different departments. For example, the sales department's section could be based on a different master than the marketing department's, enabling each to add their own identity to their section of the site. VS 2012 has excellent Design View support for nested master pages, giving you a good look at how the final page will end up.

Creating a nested master page is easy: check the Select Master Page check box when you add a master page just as you do when you add a normal content page to the site. Then add markup and ContentPlaceHolder controls to the Content controls at locations that you want to override in the content pages. Finally, you choose your nested master page as the master for new content pages you create. Inside the content page, you only see the ContentPlaceHolder controls from the nested master page, not from its parent.

Master Page Caveats

Although master pages are great and can save you a lot of work, you need to be aware of some caveats.

For starters, the ASP.NET run time changes the *client ID* of your controls in the page. This is the `id` attribute that is used in client script to access controls from JavaScript in the browser and with CSS ID selectors. With normal ASPX pages, the server-side ID of a control is usually inserted one-on-one in the final HTML. For example, a Button control with a server-side ID of `Button1` in a normal ASPX page defined with this code,

```
<asp:Button ID="Button1" runat="server" Text="Click Me" />
```

ends up with a client-side ID like this in the final HTML:

```
<input type="submit" name="Button1" value="Click Me" id="Button1" />
```

However, the same button inside an `<asp:Content>` control ends up like this:

```
<input type="submit" name="ctl00$cpMainContent$Button1"
value="Click Me" id="cpMainContent_Button1" />
```

The name attribute has been prefixed with the auto-generated ID of the master page (`ctl00`) and both the name and the `id` attributes contain the ID of the ContentPlaceHolder control (`cpMainContent`).

This means that any client-side code that previously referred to `Button1` should now refer to `cpMainContent_Button1`.

Note that this is not just a master page problem. You'll also run into this behavior in other situations; for example, when working with user controls (discussed in Chapter 8) and data-bound controls (discussed in Chapter 13 and onward).

The second caveat is related to the first. Because the name and `id` of the HTML elements are changed, they add considerably to the size of the page. This may not be problematic for a single control, but once you have pages with lots of controls, this could impact the performance of your site. The problem gets worse with nested master pages, where both content controls are appended to the ID. The same button inside a nested master page can end up like this:

```
<input type="submit" name="ctl00$ctl00$cpMainContent$ContentPlaceHolder1$Button1"
value="Click Me" id="cpMainContent_ContentPlaceHolder1_Button1" />
```

To mitigate the problem, you should keep the IDs of your `ContentPlaceHolder` and `Content` controls as short as possible. To improve readability, this book uses longer names, like `cpMainContent`. However, in your own sites, you could reduce this to `MC` or `cpMC` to save some bandwidth on every request.

NOTE ASP.NET 4 introduced a new feature called `ClientIDMode` that helps minimize the problems typically associated with changing client-side IDs. You learn more about this feature in Chapter 8.

Master pages enable you to define the general look and feel of your site in a single location, thus improving the consistency and maintainability of your site. However, there is another way to improve consistency: centralize the behavior of the pages in your website. You can do this with a so-called base page, which is discussed next.

USING A CENTRALIZED BASE PAGE

In Chapter 5 you learned that, by default, all ASPX pages derive from a class called `System.Web.UI.Page`. This means all of your pages have at least the behavior defined in this class.

However, in some circumstances this behavior is not enough and you need to add your own stuff to the mix. For example, you may have the need to add some behavior that applies to all the pages in your site. Instead of adding this behavior to each and every individual page, you can create a common *base page*. All the pages in your site can then inherit from this intermediate page instead of from the standard `Page` class. The left half of Figure 6-7 shows how an ASPX page called `MyWebPage` inherits from the `Page` class directly. The right half shows a situation where the ASPX page inherits from a class called `BasePage`, which in turn inherits from `Page`.

