An Introduction to ASP.NET 2.0

When Microsoft released the .NET Framework 1.0 Technology Preview in July 2000, it was immediately clear that Web development was going to change. The company’s then current technology, Active Server Pages 3.0 (ASP), was powerful and flexible, and it made the creation of dynamic Web sites easy. ASP spawned a whole series of books, articles, Web sites, and components, all to make the development process even easier. What ASP didn’t have, however, was an application framework; it was never an enterprise development tool. Everything you did in ASP was code oriented—you just couldn’t get away without writing code.

ASP.NET was designed to counter this problem. One of its key design goals was to make programming easier and quicker by reducing the amount of code you have to create. Enter the declarative programming model, a rich server control hierarchy with events, a large class library, and support for development tools from the humble Notepad to the high-end Visual Studio .NET. All in all, ASP.NET was a huge leap forward.

What’s Wrong with ASP.NET 1.x?

So if ASP.NET 1.0 and 1.1 are so great, what’s wrong with them? Well, nothing, actually, but when developing software there is always a trade-off between how much can be done, how many resources you have, and how
much time you have to do it. There is an almost never-ending supply of features you can add, but at some stage you have to ship the product. You cannot doubt that ASP.NET 1.0 shipped with an impressive array of features, but the ASP.NET team members are ambitious, and they not only had plans of their own but also listened to their users.

ASP.NET 2.0, code-named “Whidbey,” addresses the areas that both the development team and users wanted to improve. The aims of the new version are listed below.

- **Reduce the number of lines of code required by 70%**. The declarative programming model freed developers from having to write reams of code, but there are still many scenarios where this cannot be avoided. Data access is a great example, where the same Connection, DataAdapter/DataSet, and Command/DataReader code is used regularly.

- **Increase developer productivity**. This partly relates to reducing the amount of code required but is also affected by more server controls encompassing complex functionality, as well as providing better solutions for common Web site scenarios (such as portals and personalized sites).

- **Use a single control set for all devices**. Mobile devices are becoming more pervasive, with an increasing number of new devices. Many of the server controls render appropriately for small screens, but there are two major problems with the current support for mobile devices: (1) having a separate set of server controls purely for mobile devices is not only confusing but also costly, and (2) adding support for new devices requires additional development work and maintenance. ASP.NET 2.0 will provide a single set of controls and an extensible architecture to allow them (and other controls) to support multiple devices.

- **Provide the fastest Web server platform**. Although ASP.NET 1.0 offers a fast server platform, ASP.NET 2.0 will improve areas such as application start-up times and provide better application tracing and performance data. Innovative caching features will enhance application performance, especially when SQL Server is used.

- **Provide the best hosting solution**. With the large number of Internet applications being hosted, it’s important to provide better solutions for hosters. For example, better management features to
identify and stop rogue applications will give hosters more control over their current environment. More control can also be given to hosted companies by use of the new Web-based administration tool, allowing users to easily control the configuration of applications remotely.

- **Provide easier and more sophisticated management features.** Administration of ASP.NET applications under version 1.x required manual editing of the XML configuration file, which is not a great solution for administrators. Version 2.0 brings a graphical user interface–based administration tool that is integrated with the Internet Information Services (IIS) administration tool.

- **Ease implementation of entire scenarios.** The better management features are built on top of a management application programming interface (API), allowing custom administration programs to be created. Along with application packaging this will provide support for easily deployable applications, with or without source.

Even from this broad set of aims you can see that ASP.NET 2.0 is a great advance from 1.x for both developers and administrators.

**New Features**

This chapter isn’t an in-depth look at any specific feature—instead we are going to give you a taste of what’s to come so you can see how much easier Web development is going to be. For this outlook we’ve broken down the new features into rough end-to-end scenarios.

**Templates for a Consistent Look and Feel**

ASP.NET 1.x provides an easy way to develop Web sites, but one thing that has become apparent is the lack of an architecture for applying a consistent look and feel. Several workaround techniques emerged:

- Creating a custom class object that inherits from Page and having this custom page preload controls
- Creating a templated server control, where the templates provide the layout areas for each page, and using this control on every page
- Having User Controls for common areas of the site, such as headings, menus, and footers
Of these, the first two require knowledge of creating server controls, and while this is a topic most ASP.NET developers could master, it may not be one they’ve had experience with. Therefore a solution using custom server controls tends to be avoided. The last option, though, is a simple solution, easy to create and implement. User Controls were created to provide reusable functionality, and this is a great use for them. However, to apply a consistent look and feel you need to first place the User Controls on each page, then ensure that they are placed in the same place on each page. In other words, you really need a page template, and in reality this manifests itself as an ASP.NET file that you simply copy for each new page. The danger of this approach is that it’s too easy to modify a page and change the layout for that single page.

**Figure 1.1. Combining a master page and a child page**
To provide a templating solution, ASP.NET 2.0 has the concept of master pages, which provide a template for the look and implementation of a page. A master page is an ASP.NET page that provides a template for other pages, giving shared page-level layout and functionality. The master page defines placeholders for the content, which can be overridden by child pages. The resultant page is a combination of the master page and the child page, as shown in Figure 1.1.

Master pages are covered in Chapters 2 and 5.

Styles for Controls
The second major feature of ASP.NET 2.0 that deals with the look and feel of a site is that of themes. Theming, or skinning, has become very popular, allowing users to create a customized look for applications. On the Windows desktop two of the most popular themed applications are audio players (WinAmp and Windows Media Player), and with some additional software, even Windows XP can be themed.

The popularity of theming is due to the nature of humans—we like to choose the way things look, and we like to express our individuality. This is easy on the desktop, where users generally have a single machine each. With Web sites, however, theming becomes a harder issue because of the number of users. Tracking which users have which themes and managing those themes becomes an overhead that site administrators don’t want to get involved with.

Some Web sites provide forms of theming, but these are relatively limited in terms of customization, perhaps allowing only a choice of color scheme. Other sites provide a selection of stylesheets for users to pick from, assuming their browsers support this feature, or alternatively change the stylesheet on the server. This allows not only color schemes to be selected but also complete style choices, such as fonts, style of borders, and so on.

In ASP.NET 2.0 the goals for theming are quite simple.

- Make it simple to customize the appearance of a site or page, using the same design tools and methods used when developing the page itself. This means there’s no need to learn any special tools or techniques to add themes to a site.
• Allow themes to be applied to controls, pages, and even entire sites. For example, this allows users to customize parts of a site while ensuring that other parts (such as corporate identity) aren’t customized.
• Allow all visual properties to be customized, thus ensuring that when themed, pages and controls can achieve a consistent style.

The implementation of this in ASP.NET 2.0 is built around two areas: skins and themes. A skin is a set of properties and templates that can be applied to controls. A theme is a set of skins and any other associated files (such as images or stylesheets). Skins are control specific, so for a given theme there could be a separate skin for each control within that theme. Any controls without a skin inherit the default look. The implementation is simple because a skin uses the same definition as the server control it is skinning, and themes are just a set of files in a directory under the application root. For example, consider the sample directory structure shown below:

default.aspx
Themes
  \MyTheme
    MySkin.skin
  \YourTheme
    YourSkin.skin

Each theme consists of a directory under the Themes directory. Within each theme there is a file with a .skin suffix, which contains the skin details for that theme. For example, MySkin.skin might contain:

<asp:Label SkinID="Normal" runat="server"
  Font-Bold="True" BackColor="#FFC080" />
<asp:Label SkinID="Comic" runat="server"
  Font-Italic="True" Font-Names="Comic Sans MS" />

This defines two skins for the Label control, each with different visual properties. The theme can be chosen by setting a page-level property, and the skin is chosen by setting a control-level property, as demonstrated below.

<%@ Page Theme="MyTheme" %>
<form runat="server">
  <asp:Label SkinID="Comic" Text="A Label" />
</form>
Both of these can be set at runtime as well as design time, so this provides an extremely powerful solution, especially when connected with the new Personalization features.

Personalization and themes are covered in Chapter 7.

**Securing Your Site**

With the large amount of business being done on the Web, security is vitally important for protecting not only confidential information such as credit card numbers but also users’ personal details and preferences. Thus you have to build into your site features to authenticate users. This was easy to do in ASP.NET 1.x, although you still had to write code. Security was created by picking your preferred security mechanism (most often Forms Authentication) and then adding controls to your page to provide the login details—user name, password, “remember me” checkbox, and so on. There was no built-in mechanism for storing personal details, so this was a roll-it-yourself affair.

With ASP.NET 2.0, the pain has been taken out of both areas. For login functionality, there is now:

- A **Login** control, providing complete functionality for logging into a site
- A **LoginStatus** control, which indicates the login status and can be configured to provide automatic links to login and logout pages
- A **LoginName** control to display the current (or anonymous) name
- A **LoginView** control, providing templated views depending on the login status
- A **PasswordRecovery** control, encompassing the “I forgot my password” functionality

For example, to add login features to your page all you need to do is add the following code:

```html
<form runat="server">
  <asp:Login runat="server" />
</form>
```
This gives us the simple interface shown in Figure 1.2.

