Pro ASP.NET 2.0 Website Programming

Damon Armstrong

Apress®

Pro ASP.NET 2.0 Website Programming

Copyright © 2005 by Damon Armstrong

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-546-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Tony Davis
Technical Reviewer: Damien Foggon
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Denise Santoro Lincoln
Copy Edit Manager: Nicole LeClerc
Copy Editor: Julie McNamee
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Composition, proofreading, and indexing: Argosy Publishing
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section. You will need to answer questions pertaining to this book to successfully download the code.

CHAPTER 3

Master Pages, Themes, and Control Skins

Appearance matters. Whether you like it or not, people make decisions about how well they like your application and how well they think it works based on its visual appearance. Web applications exhibiting a consistent look and feel exude professionalism and instill confidence in users that the application is well built and reliable. People naturally conclude that the attention to the visual design is on par with that of the functional capabilities of the application. First glances aside, consistency also enhances the user experience because people feel most comfortable when they are in a familiar environment. By creating a consistent location for certain page elements, users will inherently know how to get around in your application even when accessing pages to which they are not normally accustomed. My favorite news site, for example, always lists the news categories and stories in a menu structure on the left hand side of every page. No matter where I am on their site, I have the comfort of knowing that I can navigate to a different news category or story using that menu.

Aside from the CSS (Cascading Style Sheet) support inherent in HTML, ASP.NET 1.x did not have any features to help you maintain a consistent look and feel throughout your application. ASP.NET 2.0 remedies that shortcoming with the introduction of Master Pages, themes, and control skins. Master Pages, a much needed and highly anticipated feature, allow you to define and maintain page layouts and common page content from a single file. Themes enable you to apply different CSS and control skins to your application. Control skins allow you to apply properties to specific ASP.NET controls throughout the entire application.

This chapter briefly covers each technology and how to use it in your applications. Here is an outline of what you'll find inside:

- *Master Page:* Demonstrates how you can use Master Pages to create page templates to control the look, feel, and layout of pages in your application.
- *Themes and Control Skins:* Discusses how to create and apply different visual styles to your application using CSS and control skins.

Master Pages are, by far, the most popular of these new additions to ASP.NET 2.0, so we'll begin by taking a look at them and how they can really simplify development.

Master Pages

One of my current projects truly exemplifies the need for Master Pages. We are building an article-management system for an online publisher whose existing site is managed entirely in static HTML. All the pages on the static site look very professional because the layouts, fonts, colors, and other visual styles are consistent across all the pages. The problem is that they achieved that consistency by copying the base HTML layout into each and every page. As a result, any changes to the site layout have to be made on hundreds of individual pages to maintain consistency.

Master Pages allow you to control the look, feel, and behavior of multiple pages in your application from a single location. In a Master Page, you define the basic page layout using HTML and ASP.NET controls and the behavior using server-side code. Building a Master Page is, for the most part, just like working with any standard ASP.NET web form. The most notable difference is that you can add `ContentPlaceHolder` controls to a Master Page. A `Content➥ PlaceHolder` control defines a region where you can inject content into the Master Page. All other content on the Master Page is locked, so the `ContentPlaceHolder` controls represent the only locations where content differs from page to page. Pages that employ Master Pages define `Content` controls that contain the page content. ASP.NET processes requests for these pages by injecting the `Content` controls on the page into the appropriate `ContentPlaceHolder` controls on the Master Page at runtime. Figure 3-1 depicts the process.