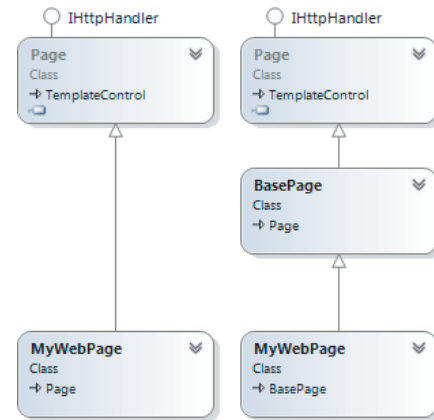


FIGURE 6-7

To have your pages inherit from this base page, you need to do two things:

- Create a class that inherits from `System.Web.UI.Page` in the `App_Code` folder of your website.
- Make the web pages in your site inherit from this base page instead of the standard `Page` class.

In an upcoming exercise you create a new base page class inside the `App_Code` folder. For now, the sole purpose of this class is to check the `Title` of the page at run time to stop pages with an

empty title or a meaningless title like “Untitled Page” making it to the browser. Giving your pages a unique and helpful title helps the major search engines to index them, so it’s recommended to always include a title in your web pages. Checking the title programmatically is relatively easy to do, which enables you to focus on the concept of inheritance rather than on the actual code. In the section that discusses themes later in this chapter, you modify the base page once more, this time to retrieve the user’s preference for a theme.

NOTE Older versions of VS used “Untitled Page” as the default title for new Web Forms. However, starting with the Service Pack 1 release of Visual Studio 2008, the default title is an empty string. I decided to leave the check for “Untitled Page” in the base page so you can see how you can check for unwanted titles.

Before you can implement the base class, you need to know more about the *ASP.NET page life cycle*, an important concept that describes the process a web page goes through when requested by a browser.

An Introduction to the ASP.NET Page Life Cycle

When you think about how a page is served by a web server to the browser and think of this process as the *life cycle of a page*, you can probably come up with a few important moments in the page’s life. For example, the initial request by the browser is the starting point for the page’s “life.” Similarly, when the page has sent its entire HTML to the browser, its life may seem to end. However, more interesting events are going on in the page’s life cycle. The following table describes eight broad phases the page goes through. Within each phase, at least one event is raised that enables a page developer to hook into the page’s life cycle and perform actions at the right moment. You see an example of this in the next exercise.

PHASE	DESCRIPTION
Page request	A request to an ASPX page starts the life cycle of that page. When the web server is able and allowed to return a cached copy of the page, the entire life cycle is not executed. In all other situations, the page enters the start phase.
Start	In this phase, the page gets access to properties like <code>Request</code> and <code>Response</code> that are used to interact with the page’s environment. In addition, during this phase the <code>PreInit</code> event is raised to signal that the page is about to go into the initialization phase. You use this event later to set the theme of a page.
Page initialization	During this phase, the controls you have set up in your page or added programmatically become available. Additionally, the <code>Page</code> class fires three events: <code>Init</code> , <code>InitComplete</code> , and <code>PreLoad</code> .

continues

(continued)

PHASE	DESCRIPTION
Load	During this phase, the control properties are loaded from View State and Control State during a postback. For example, when you change the selected item in a <code>DropDownList</code> and then cause a postback, this is the moment where the correct item gets preselected in the drop-down list again, which you can then work with in your server-side code. Also, during this phase the page raises the <code>Load</code> event.
Validation	In the validation phase, the Validation controls used to validate user input are processed. You learn about validators in Chapter 9.
Postback event handling	During this phase, the controls in your page may raise their own events. For example, the <code>DropDownList</code> may raise a <code>SelectedIndexChanged</code> event when the user has chosen a different option in the list. Similarly, a <code>TextBox</code> may raise the <code>TextChanged</code> event when the user has changed the text before she posted back to the server. When all event processing is done, the page raises the <code>LoadComplete</code> event. Also during this phase the <code>PreRender</code> event is raised to signal that the page is about to render to the browser. Shortly after that, <code>SaveStateComplete</code> is raised to indicate that the page is done storing all the relevant data for the controls in View State.
Rendering	Rendering is the phase where the controls (and the page itself) output their HTML to the browser.
Unload	The unload phase is really a clean-up phase. This is the moment where the page and controls can release resources they were holding on to. During this phase, the <code>Unload</code> event is raised so you can handle any cleanup you may need to do.

One thing that's important to realize is that all these events fire *at the server*, not at the client. So, even if you change, say, the text of a text box at the client, the `TextChanged` event of the `TextBox` control will fire at the server after you have posted back the page.