This could be achieved easily in previous versions of ASP.NET, but not with such simplicity. You needed labels, text entry boxes, buttons, and validation, whereas it’s now all rolled into one control. Sure it looks raw, but this is the basic unformatted version. Using the design tool Visual Studio .NET (more on that in Chapter 2), you can auto-format this for a better look. You can also skin the interface, as shown in Figure 1.3, or even template it to provide your own customized look. Along with the other login controls you get a complete solution for handling user logins.

The user interface isn’t the only part of logging into a site; there’s also the code needed to validate the user against a data store. With ASP.NET 1.x this required not only code to be written but also knowledge of what that data store was and how it stored data. ASP.NET 2.0 introduces a new Membership API, whose aim is to abstract the required membership functionality from the storage of the member information. For example, all of the data handling we’d have done in previous versions to validate a user can now be replaced with the code shown in Listing 1.1.

**Listing 1.1. Validating User Credentials**

```vbnet
Sub Login_Click(Sender As Object, E As EventArgs)
    If Membership.ValidateUser(Email.Text, Password.Text) Then
        FormsAuthentication.RedirectFromLoginPage(Email.Text, False)
    Else
        LoginMessage.Text = "Invalid credentials. Please try again."
    End If
End Sub
```

**Figure 1.2. The Login control**

**Figure 1.3. A skinned Login control**
What’s even better is that when using the Login control you don’t even have to do this—the control handles it for you.

The great strength of the Membership API is that it is built on the idea of Membership Providers, with support for Microsoft SQL Server and Access supplied by default. To integrate custom membership stores you simply need to provide a component that inherits from the Membership interface and add the new provider details to the configuration file.

The Membership API has some simple goals.

- Offer an easy solution for authenticating and managing users, requiring no knowledge of the underlying storage mechanism.
- Provide support for multiple data providers, allowing data stored about users to come from different data stores.
- Provide comprehensive user management in a simple-to-use API, giving an easy way for developers to store and access user details.
- Give users a unique identity, allowing integration with other services such as the Personalization and Role Manager features.

Security, membership, and role management are covered in Chapter 6.

**Personalizing Your Site**

One of the areas driving changes on the Internet is that of communities. People like to belong, and the Internet is a big, lonely place. Community sites give you a home, a sense of belonging. Part of that comes from being in contact with like-minded people, and part comes from the features some of these sites offer. Our houses are decorated to our style, and many of us customize our Windows desktop, so why shouldn’t our favorite Web sites offer the same opportunity?

Hand in hand with the Membership API lie the Personalization features. These provide a simple programming model for storing user details (including those of anonymous users), with easy customization. Like Membership, Personalization can be configured to work with multiple data providers and provides an easy way to define custom properties for each user. This leads to a user profile with strong types, allowing easy access within ASP.NET pages. For example, you can create a profile with Name, Address, and Theme as properties and a page that allows the user to update them, as shown in Listing 1.2.
List 1.2. Using the Profile Custom Properties

```csharp
<script runat="server">
    Sub Page_Load(Sender As Object, E As EventArgs)
    
        Name.Text = Profile.Name
        Address.Text = Profile.Address
        Theme.Text = Profile.Theme
    
    End Sub

    Sub Update_Click(Sender As Object, E As EventArgs)
    
        Profile.Name = Name.Text
        Profile.Address = Address.Text
        Profile.Theme = Theme.Text
    
    End Sub

    </script>

<form runat="server">
    Name:  <asp:TextBox id="Name" runat="server" /> <br />
    Address: <asp:TextBox id="Address" runat="server" /> <br />
    Theme:  <asp:TextBox id="Theme" runat="server" /> <br />
    <asp:Button Text="Update" onClick="Update_Click" runat="server" />
</form>

The simplicity of this method means we only have to deal with the user profile. We don’t need to know how it stores the data—we just deal with the properties each profile has. This personalization also allows us to easily use the theming capabilities, changing the theme when the page is created, as demonstrated below.

Sub Page_PreInit(Sender As Object, E As EventArgs)

    Me.Theme = Profile.Theme

End Sub

To ensure that the theme customization is applied before the controls are created we use the new PreInit event.

Personalization is covered in Chapter 7.
Creating Portals

As if customization of a site’s look weren’t enough, ASP.NET 2.0 also brings a way to alter the structure with its new portal framework.

The success of the ASP.NET IBuySpy portal application and its offshoots shows that customized sites are popular. The trouble has always been how to provide a consistent look while still allowing user customization not only of the style but also of the content and placement of content. Microsoft has already implemented solutions to provide this functionality, including SharePoint Server and Outlook Web Parts.

In ASP.NET 2.0, Web Parts become the underlying technology for all Microsoft portal applications, allowing a single easy-to-use, extensible framework. The concept revolves around two key controls—the WebPartZone and the WebPart. The WebPartZone identifies areas on the page in which content can be changed, and the WebPart identifies the part (or module) within the zone. There are different types of WebPart controls for different purposes, for example:

- ContentWebPart, for arbitrary controls and content
- CatalogPart, which contains a catalog of parts not currently on the page
- EditorPart controls, such as AppearanceEditorPart and LayoutEditorPart, allowing customization of the parts

For example, consider an intranet site that needs a selection of areas of content—links, announcements, and so on. Figure 1.4 shows a sample site.

This site has two main areas of content—the left area with the welcome message and the announcements, and the right area showing weather and links. Each of these main areas is a WebPartZone and the content with them a ContentWebPart. The code for this page appears in Listing 1.3.

Listing 1.3. Sample Intranet Site Using Web Parts

```xml
<table>
  <tbody>
    <tr>
      <td valign="top" align="left">
        <asp:WebPartZone id="LeftZone" runat="server" Title="Left Zone" partFrameType="TitleOnly" lockLayout="False" borderColor="White">
          <ZoneTemplate>
            <asp:ContentWebPart id="ContentWebPart1" title="Welcome" runat="server">
              <ContentTemplate>
                1.2 NEW FEATURES continues
              </ContentTemplate>
            </asp:ContentWebPart>
          </ZoneTemplate>
        </asp:WebPartZone>
      </td>
    </tr>
  </tbody>
</table>
```
This project ...

Figure 1.4. Sample intranet site using the portal framework
Here you can see two WebPartZone controls separating the left and right content. Within each there is a mixture of content, including static text, user controls, and custom server controls.

At first glance this doesn’t look like much improvement over existing layout methods such as user controls—in fact, it looks more complex. However, the framework on which Web Parts is built is great for developers and users alike. Developers only have to drop user controls or server controls into a ZoneTemplate to automatically receive Web Parts functionality. To enhance this functionality you can add verbs to the WebPartZone to indicate which features the framework should add to each part within the template. Listing 1.4 shows an example.

Listing 1.4. Web Part Verbs

```xml
<WebPartCloseVerb checked="False"
    imageUrl="images/CloseVerb.gif"
    enabled="True" text="Close"
    description="Closes the WebPart"
    visible="True" />

<WebPartRestoreVerb checked="False"
    imageUrl="images/RestoreVerb.gif"
    enabled="True" text="Restore"
    description="Restores the WebPart"
    visible="True" />

<WebPartMinimizeVerb checked="False"
    imageUrl="images/MinimizeVerb.gif"
    enabled="True" text="Minimize"
    description="Minimizes the WebPart"
    visible="True" />

<WebPartHelpVerb checked="False"
    enabled="True" text="Help"
    description="Shows help for the WebPart"
    visible="True" />

<WebPartEditVerb checked="False"
    imageUrl="images/EditVerb.gif"
    enabled="True" text="Edit"
    description="Edits the WebPart"
    visible="True" />
```

Here there are verbs that allow minimizing and maximizing the WebPart controls, editing, help, and so on.

For the user, the Personalization features allow each WebPart to be moved to other WebPartZone controls or edited. For example, moving a
WebPart is simply a matter of drag and drop, as shown in Figure 1.5, where the Daily Links section is being moved to the Left Zone.

Editing of WebPart controls is also part of the portal framework, where by default the user can alter the title, height, width, and frame style. Each WebPart can also provide custom properties that can be edited. For example, the WebPart for the My Weather section allows the zip code to be set, upon which the weather for that zip code is displayed. Editing of WebPart controls is provided by the inclusion of an EditorZone, which details what can be edited. For example, for our sample intranet site, the EditorZone might include the code shown in Listing 1.5.

**Listing 1.5. Adding Editor Parts to an EditorZone**

