Table of Contents

Chapter 6. Rich Page Composition

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working with Master Pages</td>
<td>2</td>
</tr>
<tr>
<td>Working with Themes</td>
<td>19</td>
</tr>
<tr>
<td>Working with Wizards</td>
<td>31</td>
</tr>
<tr>
<td>Conclusion</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 6

Rich Page Composition

In this chapter:

Working with Master Pages ........................................... 248
Working with Themes ................................................ 265
Working with Wizards ................................................ 277
Conclusion .......................................................... 290

A large number of Web sites these days contain similar-looking, rich pages that share the same graphics, appearance, user interface (UI) widgets, and perhaps some navigational menus or search forms. These pages are rich in content and functionality, are visually appealing, and more importantly, have an overall look and feel that abides by the golden rule of Web usability: “Be consistent.” What’s the recommended approach for building such pages and Web sites?

One possibility is wrapping these UI elements in user controls and referencing them in each page. Although such a model is extremely powerful and produces modular code, when you have hundreds of pages to work with, it soon becomes unmanageable. Both classic ASP and ASP.NET 1.x provide some workarounds for this type of issue, but neither tackles such a scenario openly and provides a definitive, optimal solution. Starting with version 2.0, ASP.NET faces up to the task through a new technology—master pages—and basically benefits from the ASP.NET Framework’s ability to merge a “supertemplate” with user-defined content replacements.

With themes, you can easily give the whole site a consistent (and, you hope, appealing) user interface and easily export that look from one application to the next. Much like Microsoft Windows XP themes, ASP.NET themes assign a set of styles and visual attributes to elements of the site that can be customized. Themes are a superset of cascading style sheets (CSS) and are not supported in versions of ASP.NET prior to 2.0.

A recurring task in Web development is collecting user input by using forms. When the input to collect is large and pretty much articulated (read, easy to categorize), multiple forms are typically used to accomplish the task. The whole procedure is divided into various steps, each of which takes care of collecting and validating a particular subset of the expected data. This multistep procedure is often called a “wizard.” With version 2.0, ASP.NET introduces a new view control that makes building wizards a snap.
Overall, building rich pages is a much more approachable task in ASP.NET today than it was with previous versions. With master pages, you build pages based on an existing template of code and markup; with themes, you use skins to control pages and achieve visual consistency as well as profile capabilities. Finally, with wizards you add rich functionality to pages.

**Working with Master Pages**

As a matter of fact, since the beginning ASP.NET and Microsoft Visual Studio greatly simplified the process of authoring Web pages and Web sites and made it affordable to a wide range of people with different skills. However, after a few months of real-world experience, many developers recognized that something was missing in the ASP.NET approach to page authoring. While building simple sites is easy, architecting real-world sites with hundreds of complex and rich pages still requires additional work and, more importantly, key decisions to be made without guidance.

Almost all Web sites use a similar graphical layout for all their pages. This doesn’t happen by chance—it grows out of accepted guidelines for design and usability. A consistent layout is characteristic of all cutting-edge Web sites, no matter how complex. For some Web sites, the layout consists of the header, body, and footer; for others, it is a more sophisticated aggregation of navigational menus, buttons, and panels that contain and render the actual content. Needless to say, manual duplication of code and HTML elements is simply out of the question. Making code automatically reusable clearly represents a better approach, but how do you implement it in practice?

**Authoring Rich Pages in ASP.NET 1.x**

In ASP.NET 1.x, the best approach to authoring pages with a common layout is to employ user controls. User controls are aggregates of ASP.NET server controls, literal text, and code. (We’ll cover user controls in my other book Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics (Microsoft Press, 2006) The ASP.NET runtime exposes user controls to the outside world as programmable components. The idea is that you employ user controls to tailor your own user interface components and share them among the pages of the Web site. For example, all the pages that need a navigational menu can reference and configure the user control that provides that feature.

**What’s Good About User Controls**

User controls are like embeddable pages. Turning an existing ASP.NET page into a user control requires only a few minor changes. User controls can be easily linked to any page that needs their services. Furthermore, changes to a user control’s implementation do not affect the referencing page and only require you (or the runtime) to recompile the user control into an assembly.
Chapter 6. Rich Page Composition

What’s Bad About User Controls

If you change the internal implementation of the user control, no referencing page will be affected. However, if you alter any aspect of the control’s public interface (such as the class name, properties, methods, or events), all the pages that reference the control must be updated. This means you must manually retouch all the pages in the application that use the control. Then you must recompile these pages and deploy the assemblies. In addition, the next time a user views each page, the ASP.NET runtime will take a while to respond because the dynamic assembly for the page must be re-created.

Architecturally speaking, the solution based on user controls works just fine. In practice, though, it is not a very manageable model for large-scale applications—it’s effectiveness decreases as the complexity of the application (the number of pages involved) increases. If your site contains hundreds of pages, handling common elements through user controls can quickly become inefficient and unmanageable.

Visual Inheritance

ASP.NET pages are built as instances of special classes—code-behind or code file classes. Because pages are ultimately classes, what happens if you stuff part of the common UI in some base class and inherit new pages from there? This approach resembles the visual inheritance feature that Windows Forms developers have been familiar with for a long time.

Pure visual inheritance à la Windows Forms is impractical in ASP.NET. This is because ASP.NET pages are made of code and markup. The markup determines the position of the controls, while code adds logic and functionality. Building predefined graphic templates in the base class doesn’t pose issues, but how would you import those standard UI blocks in derived pages and, more importantly, how would you merge those with controls local to the derived page?

In Windows Forms, controls have an absolute position that the designer reproduces, making it easy for developers to insert new controls anywhere. Web Forms, though, typically use relative positioning, which leads to either of the next two design choices. Option one is to supply predefined and named UI blocks in base classes and have derived classes load them in matching placeholders. Option two involves using master pages as defined in ASP.NET 2.0. To implement the former technique do the following:

1. Derive your page from a base class that knows how to create special UI blocks such as toolbars, headers, and footers. Each of these UI blocks has a unique name.
2. Add `<asp:placeholder>` controls to the derived page whose ID matches any of the predefined names. The base class contains the code to explore the control’s tree and expand placeholders with predefined UI blocks.
This approach exploits inheritance but provides no WYSIWYG facilities, and it forces you to create UI blocks in code-only mode with no markup. This option is demonstrated in the companion code, but it should be considered only for ASP.NET 1.x applications. The second option mentioned—using master pages—is described in the following section.

Writing a Master Page

Available in ASP.NET 2.0 and newer versions, a master page is a distinct file referenced at the application level, as well as at the page level, that contains the static layout of the page. Regions that each “derived” page can customize are referenced in the master page with a special placeholder control. A derived page is simply a collection of blocks the runtime will use to fill the holes in the master. True visual inheritance à la Windows Forms is not a goal of ASP.NET master pages. The contents of a master page are merged into the content page, and they dynamically produce a new page class that is served to the user upon request. The merge process takes place at compile time and only once. In no way do the contents of the master serve as a base class for the content page.

What’s a Master Page, Anyway?

A master page is similar to an ordinary ASP.NET page except for the top @Master directive and the presence of one or more ContentPlaceHolder server controls. A ContentPlaceHolder control defines a region in the master page that can be customized in a derived page. A master page without content placeholders is technically correct and will be processed correctly by the ASP.NET runtime. However, a placeholderless master fails in its primary goal—to be the supertemplate of multiple pages that look alike. A master page devoid of placeholders works like an ordinary Web page but with the extra burden required to process master pages. Here is a simple master page:

```<%@ Master Language="C#" CodeFile="Simple.master.cs" Inherits="Simple" %>
<%@ Master %>
<title>Hello, master pages</title>
<body>
<form id="form1" runat="server">
   <asp:Panel ID="HeaderPanel" runat="server" BackImageUrl="images/bkgnd.png" Width="100%">
      <asp:Label ID="TitleBox" runat="server" Text="Programming ASP.NET 3.5" />
   </asp:Panel>
   <asp:contentplaceholder id="PageBody" runat="server">
      <!-- derived pages will define content for this placeholder -->
   </asp:contentplaceholder>
</form>
</body>
</html>```
As you can see, the master page looks like a standard ASP.NET page. Aside from the identifying @Master directive, the only key differences are ContentPlaceHolder controls. A page bound to this master automatically inherits all the contents of the master (the header and footer, in this case) and can attach custom markup and server controls to each defined placeholder. The content placeholder element is fully identified by its ID property and normally doesn’t require other attributes.

The @Master Directive

The @Master directive distinguishes master pages from content pages and allows the ASP.NET runtime to properly handle each. A master page file is compiled to a class that derives from the MasterPage class. The MasterPage class, in turn, inherits UserControl. So, at the end of the day, a master page is treated as a special kind of ASP.NET user control.

The @Master directive supports quite a few attributes. For the most part, though, they are the same attributes that we reviewed in Chapter 3 for the @Page directive. Table 6-1 details the attributes that have a special meaning to master pages.

**TABLE 6-1 Attributes of the @Master Directive**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassName</td>
<td>Specifies the name for the class that will be created to render the master page. This value can be any valid class name but should not include a namespace. By default, the class name for simple.master is ASP.simple_master.</td>
</tr>
<tr>
<td>CodeFile</td>
<td>Indicates the URL to the file that contains any source code associated with the master page.</td>
</tr>
<tr>
<td>Inherits</td>
<td>Specifies a code-behind class for the master page to inherit. This can be any class derived from MasterPage.</td>
</tr>
<tr>
<td>MasterPageFile</td>
<td>Specifies the name of the master page file that this master refers to. A master can refer to another master through the same mechanisms a page uses to attach to a master. If this attribute is set, you will have nested masters.</td>
</tr>
</tbody>
</table>
The master page is associated with a code file that looks like the following:

```csharp
public partial class Simple : System.Web.UI.MasterPage
{
    protected void Page_Load(object sender, EventArgs e)
    {
        ...
    }
    ...
}
```

The `@Master` directive doesn’t override attributes set at the `@Page` directive level. For example, you can have the master set the language to Visual Basic .NET and one of the content pages can use C#. The language set at the master page level never influences the choice of the language at the content page level. You can use other ASP.NET directives in a master page—for example, `@Import`. However, the scope of these directives is limited to the master file and does not extend to child pages generated from the master.

### The `ContentPlaceHolder` Container Control

The `ContentPlaceHolder` control acts as a container placed in a master page. It marks places in the master where related pages can insert custom content. A content placeholder is uniquely identified by an ID. Here’s an example:

```html
<asp:contentplaceholder runat="server" ID="PageBody" />
```

A content page is an ASP.NET page that contains only `<asp:Content>` server tags. This element corresponds to an instance of the `Content` class that provides the actual content for a particular placeholder in the master. The link between placeholders and content is established through the ID of the placeholder. The content of a particular instance of the `Content` server control is written to the placeholder whose ID matches the value of the `ContentPlaceHolderID` property, as shown here:

```html
<asp:Content runat="server" contentplaceholderID="PageBody">
    ...
</asp:Content>
```

In a master page, you define as many content placeholders as there are customizable regions in the page. A content page doesn’t have to fill all the placeholders defined in the bound master. However, a content page can’t do more than just fill placeholders defined in the master.
Note: A placeholder can’t be bound to more than one content region in a single content page. If you have multiple `<asp:Content>` server tags in a content page, each must point to a distinct placeholder in the master.

**Specifying Default Content**

A content placeholder can be assigned default content that will show up if the content page fails to provide a replacement. Each `ContentPlaceHolder` control in the master page can contain default content. If a content page does not reference a given placeholder in the master, the default content will be used. The following code snippet shows how to define default content:

```xml
<asp:contentplaceholder runat="server" ID="PageBody">
    <!-- Use the following markup if no custom content is provided by the content page -->
    ...
</asp:contentplaceholder>
```

The default content is completely ignored if the content page populates the placeholder. The default content is never merged with the custom markup provided by the content page.

Note: A `ContentPlaceHolder` control can be used only in a master page. Content placeholders are not valid on regular ASP.NET pages. If such a control is found in an ordinary Web page, a parser error occurs.

**Writing a Content Page**

The master page defines the skeleton of the resulting page. If you need to share the layout or any UI block among all the pages, placing it in a master page will greatly simplify management of the pages in the application. You create the master and then think of your pages in terms of a delta from the master. The master defines the common parts of a certain group of pages and leaves placeholders for customizable regions. Each content page, in turn, defines what the content of each region has to be for a particular ASP.NET page. Figure 6-1 shows how to create a content page in Visual Studio.
The Content Control

The key part of a content page is the Content control—a mere container for other controls. The Content control is used only in conjunction with a corresponding ContentPlaceHolder and is not a standalone control. The master file that we considered earlier defines a single placeholder named PageBody. This placeholder represents the body of the page and is placed right below an HTML table that provides the page's header. Figure 6-2 shows a sample content page based on the aforementioned master page.
Let's take a look at the source code of the content page:

```csharp
<asp:Content ID="Content1" ContentPlaceHolderID="PageBody" runat="server">
  <h1>Welcome to this page!</h1>
  <h3>The rest of the page is kindly offered by our sponsor page -- the master!</h3>
</asp:Content>
```

The content page is the resource that users invoke through the browser. When the user points her or his browser to this page, the output in Figure 6-3 is shown.

The replaceable part of the master is filled with the corresponding content section defined in the derived pages. A content page—that is, a page bound to a master—is a special breed of page in that it can only contain `<asp:Content>` controls. A content page is not permitted to host server controls outside of an `<asp:Content>` tag.

Let's explore the attachment of pages to masters in a bit more detail.

### Attaching Pages to a Master

In the previous example, the content page is bound to the master by using the `MasterPageFile` attribute in the `@Page` directive. The attribute contains a string representing the path to the master page. Page-level binding is just one possibility—although it is the most common one.

You can also set the binding between the master and the content at the application or folder level. Application-level binding means that you link all the pages of an application to the
same master. You configure this behavior by setting the Master attribute in the <pages> element of the principal web.config file:

```xml
<configuration>
  <system.web>
    <pages master="MyApp.master" />
  </system.web>
</configuration>
```

If the same setting is expressed in a child web.config file—a web.config file stored in a site subdirectory—all ASP.NET pages in the folder are bound to a specified master page.

Note that if you define binding at the application or folder level, all the Web pages in the application (or the folder) must have Content controls mapped to one or more placeholders in the master page. In other words, application-level binding prevents you from having (or later adding) a page to the site that is not configured as a content page. Any classic ASP.NET page in the application (or folder) that contains server controls will throw an exception.

**Device-Specific Masters**

Like all ASP.NET pages and controls, master pages can detect the capabilities of the underlying browser and adapt their output to the specific device in use. ASP.NET makes choosing a device-specific master easier than ever. If you want to control how certain pages of your site appear on a particular browser, you can build them from a common master and design the master to address the specific features of the browser. In other words, you can create multiple versions of the same master, each targeting a different type of browser.

How do you associate a particular version of the master and a particular browser? In the content page, you define multiple bindings using the same MasterPageFile attribute, but you prefix it with the identifier of the device. For example, suppose you want to provide ad hoc support for Microsoft Internet Explorer and Netscape browsers and use a generic master for any other browsers that users employ to visit the site. You use the following syntax:

```xml
<%@ Page masterpagefile="Base.master"
  ie:masterpagefile="ieBase.master"
  netscape6to9:masterpagefile="nsBase.master" %>
```

The ieBase.master file will be used for Internet Explorer; the nsBase.master, on the other hand, will be used if the browser belongs to the Netscape family, version 6.x to 9.0. In any other case, a device-independent master (base.master) will be used. When the page runs, the ASP.NET runtime automatically determines which browser or device the user is using and selects the corresponding master page, as shown in Figure 6-4.
The prefixes you can use to indicate a particular type of browser are those defined in the ASP.NET configuration files for browsers. Table 6-2 lists the most commonly used IDs.

**TABLE 6-2 ID of Most Common Browsers**

<table>
<thead>
<tr>
<th>Browser ID</th>
<th>Browser Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>Any version of Internet Explorer</td>
</tr>
<tr>
<td>Netscape3</td>
<td>Netscape Navigator 3.x</td>
</tr>
<tr>
<td>Netscape4</td>
<td>Netscape Communicator 4.x</td>
</tr>
<tr>
<td>Netscape6to9</td>
<td>Any version of Netscape higher than 6.0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Firefox</td>
</tr>
<tr>
<td>Opera</td>
<td>Opera</td>
</tr>
<tr>
<td>Up</td>
<td>Openwave-powered devices</td>
</tr>
</tbody>
</table>

It goes without saying that you can distinguish not just between up-level and down-level browsers but also between browsers and other devices such as cellular phones and personal digital assistants (PDAs). If you use device-specific masters, you must also indicate a device-independent master.

**Warning** Browser information is stored differently in ASP.NET 1.x and ASP.NET 2.0 and newer versions. In ASP.NET 1.x, you find it in the `<browserCaps>` section of the `machine.config` file. In newer versions, it is stored in text files with a `.browser` extension located in the `Browsers` folder under the ASP.NET installation path on the Web server. It's the same folder that contains `machine.config` and `WINDOWS\Microsoft.NET\Framework\[version]\Config\Browsers`.
Setting the Title of a Page

As a collection of <asp:Content> tags, a content page is not allowed to include any markup that can specify the title of the page. Using the <title> tag is possible in the master page, but the master page—by design—works as the base for a variety of pages, each requiring its own title. The trick to setting the title is in using the Title property of the @Page directive in the content page:

```<@Page MasterPageFile="simple.master" Title="Hello, master" %>
```

Note, though, setting the title of the page is possible only if the <title> or <head> tag in the master is flagged as runat=server. When adding a new master page to your Web solution, Visual Studio 2008 automatically marks both tags with the runat=server attribute.

Processing Master and Content Pages

The use of master pages slightly changes how pages are processed and compiled. For one thing, a page based on a master has a double dependency—one on the .aspx source file (the content page) and on the .master file (the master page). If either of these pages changes, the dynamic page assembly will be re-created. Although the URL that users need is the URL of the content page, the page served to the browser results from the master page fleshed out with any replacement provided by the content page.

Compiling Master Pages

When the user requests an .aspx resource mapped to a content page—that is, a page that references a master—the ASP.NET runtime begins its job by tracking the dependency between the source .aspx file and its master. This information is persisted in a local file created in the ASP.NET temporary files folder. Next, the runtime parses the master page source code and creates a Visual Basic .NET or C# class, depending on the language set in the master page. The class inherits MasterPage, or the master’s code file, and is then compiled to an assembly.

If multiple .master files are found in the same directory, they are all processed at the same time. Thus a dynamic assembly is generated for any master files found, even if only one of them is used by the ASP.NET page whose request triggered the compilation process. Therefore, don’t leave unused master files in your Web space—they will be compiled anyway. Also note that the compilation tax is paid only the first time a content page is accessed within the application. When a user accesses another page that requires the second master, the response is faster because the previously compiled master is cached.
Serving the Page to Users

As mentioned, any ASP.NET page bound to a master page must have a certain structure—no server controls or literal text is allowed outside the `<asp:Content>` tag. As a result, the layout of the page looks like a plain collection of content elements, each bound to a particular placeholder in the master. The connection is established through the ID property. The `<asp:Content>` element works like a control container, much like the Panel control of ASP.NET or the HTML `<div>` tag. All the markup text is compiled to a template and associated with the corresponding placeholder property on the master class.

The master page is a special kind of user control with some templated regions. It’s not coincidental, in fact, that the MasterPage class inherits from the UserControl class. Once instantiated as a user control, the master page is completed with templates generated from the markup defined in the content page. Next, the resulting control is added to the control tree of the current page. No other controls are present in the final page except those brought in by the master. Figure 6-5 shows the skeleton of the final page served to the user.

![Figure 6-5](image)

**FIGURE 6-5** The structure of the final page in which the master page and the content page are merged.

Nested Master Pages

So far we’ve seen a pretty simple relationship between a master and a collection of content pages. However, the topology of the relationship can be made as complex and sophisticated as needed. A master can, in fact, be associated with another master and form a hierarchical,
nested structure. When nested masters are used, any child master is seen and implemented as a plain content page in which extra <asp:ContentPlaceHolder> controls are defined for an extra level of content pages. Put another way, a child master is a kind of content page that contains a combination of <asp:Content> and <asp:ContentPlaceHolder> elements. Like any other content page, a child master points to a master page and provides content blocks for its parent’s placeholders. At the same time, it makes available new placeholders for its child pages.

---

**Note** There’s no architectural limitation in the number of nesting levels you can implement in your Web sites. Performance-wise, the depth of the nesting has a negligible impact on the overall functionality and scalability of the solution. The final page served to the user is always compiled on demand and never modified as long as dependent files are not touched.

Let’s expand on the previous example to add an intermediate master page. The root master page—named *parent.master*—defines the header, the footer, and a replaceable region. Except for the class names, the source code is identical to the example we considered earlier. Let’s have a closer look at the intermediate master—named *content.master*:

```csharp
<%@ Master Language="C#" MasterPageFile="Parent.master" 
CodeFile="Content.master.cs" Inherits="ContentMaster" %>
<asp:Content runat=Server ContentPlaceHolderID="ContentOfThePage" >
  <table width="100%"><tr>
    <td>
      <h1>Welcome to this page!</h1>
      <h3>The rest of the page is kindly offered by our sponsor page — the master!</h3>
    </td>
    <td align="center">
      <h2>Select Your Favorite Chapter</h2>
      <asp:ContentPlaceHolder runat="server" ID="ChapterMenu" />
    </td>
  </tr></table>
</asp:Content>
```

As you can see, the master contains both a collection of `<asp:Content>` and `<asp:ContentPlaceHolder>` tags. The top directive is that of a master but contains the `MasterPageFile` attribute, which typically characterizes a content page.

The `content.master` resource is not directly viewable because it contains a virtual region. If you’re familiar with object-oriented programming (OOP) terminology, I’d say that an intermediate master class is much like an intermediate virtual class that overrides some methods on the parent but leaves other abstract methods to be implemented by another derived class. Just as abstract classes can’t be instantiated, nested master pages can’t be viewed through a browser. In any case, the `content.master` resource is undoubtedly a master class, and its code file contains a class that inherits from `MasterPage`.
Warning  Visual Studio 2008 is currently the only Microsoft tool capable of fully supporting nested master pages. If you’re using older versions, you have to create an intermediate master page as a content page and then change the top directive to @Master, remove the Title attribute and, last but not least, change the base class of the code file to MasterPage.

The following code illustrates a content page that builds on two masters:

```xml
<asp:Content ContentPlaceHolderID="ChapterMenu" runat="server">
    <small>This is the nested master</small><hr />
    <asp:TextBox runat="server" Text="[Enter keywords]" />
    <asp:LinkButton runat="server" Text="Search ..." />
</asp:Content>
```

Figure 6-6 shows the results.

Admittedly, there’s nothing in the figure that clearly indicates the existence of two masters; for your information, the innermost master controls the left-most area where the drop-down list is laid out. This means that writing another page that offers an alternative technique to find a chapter is particularly easy. Have a look at the code and Figure 6-7:

```
<asp:Content ContentPlaceHolderID="ChapterMenu" runat="server">
    <asp:DropDownList runat="server">
        ...
    </asp:DropDownList>
</asp:Content>
```
A sapient use of master and content pages leads straight to an obvious conclusion: slightly different pages require slightly different code.

Programming the Master Page

You can use code in content pages to reference properties, methods, and controls in the master page, with some restrictions. The rule for properties and methods is that you can reference them if they are declared as public members of the master page. This includes public page-scope variables, public properties, and public methods.

Exposing Master Properties

To give an identity to a control in the master, you simply set the runat attribute and give the control an ID. Can you then access the control from within a content page? Not directly. The only way to access the master page object model is through the Master property. Note, though, that the Master property of the Page class references the master page object for the content page. This means that only public properties and methods defined on the master page class are accessible.

The following code enhances the previous master page to make it expose the text of the header as a public property:

```csharp
public partial class SimpleWithProp : System.Web.UI.MasterPage
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```
The header text of Figure 6-3 (shown earlier) is represented by a `Label` control named `TitleBox`. The control's protection level makes it inaccessible from the outside world, but the public property `TitleBoxText` defined in the preceding code represents a public wrapper around the `Label`'s `Text` property. In the end, the master page has an extra public property through which programmers can set the text of the header.

### Invoking Properties on the Master

The `Master` property is the only point of contact between the content page and its master. The bad news is that the `Master` property is defined to be of type `MasterPage`; as such, it doesn't know anything about any property or method definition specific to the master you're really working with. In other words, the following code wouldn't compile because no `TitleBoxText` property is defined on the `MasterPage` class:

```csharp
public partial class HelloMaster : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Master.TitleBoxText = "Programming ASP.NET 3.5";
    }
}
```

What's the real type behind the `Master` property?

The `Master` property represents the master page object as compiled by the ASP.NET runtime engine. This class follows the same naming convention as regular pages—`ASP.XXX_master`, where `XXX` is the name of the master file. Developers can override the default class name by setting the `ClassName` attribute on the `@Master` directive. The attribute lets you assign a user-defined name to the master page class:

```html
<%@ Master Inherits="SimpleWithProp" ... Classname="MasterWithProp" %>
```

In light of this, to be able to call custom properties or methods, you must first cast the object returned by the `Master` property to the actual type:

```csharp
((ASP.MasterWithProp)Master).TitleBoxText = "Programming ASP.NET 3.5";
```

Interestingly enough, Visual Studio provides some facilities to let you identify the right dynamically generated type already at design time. See Figure 6-8.
Part I Building an ASP.NET Page

FIGURE 6-8 Visual Studio pops up names of classes that will be created only during the page execution.

The ASP namespace is the system namespace that all system dynamically defined types belong to. In Visual Studio, that namespace is properly recognized and handled by IntelliSense.