Now you may wonder why you need to know all of this. The biggest reason to have some understanding of the page life cycle is that certain actions can be performed only at specific stages in the page life cycle. For example, dynamically changing the theme has to take place in `PreInit`, as you'll see later. To really understand the ASP.NET page life cycle, you need to know a little more about controls, state, events, and so on. Therefore, you'll revisit the page life cycle again in Chapter 15 where you get a good look at all the different events that fire, and in what order.

In the next exercise, you use the `PreRender` event of the `Page` class to check the title. Because a developer could set the page's title programmatically during many events, checking for a correct title should be done as late as possible in the page's life cycle, which is why `PreRender` is the best event for this.

Implementing the Base Page

Implementing a base page is pretty easy: all you need to do is add a class file to your `App_Code` folder, add some code to it, and you're done. What's often a bit more difficult is to make sure each page in your site inherits from this new base page instead of from the standard `System.Web.UI.Page` class. Unfortunately, there is no way to configure the application to do this for you automatically when using Code Behind, so you need to modify each page manually. Visual Studio makes it a little easier for you by enabling you to export a page template that already contains this code. In the next exercise you add a base page to the site and in a later exercise you see how to export a web form to a template so you can add files that use the base page in no time.

TRY IT OUT Creating a Base Page

1. Right-click the `App_Code` folder in the Solution Explorer and choose `Add ⇨ Add New Item`. Select `Class` in the Templates list and name the file `BasePage`. You could choose another name if you like but `BasePage` clearly describes the purpose of the class, making it easier to understand what it does.
2. Clear the contents of the file, and then add the following code:

VB.NET

```
Public Class BasePage
    Inherits System.Web.UI.Page

    Private Sub Page_PreRender(sender As Object, e As EventArgs) Handles Me.PreRender
        If String.IsNullOrEmpty(Me.Title) OrElse Me.Title.Equals("Untitled Page",
            StringComparison.CurrentCultureIgnoreCase) Then
            Throw New Exception(
                "Page title cannot be ""Untitled Page"" or an empty string.")
        End If
    End Sub

End Class
```

C#

```
using System;

public class BasePage : System.Web.UI.Page
{
    private void Page_PreRender(object sender, EventArgs e)
    {
        if (string.IsNullOrEmpty(this.Title) || this.Title.Equals("Untitled Page",
            StringComparison.CurrentCultureIgnoreCase))
        {
            throw new Exception(
                "Page title cannot be \"Untitled Page\" or an empty string.");
        }
    }
}
```

```

    }

    public BasePage()
    {
        this.PreRender += Page_PreRender;
    }
}

```

3. Save the file and close it, and then open the `Login.aspx` page that you created earlier. Open its Code Behind file and change the `Inherits` code (the colon `[:]` in `C#`) so the login page inherits from the `BasePage` class you created earlier:

VB.NET

```

Partial Class Login
    Inherits BasePage
    ...
End Class

```

C#

```

public partial class Login : BasePage
{
    ...
}

```

4. Save the page and then request it in the browser by pressing `Ctrl+F5`. If you haven't changed the title of the page earlier, you should be greeted by the error shown in Figure 6-8 in your browser.

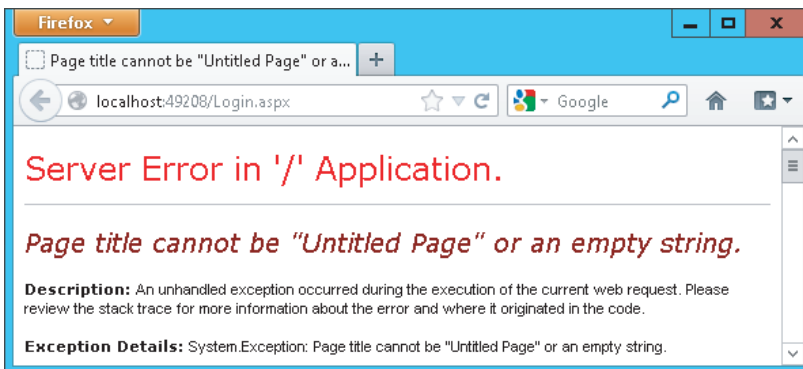


FIGURE 6-8

Instead of this error, you may see an error that displays the source for the `BasePage` class where the title is checked.