```xml
<asp:EditorZone id="EditorZone1" title="Edit WebParts" runat="server" partFrameType="TitleOnly" cssClass="EditorZoneBody">
  <ZoneTemplate>
    <asp:AppearanceEditorPart id="AppearanceEditorPart1" runat="server" font-size="8pt" font-bold="true" />
    <asp:PropertyGridEditorPart id="PropertyGridEditorPart1" title="Custom Settings" font-size="8pt" runat="server" />
  </ZoneTemplate>
</asp:EditorZone>
```

Figure 1.5. Dragging a **WebPart** to another location
This indicates that there are two editor parts—one for the appearance, and one for the property grid—for properties of the WebPart that are marked as personalizable. Selecting the edit button invokes the editing features and the EditorZone is made visible, as shown in Figure 1.6. Once edited for zip code 02116, the WebPart shows the weather for Boston (Figure 1.7).

The portal framework is covered in Chapter 8.

**Figure 1.6. Editing the My Weather WebPart**

**Figure 1.7. The My Weather WebPart**

**Setting Up and Managing Your Site**

ASP.NET 1.x made deployment of Web sites easy with its xcopy deployment model. This removed the need for some administrative tasks, such as registering COM components, but still left other tasks, such as site administration, as more manual affairs. The XML-based configuration file obeyed the xcopy rule, but there are three major problems with it. First, there is no easy-to-use administration tool, meaning you must have knowledge of the
XML schema before you can modify it. Second, you need some way to actually fetch the file, edit it, and then upload it. This is a problem particularly for hosted scenarios, where users are always remote, and administration of many sites can become a management nightmare. Finally, you cannot create a Web Application, which is required for sites that require security.

Three features in ASP.NET 2.0 help solve these issues. The first is the Microsoft Management Console (MMC) Snap-in for configuration, as shown in Figure 1.8.

The second feature is a Management API, providing a programmable interface to manage a site. For example, Listing 1.6 sets the authorization mode using the API.

**Listing 1.6. Setting the Authorization Mode**

```vbscript
Dim cfg As Configuration
Dim ms As AuthenticationSection

cfg = Configuration.GetConfigurationForUrl(Request.ApplicationPath)
ms = CType(cfg.GetConfigurationSection("system.web/authentication"), AuthenticationSection)
ms.Mode = HttpAuthenticationMode.Windows
cfg.Update()
```

Figure 1.8. ASP.NET Configuration MMC Snap-in
The Management API provides access to all areas of the configuration, both at the machine level (machine.config) and the application level (web.config). This allows utilities to be written not only to manage a single site but also to manage all sites.

The third aspect of site management is the creation of a Web-based tool, wrapping much of the Management API. This provides a simple way to remotely administer a site, as shown in Figure 1.9.

**Welcome to the Web Site Administration Tool**

*Application: /temp*

*Current User Name: HUNDRED-ACRE\DAVIDS*

This Web Administration Tool enables you to manage application security, add personalization, view reports, and select or change application data sources.

To close this tool and return to your Web site, click cancel.

**Figure 1.9. The Web management tool**

Here you have a simple Web interface that allows configuration of all aspects of a site. The interface is designed to be customized, so corporations and hosts can give it a company look.

Administration is covered in Chapter 13.

**Using Images on Your Site**

Using images isn’t a particularly difficult area of site design, but their use has been eased with two new server controls. First, the ImageMap control provides easy support for image maps, as demonstrated on the next page.
<asp:ImageMap runat="server"
    onClick="Map_Click"
    ImageUrl="images/states.jpg">
    <asp:CircleHotSpot X="100" Y="100" Radius="25"
        Value="Other State" />
    <asp:RectangleHotSpot Top="200" Left="150" Right="200" Bottom="150"
        Value="More State"/>
    <asp:PolygonHotSpot Coordinates="3,4, 15,18, 45,18, 15,70, 3,4"
        Value="State 1" />
</asp:PolygonHotSpot>

The detection of the hot spot is handled in the postback event:

Sub Map_Click(Sender As Object, E As ImageMapEventArgs)
    Select Case e.Value
    Case "State 1"
        ' ...
    Case "Other State"
        ' ...
    Case "More States"
        ' ...
    End Select
End Sub

The second new image-handling feature is that of dynamic images, designed specifically to render images appropriate to the calling browser. This is necessary because images displayed in Web browsers generally aren’t suitable for smaller devices, such as PDAs or phones. The new DynamicImage control uses an HttpHandler to sniff the browser type and render the appropriate image. For example, consider the following code:

<form runat="server">
    <asp:DynamicImage DynamicImageType="ImageFile"
        ImageFile="car.gif" runat="server" />  
</form>

For a standard Web browser the image is rendered as expected, but for a Wireless Access Protocol (WAP) phone, the image is rendered as a Wireless Bitmap (WBMP). This removes any need for the developer to specifically target images to browser types.

Images are covered in Chapter 12.
Using Data on Your Site

It’s probably no exaggeration to say that most, if not all, Web sites use some form of data to drive them. Whether XML files, a database, or another dynamic form of storage, the data allows a site to respond to the user and to be up to date. ASP.NET 1.x provided some great data binding capabilities, but they always involved code, often the same code used over and over. One of the key goals of ASP.NET 2.0 is to reduce code and to ease the use of databases, especially for beginner programmers. To achieve this a new set of data controls has been introduced, removing the need for in-depth knowledge of ADO.NET.

Data source controls provide a consistent and extensible method for declaratively accessing data from Web pages. There are several data source controls, including AccessDataSource, SqlDataSource, XmlDataSource, and ObjectDataSource, and it’s likely that others (perhaps for Excel and Exchange Server) will appear as ASP.NET 2.0 nears release, along with third-party data sources. The use of data controls is simple, as shown below.

```xml
<asp:SqlDataSource id="ds1" runat="server"
    ConnectionString="localhost;database=pubs;Trusted_Connection=True"
    SelectCommand="SELECT * FROM authors"/>
<asp:DataGrid DataSourceId="ds1" runat="server" />
```

This just encapsulates the code everyone used to put in the Page_Load event—it connects to the database, fetches the data, and binds the grid. The contents of the SelectCommand can be a stored procedure as well as a SQL command, thus preserving the separation of data access from the page itself. There are commands for updating, inserting, and deleting.

This model is extended by use of a parameter collection, allowing parameters to be passed into the command from a variety of sources. For example, the code in Listing 1.7 automatically takes the value from the TextBox control txtState and feeds this into the parameter @state.

**Listing 1.7. Using a ControlParameter**

```xml
<asp:SqlDataSource id="ds1" runat="server"
    ConnectionString="localhost;database=pubs;Trusted_Connection=True"
    SelectCommand="SELECT * FROM authors WHERE state=@state">
    <SelectParameters>
        <ControlParameter name="@state" ControlID="txtState" />
    </SelectParameters>
</asp:SqlDataSource>

<asp:TextBox id="txtState" runat="server" />
<asp:DataGrid DataSourceId="ds1" runat="server" />
```
There are also other parameter types, allowing parameter information to be taken directly from Session variables, Cookies, the Request (QueryString), and the HTML Form.

**Data Binding**

Data binding in ASP.NET 1.x was simple, but it did cause confusion in some areas. For example, should you use early binding, for which you have to know the underlying data structure? Or should you take the development shortcut and use late binding, like this:

```
<%# DataBinder.Eval(Container.DataItem, "au_lname") %>
```

With ASP.NET 2.0 this syntax has been simplified:

```
<%# Eval("au_lname") %>
```

There is also an equivalent XPath syntax for XPath expressions when binding to XML documents:

```
<%# XPath("@au_lname") %>
```

**Binding to Objects**

One of the most requested features has been the ability to bind data directly to objects. Good design dictates that you separate your data access layer from your presentation layer, and this is often done as a set of classes. The new `ObjectDataSource` allows you to simply bind directly to existing objects, such as classes, thus allowing you to have a strongly typed data layer but still participate in the easy data binding that ASP.NET 2.0 brings.

Data source controls and data binding are covered in Chapter 3.

**Adding Mobility Support**

Mobile devices are becoming more pervasive. It seems everyone has a mobile phone, many people have PDAs, and some great devices now combine the functionality of both. From the development perspective the problem with these devices is their screen size and rendering capabilities. Not only do many of them not accept HTML, but with their tiny screens some also can’t display images, tables, and so on.
In ASP.NET 1.x, the Microsoft Mobile Internet Toolkit (MMIT in version 1.0 and ASP.NET Mobile Controls in version 1.1) provided this support, including separate controls for building Web pages suitable for small-screen browsers. In ASP.NET 2.0, the MMIT is no longer required because mobile support is built into all controls. This reduces not only the amount of code required but also the need for specialist knowledge about mobile platforms. This might seem relatively unimportant while the number of sites that target mobile browsers is small, but this is bound to increase as the features of small devices improve and prices drop.

The really important part of the changes is to the infrastructure of the ASP.NET server controls. All controls are now built on a control adapter architecture, where there is an adapter for each specific device. The adapters have knowledge of each device and perform the rendering appropriate for its markup language and screen size. Since the controls are derived from adapters, they don’t need to perform any special action to choose what to render—the adapter intelligently renders the appropriate content based on the calling device. New devices are easily supported because they require only the addition of an adapter, which the controls can then take advantage of.

**Device Filters**

This architecture is taken further by allowing adapter-specific attributes for controls, enabling the page designer to provide different content for specific devices. For example, the following code shows how different text and cascading style sheet (CSS) styling can be defined for a mobile device.