The @MasterType Directive

By adding the @MasterType directive in the content page, you can avoid all the casting just shown. The @MasterType informs the compiler about the real type of the Master property. The Master property is declared of the right type in the dynamically created page class, and this allows you to write strong-typed code, as follows:

```csharp
<%@ Page Language="C#" MasterPageFile="SimpleWithProp.master" CodeFile="HelloMasterType.aspx.cs" Inherits="HelloMasterType" %>
<%@ MasterType VirtualPath="SimpleWithProp.master" %>
```

In the code file, you can have the following statements:

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    Master.TitleBoxText = "Programming ASP.NET 3.5";
}
```

The @MasterType directive supports two mutually exclusive attributes—VirtualPath and TypeName. Both serve to identify the master class to use. The former does it by URL, the latter by type name.
Changing the Master Page Dynamically

To associate an ASP.NET content page with a master page—keeping in mind that in no case can you associate a classic ASP.NET page with a master—you use the `MasterPageFile` attribute of the `@Page` directive. `MasterPageFile`, though, is also a read-write property on the `Page` class that points to the name of the master page file. Can you dynamically select the master page via code and based on runtime conditions?

Using a dynamically changing master page is definitely possible in ASP.NET and is suitable, for example, for applications that can present themselves to users through different skins. However, programmatically selecting the master page is not a task that you can accomplish at any time. To be precise, you can set the `MasterPageFile` property only during the `PreInit` page event—that is, before the runtime begins working on the request.

```csharp
protected void Page_PreInit(object sender, EventArgs e)
{
    MasterPageFile = "simple2.master";
}
```

If you try to set the `MasterPageFile` property in `Init` or `Load` event handlers, an exception is raised.

**Note** The `Master` property represents the current instance of the master page object, is a read-only property, and can’t be set programmatically. The `Master` property is set by the runtime after loading the content of the file referenced by the `MasterPageFile` property.

Working with Themes

For years, CSS styles have helped site developers to easily and efficiently design pages with a common and consistent look and feel. Although page developers can select the CSS file programmatically on the server, at its core CSS remains an inherent client-side technology, devised and implemented to apply skins to HTML elements. When you build ASP.NET pages, though, you mostly work with server controls.

CSS styles can be used to style server controls, but they’re not the right tool for the job. The main issue here is that ASP.NET controls can have properties that are not the direct emanation of a CSS style property. The appearance of an ASP.NET control can be affected by an array of resources—images, strings, templates, markup, combinations of various CSS styles.

To properly apply skins to ASP.NET server controls, CSS files are necessary but not sufficient. Enter ASP.NET themes.
Part I Building an ASP.NET Page

ASP.NET themes are closely related to Windows XP themes. Setting a theme is as simple as setting a property, and all the settings the theme contains are applied in a single shot. Themes can be applied to individual controls and also to a page or an entire Web site.

Understanding ASP.NET Themes

More often than not, when you author a page you don’t just focus on the tasks a certain set of controls must be able to accomplish. You also consider their appearance. Most of the time, you end up setting visual attributes such as colors, font, borders, and images. The more sophisticated the control, the more time you spend making it look nice rather than just functional.

In ASP.NET, the DataGrid control—one of the most popular and customizable controls—provides a gallery of predefined styles from which you choose the most appealing. This gallery of predefined styles is the DataGrid’s auto-format feature. DataGrid’s built-in styles are implemented through a set of predefined settings that Visual Studio applies to the control at design time. The auto-format feature saves testing and typing and lets you choose the style visually. Added as a time-saving feature, auto-format addresses the issue only partially, as it has two main drawbacks. First, a lot of visual attributes are still persisted to the .aspx source file, making rich pages hard to read and maintain. Second, the list of available formats is kind of closed and can’t be further extended or personalized if you’re not writing Visual Studio ad hoc designers for the specific control.

Wouldn’t it be great if you could compose your pages just by picking controls off the toolbox and connecting them, without even bothering about their final look? Wouldn’t it be nice if you could then simply create an additional file in the project to define visual attributes for each type of control? In this way, the .aspx source file would be free of verbose visual attributes and you could change the style of controls at will, even dynamically, while performing little or no modifications to the original page. ASP.NET themes provide exactly this capability.

What’s a Theme, Anyway?

A theme is a set of skins and associated auxiliary files such as style sheets and images—a sort of super CSS file. Once enabled, the theme determines the appearance of all controls under its jurisdiction. Consider the following simple markup:

```xml
<asp:Calendar ID="Calendar1" runat="server" />
```

Without themes, the calendar will look grayish, spare, and spartan. With a theme added, the same markup renders a more colorful and appealing calendar. As you can see, a neat separation exists between the page contents and formatting rules. Look at Figure 6-9. Which do you think is the unthemed calendar?
To fully understand ASP.NET themes, you must be familiar with a few terms, which are detailed in Table 6-3.

**TABLE 6-3 ASP.NET Themes Terminology**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skin</td>
<td>A named set of properties and templates that can be applied to one or more controls on a page. A skin is always associated with a specific control type.</td>
</tr>
<tr>
<td>Style sheet</td>
<td>A CSS or server-side style sheet file that can be used by pages on a site.</td>
</tr>
<tr>
<td>StyleSheet Theme</td>
<td>A theme used to abstract control properties from controls. The application of this theme means that the control can still override the theme.</td>
</tr>
<tr>
<td>Customization Theme</td>
<td>A theme used to abstract control properties from controls, but the theme overrides the control and any style sheet theme.</td>
</tr>
</tbody>
</table>

Imagine you are creating a new Web site and would like it to be visually appealing from the start. Instead of having to learn all the available style properties of each employed control, you just use ASP.NET themes. Using a built-in theme in a page is as easy as setting a property, as we’ll see in a moment. With this change, pages automatically inherit a new, and hopefully attractive, appearance. For example, if you add a Calendar control to a page, it automatically renders with the default appearance defined in the theme.

Selecting a theme for one or more pages doesn’t necessarily bind you to the settings of that theme. Through the Visual Studio designer, you can review the pages and manually adjust some styles in a control if you want to.

---

**Note** The following convention holds true in this book and, in general, in related literature. Unless otherwise suggested by the context, the word “theme” indicates a customization theme. A style sheet theme is usually referred to as “a style sheet theme.”
Structure of a Theme

Themes are expressed as the union of various files and folders living under a common root directory. Themes can be global or local. Global themes are visible to all Web applications installed on a server machine. Local themes are visible only to the application that defines them. Global themes are contained in child directories located under the following path. The name of the directory is the name of the theme.

%WINDOWS%\Microsoft.NET\Framework\[version]\ASP.NETClientFiles\Themes

Local themes are specialized folders that live under the App_Themes folder at the root of the application. Figure 6-10 shows a sample theme (named Core35-Basic) in a Web application.

As you can see, the theme in the figure consists of a .css file and a .skin file, plus a subdirectory of images. Generally, themes can contain a mix of the following resources:

- **CSS files**  Also known as style sheets, CSS files contain style definitions to be applied to elements in an HTML document. Written according to a tailormade syntax, CSS styles define how elements are displayed and where they are positioned on your page. Web browsers that support only HTML 3.2 and earlier will not apply CSS styles. The World Wide Web Consortium (W3C) maintains and constantly evolves CSS standards. Visit http://www.w3.org for details on current CSS specifications. CSS files are located in the root of the theme folder.

- **Skin files**  A skin file contains the theme-specific markup for a given set of controls. A skin file is made by a sequence of control definitions that include predefined values for
most visual properties and supported templates. Each skin is control-specific and has a
unique name. You can define multiple skins for a given control. A skinned control has
the original markup written in the .aspx source file modified by the content of the skin.
The way the modification occurs depends on whether a customization or a style sheet
theme is used. Skin files are located in the root of the theme folder.

- **Image files** Feature-rich ASP.NET controls might require images. For example, a
  pageable DataGrid control might want to use bitmaps for first or last pages that are
  graphically compliant to the skin. Images that are part of a skin are typically located in
  an Images directory under the theme folder. (You can change the name of the folder as
  long as the name is correctly reflected by the skin's attributes.)

- **Templates** A control skin is not limited to graphical properties but extends to define
  the layout of the control—for templated controls that support this capability. By stuff-
  ing template definitions in a theme, you can alter the internal structure of a control
  while leaving the programming interface and behavior intact. Templates are defined as
  part of the control skin and persisted to skin files.

The content types just listed are not exhaustive, but they do cover the most commonly used
data you might want to store in a theme. You can have additional subdirectories filled with
any sort of data that makes sense to skinned controls. For example, imagine you have a
custom control that displays its own user interface through the services of an external ASP.
NET user control (.ascx). Skinning this control entails, among other things, indicating the URL
to the user control. The user control becomes an effective part of the theme and must be
stored under the theme folder. Where exactly? That is up to you, but opting for a Controls
subdirectory doesn't seem to be a bad idea. We'll return to this point later when building a
sample theme.

**Customization Themes vs. Style Sheet Themes**

There are two forms of themes—customization themes and style sheet themes.

Customization themes are used for post customization of a site. The theme overrides any
property definition on the control found in the .aspx source. By changing the page's theme,
you entirely modify the appearance of the page without touching the source files. If you opt
for customization theming, you just need minimal markup for each control in the ASP.NET
page.

Style sheet themes are similar to CSS style sheets, except that they operate on control prop-
erties rather than on HTML element styles. Style sheet themes are applied immediately after
the control is initialized and before the attributes in the .aspx file are applied. In other words,
with a style sheet theme developers define default values for each control that are in fact
overridden by settings in the .aspx source.
Important Customization themes and style sheet themes use the same source files. They differ only in how the ASP.NET runtime applies them to a page. The same theme can be applied as a customization theme or a style sheet theme at different times.

The difference between customization and style sheet themes is purely a matter of which takes priority over which. Let’s review the resultant form of a control when a customization theme and style sheet theme are applied. Imagine you have the following markup:

```html
<asp:Calendar ID="Calendar1" runat="server" backcolor="yellow" />
```

If the page that contains this markup is bound to a customization theme, the calendar shows up as defined in the theme. In particular, the background of the calendar will be of the color defined by the theme.

If the page is bound to a style sheet theme, instead, the background color of the calendar is yellow. The other properties are set in accordance with the theme.

**Theming Pages and Controls**

You can apply themes at various levels—application, folder, and individual pages. In addition, within the same theme you can select different skins for the same type of control.

Setting a theme at the application level affects all the pages and controls in the application. It’s a feature you configure in the application’s `web.config` file:

```xml
<system.web>
    <pages theme="Core35-Basic" />
</system.web>
```

The `theme` attribute sets a customization theme, while the `styleSheetTheme` attribute sets a style sheet theme. Note that the case is important in the `web.config`'s schema. Likewise, a theme can be applied to all the pages found in a given folder and below that folder. To do so, you create a new `web.config` file in an application’s directory and add the section just shown to it. All the pages in that directory and below it will be themed accordingly. Finally, you can select the theme at the page level and have styles and skins applied only to that page and all its controls.

**Enabling Themes on a Page**

To associate a theme with a page, you set the `Theme` or `StyleSheetTheme` attribute on the `@Page` directive, and you’re all set:

```html
<% @Page Language="C#" Theme="Core35-Basic" %>
<% @Page Language="C#" StyleSheetTheme="Core35-Basic" %>
```
Also in this case, Theme sets a customization theme, whereas StyleSheetTheme indicates a style sheet theme.

Bear in mind that the name of the selected theme must match the name of a subdirectory under the App_Themes path or the name of a global theme. If a theme with a given name exists both locally to the application and globally to the site, the local theme takes precedence. Figure 6-11 shows IntelliSense support for themes in Visual Studio.

FIGURE 6-11 IntelliSense support for themes in Visual Studio.

While we’re speaking of precedence, it is important to note that themes have a hierarchical nature: directory-level themes takes precedence over application-level themes, and page-level themes override any other themes defined around the application. This hierarchy is independent of which attributes are used—Theme or StyleSheetTheme—to enable theming.

Note Setting both Theme and StyleSheetTheme attributes is not prohibited, even though it is not a recommended practice. There’s a behavioral gap between the two forms of themes that should make clear which one you need in any situation. However, if you set both attributes, consider that both themes will be applied—first the style sheet theme and then the customization theme. The results depend on the CSS cascading mechanism and ultimately is determined by the CSS settings of each theme.

Applying Skins

A skin file looks like a regular ASP.NET page as it is populated by control declaration and import directives. Each control declaration defines the default appearance of a particular control. Consider the following excerpt from a skin file:

```xml
<!-- This is a possible skin for a Button control -->
<asp:Button runat="server"
    BorderColor="darkgray"
    Font-Bold="true"
    BorderWidth="1px"
    BorderStyle="outset"
    ForeColor="DarkSlateGray"
    BackColor="gainsboro" />
```

Chapter 6. Rich Page Composition
Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
Part I Building an ASP.NET Page

The net effect of the skin is that every Button control in a themed page will be rendered as defined by the preceding markup. If the theme is applied as a style sheet, the settings just shown will be overridable by the developer; if the theme is a customization theme, those settings determine the final look and feel of the control. Properties that the theme leaves blank are set according to the control’s defaults or the .aspx source.

Important Whatever theme you apply—customization or style sheet—control properties can always be modified through code in page events such as Init and Load.

A theme can contain multiple skins for a given control, each identified with a unique name—the SkinID attribute. When the SkinID attribute is set, the skin is said to be a named skin. A theme can contain any number of named skins per control, but just one unnamed (default) skin. You select the skin for a control in an ASP.NET themed page by setting the control’s SkinID property. The value of the control’s SkinID property should match an existing skin in the current theme. If the page theme doesn’t include a skin that matches the SkinID property, the default skin for that control type is used. The following code shows two named skins for a button within the same theme:

```xml
<!-- Place these two definitions in the same .skin file -->
<asp:button skinid="skinClassic" BackColor="#808080" />
<asp:button skinid="skinTrendy" BackColor="#CCFFFF" />
```

When you enable theming on a page, by default all controls in that page will be themed except controls, and individual control properties, that explicitly disable theming.

Important Note The automatic application of themes to all controls in a page makes it easy to customize a page that has no knowledge of skins, including existing pages written for ASP.NET 1.x.

Taking Control of Theming

The ASP.NET theming infrastructure provides the EnableTheming Boolean property to disable skins for a control and all its children. You can configure a page or control to ignore themes by setting the EnableTheming property to false. The default value of the property is true. EnableTheming is defined on the Control class and inherited by all server controls and pages. If you want to disable theme support for all controls in a page, you can set the EnableTheming attribute on the @Page directive.

Important Note that the EnableTheming property can be set only in the Page_PreInit event for static controls—that is, controls defined in the .aspx source. For dynamic controls—that is, controls created programmatically—you must have set the property before adding the control to the page’s control tree. A control is added to the page’s control tree when you add to the Controls collection of the parent control—typically, the form or another control in the form.
When is disabling themes useful? Themes are great at ensuring that all page controls have a consistent look and feel, but at the same time themes override the visual attributes of any control for which a skin is defined. You can control the overriding mechanism a bit by switching style sheet and customization themes. However, when you want a control or page to maintain its predefined look, you just disable themes for that page or control.

Note that disabling themes affects only skins, not CSS styles. When a theme includes one or more CSS style-sheet files, they are linked to the `<head>` tag of the resulting HTML document and, after that, are handled entirely by the browser. As you can easily guess, there’s not much a Web browser can know about ASP.NET themes!

### Theming Controls
Themes style server controls to the degree that each control allows. By default, all control properties are themeable. Theming can be disabled on a particular property by applying the `Themeable` attribute on the property declaration, as follows:

```
[Themeable(false)]
public virtual bool CausesValidation
{
    get { ... }
    set { ... }
}
```

You can’t change the `Themeable` attribute for built-in server controls. You have that option for custom controls instead. Moreover, for custom controls you should use the `Themeable` attribute to prevent theming of behavioral properties such as the `CausesValidation` property just shown. Themes should be used only on visual properties that uniquely affect the appearance of the control. Finally, the `Themeable` attribute can be applied to the class declaration of a custom control to stop it from ever bothering about themes:

```
[Themeable(false)]
public MyControl : Control
{
    ...
}
```

### Putting Themes to Work
Finding a bunch of themes that suit your needs, free or for a small fee, shouldn’t be a problem. However, this is a bad reason for not learning how to build your own themes. As mentioned, themes consist of several supporting files, including CSS style sheets and control skins to decorate HTML elements and server controls, respectively, and any other supporting images or files that make up the final expected result.
I firmly believe that building nice-looking, consistent, usable themes is not a programmer’s job. It is a task that designers and graphics people can easily accomplish ten times better and faster. However, themes are more than CSS files, and what’s more, they are in the area of control properties—exactly the realm of the developer. In short, developers should provide guidance to theme designers much more than we did in the past with CSS authors.

As a first step, let’s review the key differences between CSS files and themes.

CSS vs. Themes
Themes are similar to CSS style sheets in that both apply a set of common attributes to any page where they are declared. Themes differ from CSS style sheets in a few key ways, however.

First and foremost, themes work on control properties, whereas CSS style sheets operate on styles of HTML elements. Because of this, with themes you can include auxiliary files and specify standard images for a TreeView or Menu control, the paging template of a DataGrid, or the layout of a Login control. In addition, themes can optionally force overriding of local property values (customization themes) and not cascade as CSS style sheets do.

Because themes incorporate CSS style-sheet definitions and apply them along with other property settings, there’s no reason for preferring CSS style sheets over themes in new ASP.NET applications.

Note: In Visual Studio 2008, you’ll find a brand new, and extremely effective, CSS Designer tool. The tool allows you to manage CSS styles applied to the various elements of a page. It is interesting to note, though, that the tool doesn’t capture CSS styles applied through a stylesheet defined in a theme. It works perfectly with CSS, however, when you link to the page using the <head> node and from a non-theme folder.

Creating a Theme
To create a new theme in a Visual Studio ASP.NET solution, you start by creating a new folder under App_Themes. The simplest way to do this is by right-clicking on the App_Themes node and selecting a theme folder. Next, you add theme files to the folder, and when you’re done, you can even move the entire directory to the root path of global themes on the Web server.

Typical auxiliary files that form a theme are listed in Figure 6-12. They are CSS style-sheet files, skin files, XML or text files, and extensible style-sheet files (XSLT). Empty files of the specified type are created in the theme folder and edited through more or less specialized text editors in Visual Studio.
A skin file is a collection of a control’s markup chunks, optionally named through the SkinID attribute. You can create a skin file by cutting and pasting the markup of controls you visually configured in a sample page. If some properties of the skinned controls require resources, you can point them to a path inside the theme folder. Here’s an example:

```xml
<asp:BulletedList runat="server"
    Font-Names="Verdana"
    BulletImageURL="Images/smokeandglass_bullet2.gif"
    BulletStyle="CustomImage"
    BackColor="transparent"
    ForeColor="#585880" />
```

This skin of the BulletedList control points to a theme-specific URL for the bullet image. The directory Images is intended to be relative to the theme folder. Needless to say, the name Images is totally arbitrary. Should the skin require other external files, you could group them in other theme subdirectories.

A skin file can define the appearance of built-in server controls as well as custom controls. To skin a custom control, though, you must first reference it in the skin file (just as you would reference it in a typical page file), as follows.

```xml
<%@ Register TagPrefix="x"
    Namespace="Samples.Core35.Controls"
    Assembly="Core35.Controls" %>
```

Next, you add the desired default markup for any control defined in the specified assembly and namespace.
Part I Building an ASP.NET Page

Loading Themes Dynamically

You can apply themes dynamically, but this requires a bit of care. The ASP.NET runtime loads theme information immediately after the `PreInit` event fires. When the `PreInit` event fires, the name of any theme referenced in the `@Page` directive is already known and will be used unless it is overridden during the event. If you want to enable your users to change themes on the fly, you create a `Page_PreInit` event handler. The following code shows the code file of a sample page that changes themes dynamically:

```csharp
public partial class TestThemes : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack) {
            ThemeList.DataSource = GetAvailableThemes();
            ThemeList.DataBind();
        }
    }

    void Page_PreInit(object sender, EventArgs e)
    {
        string theme = "";
        if (Page.Request.Form.Count > 0)
            theme = Page.Request("ThemeList").ToString();
        if (theme == "None")
            theme = "";
        this.Theme = theme;
    }

    protected StringCollection GetAvailableThemes()
    {
        string path = Request.PhysicalApplicationPath + "App_Themes";
        DirectoryInfo dir = new DirectoryInfo(path);
        StringCollection themes = new StringCollection();
        foreach (DirectoryInfo di in dir.GetDirectories())
            themes.Add(di.Name);
        return themes;
    }
}
```

Figure 6-13 shows the page in action. The drop-down list control enumerates the installed application themes and lets you choose the one to apply. The selected theme is then applied in the `PreInit` event and immediately reflected. In the `PreInit` event, no view state has been restored yet; so `Request.Form` is the only safe way to access a posted value like the selected theme.
Master pages and themes give you the power of building similar-looking, rich pages that share graphics, control layout, and even some functionality. A special type of rich page is the page that implements a wizard. More common in Windows desktop applications than in Web scenarios, wizards are typically used to break up large forms to collect user input. A wizard is a sequence of related steps, each associated with an input form and a user interface. Users move through the wizard sequentially, but they are normally given a chance to skip a step or jump back to modify some of the entered values. A wizard is conceptually pretty simple, but implementing it over HTTP connections can be tricky. Everybody involved with serious Web development can only heartily welcome the introduction of the Wizard control.

An Overview of the Wizard Control

The Wizard control supports both linear and nonlinear navigation. It allows you to move backward to change values and to skip steps that are unnecessary due to previous settings or because users don’t want to fill those fields. Like many other ASP.NET controls, the Wizard supports themes, styles, and templates.

The Wizard is a composite control and automatically generates some constituent controls such as navigation buttons and panels. As you’ll see in a moment, the programming interface of the control has multiple templates that provide for in-depth customization of the overall
user interface. The control also guarantees that state is maintained no matter where you
move—backward, forward, or to a particular page. All the steps of a wizard must be declared
within the boundaries of the same Wizard control. In other words, the wizard must be self-
contained and not provide page-to-page navigation.

Structure of a Wizard

As shown in Figure 6-14, a wizard has four parts: header, view, navigation bar, and sidebar.

The header consists of text you can set through the HeaderText property. You can change the
default appearance of the header text by using its style property; you can also change the
structure of the header by using the corresponding header template property. If HeaderText
is empty and no custom template is specified, no header is shown for the wizard.

The view displays the contents of the currently active step. The wizard requires you to define
each step in an <asp:WizardStep> element. An <asp:WizardStep> element corresponds to a
WizardStep control. Different types of wizard steps are supported; all wizard step classes
inherit from a common base class named WizardStepBase.
All wizard steps must be grouped in a single `<wizardsteps>` tag, as shown in the following code:

```xml
<asp:wizard runat="server" DisplaySideBar="true">
    <wizardsteps>
        <asp:wizardstep runat="server" steptype="auto" id="step1">First step</asp:wizardstep>
        <asp:wizardstep runat="server" steptype="auto" id="step2">Second step</asp:wizardstep>
        <asp:wizardstep runat="server" steptype="auto" id="finish">Final step</asp:wizardstep>
    </wizardsteps>
</asp:wizard>
```

The navigation bar consists of auto-generated buttons that provide any needed functionality—typically, going to the next or previous step or finishing. You can modify the look and feel of the navigation bar by using styles and templates.

The optional sidebar is used to display content in the left side of the control. It provides an overall view of the steps needed to accomplish the wizard’s task. By default, it displays a description of each step, with the current step displayed in boldface type. You can customize the sidebar using styles and templates. Figure 6-15 shows the default user interface. Each step is labeled using the ID of the corresponding `<asp:wizardstep>` tag.

![Simple Wizard](image)

**FIGURE 6-15** A wizard with the default sidebar on the left side.
Wizard Styles and Templates

You can style all the various parts and buttons of a Wizard control by using the properties listed in Table 6-4.

**TABLE 6-4 The Wizard Control’s Style Properties**

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CancelButtonStyle</td>
<td>Sets the style properties for the wizard’s Cancel button</td>
</tr>
<tr>
<td>FinishCompleteButtonStyle</td>
<td>Sets the style properties for the wizard’s Finish button</td>
</tr>
<tr>
<td>FinishPreviousButtonStyle</td>
<td>Sets the style properties for the wizard’s Previous button when at the finish step</td>
</tr>
<tr>
<td>HeaderStyle</td>
<td>Sets the style properties for the wizard’s header</td>
</tr>
<tr>
<td>NavigationButtonStyle</td>
<td>Sets the style properties for navigation buttons</td>
</tr>
<tr>
<td>NavigationStyle</td>
<td>Sets the style properties for the navigation area</td>
</tr>
<tr>
<td>SideBarButtonStyle</td>
<td>Sets the style properties for the buttons on the sidebar</td>
</tr>
<tr>
<td>SideBarStyle</td>
<td>Sets the style properties for the wizard’s sidebar</td>
</tr>
<tr>
<td>StartStepNextButtonStyle</td>
<td>Sets the style properties for the wizard’s Next button when at the start step</td>
</tr>
<tr>
<td>StepNextButtonStyle</td>
<td>Sets the style properties for the wizard’s Next button</td>
</tr>
<tr>
<td>StepPreviousButtonStyle</td>
<td>Sets the style properties for the wizard’s Previous button</td>
</tr>
<tr>
<td>StepStyle</td>
<td>Sets the style properties for the area where steps are displayed</td>
</tr>
</tbody>
</table>

The contents of the header, sidebar, and navigation bar can be further customized with templates. Table 6-5 lists the available templates.

**TABLE 6-5 The Wizard Control’s Template Properties**

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FinishNavigationTemplate</td>
<td>Specifies the navigation bar shown before the last page of the wizard. By default, the navigation bar contains the Previous and Finish buttons.</td>
</tr>
<tr>
<td>HeaderTemplate</td>
<td>Specifies the title bar of the wizard.</td>
</tr>
<tr>
<td>SideBarTemplate</td>
<td>Used to display content in the left side of the wizard control.</td>
</tr>
<tr>
<td>StartNavigationTemplate</td>
<td>Specifies the navigation bar for the first view in the wizard. By default, it contains only the Next button.</td>
</tr>
<tr>
<td>StepNavigationTemplate</td>
<td>Specifies the navigation bar for steps other than first, finish, or complete. By default, it contains Previous and Next buttons.</td>
</tr>
</tbody>
</table>

In addition to using styles and templates, you can control the programming interface of the Wizard control through a few properties.
The Wizard’s Programming Interface

Table 6-6 lists the properties of the Wizard control, excluding style and template properties and properties defined on base classes.

**TABLE 6-6 Main Properties of the Wizard Control**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveStep</td>
<td>Returns the current wizard step object. The object is an instance of the WizardStep class.</td>
</tr>
<tr>
<td>ActiveStepIndex</td>
<td>Gets and sets the 0-based index of the current wizard step.</td>
</tr>
<tr>
<td>DisplayCancelButton</td>
<td>Toggles the visibility of the Cancel button. The default value is false.</td>
</tr>
<tr>
<td>DisplaySideBar</td>
<td>Toggles the visibility of the sidebar. The default value is false.</td>
</tr>
<tr>
<td>HeaderText</td>
<td>Gets and sets the title of the wizard.</td>
</tr>
<tr>
<td>SkipLinkText</td>
<td>The ToolTip string that the control associates with an invisible image, as a hint to screen readers. The default value is “Skip Navigation Links” and is localized based on the server’s current locale.</td>
</tr>
<tr>
<td>WizardSteps</td>
<td>Returns a collection containing all the WizardStep objects defined in the control.</td>
</tr>
</tbody>
</table>

A wizard in action is fully represented by its collection of step views and buttons. In particular, you’ll recognize the following buttons: StartNext, StepNext, StepPrevious, FinishComplete, FinishPrevious, and Cancel. Each button is characterized by properties to get and set the button’s image URL, caption, type, and destination URL after a click. The name of a property is the name of the button followed by a suffix. The available suffixes are listed in Table 6-7.

**TABLE 6-7 Suffix of Button Properties**

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonImageUrl</td>
<td>Gets and sets the URL of the image used to render the button</td>
</tr>
<tr>
<td>ButtonText</td>
<td>Gets and sets the text for the button</td>
</tr>
<tr>
<td>ButtonType</td>
<td>Gets and sets the type of the button: push button, image, or link button</td>
</tr>
<tr>
<td>DestinationPageUrl</td>
<td>Gets and sets the URL to jump to once the button is clicked</td>
</tr>
</tbody>
</table>

Note that names in Table 6-7 do not correspond to real property names. You have the four properties in this table for each distinct type of wizard button. The real name is composed by the name of the button followed by any of the suffixes—for example, CancelButtonText, FinishCompleteDestinationPageUrl, and so on.

The Wizard control also supplies a few interesting methods—for example, GetHistory, which is defined as follows:

```csharp
public ICollection GetHistory()
```
GetHistory returns a collection of WizardStepBase objects. The order of the items is determined by the order in which the wizard’s pages were accessed by the user. The first object returned—the one with an index of 0—is the currently selected wizard step. The second object represents the view before the current one, and so on.

The second method, MoveTo, is used to move to a particular wizard step. The method’s prototype is described here:

```csharp
public void MoveTo(WizardStepBase step)
```

The method requires you to pass a WizardStepBase object, which can be problematic. However, the method is a simple wrapper around the setter of the ActiveStepIndex property. If you want to jump to a particular step and not hold an instance of the corresponding WizardStep object, setting ActiveStepIndex is just as effective.

Table 6-8 lists the key events in the life of a Wizard control in an ASP.NET page.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveViewChanged</td>
<td>Raised when the active step changes</td>
</tr>
<tr>
<td>CancelButtonClick</td>
<td>Raised when the Cancel button is clicked</td>
</tr>
<tr>
<td>FinishButtonClick</td>
<td>Raised when the Finish Complete button is clicked</td>
</tr>
<tr>
<td>NextButtonClick</td>
<td>Raised when any Next button is clicked</td>
</tr>
<tr>
<td>PreviousButtonClick</td>
<td>Raised when any Previous button is clicked</td>
</tr>
<tr>
<td>SideBarButtonClick</td>
<td>Raised when a button on the sidebar is clicked</td>
</tr>
</tbody>
</table>

As you can see, there’s a common click event for all Next and Previous buttons you can find on your way. A Next button can be found on the Start page as well as on all step pages. Likewise, a Previous button can be located on the Finish page too. Whenever a Next button is clicked, the page receives a NextButtonClick event; whenever a Previous button is clicked, the control raises a PreviousButtonClick event.

Adding Steps to a Wizard

A WizardStep object represents one of the child views that the wizard can display. The WizardStep class ultimately derives from View and adds just a few public properties to it. A View object represents a control that acts as a container for a group of controls. A view is hosted within a MultiView control. (See Chapter 4.) To create its output, the wizard makes internal use of a MultiView control. However, the wizard is not derived from the MultiView class.
You define the views of a wizard through distinct instances of the `WizardStep` class, all grouped under the `<WizardSteps>` tag. The `<WizardSteps>` tag corresponds to the `WizardSteps` collection property exposed by the `Wizard` control:

```xml
<wizardSteps>
    <asp:WizardStep>...
    ...
    </asp:WizardStep>
</wizardSteps>
```

Each wizard step is characterized by a title and a type. The `Title` property provides a brief description of the view. This information is not used unless the sidebar is enabled. If the sidebar is enabled, the title of each step is used to create a list of steps. If the sidebar is enabled but no title is provided for the various steps, the ID of the `WizardStep` objects is used to populate the sidebar, as shown earlier in Figure 6-15.

While defining a step, you can also set the `AllowReturn` property, which indicates whether the user is allowed to return to the current step from a subsequent step. The default value of the property is `true`.

**Types of Wizard Steps**

The `StepType` property indicates how a particular step should be handled and rendered within a wizard. Acceptable values for the step type come from the `WizardStepType` enumeration, as listed in Table 6-9.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto</td>
<td>The default setting, which forces the wizard to determine how each contained step should be treated.</td>
</tr>
<tr>
<td>Complete</td>
<td>The last page that the wizard displays, usually after the wizard has been completed. The navigation bar and the sidebar aren’t displayed.</td>
</tr>
<tr>
<td>Finish</td>
<td>The last page used for collecting user data. It lacks the Next button, and it shows the Previous and Finish buttons.</td>
</tr>
<tr>
<td>Start</td>
<td>The first screen displayed, without a Previous button.</td>
</tr>
<tr>
<td>Step</td>
<td>All other intermediate pages, in which the Previous and Next buttons are displayed.</td>
</tr>
</tbody>
</table>

When the wizard is in automatic mode—the default type `Auto`—it determines the type of each step based on the order in which the steps appear in the source code. For example, the
first step is considered to be of type Start and the last step is marked as Finish. No Complete step is assumed. If you correctly assign step types to your wizard steps yourself, rather than use the Auto type, the order in which you declare your steps in the .aspx source is not relevant.

Creating an Input Step

The following code shows a sample wizard step used to collect the provider name and the connection string to connect to a database and search for some data. For better graphical results, the content of the step is encapsulated in a fixed-height <div> tag. If all the steps are configured in this way, users navigating through the wizard won’t experience sudden changes in the overall page size and layout.

```html
<asp:wizardstep ID="Wizardstep1" runat="server" title="Connect">
    <div>
        <table>
            <tr><td>Provider</td><td>
                <asp:textbox runat="server" id="ProviderName" text="System.Data.SqlClient" />
            </td></tr>
            <tr><td>Connection String</td><td>
                <asp:textbox runat="server" id="ConnString" text="SERVER=(local);DATABASE=northwind... " />
            </td></tr>
            <tr height="100px"></tr>
        </table>
    </div>
</asp:wizardstep>
```

Figure 6-16 shows a preview of the step. As you can guess, the step is recognized as a Start step. As a result, the wizard is added only to the Next button.

![A sample Start wizard step.](image)
A wizard is usually created for collecting input data, so validation becomes a critical issue. You can validate the input data in two nonexclusive ways—using validators and using transition event handlers.

The first option involves placing validator controls in the wizard step. This guarantees that invalid input—empty fields or incompatible data types—is caught quickly and, optionally, already on the client.

```<asp:requiredfieldvalidator ID="RequiredField1" runat="server"
text="*"
errormessage="Must indicate a connection string"
setfocusonerror="true"
controltovalidate="ConnString" /></asp:requiredfieldvalidator>```

If you need to access server-side resources to validate the input data, you’re better off using transition event handlers. A transition event is an event the wizard raises when it is about to switch to another view. For example, the `NextButtonClick` event is raised when the user clicks the Next button to jump to the subsequent step. You can intercept this event, do any required validation, and cancel the transition if necessary. We’ll return to this topic in a moment.

**Defining the Sidebar**

The sidebar is a left-side panel that lists buttons to quickly and randomly reach any step of the wizard. It’s a sort of quick-launch menu for the various steps that form the wizard. You control the sidebar’s visibility through the Boolean `DisplaySideBar` attribute and define its contents through the `SideBarTemplate` property.