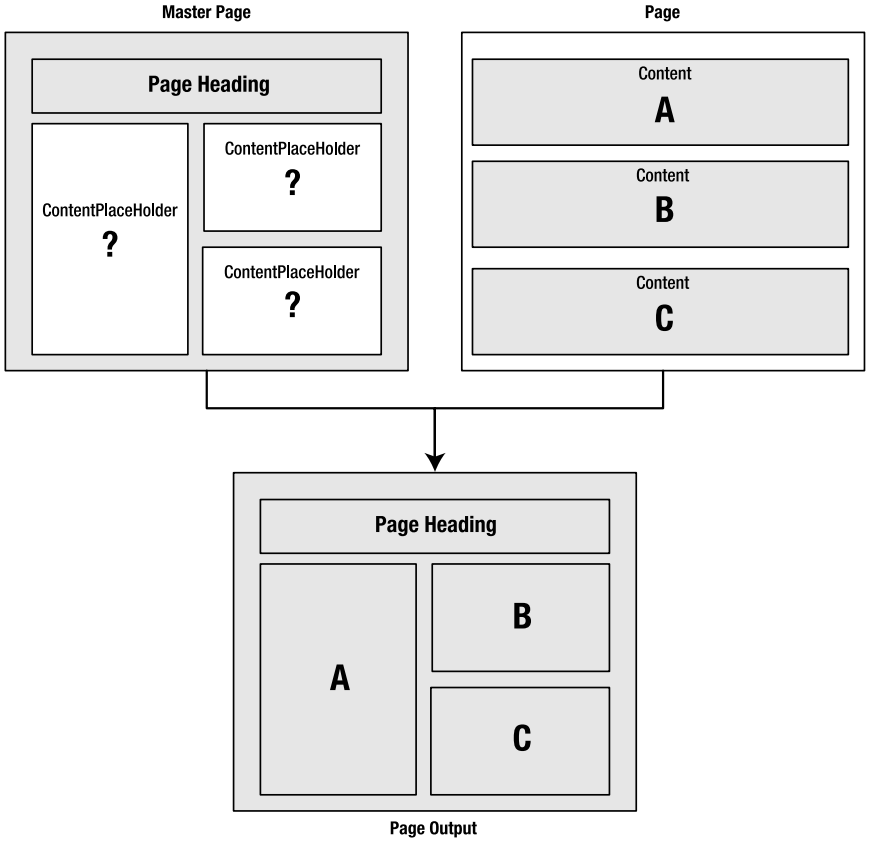

Figure 3-1. *Injecting page content into a Master Page*

Creating a Master Page

You add new Master Pages to your application by right-clicking on the web project in the Solution Explorer and selecting the **Add New Item** option from the context menu. Alternatively, you can select **Website ➤ Add New Item** in the Visual Studio IDE. Either way, Visual Studio displays the **Add New Item** dialog box (Figure 3-2), which allows you to select the type of item you want to add to your project. Select the **Master Page** item from the list and enter a name for the new Master Page in the **Name** text box. Like a normal ASP.NET page, you can put the server-side code for the Master Page directly in the markup file or in a code-behind file. If you want to use a code-behind file, make sure you check the **Place code in separate file** option. Click the **Add** button and Visual Studio adds the appropriate files to your web application.

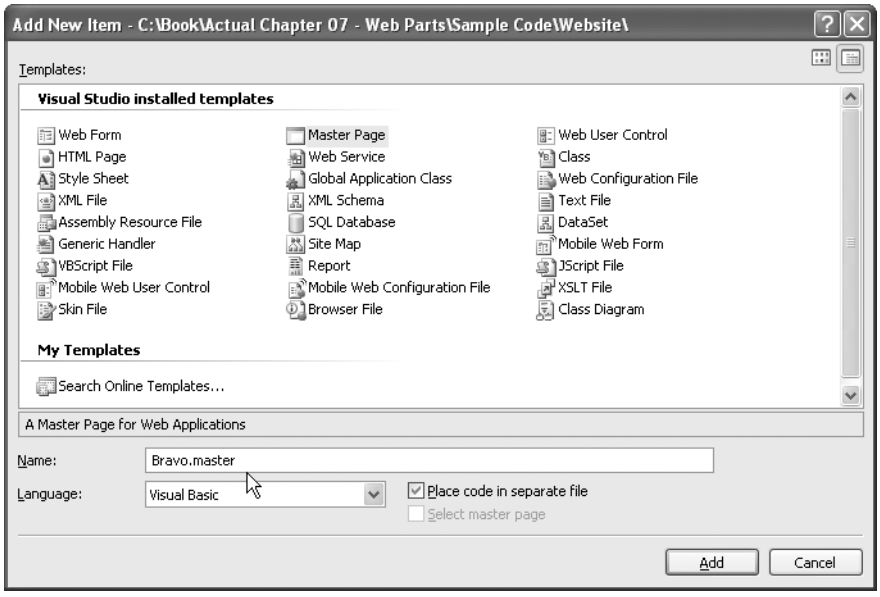

Figure 3-2. *Visual Studio's Add New Item dialog box*

After Visual Studio adds the Master Page to your application, you can interact with it in the editor as though it was a normal page. You can add HTML, styles, JavaScript, and web controls to the page and even create server-side code to respond to page and control events. As you design the Master Page, you'll run across sections that should contain page-specific content. When you do, drop in a `ContentPlaceHolder` control so you can inject page content into the Master Page at that location.

As an example, let's say you are creating an intranet application for the Bravo Corporation, a fictitious organization that makes things and then sells them to people. All the pages in the application need to have the same header and layout, but the content for each page is different. A Quick Links section in the subheader displays important links that are relevant to the current page. In other words, the page content and the quick links change from page to page, but the heading and overall page layout should be consistent across all pages. Figure 3-3 shows an example of the page layout.

Figure 3-3. *Bravo Corp page layout*

Following is the markup required to make a Master Page for the layout shown in Figure 3-3. All the code specific to the Master Page content is shown in bold, and the various sections of the page are delineated by HTML comments (`<!-- Comment -->`). Also a couple of CSS styles are used in the Listing 3-1 HTML that can be found in the `/App_Themes/Default/Default.css` file in the sample application.