```xml
<asp:Label id="MyLabel" runat="server"
    Text="Welcome to our site"
    Nokia:Text="Time to upgrade your Nokia phone!"
    cssClass="StandardStyleClass"
    Nokia:cssClass="SpecialNokiaStyleClass" />
```

**Device Templates**

Along with modified attributes, we also have the ability to provide templates for specific devices. We know that mobile devices have a small screen size, so repeated controls such as grids and lists either aren’t appropriate or need different output. By using specific templates for devices we can now provide different content to different devices, as shown in Listing 1.8.
Listing 1.8. Filtered Templates for Mobile Devices

<asp:Repeater runat="server">

    <HtmlBrowsers:HeaderTemplate>
        <table>
            <tr><td>UserName</td><td>Address</td><td>Phone</td></tr>
        </table>
    </HtmlBrowsers:HeaderTemplate>

    <HtmlBrowsers:ItemTemplate>
        <tr>
            <td><%-# Container.DataItem("UserName") %></td>
            <td><%-# Container.DataItem("Address") %></td>
            <td><%-# Container.DataItem("Phone") %></td>
        </tr>
    </HtmlBrowsers:ItemTemplate>

    <WmlBrowsers:ItemTemplate>
        <asp:Panel runat="server">
            <%-# Container.DataItem("UserName") %>
            <%-# Container.DataItem("Phone") %>
        </asp:Panel>
    </WmlBrowsers:ItemTemplate>

    <HtmlBrowsers:FooterTemplate>
        </table>
    </HtmlBrowsers:FooterTemplate>

</asp:Repeater>

These mechanisms provide a way for developers to override the built-in rendering for mobile devices. Along with automatic mobile support with the standard controls, there are controls specifically designed for mobile devices, such as PhoneLink (to launch a phone call) and Pager (to provide paging support). Standard controls also support the SoftKeyLabel attribute to allow specific text to be targeted to soft keys on phones.

Mobility is covered in Chapter 10.

Compilation and Deployment

Since the release of ASP.NET 1.0 there’s been a fairly standard approach to Web site architecture. In general there has been a separation of business logic into separate assemblies, often in a separate directory with a make file. Using Visual Studio .NET 1.0 and 2003 for this approach is fine since it provides
the compilation step for you, but stand-alone tools (such as Web Matrix) don’t, so you have to handcraft a batch file to make your assemblies.

ASP.NET 2.0 provides automatic compilation for satellite code by supporting a code directory. All files within this directory will be compiled on the first run, thus removing the need for separate compilation scripts. Files within the code directory don’t have to be just pure code, such as Visual Basic .NET or C# files. Support is also included for Web Services Description Language (WSDL) files and strongly typed DataSets (XSD) files. For WSDL files the proxy will automatically be created, and for XSD files the appropriate classes will be created.

Along with automatic compilation comes pre-compilation—an entire site (Web pages and code) can be pre-compiled. This not only provides a way to deploy compiled applications but also removes the performance hit taken by the compilation process on the first run. In addition, since only compiled files are deployed, intellectual property is protected.

Another automatic feature is that of resources, such as those used for globalization. The resources directory provides a place for these, which are included as part of the compilation process.

Compilation is covered in Chapter 2.

**Development Tools**

Having a whole raft of new features in ASP.NET is great, but what about design tools? Version 2.0 of the .NET Framework will introduce the latest version of Visual Studio .NET—Visual Studio .NET “Whidbey.” When ASP.NET 1.0 was released it quickly became apparent that a development tool targeted at Web developers was required. Visual Studio .NET provides great project and design features targeted at corporate developers. Web Matrix was released to appeal to ASP.NET developers who don’t have access to Visual Studio .NET. It’s a small stand-alone tool, specifically targeted at ASP.NET development, and provides some features that aren’t in Visual Studio .NET.

With ASP.NET 2.0, Visual Studio .NET “Whidbey” has undergone some major enhancements and now provides a far superior environment for developing Web applications than previous versions. While the design environment is very familiar, the feature set has improved, making it a premier Web development tool.
Key design features for Visual Studio .NET “Whidbey” include the following:

- Traditional in-line coding approach, plus a new code-behind model
- Support for all managed languages
- Ability to edit any file anywhere (FTP, File System, Front Page Extensions, and so on)
- Support for data controls, drag and drop, and database access, with a rich design surface
- Support for visual inheritance through master pages
- No project files, allowing projects to be manipulated outside of the tool
- Integrated Web Administration Tool
- IntelliSense included
- Debugging support
- No “build” step—ability to compile on first run

This feature set is really a combination of features from Visual Studio .NET and Web Matrix.

Visual Studio .NET “Whidbey” is covered in Chapter 2.

**SUMMARY**

Of course, there are many changes within ASP.NET 2.0—too many to mention in this introduction, although some highlights were covered in this chapter. The remainder of the book covers these changes (including items such as changes to existing controls, changes to page attributes, new controls, and so on) in detail.

It’s important to remember that this is a preview technology, still evolving and still in testing. Despite that, the initial feature set is extremely impressive and provides a leap in productivity for Web site developers.
THE LOOK AND FEEL OF A SITE can be its savior or its downfall, and there are plenty of books and Web sites that instruct in design and usability. This chapter discusses not site design but how ASP.NET 2.0 aids in the design and consistency of sites. From the development perspective, generating a site that is consistent isn’t so hard, and there are plenty of ways it can be achieved. However, these are all custom solutions, not part of the underlying .NET Framework. ASP.NET 2.0 brings a solution that not only improves ways of providing UI reuse but also aids in maintenance of the site.

Likewise, providing navigation within a site can be achieved easily, but you nearly always have to write code or buy a custom solution. ASP.NET 2.0 has a new framework that provides a simple and extensible solution for providing navigation.

Master Pages in Detail

In Chapter 1 we had a brief look at the idea of master pages, showing how they provide a template for all content pages. This provides a way to create a consistent look for a site, since the look is defined in the master page. Let’s refresh ourselves about how this works.

Figure 5.1 shows an example of two content pages using a master page. The master page defines the page layout—that is, the shared UI and code plus any default content. In this case it is the light shaded content at the
top and left, representing menus and other navigation features. The master page defines content areas using the `ContentPlaceHolder` control, and it is into these areas that content pages place their content (shown as dark shaded areas in the figure). Pages that use a master page to define the layout can place content only in the areas defined by the `ContentPlaceHolder`, thus enabling a consistent site design.

**Creating Master Pages**

In Visual Studio .NET “Whidbey,” creating master pages is simply a matter of selecting Master Page from the Add New Item dialog. The newly created master page is just an ASP.NET page with a different file extension (`.master`), so it fits with your existing knowledge. You don’t have to learn any new techniques, apart from the use of the `ContentPlaceHolder` control. Listing 5.1, for example, shows the contents of a master page newly added to a site.
You can see that this looks similar to existing ASP.NET pages and contains simple HTML and ASP.NET controls. The main difference between this page and a standard ASP.NET page is the use of the Master directive and the file suffix .master. The critical point is the use of the ContentPlaceHolder control, which dictates where content pages can insert content. The id attribute uniquely identifies the placeholder, allowing more than one placeholder to be in a master page. The master page can have code as well as content, allowing the master page to be dynamic.

Turning this empty master page into one that defines the look and feel of a site is simply a matter of adding controls to get the required look. Figure 5.2, for example, shows the addition of a table to define the page layout, with an area at the top where the logo and site description sit and a
region down the left for the menu. In the middle we have the Content PlaceHolder, which is the area we are leaving for content pages to fill in.

Using a Master Page

To create a page that uses the master page, you pick Content Page from the Add New Item dialog, at which point you get the opportunity to pick which master page you wish the content page to inherit from, as shown in Figure 5.3.

When first created, the content page contains a single line of code:

```html
<%@ Page language="VB" master="~/mysite.master" %>
```

Figure 5.4 shows this content page in design view. Notice how the content defined in the master is grayed out and disabled—the only area allowed for editing is that defined by the Content control.

The confusing thing here is that this Content control doesn’t seem to exist in the file—remember there was only a single line, the Page directive.
This is because at design time the content of the master page is rendered, but our page defines no content, so an empty region is displayed so the Designer can prompt you where to add the Content control. This can be done either by selecting the Create Empty Content option from the Common Tasks menu or by simply dragging controls from the Toolbox into this region.

Listing 5.2 shows the source view for a content page with a couple of controls added.

Listing 5.2. Using a Master Page (MyPage.aspx)