Regardless of the template, the internal layout of the sidebar is not left entirely to your imagination. In particular, the `<SideBarTemplate>` tag must contain a `DataList` control with a well-known ID—`SideBarList`. In addition, the `<ItemTemplate>` block must contain a button object with the name of `SideBarButton`. The button object must be any object that implements the `IButtonControl` interface.

**Note** For better graphical results, you might want to use explicit heights and widths for all steps and the sidebar as well. Likewise, the push buttons in the navigation bar might look better if they are made the same size. You do this by setting the `Width` and `Height` properties on the `NavigationButtonStyle` object.

**Navigating Through the Wizard**

When a button is clicked to move to another step, an event is fired to the hosting page. It’s up to you to decide when and how to perform any critical validation, such as deciding whether or not conditions exist to move to the next step.
In most cases, you’ll want to perform server-side validation only when the user clicks the Finish button to complete the wizard. You can be sure that whatever route the user has taken within the wizard, clicking the Finish button will complete it. Any code you bind to the FinishButtonClick event is executed only once, and only when strictly necessary.

By contrast, any code bound to the Previous or Next button executes when the user moves back or forward. The page posts back on both events.

Filtering Page Navigation with Events

You should perform server-side validation if what the user can do next depends on the data he or she entered in the previous step. This means that in most cases you just need to write a NextButtonClick event handler:

```html
<asp:wizard runat="server" id="QueryWizard"
    OnNextButtonClick="OnNext">
    ...
</asp:wizard>
```

If the user moves back to a previously visited page, you can usually ignore any data entered in the current step and avoid validation. Because the user is moving back, you can safely assume he or she is not going to use any fresh data. When a back movement is requested, you can assume that any preconditions needed to visit that previous page are verified. This happens by design if your users take a sequential route.

If the wizard’s sidebar is enabled, users can jump from page to page in any order. If the logic you’re implementing through the wizard requires that preconditions be met before a certain step is reached, you should write a SideBarButtonClick event handler and ensure that the requirements have been met.

A wizard click event requires a WizardNavigationEventHandler delegate (which is defined for you by ASP.NET):

```csharp
public delegate void WizardNavigationEventHandler(
    object sender,
    WizardNavigationEventArgs e);
```

The WizardNavigationEventArgs structure contains two useful properties that inform you about the 0-based indexes of the page being left and the page being displayed. The CurrentStepIndex property returns the index of the last page visited; the NextStepIndex returns the index of the next page. Note that both properties are read-only.

The following code shows a sample handler for the Next button. The handler prepares a summary message to show when the user is going to the Finish page.

```csharp
...
void OnNext(object sender, WizardNavigationEventArgs e)
{
    // Collect the input data if going to the last page
    // -1 because of 0-based indexing, add -1 if you have a Complete page
    if (e.NextStepIndex == QueryWizard.WizardSteps.Count - 2)
        PrepareFinalStep();
}
void PrepareFinalStep()
{
    string cmdText = DetermineCommandText();

    // Show a Ready-to-go message
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat("You're about to run: \n\n{0}\n\nReady to go?\n", cmdText);
    ReadyMsg.Text = sb.ToString();
}

string DetermineCommandText()
{
    // Generate and return command text here
}

Each page displayed by the wizard is a kind of panel (actually, a view) defined within a parent control—the wizard. This means that all child controls used in all steps must have a unique ID. It also means that you can access any of these controls just by name. For example, if one of the pages contains a text box named, say, ProviderName, you can access it from any event handler by using the ProviderName identifier.

The preceding code snippet is an excerpt from a sample wizard that collects input and runs a database query. The first step picks up connection information, whereas the second step lets users define tables, fields, and optionally a WHERE clause. The composed command is shown in the Finish page, where the wizard asks for final approval. (See Figure 6-17.)

The full source code of the wizard is in the companion code for this book.

![Two successive pages of the sample wizard: query details and the Finish step.](image)

**FIGURE 6-17** Two successive pages of the sample wizard: query details and the Finish step.
Part I Building an ASP.NET Page

Canceling Events

The WizardNavigationEventArgs structure also contains a read/write Boolean property named Cancel. If you set this property to true, you just cancel the ongoing transition to the destination page. The following code shows how to prevent the display of the next step if the user is on the Start page and types in sa as the user ID:

```csharp
void OnNext(object sender, WizardNavigationEventArgs e)
{
    if (e.CurrentStepIndex == 0 &&
        ConnString.Text.IndexOf("UID=sa") > -1)
    {
        e.Cancel = true;
        return;
    }
}
```

You can cancel events from within any transition event handler and not just from the NextButtonClick event handler. This trick is useful to block navigation if the server-side validation of the input data has failed. If you do cause a step to fail, though, you're responsible for showing some feedback to the user.

---

Finalizing the Wizard

All wizards have some code to execute to finalize the task. If you use the ASP.NET Wizard control, you place this code in the FinishButtonClick event handler. Figure 6-18 shows the final step of a wizard that completed successfully.

```csharp
void OnFinish(object sender, WizardNavigationEventArgs e)
{
    string finalMsg = "The operation completed successfully."
    try
    {
        // Complete the wizard (compose and run the query)
        string cmd = DetermineCommandText();
        DataTable table = ExecuteCommand(ConnString.Text, cmd);
        grid.DataSource = table;
        grid.DataBind();
        // OK color
        FinalMsg.ForeColor = Color.Blue;
    }
```
catch (Exception ex) {
    FinalMsg.ForeColor = Color.Red;
    finalMsg = String.Format("The operation cannot be completed
due to: <br />{0}", ex.Message);
}
finally {
    FinalMsg.Text = finalMsg;
}

string DetermineCommandText()
{
    // Generate and return command text here
}

DataTable ExecuteCommand()
{
    // Execute database query here
}

FIGURE 6-18 The final step of a wizard that completed successfully.

If the wizard contains a Complete step, that page should be displayed after the Finish button is clicked and the final task has completed. If something goes wrong with the update, you should either cancel the transition to prevent the Complete page from even appearing or adapt the user interface of the completion page to display an appropriate error message. Which option you choose depends on the expected behavior of the implemented operation. If the wizard’s operation can fail or succeed, you let the wizard complete and display an error message in case something went wrong. If the wizard’s operation must complete successfully unless the user quits, you should not make the transition to the Complete page; instead, provide users with feedback on what went wrong and give them a chance to try again.
Conclusion

Since version 1.0, ASP.NET has been characterized by a well-balanced mix of low-level and feature-rich tools. Using low-level tools such as events, HTTP modules, and HTTP handlers, you can plug into the ASP.NET pipeline to influence the processing of requests at every stage. At the same time, ASP.NET offers a wealth of feature-rich components for those who don't need control over every little step.

Starting with ASP.NET 2.0, the quantity and quality of application services has grown significantly. The introduction of rich composition tools for building pages like the ones we examined in this chapter is just a confirmation of the trend. In this chapter, we explored master pages to create content pages based on a predefined template made of graphics and, optionally, code. Master pages are not pure object-oriented visual inheritance à la Windows Forms; instead, they benefit from aggregation and let derived pages personalize well-known regions of the master. With full support from the Visual Studio environment, master pages are a time-saving feature that brings concrete added value to ASP.NET solutions.

Likewise, themes let developers code pages and controls that allow users to apply skins at will. ASP.NET themes work like Windows XP themes, and overall they're a superset of CSS that covers control properties in addition to HTML element styles. Themes work well in conjunction with the user profile API we discussed in Chapter 5. Using both, developers can let end users choose the theme and persist its name back to the personalization storage layer.

Finally, wizards are made-to-measure controls to quickly and efficiently write multistep input forms that divide complex operations into simple steps.

With this chapter, we completed the first part of the book, dedicated to building ASP.NET pages. With the next chapter, we approach the world of data access and explore ways to add data to a Web site.

Just the Facts

- A master page is a distinct file referenced at the application or page level that contains the static layout of the page.
- A master page contains regions that each derived page can customize.
- A derived page, or named content page, is a collection of markup blocks that the runtime will use to fill the regions in the master page.
- Content pages can't contain information other than contents for the master's placeholders.
- Regions in the master page can have a default content to be used if the content page doesn't provide any.
You can define various masters for a page and have the system automatically pick up a particular one based on the browser’s user agent string.

Master pages can be nested and expose a strongly-typed object model.

Themes are a collection of settings spread over various files that the ASP.NET runtime uses to give the whole site (or page) a consistent user interface.

Themes become a kind of attribute and can be exported from one application to the next and applied to pages on the fly.

Themes differ from CSS files because they let you style ASP.NET control properties and not just HTML elements.

A theme contains skin files, CSS files, images, plus any other auxiliary file you might find useful.

A skin file is a collection of ASP.NET control declarations. The system ensures that after instantiation each control of that type in the page will have exactly the same set of attributes.

The wizard control manages multiple views inside a single control and provides an auto-generated user interface for you to move back and forth like in a desktop wizard.