5. Go back to VS and open the login page in Markup View. Locate the `Title` attribute in the `@ Page` directive (or add one if it isn't there) and set its value to `Log in to Planet Wrox`. The following snippet shows the VB.NET version of the `@ Page` directive but the `C#` version is almost identical:

```

<%@ Page Title="Log in to Planet Wrox" Language="VB"
    MasterPageFile="~/MasterPages/Frontend.master" AutoEventWireup="false"
    CodeFile="Login.aspx.vb" Inherits="Login" %>

```


6. Repeat steps 3 and 5 for all the pages in your site. To make this a bit quicker, you can use Find and Replace to quickly replace all the occurrences of `System.Web.UI.Page` with `BasePage`. Make sure you don't accidentally replace it in the `BasePage` file in the `App_Code` folder itself. To prevent this from happening, make sure you search only in Code Behind files, like this:
 - Open the Replace in Files dialog box (press `Ctrl+Shift+H` or select `Edit ⇨ Find and Replace ⇨ Replace in Files`).
 - In the Find What box, enter `System.Web.UI.Page`. In the Replace With text box, enter `BasePage`.
 - Expand the Find Options section and in the Look at These File Types text box, enter `*.aspx.vb` or `*.aspx.cs` depending on the language you use. This leaves the `BasePage` file, which has a single extension of `.vb` or `.cs`, alone.
 - Click `Replace All` and then click `Yes` to confirm the Replace operation.
7. Save the changes you made to any open page and then browse to `Login.aspx` again. If everything worked out as planned, the error should be gone and you now see the login page.

Remember, though, that all other pages in your site now throw an error when you try to access them. The fix is easy; just give them all a valid `Title`. For pages without a `Title` attribute in their page directive, you need to do this manually. For other pages, with an empty `Title=""` attribute, you can quickly do this by searching the site for `Title=""` and replacing it with something like `Title="Planet Wrox"`. (Don't forget to reset `Look at These File Types` back to `*.*`). For pages other than the demo pages you've created so far, you're better off giving each page a unique title, clearly describing the content it contains.

How It Works

By default, all pages in your website inherit from the `Page` class defined in the `System.Web.UI` namespace. This gives them the behavior required to make them act as web pages that can be requested by the browser and processed by the server. Because the inheritance model in .NET enables you to create a chain of classes that inherit from each other, you can easily insert your own base page class between a web page and the standard `Page` class. You do this by changing the `Inherits` statement (in VB) and the colon (in C#) to your new `BasePage`:

VB.NET

```
Partial Class Login
    Inherits BasePage
```

C#

```
public partial class Login : BasePage
```

Inside this new `BasePage` class you add an event handler that is called when the class fires its `PreRender` event. As you learned earlier, this event is raised quite late in the page's life cycle, when the entire page has been set up and is ready to be rendered to the client:

VB.NET

```
Private Sub Page_PreRender(sender As Object, e As EventArgs) Handles Me.PreRender
    ' Implementation here
```

```
End Sub
```

C#

```
private void Page_PreRender(object sender, EventArgs e)
{
    // Implementation here
}
```

Note that Visual Basic uses the `Handles` keyword to tell the compiler that the `Page_PreRender` method will be used to handle the event. In C#, you need to hook up this handler manually. A good place to do this is in the class's constructor:

```
public BasePage()
{
    this.PreRender += Page_PreRender;
}
```

This highlighted line of code serves the same purpose as the `Handles` keyword in VB.NET: it tells the compiler what method to run when the page raises its `PreRender` event.

Inside the event handler, the code checks the current page title. If the page title is still an empty string (the default for any new page you add to your web project) or `Untitled Page` it *throws an exception*.