```xml
<%@ Page Language="VB" Master="MySite.master" %>
<asp:Content id="Content1" ContentPlaceHolderId="ContentPlaceHolder1" runat="server">
    <asp:Button id="Button1" runat="server" text="Button" />
    <asp:ListBox id="ListBox1" runat="server">
        <asp:ListItem>Item 1</asp:ListItem>
        <asp:ListItem>Item 2</asp:ListItem>
    </asp:ListBox>
</asp:Content>
```

The local content is within the Content control—content in a page that has a master page cannot be outside a Content control. This ensures that all content pages using a master have a consistent look. Since master pages can contain multiple content areas, the id of the ContentPlaceHolder control is used to link the Content control to the ContentPlaceHolder control in the master page. When the page is constructed, ASP.NET first adds all of the content from the master page. Then it loops through the ContentPlaceHolder controls and, for each, looks in the content page for a Content control where the ContentPlaceHolderId matches the id of the ContentPlaceHolder. This ensures that the correct content is placed in the correct holder.

**Default Content**

Along with layout and code that applies to all pages, the master page can also supply default content, which can be overridden by content pages or displayed if not overridden. This is achieved by simply inserting the content within the ContentPlaceHolder element. For example, our MySite.master page could have the following default content:

```xml
<asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
    <h2>Welcome</h2>
    Welcome to my site, where you'll find lots of interesting stuff.
</asp:ContentPlaceHolder>
```
Creating a new content file based on this master would give us the following line of code:

```csharp
<%@ Page Language="VB" master="/MySite.master" %>
```

Since we haven’t specified a `Content` control, all content is provided by the master page, as shown in Figure 5.5.

**Nested Master Pages**

Master pages aren’t limited to a single master and content pages; the architecture allows for nested master pages, where a master page can have a master page. This is particularly useful for sites that require some overall branding and look but that also have subsites whose look must be consistent. For example, consider a corporation with group intranets—perhaps one for the sales department and one for research. The company wishes to have an overall look, including menus between the subsites, but allows the departments to design their parts of the site to fit their business needs. In this situation you could have three master pages—the top-level master defining the corporate site image, a submaster for the sales department, and a submaster for the research department. The sales and research sub-master pages would define the corporate master as their master page. The inheritance rules of master pages mean that any pages using one of the submaster pages receives content from all master pages in the hierarchy (see Figure 5.6).

Notice that you can inherit from any level of the hierarchy—you aren’t limited to using the bottom level. This allows you to provide generic content pages that apply to the whole site, as well as pages that apply to individual site areas. Let’s take a look at the code that makes this possible, starting with the site master as shown in Listing 5.3.
Listing 5.3. The Site Master Page (BigCorp.master)

```csharp
<%@ Master %>

<html>
<head>
  <link rel="stylesheet" type="text/css" href="MySite.css">
</head>

<body>
<form runat="server">
<table width="100%" border="0">
  <tr>
    <td>
      <asp:Hyperlink ImageUrl="home.gif" runat="server"
                      NavigateUrl="BigCorp_Default.aspx" />
    </td>
    <td>
      <h1>Big Corp Intranet</h1>
    </td>
    <td>
      <a href="BigCorp_SalesDefault.aspx">Sales</a>
    </td>
  </tr>
</table>
</form>
</body>
</html>

Figure 5.6. Using nested master pages

continues
This simple master page, containing some content and a placeholder, is shown in design view in Figure 5.7.

Now consider Listing 5.4, which shows a submaster page that inherits from the first master page.

**Listing 5.4. A Submaster Page (BigCorp_Sales.master)**

```html
<%@ Master Master="BigCorp.master" %>

<asp:Content ContentPlaceHolderId="MainContentRegion" runat="server">
  <table border="0" width="100%">
    <tr>
      <td>
        <h2>Big Corp Sales</h2>
      </td>
    </tr>
    <tr>
      <td>
        <table border="0" width="100%">
          <tr>
            <td><a href="sales/page1.aspx" Menu 1</a></td>
            <td><a href="sales/page2.aspx" Menu 2</a></td>
            <td><a href="sales/page3.aspx" Menu 3</a></td>
            <td><a href="sales/page4.aspx" Menu 4</a></td>
          </tr>
        </table>
      </td>
    </tr>
  </table>
</asp:ContentPlaceHolder>
</body>
</html>
```

**Figure 5.7. The site-wide master page (BigCorp.master)**
Because this page inherits from a master page, all content must be within a Content control, with a ContentPlaceHolderId matching that of the master ContentPlaceHolder. However, we can also include ContentPlaceHolder controls in this master, allowing content pages to add content to our content. The design view for BigCorp_Sales.master is shown in Figure 5.8, where you can see that Content1 is the content area defined by BigCorp.master, and Content2 is the region defined for content pages to use.

Using the nested master page is the same as creating any other content page. For example, Listing 5.5 shows a content page using BigCorp_Sales.master.

**Listing 5.5. A Content Page Using a Nested Master Page**

```jsp
<%@ Page Master="BigCorp_Sales.master" %>
<script runat="server">
```

Figure 5.8. The nested sales master page (BigCorp_Sales.master)
Sub Page_Load(sender As Object, e As EventArgs)
End Sub
</script>

<asp:Content ContentPlaceHolderId="SalesContentRegion"
    runat="server">
    Welcome to the Big Corp Sales Intranet
</asp:Content>

Here the ContentPlaceHolderId matches the immediate parent, and since the parent inherits from another page, the ultimate result is a combination of both master pages and the child ASP.NET content page. So, if we have two child content pages, one for the sales division and one for the research division, we’ll end up with a site as shown in Figure 5.9.

Figure 5.9. Using nested master pages
In Figure 5.9 you can see that although both departments have chosen a different style of menu (one vertical and one horizontal), the top of the page remains constant because it is defined in the top-level master.

**Master Page Configuration**
Attaching a master page directly to a page provides great flexibility but does have a downside. Not only must developers know the name of the master but they are also free to not use it, which could result in pages not fitting with the overall site design. To ensure that a master page cannot be omitted, master pages can be attached globally by modifying the Web configuration file, as shown in the following fragment:

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

A master page attached locally via the `Master` attribute, however, overrides any global master pages defined in `web.config`.

**Device-Specific Master Pages**
ASP.NET has a new architecture for detecting and rendering content to mobile devices, and this control architecture has also been implemented by the master page processing, enabling different master pages to be used for different devices. For example, it would be possible to supply different master pages for Internet Explorer and Mozilla. This is achieved by creating separate master pages and then in the page prefixing the `master` attribute with the name of the device, as shown below:

```csharp
<%@ Page master="default.master"
    Mozilla:master="mozilla.master" %>
```

When this page is accessed by a Mozilla browser, `mozilla.master` is used. All other browsers use the default master page. The results can be seen in Figure 5.10, where the content simply differs for each browser. Where this comes into its own is in supporting multiple devices, such as small-screen devices, where your master page might need to be very different, perhaps to incorporate a different menu structure.

Mobile device support is covered in more detail in Chapter 10.
Event Ordering

Because events can be present in both the master and content pages, the event order follows that of User Controls. So, for events that are captured twice, such as the Page_Load event, the content page event is fired first.

Accessing the Master Page

Content pages that have a master have a property, Master, allowing access to the master page. The Master property returns an instance of Page (from System.Web.UI), and thus you can access all of the properties and methods in the same way as for other pages. For example, to access a control from the master page you can do one of two things. The first option is to expose the control through a public property on the master page (Listing 5.6) and access that property from the content page (Listing 5.7).
Listing 5.6. Exposing Master Page Properties (MySite.master)

```html
<%@ Master %>
<script runat="server">
Public ReadOnly Property Home() As Hyperlink
    Get
        Return homeUrl
    End Get
End Property
</script>
<form runat="server">
    <asp:Hyperlink id="homeUrl"
        NavigateUrl="default.aspx" />
</form>
```

Listing 5.7. Accessing Exposed Master Page Properties (MyPage.aspx)

```html
<%@ Page Master="MySite.master" %>
<script runat="server">
Sub Page_Load(sender As Object, e As EventArgs)
    Dim Url As String = Master.Home.NavigateUrl
End Sub
</script>
```

Listing 5.6 shows a master page with a control exposed as a **Public Property**, and Listing 5.7 shows a content page accessing that control through the exposed property.

The second approach is to access the controls late bound and use the `FindControl` method to find the controls, as shown in Listing 5.8.

Listing 5.8. Accessing Master Page Contents Late Bound

```html
<%@ Page Master="MySite.master" %>
<script runat="server">
Sub Page_Load(sender As Object, e As EventArgs)
    Dim Url As String = 
        CType(Master.FindControl("homeUrl"), _
             Hyperlink).NavigateUrl
End Sub
```

Continues...
While the first solution does require you to expose controls as properties, this is required only for controls that are needed external to the master page, and this approach does provide a more efficient solution than the late-bound approach.

**Navigation**

The importance of good navigation on a site cannot be underestimated. It doesn’t matter how great your site looks or how well it was developed—if it’s hard to navigate, users won’t like using it. It’s easy to see how seriously navigation is taken just by looking at the number of menu controls that have been written since ASP.NET 1.0 was released—there are now controls that use tree views, vertical expansion, horizontal layouts, flashy graphics, and so on.