Listing 3-1. *Master Page Example*

```
<%@ Master Language="VB" CodeFile="Bravo.master.vb" Inherits="Bravo" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="PageHeading" runat="server">
    <title>Bravo Company</title>
</head>
<body topmargin=0 bottommargin=0 leftmargin=0 rightmargin=0>
  <form id="formMain" runat="server">
    <table style="width:100%;" border=0 cellpadding=0 cellspacing=0>

      <!--Header Section -->
      <tr>
        <td class="PageHeading">Bravo Corp Employee Website</td>
      </tr>
```

```
      <!--Sub-Header Section -->
      <tr>
        <td class="PageSubHeading">
          <table style="width:100%;" cellpadding=0 cellspacing=0>
            <tr>
              <td align=left class="TagLine">
                We make things and then sell them to people!
              </td>
              <td align=right class="TagLine">
                <asp:contentplaceholder id="QuickLinks" runat="server">
                </asp:contentplaceholder>
              </td>
            </tr>
          </table>
        </td>
      </tr>

      <!-- Main Body Section -->
      <tr>
        <td>
          <table cellpadding=5>
            <tr>
              <td>
                <asp:contentplaceholder id="MainContent" runat="server">
                </asp:contentplaceholder>
              </td>
            </tr>
          </table>
        </td>
      </tr>
    </table>
  </form>
</body>
</html>
```

At the top of the Master Page, the `<%@ Master %>` directive tells ASP.NET how to handle the page. Visual Studio generates this section for you automatically. Know, however, that it is almost identical to the `<%@ Page %>` directive.

Inside the page markup, you'll find two `ContentPlaceHolder` controls: one in the subheader section and one in the main body section. The one in the subheader is named `QuickLinks` and allows you to inject links into the subheader area; the second one in the main body section is named `MainContent` and allows you to create the body of the page. Next, you'll see how to create a content page that uses a Master Page.

Creating Content Pages

Any page that uses a Master Page is known as a *content page*. You add new content pages to your application using the **Add New Item** dialog box as described earlier with Master Pages. But, instead of selecting Master Page from the list of items, you select the **Web Form** item (see Figure 3-4). After you select Web Form, follow these steps to create a content page:

1. Specify the page name in the **Name** text box.
2. Check the **Place code in a separate file option** if you want to use a code-behind file.
3. Make sure you check the **Select Master Page** option. This tells the Add New Item dialog box that the new page is a content page that requires the uses a Master Page.
4. Click the **Add** button.

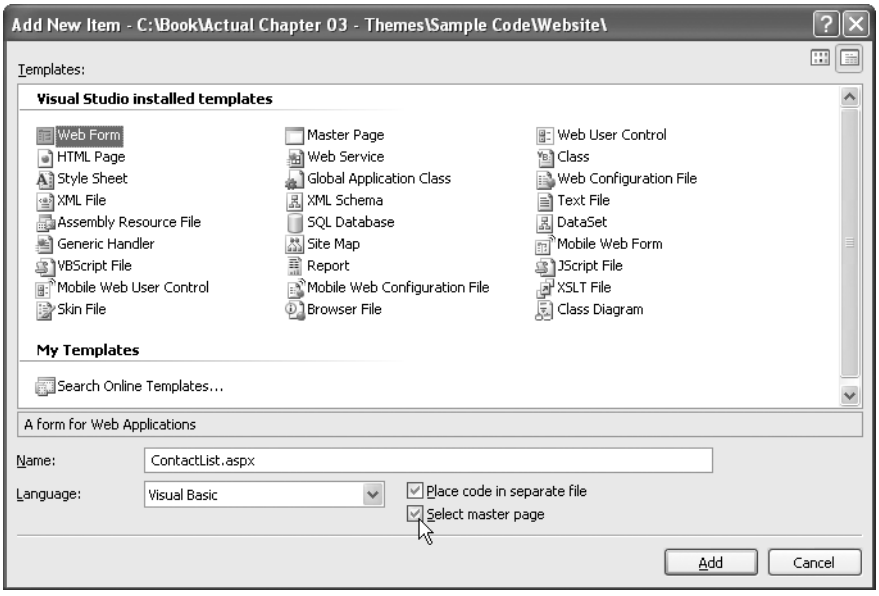

Figure 3-4. *Add New Item dialog box showing the Select master page check box*

After you click on the **Add** button, Visual Studio displays the **Select a Master Page** dialog box as shown in Figure 3-5. This dialog box displays your application folder structure and allows you to select which Master Page you want to use for the new page. As you select directories on the left side of the dialog box, Master Pages in that folder appear on the right-hand side. When you locate the one you want to use, click to select it and then click the **OK** button.