VB.NET

```
If String.IsNullOrEmpty(Me.Title) OrElse Me.Title.Equals("Untitled Page",
    StringComparison.CurrentCultureIgnoreCase) Then
    Throw New Exception(
        "Page title cannot be " & "Untitled Page" & " or an empty string.")
End If
```

C#

```
if (string.IsNullOrEmpty(this.Title) || this.Title.Equals("Untitled Page",
    StringComparison.CurrentCultureIgnoreCase))
{
    throw new Exception(
        "Page title cannot be \"Untitled Page\" or an empty string.");
}
```

This code uses the handy `IsNullOrEmpty` method of the `String` class to check if a value is null (Nothing in VB) or an empty string. It also uses the `Equals` method to check if the page title is equal to `Untitled Page`. It uses `StringComparison.CurrentCultureIgnoreCase` to do a case-insensitive comparison, so `untitled page` or `Untitled Page` would both match.

Notice how the keywords `Me` (in VB.NET) and `this` (in C#) are used. These keywords are context-sensitive and always refer to the instance of the class where they are used. In this example, `Me` and `this` refer to the current instance of the `BasePage` class. This `BasePage` instance has a `Title` property (which it inherits from `Page`) that can be checked for unwanted values. If it still contains the default title (an empty string) or the text “Untitled Page,” the code raises (or throws) an exception. This immediately stops execution of the page so you as a page developer can fix the problem by providing a valid title before the page ends up in public. In Chapter 18 you learn more about exceptions and how to prevent and handle them.

To display a double quote (") in the error message, both languages use a different format. In Visual Basic, you need to double the quotes. In C#, you need to prefix the double quote with a backslash (\) to *escape* the double quote. In both cases, a double quote character ends up in the error message.

Because every new page you add to the site should now inherit from this new base page, you should create a page template that already has the correct code in its Code Behind and markup, making it easy to add the correct page to the site right from the start. This is discussed next.

Creating Reusable Page Templates

Visual Studio comes with a great tool to export templates for a number of different file types including ASPX pages, class files, and even CSS files. By creating a custom template, you define the code or markup that you need in every file once and then create new files based on this template, giving you a jump start with the file and minimizing the code you need to type. The next exercise shows you how to create your own templates.

TRY IT OUT Creating a Reusable Page Template

In this exercise you see how to create a template file for all new ASPX pages you add to your site. To avoid conflicts with existing pages in your current site, you create a new temporary page and use that for the template. Afterward, you can delete the temporary file.

1. Add a new Web Form to the root of the site and call it **Temporary.aspx**. Make sure it uses Code Behind, uses your programming language, and is based on the master page in the `MasterPages` folder.
2. Open the Code Behind of this new page (by pressing F7) and change the `Inherits` line (the colon in C#) so the page inherits from `BasePage` instead of from `System.Web.UI.Page`. Also rename the class from `Temporary` to `$relurlnamespace$_$safeitemname$`:

VB.NET

```
Partial Class $relurlnamespace$_$safeitemname$
    Inherits BasePage
    ...
End Class
```

C#

```
public partial class $relurlnamespace$_$safeitemname$ : BasePage
{
    ...
}
```

Make sure you don't remove any of the existing code, like the `using` statements or the `Page_Load` method in the C# version.

Don't worry about any compile errors you may get about unexpected characters like `$`. Once you start adding pages based on this template, `$relurlnamespace$_$safeitemname$` will be replaced by the name of the page you're adding.