Providing a good menu isn’t the end of site navigation because it’s important to ensure visitors know where they are within the site hierarchy. Too often we see sites with pages three or four levels deep within the menu structure, but when we navigate to those pages there’s no indication of where we are. We are left wondering how to navigate back up the structure; at worst, we have to go back to the home page to navigate down again.

**Site Maps**

There are plenty of ways to implement navigation on a site, but none that are an intrinsic part of ASP.NET 1.x. With ASP.NET 2.0, there are controls and configuration files for providing a set way to define site structure and techniques for displaying the navigation information and extracting the current navigation path.

Like the rest of ASP.NET, the architecture for navigation has been broken down into logical parts, allowing customization. First, there is a configurable provider supplying the site map information, and then a set of controls that can take advantage of the data supplied by the provider. The provider not only exposes the site structure to other controls but also keeps track of the current navigation, allowing pages to identify where in the hierarchy they are. The entire structure and the current details can be exposed to users by binding controls to the provider. This pluggable architecture means that data defining the structure of a site can come from
any data source—the site map provider is the link between the data and the navigation within a site.

**Site Map Providers**

A site map provider is a data provider that exposes the site structure by way of a set interface. Site maps are pluggable within the application configuration file, within the `system.web` section. The syntax for this section is shown in Listing 5.9.

**Listing 5.9. Site Map Configuration Syntax**

```xml
<siteMap
  defaultProvider="string"
  enabled="[true|false]">
  <providers>
    <add
      name="string"
      description="string"
      provider-specific-configuration />
    <remove
      name="string" />
  </providers>
</siteMap>
```

The attributes for the `siteMap` element are shown in Table 5.1.

The attributes for the `providers` element are shown in Table 5.2.

**Table 5.1. SiteMap Configuration**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultProvider</td>
<td>The name of the default provider. This should match one of the names supplied in the providers section.</td>
</tr>
<tr>
<td>enabled</td>
<td>A Boolean value indicating whether or not site maps are enabled.</td>
</tr>
</tbody>
</table>

**Table 5.2. SiteMap providers Configuration**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the site map provider.</td>
</tr>
<tr>
<td>description</td>
<td>A description of the provider.</td>
</tr>
<tr>
<td>type</td>
<td>A string containing the full .NET type of the provider.</td>
</tr>
</tbody>
</table>
Table 5.3. XmlSiteMapProvider-Specific Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>siteMapFile</td>
<td>The name of the XML file containing the site structure.</td>
</tr>
<tr>
<td></td>
<td>The filename is configured as app.SiteMap.</td>
</tr>
</tbody>
</table>

With the Technology Preview of ASP.NET 2.0, the only provider is the XmlSiteMapProvider (in System.Web), allowing site navigation structure to be stored in an XML file. For a full description of the type attribute, see the machine.config file. The XmlSiteMapProvider has one provider-specific attribute, as shown in Table 5.3.

The pluggable architecture makes it extremely easy to add support for additional methods of site map storage. For example, you could write a FrontPage site map provider to read the site structure from the format used by Microsoft FrontPage, or perhaps one to build the structure from the file system, directly reading the names of the files and directories. To write your own site map provider you need to implement the ISiteMapProvider interface. A discussion of this is outside the scope of the book, but details of the interface can be found in the documentation.

Site Map Configuration Files
The XmlSiteMapProvider defines a set schema for the app.SiteMap file, as shown in Listing 5.10.

Listing 5.10. XmlSiteMapProvider Schema

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified">
  <xs:element name="siteMap">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="siteMapNode" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="siteMapNode">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="siteMapNode" minOccurs="0" MaxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
This defines a structure consisting of a root `siteMap` element, with the site structure being contained by `siteMapNode` elements. There has to be one top-level `siteMapNode` element, and within that can be any number of `siteMapNode` elements of any depth. The attributes for the `siteMapNode` element are shown in Table 5.4.

The use of `SiteMapFile` allows the site map information to be split among different sources. This is especially useful when different divisions supply sections of a corporate site—each part of the site map can be authored independently and even stored in different providers.

Listing 5.11 shows a sample site map file. To create one within Visual Studio .NET “Whidbey,” you simply create a new XML file and call it `app.SiteMap`—there isn’t a template for this.

**Table 5.4. siteMapNode Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>url</code></td>
<td>The URL to be used to navigate to the node. This must be unique within the entire site map file.</td>
</tr>
<tr>
<td><code>title</code></td>
<td>The title of the node.</td>
</tr>
<tr>
<td><code>description</code></td>
<td>A description of the node.</td>
</tr>
<tr>
<td><code>keywords</code></td>
<td>Keywords used to describe the node. Multiple keywords can be separated by semicolons (;) or commas (,).</td>
</tr>
<tr>
<td><code>roles</code></td>
<td>A list of roles allowed to view the node. Multiple roles can be separated by semicolons (;) or commas (,).</td>
</tr>
<tr>
<td><code>SiteMapFile</code></td>
<td>An external file containing additional <code>siteMap</code> nodes.</td>
</tr>
<tr>
<td><code>Provider</code></td>
<td>The name of the site map provider that will supply additional nodes specified in <code>SiteMapFile</code>.</td>
</tr>
</tbody>
</table>
Listing 5.11. Sample app.SiteMap File

```xml
<siteMap>
<siteMapNode title="Home"
    description="Home"
    url="SiteMaps.aspx?id=1">
  <siteMapNode title="Sales"
      description="The Sales Site"
      url="SiteMaps.aspx?id=2">
    <siteMapNode title="Customers"
      url="SiteMaps.aspx?id=3"/>
    <siteMapNode title="Products"
      url="SiteMaps.aspx?id=4"/>
    <siteMapNode title="Region"
      url="SiteMaps.aspx?id=5"/>
    <siteMapNode title="Futures"
      url="SiteMaps.aspx?id=6"/>
  </siteMapNode>
  <siteMapNode title="Research"
      description="The Research Site"
      url="SiteMaps.aspx?id=7">
    <siteMapNode title="Widgets"
      url="SiteMaps.aspx?id=8"/>
    <siteMapNode title="Doodads"
      url="SiteMaps.aspx?id=9"/>
    <siteMapNode title="Thingies"
      url="SiteMaps.aspx?id=10"/>
  </siteMapNode>
</siteMapNode>
</siteMap>
```

This provides the following structure for the site:

Home
  Sales
    Customers
    Products
    Region
    Futures
  Research
    Widgets
    Doodads
    Thingies

Using a Site Map File

Once the structure of your site is defined in the site map file, you then need a way to make use of it. For this you use a SiteMapDataSource control, which provides data access to the site map data, and then a control to display that data. From within Visual Studio .NET “Whidbey,” you can just
drag a `SiteMapDataSource` control onto the design surface—there’s no need to set any properties because it defaults to using `app.SiteMap` as its data source. You can then drag a `TreeView` control onto the page and set its `DataSourceId` property to the `id` of the `SiteMapDataSource` control. Figure 5.11, for example, shows how our Big Corp site could be constructed using a single menu.

Other controls can be bound to site map data, but in the Technology Preview release, the `TreeView` provides the best option because of its hierarchical display. It’s possible that a dedicated menu control will appear in future versions.

**Site Maps in Depth**

At its simplest, the use of site maps needs nothing more than has been discussed above, but there’s actually more to them. Adding a `SiteMapDataSource` control to a page provides all that’s needed for site map handling, but there are properties that allow for more control over how the data is supplied from the `SiteMapDataSource` to controls. For example, the syntax of the `SiteMapDataSource` control is shown in Listing 5.12.
Listing 5.12. SiteMapDataSource Syntax

<asp:SiteMapDataSource id="String" runat="server"
  FlatDepth="Integer"
  SiteMapProvider="String"
  SiteMapViewType="[Flat|Path|Tree]"
  StartingDepth="Integer"
  StartingNodeType="[Current|Parent|Root]"
  StartingNodeUrl="String"
/>  

The attributes are shown in Table 5.5.

Table 5.5. SiteMapDataSource Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlatDepth</td>
<td>A number that defines how many nodes deep in the hierarchical structure are flattened. The default is -1, which indicates all nodes.</td>
</tr>
<tr>
<td>SiteMapProvider</td>
<td>The name of the provider supplying the site map data.</td>
</tr>
</tbody>
</table>
| SiteMapViewType  | One of the SiteMapViewType enumerations, whose values are:  
  • Flat: Indicates that the data is presented without any structure.  
  • Path: Indicates the data presented is a list of nodes between the root node and the current node.  
  • Tree: Indicates the data is presented in the same hierarchical structure as the original data source. Binding nonhierarchical controls when using this mode shows only the top level of the hierarchy. This is the default value. |
| StartingDepth    | Specifies the node depth at which to start representing data. The default is 0, which is the first node. |
| StartingNodeType | One of the SiteMapNodeType enumerations, whose values are:  
  • Current: Indicates the node that represents the currently displayed page.  
  • Parent: Indicates the parent node of the currently displayed page.  
  • Root: Indicates the root node. This is the default value. |
| StartingNodeUrl  | The URL of the node at which to start representing data. |
The effects of some of these properties are not immediately apparent and depend on which control you bind to the data source and where you are in the navigation hierarchy. Probably the most useful control is the TreeView, which naturally displays hierarchical data, but the ListBox is also good for displaying site map data in a flat view. A good way to see the effects of these properties is to build a grid with three SiteMapDataSource controls, each set to a different SiteMapViewType. Then you can bind a TreeView and a ListBox to each type view of the site map, as shown in Listing 5.13.