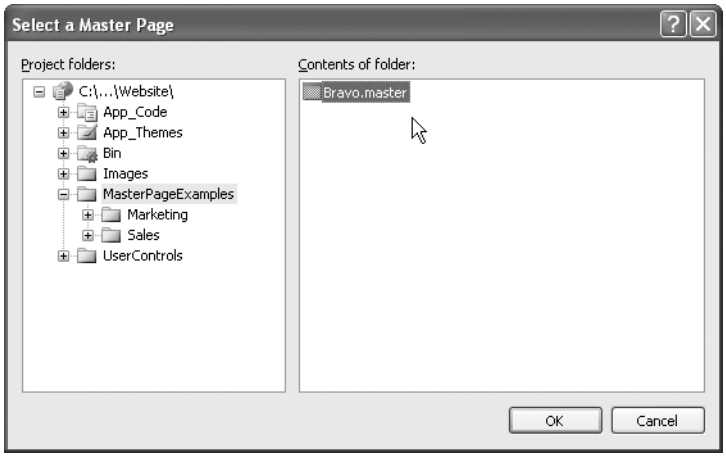

Figure 3-5. *The Select a Master Page dialog box*

After you click the **OK** button, Visual Studio creates a new content page with the appropriate `<@% Page %>` directive parameters that define which Master Page the content page uses. It also automatically creates `Content` controls that match up with the `ContentPlaceHolders` in the Master Page. `Content` controls have a property named `ContentPlaceHolderID`, which identifies the `Content`➥ `PlaceHolder` control on the Master Page where the content should be injected. Listing 3-2 shows the content page generated by Visual Studio for use with the Master Page shown earlier.

Listing 3-2. *Content Page*

```
<%@ Page Language="VB" MasterPageFile="~/MasterPageExamples/Bravo.master"
    AutoEventWireup="false" CodeFile="ContactList.aspx.vb" Inherits="ContactList"
    title="Untitled Page" %>

<asp:Content ID="Content1" ContentPlaceHolderID="QuickLinks" Runat="Server">
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
```

Notice the `MasterPageFile` property in the `<@% Page %>` directive at the top of the code listing. This property points to the Master Page file that the page should use in conjunction with its content. Also note that the page does not contain any standard `<HTML>` or `<BODY>` tags normally found in a page. Instead, it has two `Content` controls. You cannot specify any HTML or web controls for a content page outside of a `Content` control, although you do have free reign inside the `Content` control. Finally, observe that the `ContentPlaceHolderID` values in the `Content` controls match up with the `ID` values for the `ContentPlaceHolder` controls from the Master Page. If you accidentally specify a nonexistent `ContentPlaceHolderID` value, then you should receive an error in the task list. Your project will still build successfully, but if you run the page, it will throw an exception.

Returning to the example, let's say that you wanted to create a company contact list using the Master Page discussed earlier. The main content should include the names and extensions of the people in the company. The Quick Links section should contain a link to the Phone System Help page and a link to an Update My Contact Info page. The content page markup would look like Listing 3-3.

Listing 3-3. *ContactList.aspx Example Content Page*

```
<%@ Page Language="VB" MasterPageFile="~/MasterPageExamples/Bravo.master"
    AutoEventWireup="false" CodeFile="ContactList.aspx.vb" Inherits="ContactList"
    title="Company Contact List" %>

<asp:Content ID="Content1" ContentPlaceHolderID="QuickLinks" Runat="Server">
    <a href="">Phone System Help</a> |
    <a href="">Update My Contact Info</a>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
    <div style="width:400px;">
        Welcome to the employee directory page.  You can find a list of
        phone numbers in the table below.  Have fun calling people!<br /><br />
    </div>
    <table cellpadding=5 style="border:1px solid black;">
        <tr style="font-weight:bold; color:White; background-color:DarkBlue;">
            <td>Name</td><td>Extension</td>
        </tr>
        <tr><td>Anderson, Ty</td><td>x 5891</td></tr>
        <tr><td>Armstrong, Teresa</td><td>x 1212</td></tr>
        <tr><td>Haynes, Tim</td><td>x 2911</td></tr>
        <-- etc -->
    </table>
</asp:Content>
```

As you can see, only content is specified in the `ContactList.aspx` content page. No layout, no header, and no subheading are defined in the content page. When ASP.NET renders the page, it injects the content into the Master Page, and the final result looks like Figure 3-6.

Figure 3-6. *ContactList.aspx displayed in the browser*

Accessing Master Pages from Content Pages

Many scenarios occur in which a content page may need access to controls or functionality in the Master Page. You can do this via the `Master` property of the `Page` object. Every ASP.NET page stores a reference to its Master Page in the `Master` property. If the page does not use a Master Page, then the reference is set to `Nothing`. By default, the `Master` property is a `MasterPage` type. This allows you to access any method or property normally found in the `MasterPage` base class from which all Master Pages ultimately inherit.