3. Switch to Markup View, and change the `Inherits` attribute from `Temporary` to `$relurlnamespace$_$safeitemname$` as shown in this C# example:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/MasterPages/Frontend.master"
    AutoEventWireup="true" CodeFile="Temporary.aspx.cs"
    Inherits="$relurlnamespace$_$safeitemname$" %>
```

You must leave the `CodeFile` attribute alone; VS will change it to the right Code Behind file automatically whenever you add a new page to the site.

4. Optionally, add other code you want to add to your pages by default, like a comment block with a copyright notice.
5. Save all changes and then choose `File` ⇨ `Export Template`. In the dialog box that follows, select `Item Template` and choose your programming language from the drop-down list at the bottom of the screen, shown in Figure 6-9.

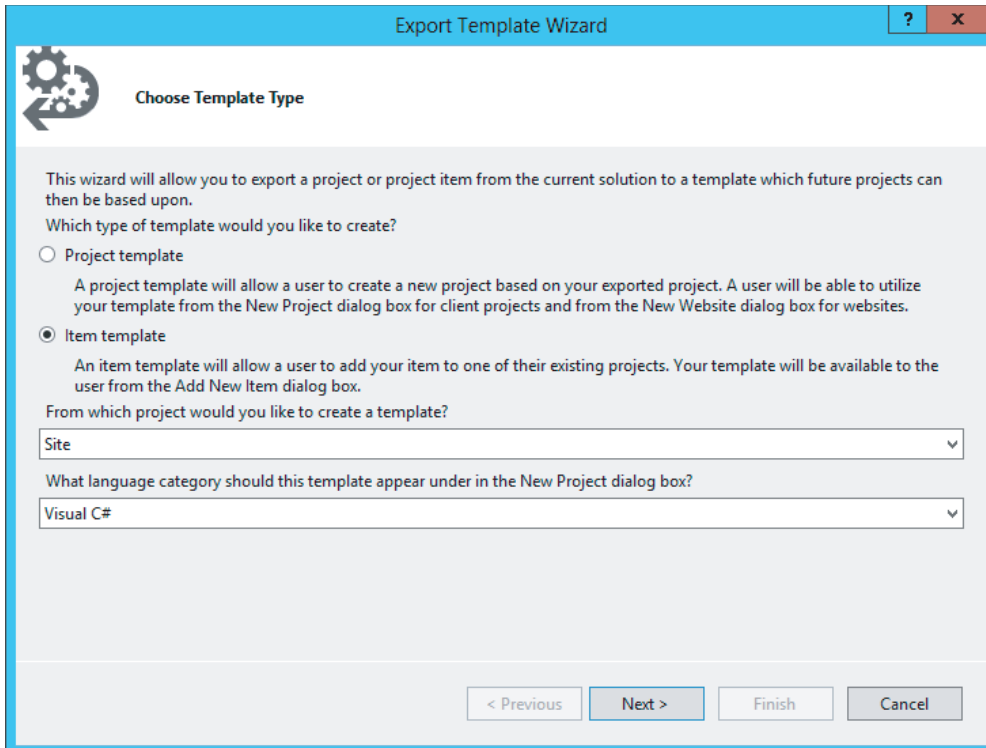


FIGURE 6-9

6. Click `Next` and place a check mark in front of `Temporary.aspx`, which you find near the bottom of the list. Click `Next` again to go to the `Select Item References` dialog box.
7. There is no need to set anything in the `Select Item References` dialog box now. If you had a web page referencing specific assemblies (`.dll` files) you could pick them here, so VS adds the references for you automatically next time you add a file based on this template. Click `Next` again to go to the `Select Template Options` screen. Type `MyBasePage` as the new template name, and optionally

type a short note describing the purpose of the template. Make sure the Automatically Import the Template into Visual Studio option is checked. Figure 6-10 shows the final dialog box.

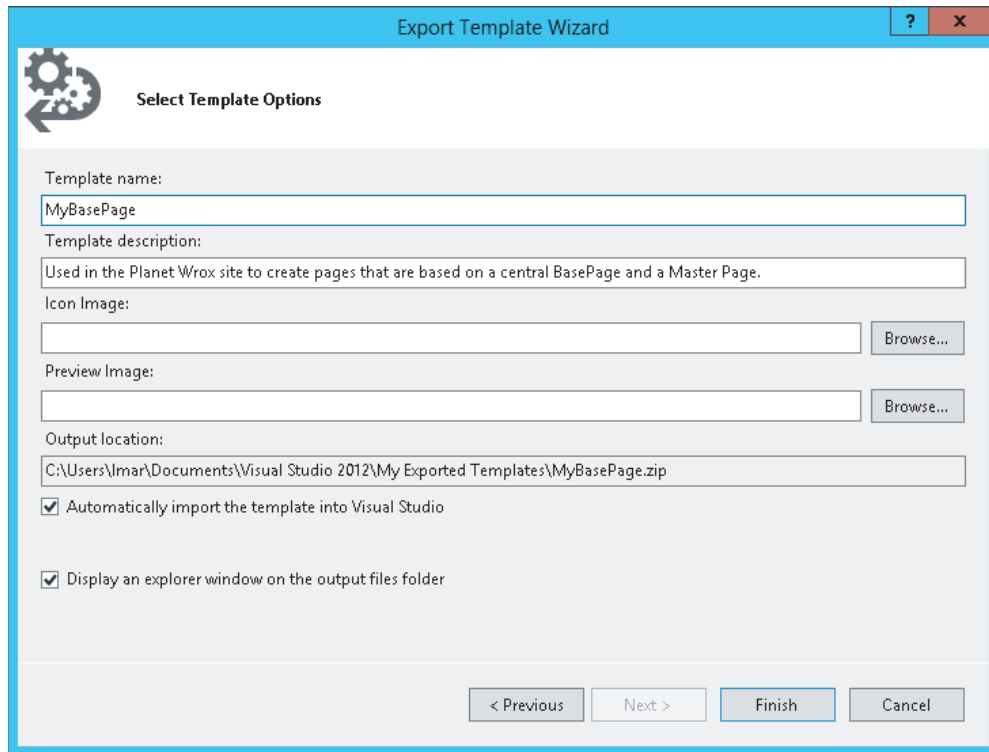


FIGURE 6-10

8. Click Finish to create the template. VS opens a File Explorer (Windows Explorer in Windows 7) showing a copy of the new template as a zip file. You can close that window, because you don't need it.

If you want to carry out this exercise for both VB.NET and C#, be sure to rename the resulting zip file first before you make an export for the second language; otherwise the zip file gets overwritten. To rename the file, open File Explorer, go to your Documents folder and then browse to Visual Studio 2012\Templates\ItemTemplates. You'll find a file called MyBasePage.zip, which you can rename to something like MyBasePageCS.zip. Note that the file's location is different from the one you see in Figure 6-10; the output location contains just a copy of the exported template that you can use as a backup.