**Listing 5.13. Sample Site Map Displays**

```xml
<asp:SiteMapDataSource id="SiteDataFlat" runat="server"
    SiteMapViewType="Flat" />
<asp:SiteMapDataSource id="SiteDataPath" runat="server"
    SiteMapViewType="Path" />
<asp:SiteMapDataSource id="SiteDataTree" runat="server"
    SiteMapViewType="Tree" />

<table border="1" width="50%">
    <tr>
        <td>Flat</td>
        <td>Path</td>
        <td>Tree</td>
    </tr>
    <tr>
        <td>
            <asp:TreeView runat="server"
                DataSourceId="SiteDataFlat" />
        </td>
        <td>
            <asp:TreeView runat="server"
                DataSourceId="SiteDataPath" />
        </td>
        <td>
            <asp:TreeView runat="server"
                DataSourceId="SiteDataTree" />
        </td>
    </tr>
    <tr>
        <td>
            <asp:ListBox runat="server"
                DataSourceId="SiteDataFlat" />
        </td>
        <td>
            <asp:ListBox runat="server"
                DataSourceId="SiteDataPath" />
        </td>
        <td>
            continues
        </td>
    </tr>
</table>
```
The initial display is shown in Figure 5.12. By default the TreeView binds the title attribute of the site map to the Text property and the url attribute to the NavigateUrl property, giving you an instant menu control. For the ListBox the title attribute is bound to both the DataTextField and DataValueField properties.

You can see from this that when in Tree mode, the TreeView displays as you expect it to. However, the Flat view shows how all nodes (at whatever level) are shown. Nodes with children are expandable in the normal TreeView style. For the Path mode, nothing is shown because we haven’t yet performed any navigation.

For the ListBox control, the Tree mode shows only the first node because it is a naturally flat control and can deal only with a single level of the hierarchy. However, in Flat mode you see all nodes because they have been flattened and therefore appear at the top level.

The results of navigating to the Sales Region page are shown in Figure 5.13.

Here you can see that the Tree and Flat views are essentially the same as their initial settings, and the Path view has now been filled. In the Path view column, note that the TreeView contains the same data as the Tree mode, but the ListBox shows only those nodes in the path between the root node and the selected node.
**Flattening Nodes**

Setting the FlatDepth property limits the depth of the nodes that are flattened. For example, on the left in Figure 5.14 you see a FlatDepth of 1, so only one node is flattened. On the right a FlatDepth of 2 causes three nodes to be flattened—the top node, plus its two child nodes.

**Setting the Starting Depth**

The StartingDepth property indicates at which node level the data is displayed from, and it affects all three modes (Flat, Path, and Tree). For example, setting the StartingDepth to 1 (where no FlatDepth is set) is shown in Figure 5.15.

Here you can see that only nodes from level 1 down are shown and only those from our navigation point—remember, the SiteMapData Source keeps track of where we are in the navigational structure.
Setting the Starting Node Type

The `StartingNodeType` property identifies what type of node to start displaying data from. For example, setting this property to `Parent` would give the results in Figure 5.16. We’ve navigated to the `Region` node, a node that is underneath `Sales`. In the `ListBox`, for the `Flat` view we see only the `Parent` of the current node, plus its children; for the `Path` view, we see only the current node and its parent; and for the `Tree` view, we see only the parent.

Setting the `StartingNodeType` to `Current` means that only the current node is displayed, as shown in Figure 5.17. Setting the `CurrentNodeType` to `Root` means that the current node becomes the root node as far as displaying the node hierarchy is concerned.

<table>
<thead>
<tr>
<th>Flat</th>
<th>Path</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales</td>
<td>Sales</td>
<td>Sales</td>
</tr>
<tr>
<td>Customers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Products</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Region</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Futures</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.15. Results of setting the StartingDepth property to 1**

**Figure 5.16. Results of setting the StartingNodeType property to Parent**
Setting the Start Node URL

The `StartingNodeUrl` property allows us to set the starting point, given the URL of a page. Since URLs in the site map file must be unique, this allows us to navigate to a given node knowing only the URL, rather than its location in the hierarchy.

Showing a Navigation Path

When a site map provides the navigational architecture for a site, it’s easy to add features that take advantage of this. With a hierarchy three deep or more, it has always been hard for users to remember where they are within that structure, so the idea of breadcrumbs came about, laying a trail of the path back to the root of the site.

With ASP.NET 2.0 this is simple: We have the `SiteMapPath` control, which automatically hooks into the site map architecture, so all you have to do is drop it on a page, as shown in Figure 5.18.

This figure shows the default implementation, just from adding the following line of code to our page:

```html
<asp:SiteMapPath runat="server" />
```

To use the `SiteMapPath` control you don’t need a `SiteMapDataSource` control because it works directly with the site map provider.

The current node is shown as simple text, and parent nodes are shown as hyperlinks, allowing quick navigation up the tree. The text for the tooltip is set to the `description` attribute from the site map file.

There are plenty of ways to customize this control to fit it to your site. The syntax is shown in Listing 5.14.
LISTING 5.14. SiteMapPath Syntax

```xml
<SiteMapPath id="String" runat="server"
    CurrentNodeStyle="Style"
    CurrentNodeTemplate="Template"
    HoverNodeStyle="Style"
    NodeStyle="Style"
    NodeTemplate="Template"
    ParentLevelsDisplayed="Integer"
    PathDirection="[CurrentToRoot|RootToCurrent]"
    PathSeparator="String"
    PathSeparatorStyle="Style"
    PathSeparatorTemplate="Template"
    RenderCurrentNodeAsLink="Boolean"
    RootNodeStyle="Style"
    RootNodeTemplate="Template"
    ShowToolTips="Boolean"
    SiteMapProvider="String"
/>
```

These are just the unique properties for this control, described in Table 5.6. All other properties are inherited and are described in the documentation.

### Table 5.6. SiteMapPath Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CurrentNodeStyle</code></td>
<td>Sets or returns the <code>Style</code> object that defines how the current node is displayed.</td>
</tr>
<tr>
<td><code>CurrentNodeTemplate</code></td>
<td>Sets a <code>Template</code>, allowing customization of how the current node is displayed.</td>
</tr>
<tr>
<td><code>NodeStyle</code></td>
<td>Sets or returns the <code>Style</code> to be used for nodes.</td>
</tr>
<tr>
<td><code>NodeTemplate</code></td>
<td>Sets a <code>Template</code>, allowing customization of how a node is displayed.</td>
</tr>
<tr>
<td><code>ParentLevelsDisplayed</code></td>
<td>Sets or returns the number of parent levels displayed. By default all parent levels are displayed.</td>
</tr>
</tbody>
</table>
### Table 5.6. SiteMapPath Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PathDirection</strong></td>
<td>Gets or sets the direction in which the nodes are displayed. This can be one of the PathDirection enumerations, whose values are:</td>
</tr>
<tr>
<td></td>
<td>• CurrentToRoot: The current node is shown on the left, and child nodes are shown to the right.</td>
</tr>
<tr>
<td></td>
<td>• RootToCurrent: The current node is shown on the left, and parent nodes are shown on the right. This is the default value.</td>
</tr>
<tr>
<td></td>
<td>Setting the direction has no effect on the separator between nodes.</td>
</tr>
<tr>
<td><strong>PathSeparator</strong></td>
<td>Sets or returns a string to be used as a separator between nodes. This is replaced by the contents of the PathSeparatorTemplate if present. The default is <code>&gt;</code>.</td>
</tr>
<tr>
<td><strong>PathSeparatorStyle</strong></td>
<td>Sets or returns the Style to be used for the PathSeparator string.</td>
</tr>
<tr>
<td><strong>PathSeparatorTemplate</strong></td>
<td>Sets a Template, allowing customization of the node separator.</td>
</tr>
<tr>
<td><strong>RenderCurrentNodeAsLink</strong></td>
<td>Sets or returns a Boolean that indicates whether or not the current node is rendered as a hyperlink. The default value is False.</td>
</tr>
<tr>
<td><strong>RootNodeStyle</strong></td>
<td>Sets or returns the Style to be used for the root node. Any Style values set here override those set in the NodeStyle property.</td>
</tr>
<tr>
<td><strong>RootNodeTemplate</strong></td>
<td>Sets a Template, allowing customization of the root node.</td>
</tr>
<tr>
<td><strong>ShowToolTips</strong></td>
<td>Sets or returns a Boolean indicating whether or not tooltips are shown on hyperlinks.</td>
</tr>
<tr>
<td><strong>SiteMapProvider</strong></td>
<td>Sets or returns a string indicating the site name of the provider supplying the site map data.</td>
</tr>
</tbody>
</table>
These properties give a great deal of flexibility in how the navigation path is shown. For example, consider the code shown in Listing 5.15.