One of the most useful methods on the `Master` property is the `FindControl` method, which reaches up into the Master Page to look for controls. For example, if you have a `Label` control defined on your Master Page named `lblTitle`, then you can acquire a reference to that `Label` control by using `Master.FindControl("lblTitle")` and then casting the control returned to a `Label` control. This allows you to set controls in the Master Page based on logic from the content page.

Another useful tactic for interacting with the Master Page is to strongly type the `Master` property using the `<@% MasterType %>` directive. To do this, just add the `<@MasterType>` directive below the `<%@ Page %>` directive and point the `VirtualPath` at the Master Page file:

```
<%@ Page Language="VB" MasterPageFile="~/MasterPageExamples/Bravo.master"
    AutoEventWireup="false" CodeFile="ContactList.aspx.vb" Inherits="ContactList"
    title="Company Contact List" %>
<%@ MasterType VirtualPath="~/MasterPageExamples/Bravo.master" %>
```

Strongly typing the `Master` property gives you direct access to public methods and properties defined in that Master Page without having to cast it from a `MasterPage` into the target type. For example, let's say that you have a Master Page with a menu on the left-hand side. All pages use the exact same menu, so you define it directly in the Master Page. Some pages, however, do not use the menu at all, so you want to hide it on those pages. As such, you create a public function named `HideMenu` on the Master Page. The method simply sets the `visible` property on the menu

section to `false` to hide it from view. If you do NOT strongly type the `Master` property, then you end up having to cast the `Master` property before you can use the public method you defined:

```
DirectCast(Master, ASP.Bravo_Master).HideMenu()
```

■ **Note** You can access Master Page and user control resources for the page via the ASP namespace.

When you strongly type the `Master` property, you can simply call the method without resorting to any casting:

```
Master.HideMenu()
```

This makes working with and reading the code a bit easier. As an added benefit, strongly typing the `Master` property gives you full IntelliSense support for the methods and properties specified in the Master Page.

Defining a Default Master Page for Your Application

All content pages usually specify which Master Page they use via the `MasterPageFile` property in the `<%@ Page %>` directive. If you leave it off, ASP.NET throws an error when you access the page and informs you that `Content` controls can only be used in conjunction with a Master Page. There is one exception to that rule. You can define a default Master Page in `web.config` in the `<pages>` element as shown in Listing 3-4.

Listing 3-4. *Default Master Page Defined in* `Web.config`

```
<configuration>
  ...
  <system.web>
    ...
    <pages theme="Default" masterPageFile="~/MasterPageExamples/Bravo.master" />
    ...
  </system.web>
</configuration>
```

When you specify a default Master Page in `Web.config`, all the content pages without a `masterPageFile` value defined in the `<%@ Page %>` directive default to the Master Page in `Web.config`. You can use this feature to help manage which Master Pages your application uses from a single location.

■ **Note** ASP.NET only applies the default Master Page to content pages that do not have a Master Page explicitly defined in the `<%@ Page %>` directive. ASP.NET is also smart enough to check that the page is a content page (that is, it contains `Content` controls) before applying the default Master Page setting. You can still have normal pages in your application that do not use Master Pages even if you set a default Master Page.

Changing Master Pages in Code

You can programmatically change the Master Page in code using the `MasterPageFile` property on the `Page` object; however, there are a few limitations when using this approach. First, you can only set the `MasterPageFile` property during the `PreInit` event. Attempting to change the Master Page past this point results in a runtime exception. Second, the Master Page that you specify in code must have `ContentPlaceHolder` controls whose `ID` properties match up with the `ContentPlaceHolderID` properties for the `Content` controls defined in the content page. If they don't, an exception is thrown.

Remember, this is how a content page determines where to inject content into the Master Page. Listing 3-5 demonstrates how to set the Master Page in code.

Listing 3-5. *Setting the Master Page in Code*

```
'************************************************************************
Protected Sub Page_PreInit(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles Me.PreInit
    Page.MasterPageFile = "Delta.master"
End Sub
```

Nested Master Pages

ASP.NET allows you to create nested Master Pages. In other words, you can specify a Master Page for a Master Page. One scenario in which this may be useful is creating a corporate intranet for a company with multiple departments or divisions. You can control the overall look and feel for the company using one Master Page, and you can control the overall look and feel for a department using a nested Master Page.

Listing 3-6 is the markup for the `BravoSales.master` file, an example nested Master Page that defines the look and feel for pages in the Bravo Corporation's Sales Department. It uses the `Bravo.master` file for the overall corporate look and feel. Figure 3-7 demonstrates how nested Master Pages allow you to inherit the look and feel of one Master Page into another.