9. Back in VS, delete the temporary file `Temporary.aspx` you created. Then right-click the website in the Solution Explorer and choose Add ⇄ Add New Item. Note that your custom template now shows up in the list of templates, shown in Figure 6-11. If you click it, VS shows you the description you gave it earlier. Note: you may have to restart VS and reopen your website for the template to appear.
10. Type a new name for the page, such as `TestPage.aspx`, and click Add to add it to your site. Look at the markup and the Code Behind of the file and verify that `$relurlnamespace$_$safeitemn`

ame\$ has been renamed to `_TestPage` to reflect the new name of the page. If everything looks OK, you can delete `TestPage.aspx` because it's not used in the Planet Wrox website.

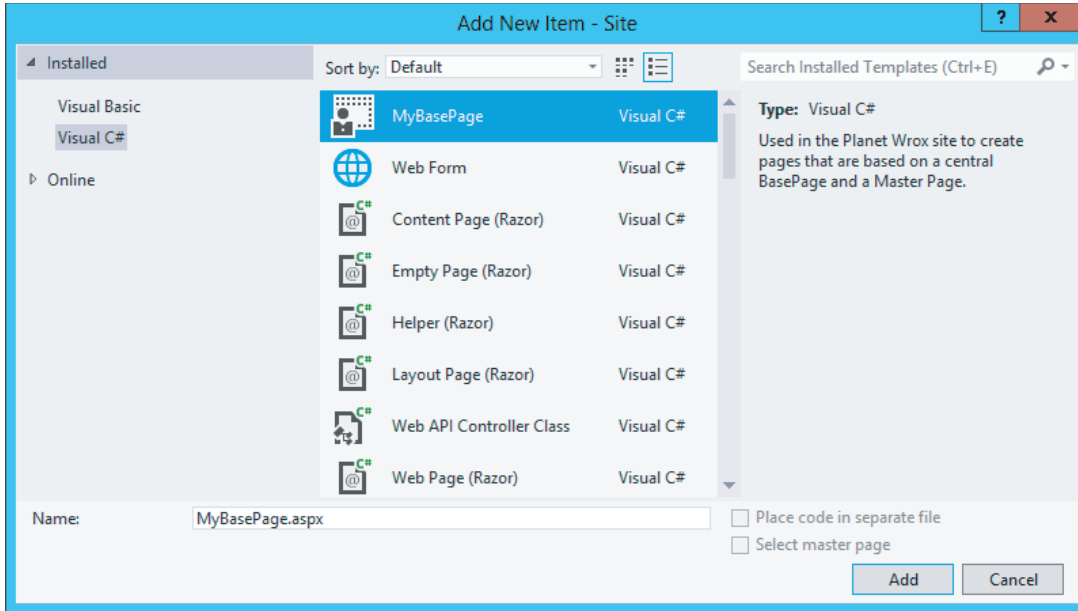


FIGURE 6-11

How It Works

When you export the template, Visual Studio creates a zip file with the necessary files—an ASPX file and a Code Behind file in this exercise. This zip file is then stored in the `ItemTemplates` subfolder of the `Visual Studio 2012` folder under your Documents folder. Some of the files in the zip file contain the placeholders `$relurlnamespace$` and `$safeitemname$`. When you add a new file to the site that is based on your template using the Add New Item dialog box, VS replaces `$relurlnamespace$` with the name of the folder (nothing, in the case of a file added to the root of the site) and `$safeitemname$` with the actual name of the page. In this exercise, you typed `TestPage.aspx` as the new name for the page, so you ended up with a class in the Code Behind called `_TestPage`, which in turn inherits from the global `BasePage`. The underscore (`_`) is hard-coded between the two placeholders and is really only needed when adding a Web Form based on this template to a subfolder. However, it's a valid start of a class identifier so you can safely leave it in for pages at the root of your website. If you add a file to a subfolder, such as the `Demos` folder, the class name is prefixed with the name of the folder so you end up with a class called `Demos_TestPage`. In addition to `$relurlnamespace$` and `$safeitemname$`, you can use a few other placeholders. Search the MSDN site at <http://msdn.microsoft.com> for the term `$safeitemname$` to find the other template parameters.