**Listing 5.15. Setting the SiteMapPath Properties**

```xml
<asp:SiteMapPath ID="SiteMapPath1" runat="server"
    NodeStyle-Font-Name="Franklin Gothic Medium"
    NodeStyle-Font-Underline="true"
    NodeStyle-Font-Bold="true"
    RootNodeStyle-Font-Name="Symbol"
    RootNodeStyle-Font-Bold="false"
    CurrentNodeStyle-Font-Name="Verdana"
    CurrentNodeStyle-Font-Size="10pt"
    CurrentNodeStyle-Font-Bold="true"
    CurrentNodeStyle-ForeColor="red"
    CurrentNodeStyle-Font-Underline="false">
    <PathSeparatorTemplate>
        <asp:Image runat="server" ImageUrl="arrow.gif"/>
    </PathSeparatorTemplate>
</asp:SiteMapPath>
```

This defines styles for the nodes and a separator that uses a custom image. The results are shown in Figure 5.19.

Notice that the root node is underlined even though it wasn’t specified as part of the `RootNodeStyle`—the underlining was inherited from the `NodeStyle`.

**SiteMapPath Events**

The `SiteMapPath` is built dynamically from the data held by the underlying site map provider. As the tree of nodes is traversed, each item in the path, from the root node to the current node, is added to the `Controls` collection of the `SiteMapPath` control. Like other collection controls (such as the `DataList` or `DataGrid`), two events are fired when items are either created (`ItemCreated`) or bound (`ItemDataBound`) to the `SiteMapPath`. The signature for these events is the same:

```vbnet
Sub eventName(Sender As Object,
               E As SiteMapNodeItemEventArgs)
```

`SiteMapNodeItemEventArgs` has one property, `Item`, which returns an object of type `SiteMapNodeItem`, which in turn has three properties, as described in Table 5.7.

![Current Path: Home ➔ Sales ➔ Region](image)

**Figure 5.19. A customized SiteMapPath control**
### Table 5.7. SiteMapNodeItem Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ItemIndex</td>
<td>The zero-based index number of the item being added.</td>
</tr>
<tr>
<td>ItemType</td>
<td>The type of node being added, which can be one of the SiteMapNodeItemType enumerations:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Current</strong>: Indicates the current node (page) within the navigation path.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Parent</strong>: Indicates a parent of the current node. All nodes between the current node and the root node are parent nodes.</td>
</tr>
<tr>
<td></td>
<td>• <strong>PathSeparator</strong>: Indicates a separator between nodes.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Root</strong>: Indicates the root node of the navigation path.</td>
</tr>
<tr>
<td>SiteMapNode</td>
<td>The SiteMapNode that represents the node being added to the SiteMapPath.</td>
</tr>
</tbody>
</table>

Intercepting the ItemCreated and ItemDataBound events gives you a chance to change the default behavior as the items are created. For example, Listing 5.16 shows how you could build up an HTML meta tag consisting of the Keywords from the site map details. If the SiteMapPath control were embedded into the master page, this meta tag would be automatically constructed for each page.

### Listing 5.16. SiteMapPath ItemCreated Event

```csharp
<%@ Page %>

<head runat="server" id="PageHead" />

<script runat="server">

Sub ItemCreated(Sender As Object, 
    E As SiteMapNodeItemEventArgs)

    If E.Item.ItemType = _SiteMapNodeItemType.Current Then

        Dim sb As New StringBuilder()
        Dim s As String

        For Each s In E.Item.SiteMapNode.Keywords
            sb.Append(s) + " ">

    End Sub

</script>
```

continues
The SiteMapNode Object

When the site map is constructed from the data provider, each of the items is built into a SiteMapNode object. These in turn are added to a SiteMapNodeCollection, which therefore represents all pages within a Web site. The SiteMapNode object provides links to nodes up, down, and next to it in the hierarchy and thus can be used to build a treelike structure. As shown in Listing 5.16, the ItemCreated event of the SiteMapPath object allows access to the SiteMapNode, which has the properties detailed in Table 5.8.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>Returns a collection of additional attributes applicable to the node. For the XmlSiteMapProvider, the list of attributes maps to existing properties, namely Title, Description, Url, Attributes, Roles, and Keywords.</td>
</tr>
<tr>
<td>ChildNodes</td>
<td>If applicable, returns a SiteMapNodeCollection containing child nodes of the current node.</td>
</tr>
<tr>
<td>Description</td>
<td>Returns the description of the current node.</td>
</tr>
<tr>
<td>HasChildNodes</td>
<td>Indicates whether or not the current node has any child nodes.</td>
</tr>
</tbody>
</table>
### Table 5.8. SiteMapNode Properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keywords</td>
<td>Returns an <code>IList</code> containing keywords for the current node.</td>
</tr>
<tr>
<td>NextSibling</td>
<td>Returns the next node on the same level as the current node, or returns <code>null</code> (Nothing in Visual Basic) if there is no next node.</td>
</tr>
<tr>
<td>ParentNode</td>
<td>Returns the parent node of the current node, or returns <code>null</code> (Nothing in Visual Basic) if there is no parent node (i.e., the current node is the root node).</td>
</tr>
<tr>
<td>PreviousSibling</td>
<td>Returns the previous node on the same level as the current node, or returns <code>null</code> (Nothing in Visual Basic) if there is no previous node.</td>
</tr>
<tr>
<td>Roles</td>
<td>Returns an <code>IList</code> containing the roles applicable to the current node.</td>
</tr>
<tr>
<td>RootNode</td>
<td>Returns the root node.</td>
</tr>
<tr>
<td>Title</td>
<td>Returns the title of the current node.</td>
</tr>
<tr>
<td>Url</td>
<td>Returns the URL of the current node.</td>
</tr>
</tbody>
</table>

There are three methods for the `SiteMapNode` object, as described in Table 5.9.

### Table 5.9. SiteMapNode Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetAllNodes</td>
<td>Returns a <code>SiteMapNodeCollection</code> containing all child nodes of the current node.</td>
</tr>
<tr>
<td>GetDataSourceView</td>
<td>Returns a <code>SiteMapDataSourceView</code>, which is a view of the underlying site map data. This is useful for control developers who wish to interface to the site map architecture.</td>
</tr>
<tr>
<td>IsDescendantOf</td>
<td>Indicates whether or not the current node is a descendant of a supplied node.</td>
</tr>
</tbody>
</table>
Accessing the Site Map at Runtime
So far we’ve seen the site map be used by controls, but it can also be accessed directly because it is exposed through a static page property called SiteMap. For example, to access the current node within the site map, you can use the following code:

```vbscript
Dim currNode As SiteMapNode
currNode = SiteMap.CurrentNode
```

This means that even if you aren’t using a SiteMapPath control, you can easily build links pointing back to the hierarchy, as shown in Listing 5.17.

**Listing 5.17. Using the SiteMap Property of the Page**

```xml
<script runat="server">
Sub Page_Load(Sender As Object, E As EventArgs)
End Sub
</script>

<form runat="server">
    <asp:HyperLink id="ParentLink" Text="Go Back" />
</form>
```

Table 5.10 details the properties of the SiteMap class.

**Table 5.10. SiteMap Class Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentNode</td>
<td>Returns a SiteMapNode object representing the current page.</td>
</tr>
<tr>
<td>Provider</td>
<td>Returns the site map provider.</td>
</tr>
<tr>
<td>Providers</td>
<td>Returns a collection (SiteMapProviderCollection) of all site map providers.</td>
</tr>
<tr>
<td>RootNode</td>
<td>Returns a SiteMapNode object representing the root node.</td>
</tr>
</tbody>
</table>
These properties give you access to the site map details and allow you to interface into it at the programmatic level, in case more flexibility is required than the standard server controls provide.

**SUMMARY**

In this chapter we’ve looked at two very important topics: how the look and feel of sites can be implemented in ASP.NET 2.0 and how navigation around those sites can be implemented.

We’ve seen how a great deal of time-consuming work has been removed by the introduction of master pages, allowing the site layout and default implementation to be easily centralized. Not only does this make it easier to develop individual pages, but since layout and code can be centrally contained, it also eases maintenance and reduces potential errors.

We’ve also seen how ASP.NET 2.0 supplies a comprehensive architecture for site navigation. The introduction of the `SiteMapDataSource` and the underlying flexibility of providers allow site structure to be easily defined and used within Web pages. By placing this navigation within a master page you also have the simplicity of having to define the navigation in only a single place.

Now it’s time to move on to another important topic—security, and how to identify users and control what they can do once they reach your site.