Listing 3-6. `BravoSales.master` *Example*

```
<%@ Master MasterPageFile="~/MasterPageExamples/Bravo.master" Language="VB"
    CodeFile="BravoSales.master.vb" Inherits="BravoSales" %>
<%@ MasterType VirtualPath="~/MasterPageExamples/Bravo.master" %>

<asp:Content ID="Content1" ContentPlaceHolderID="QuickLinks" Runat="Server">
    <a href="SalesHome.aspx">Sales Dept. Home</a>
    <asp:ContentPlaceHolder runat=server ID="SalesQuickLinks" />
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
    <span style="font-size:20pt;">The Sales Department!</span><hr />
    <asp:ContentPlaceHolder runat=server ID="SalesMainContent" />
</asp:Content>
```

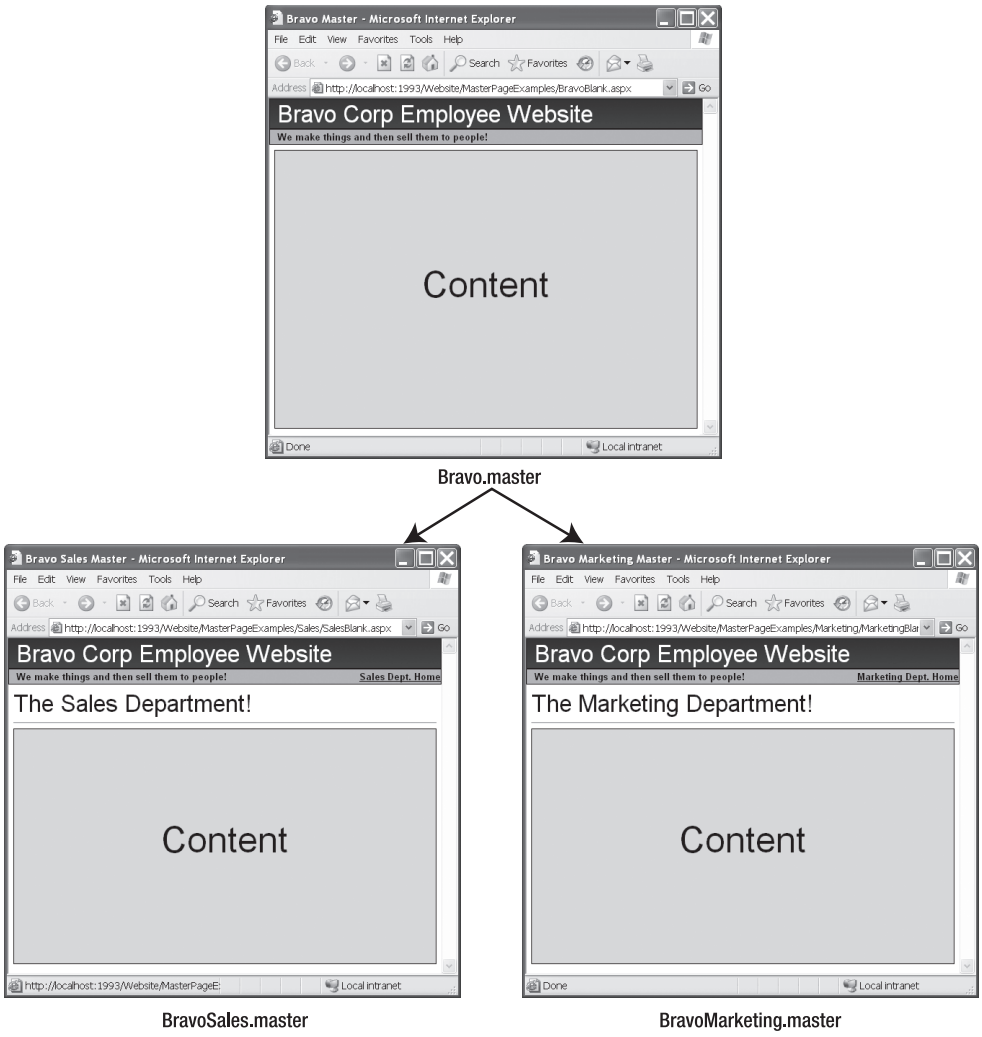

Figure 3-7. *Nested Master Pages*

Creating a nested Master Page is virtually identical to creating a content page, but instead of using a `<%@ Page %>` directive, you use a `<%@ Master %>` directive. You still specify the Master Page using the `MasterPageFile` property, you can still strongly type the Mater Page using the `<%@ MasterType %>` directive, and the `Content` controls still have to specify the appropriate `ContentPlaceHolderID` so ASP.NET knows where to inject the content into the Master Page. The main difference is that you can define any number of new `ContentPlaceHolders` in the nested Master Page, which you can then use in subsequent content pages.