If you need to make a change to the exported template, either redo the entire export process, or manually edit the files in the zip file.

With this exported template you now have a very quick way to add pages to your site that inherit from the `BasePage` class. You don't need to manually change the Code Behind of the class file or the markup of the page anymore.

In addition to master pages and the central `BasePage` class, you have more options to create consistent-looking websites. One of them is themes.

THEMES

So far you've seen how to create a master page to define the global look and feel of the pages in your site. You also saw how to centralize the behavior of your pages by using a central base page. However, you have more ways to influence the look and feel of your site: themes and skins. Skins are dealt with later in the chapter because they are an optional part of themes, which need to be discussed first.

A *theme* is a collection of files that defines the look of a page. A theme typically includes skin files, CSS files, and images. You define themes in the special `App_Themes` folder in the root of your website. Within this folder you create one or more subfolders that define the actual themes. Inside each subfolder, you can have a number of files that make up the theme. Figure 6-12 shows the Solution Explorer for a website that defines two themes: Monochrome and DarkGrey.

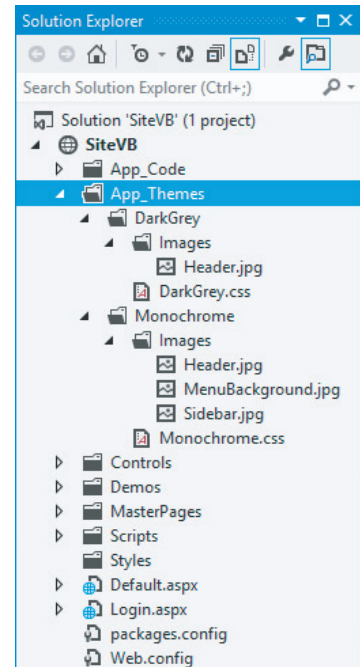


FIGURE 6-12

A link to each CSS file in the theme folder is added to your page's `<head>` section automatically whenever the theme is active. You see how this works later. The images in the theme folder can be referenced from the CSS files. You can use them to change common elements of the website, such as background images, or images used in bulleted lists or navigation lists.

To create a theme, you need to do the following:

- Create the special `App_Themes` folder if it isn't already present in your site.
- For each theme you want to create, create a subfolder with the theme's name, like `Monochrome` or `DarkGrey` in Figure 6-12.
- Optionally, create one or more CSS files that will be part of the theme. Although naming the CSS files after the theme helps in identifying the right files, this is not a requirement. Any CSS file you add to the theme's folder is added to the page at run time automatically.
- Optionally, add one or more images to the theme folder. The CSS files should refer to these images with a relative path as explained later.