■ **Caution** Visual Studio.NET 2005 (Beta 2) does not support visual editing of content pages that uses nested Master Pages. If you strongly prefer the graphical editor, then you may want to avoid using nested Master Pages.

Themes and Control Skins

Themes allow you to control the look and feel of your entire web application using CSS and control skins. If you have worked with web applications in the past, then you know that CSS styles control the way HTML elements appear in the browser by defining visual and behavioral properties. Control skins are, in many respects, style sheets for ASP.NET controls. Instead of hard-coding visual information into each ASP.NET control in your application, now you can create a control skin that defines how you want that control to appear throughout your entire application. When that type of control is rendered by the ASP.NET runtime, the control skin dictates property settings on that control that govern its visual appearance.

Creating a Theme

To add a theme to your web application, right-click on your web project and select **Theme Folder** from the **Add Folder** context menu. This adds an `App_Themes` folder to your application as well as a subfolder called `Theme1`, although you should change this name to something more meaningful to your application.

Each theme in your application requires a new subfolder under the `App_Themes` folder, and you can add a new theme by selecting **Theme Folder** from the **Add Folder** context menu. If you want a theme called Bluish Haze, then you would add a new subfolder named "Bluish Haze" to the `Themes` folder. If you wanted a theme called GreenTheme, then you would name the folder "GreenTheme." You get the idea. After you create the folder, you then need to create any control skin files for your theme and, optionally, the theme's CSS. Figure 3-8 shows an example of a Themes file structure with CSS and control skin files.

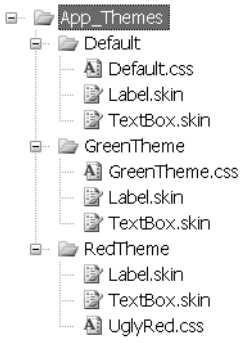

Figure 3-8. `Themes` *folder showing control skins and CSS.Note that the* `.css` *file names match the folder names.*

Adding a Cascading Style Sheet to Your Theme

After you have created your theme's folder, you can create a CSS for your theme. To do so, right-click on your theme's folder and select **Add New Item** from the context menu. Select the **Style Sheet** template from the template list and name your style sheet using the same name as your theme. If you are creating a style sheet for the GreenTheme, then name your style sheet **GreenTheme.css**. You can then edit the style sheet and add any necessary style definitions and classes that you may need.

■ **Note** You are not required to create a CSS for your theme. In fact, you can have a completely blank theme that does not have a style sheet or any control skins if you want your page to render without any CSS styles or ASP.NET properties to be applied to the page.

When you specify a theme for a page, the page is required to have a `<head RunAt=server>` tag. ASP.NET automatically places a `<link>` tag in the head section that points to the CSS for your theme, assuming that you have one specified. If the page does not have a `<head➥ RunAt=server>` tag, then an exception is thrown.

Creating Control Skins for Your Theme

As mentioned before, control skins are like style sheets for ASP.NET. You define control skins inside `.skin` files that reside in the various themes folders in your application. You can lump all your control skins into a single file, or you can create a series of `.skin` files to hold different control skins. It is mostly a matter of organizational preference. If you decide to use multiple `.skin` files, then the naming convention is normally

```
[ControlName].skin
```

Thus, the control skin file for `Textbox` controls should be named `TextBox.skin`, and the skin file for `Label` controls should be called `Label.skin`. The sample application uses the individual file-naming method.

To create a skin file for a `TextBox` control, follow these steps:

1. Right-click on your theme's folder, and then select the **Add New Item** button from the context menu.
2. When the **Add New Item** dialog box appears, select **Skin File** from the list of templates, and name it **TextBox.skin**.
3. Click the **Add** button. The `TextBox.skin` file appears in the Solution Explorer and opens in the Visual Studio IDE.

Creating the content for a skin file is as easy as defining a control on a web form. Following is an example of a `TextBox` control skin:

```
<asp:TextBox Runat="Server" BackColor="Gainsboro" Font-Bold="True"
             BorderStyle="Solid" BorderColor="Green" BorderWidth="1px"
             ForeColor="DarkGreen" Font-Italic="False" />
```

The control skin content is similar to any `TextBox` definition you would see on a web form, but notice there is no `ID` or `Text` property specified. You don't specify the `Text` property because the `Text` value changes from one control the next so you do not want to specify a global value. If you specify an `ID` property in the `.skin` file, the application will not compile. When ASP.NET encounters a server control while rendering a page with themes, it determines whether there is a control skin for that control type in the appropriate theme folder. If one is

found, it reads the skin information and automatically maps any of the properties specified in the skin file onto the control it is rendering. So, if you have a `TextBox` control defined in your application that looks like this:

```
<asp:Textbox Runat="Server" ID="MyTextbox" Text="My Text Value" />
```

It will be rendered as though it was defined like this:

```
<asp:TextBox Runat="Server" ID="MyTextbox" Text="My Text Value"
             BackColor="Gainsboro" Font-Bold="True" BorderStyle="Solid"
             BorderColor="Green" BorderWidth="1px" ForeColor="DarkGreen"
             Font-Italic="False" />
```

Notice that the only properties from the actual definition are the `ID` and the `Text` properties, all the other properties come from the skin file.

■ **Caution** If you have a property defined in both a control and in the skin file, the skin file property overwrites the property on the control. This is in stark contrast to the way CSS files work, so you need to be conscious of it, especially if you are used to working with CSS files.

Disabling Control Skins

Whenever ASP.NET encounters a control, it tries to apply the applicable control skin. Sometimes, however, you'll have a control you do not want the skin to apply to. In these instances, you just have to set the `EnableTheming` property to `False` on the control. When ASP.NET encounters a control with the `EnableTheming` property set to `False`, it does not attempt to apply any skin to the control. For example, ASP.NET does not attempt to skin the following control:

```
<asp:TextBox Runat="Server" ID="MyNonSkinnedTextBox" EnableThemeing="False"/>
```

Creating Named Control Skins for Your Theme

At times, you'll also want to apply a skin to a control, but you want it to be different than the skin you already have defined. That's when named control skins come into play. Named control skins are defined in the same file as control skins, so you can have one control skin and multiple named control skins in a single `.skin` file. Following is an example of a `TextBox` skin file that contains both a control skin and a named control skin:

```
<asp:TextBox Runat="Server" BackColor="Gainsboro" Font-Bold="True"
   BorderStyle="Solid" BorderColor="Green" BorderWidth="1px"
   ForeColor="DarkGreen" Font-Italic="False" />
```

```
<asp:TextBox Runat="Server" SkinID="MultiLine" Height="75px" Width="300px"
   BackColor="Gainsboro" Font-Bold="True" BorderStyle="Solid"
   BorderColor="Green" BorderWidth="1px" ForeColor="DarkGreen"
   Font-Italic="False" />
```

The first section of the code contains the control skin for a `TextBox`. The second section contains a named control skin. You can tell the second item is a named control skin because it contains a `SkinID` property to "name" it. Also notice that it specifies a height and width to properly display multiline text boxes (hence the name `MultiLine`). If you have a multiline text box that needs to use this skin, it would be defined as follows:

```
<asp:TextBox Runat="Server" ID="MyMultiLineTextBox" SkinID="MultiLine"
             TextMode="MultiLine" />
```

When ASP.NET encounters this control, it looks for a `TextBox` control skin specifically named `MultiLine` in the appropriate theme folder. If it cannot find the `MultiLine` named skin in the folder, then no skin is applied. Not even the unnamed control skin for the control type.

Applying Themes to Specific Pages or the Entire Application

You can apply a specific theme to a page using the `Theme` parameter of the `Page` directive. In the following example, the `GreenTheme` theme is applied to the page:

```
<%@ Page Language="VB" CompileWith="ThemeDemo.aspx.vb" ClassName="ThemeDemo_aspx"
title="Theme Demo" EnableViewState="False" Theme="GreenTheme" %>
```

If you want to apply a theme to every page in your entire application, then you can use `Web.config` to specify a theme. This is accomplished via the `theme` property of the `<pages>` element:

```
<configuration>
    ...
    <system.web>
        ...
        <pages theme="GreenTheme" />
        ...
    </system.web>
</configuration>
```

On the off chance that one of the pages in your application needs a different theme than the one specified in the `Web.config` file, you can override `Web.config` by specifying a theme in the `Page` directive, as described earlier. It has precedence over `Web.config`.

Programming with Themes

Another option for setting the theme for a page is to do it programmatically. The `Page` class exposes a property called `Theme` you can use to set the page's theme in code. Like the `MasterPageFile` property, you can only set the `Theme` property during the `PreInit` event. If you attempt to change it after the `PreInit` event, an exception is thrown.

Following is an example of how to programmatically set a page's theme. It assumes the appropriate theme name is passed in as a parameter in the query string.

```
Private Sub Page_PreInit(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles Me.PreInit
        Page.Theme = Request.QueryString("Theme")
End Sub
```

Summary

Using a consistent look and feel throughout your application helps build the perception that it is well built, well thought out, and reliable. Before ASP.NET 2.0, however, maintaining a consistent look and feel was difficult because there was no formal structure for making global layout, behavior, and visual changes aside from the inherent support for CSS in HTML. The introduction of Master Pages, themes, and control skins gives you a distinct advantage in terms of global site maintenance. As you use them on projects, you'll find they are major time-savers.