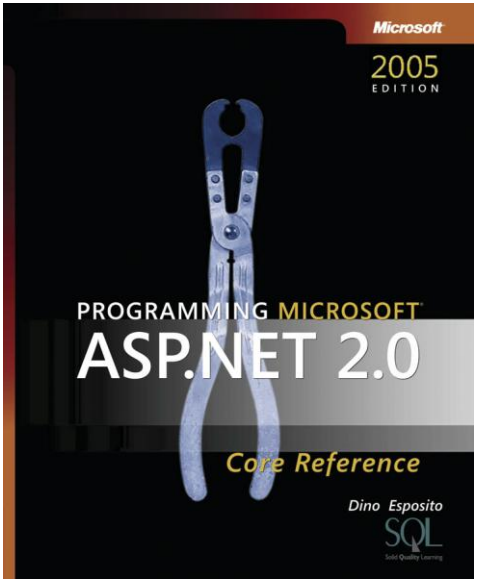


Programming Microsoft® ASP.NET 2.0 Core Reference

Dino Esposito
(Solid Quality Learning)

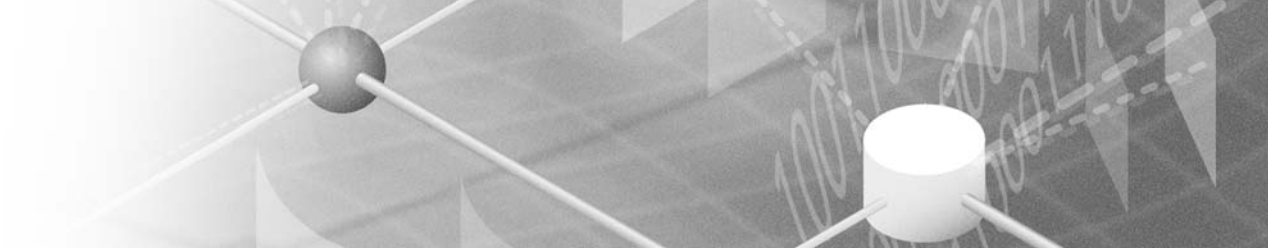
To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/8376.aspx

9780735621763
Publication Date: November 2005

Microsoft Press

Table of Contents

Part III ASP.NET Infrastructure



Chapter 2

Web Development in Microsoft Visual Studio .NET 2005

In this chapter:

No matter how you design and implement a Web application, at the end of the day it always consists of a number of pages bound to a public URL. The inexorable progress of Web-related technologies has not changed this basic fact, for the simple reason that it is the natural outcome of the simplicity of the HTTP protocol. As long as HTTP remains the underlying transportation protocol, a Web application can't be anything radically different from a number of publicly accessible pages. So in this context, what's the role of Microsoft ASP.NET and Visual Studio .NET 2005?

ASP.NET provides an abstraction layer on top of HTTP with which developers build Web sites and Web-based front-ends for enterprise systems. Thanks to ASP.NET, developers can work with high-level entities such as classes and components within the object-oriented paradigm. Development tools assist developers during the work and try to make the interaction with the ASP.NET framework as seamless and productive as possible. Development tools are ultimately responsible for the application or the front-end being created and deployed to users. They offer their own programming model and force developers to play by those rules.

The key development tool for building ASP.NET applications and front-ends is Visual Studio .NET 2005—the successor to Visual Studio .NET 2003. It has a lot of new features and goodies expressly designed for Web developers to overcome some of the limitations that surfaced from using Visual Studio .NET 2003.

In this chapter, we'll review the main characteristics and features of Visual Studio .NET 2005 as far as ASP.NET applications are concerned. We'll see changes made to the project, new IDE and editing features, and deployment capabilities.

Introducing Visual Studio .NET 2005

Visual Studio .NET is a container environment that integrates the functionality of multiple visual designers. You have a designer for building Windows Forms applications, one for building ASP.NET sites, one for building Web services, and so on. All items required by your work—such as references, connectors to data sources, folders, and files—are grouped at two levels: solutions and projects. A solution container contains multiple projects, whereas a project container typically stores multiple items. Using these containers, you manage settings for your solution as a whole or for individual projects. Each item in the project displays its own set of properties through a secondary window—the Properties window.

Before we meet Visual Studio .NET 2005 in person, let's briefly review the major shortcomings of its predecessor. In this way, you can enjoy the new set of features even more.

Visual Studio .NET 2003 Common Gripes

As you probably know from your own experiences, Visual Studio .NET 2003 has a single model for designing applications: the project-based approach. Real-world experience has shown this is not necessarily the best approach—at least as far as ASP.NET and Web applications are concerned.

The project is the logical entity that originates any type of .NET application—be it Windows Forms, the Web, a console, or a Web service. Developers build an application by creating a new project, configuring it, and then adding items such as pages, resources, classes, controls, and whatever else will help. For Web applications, a Visual Studio .NET project poses a few issues at two levels at least: machine and integrated development environment (IDE).

Constraints at the Machine Level

For Visual Studio .NET 2003 to run successfully on a development machine, you need to install Microsoft FrontPage Server Extensions (FPSE). FPSE are the only supported way to get to the files of the project, as Visual Studio .NET does not support FTP or even direct Internet Information Server (IIS) access. Among other things, an FPSE-equipped machine runs into trouble as soon as you try to install Windows SharePoint Services (WSS) on it. Additional setup work is required if you want Visual Studio .NET and ASP.NET to work on the same development machine along with WSS test sites.

Visual Studio .NET is dependent on IIS, which must be installed on the same development machine or on a connected server. In addition, each application you create must be tied to an IIS virtual folder. These limitations have a much greater impact on the development process

than one might think at first. For example, developers need administrative privileges to create new projects, and effective corporate security policies for developer machines should be defined throughout the company. Furthermore, debugging various configurations and scenarios is definitely hard and challenging, though certainly not impossible.

Constraints at the IDE Level

All in all, the number-one issue with Visual Studio .NET–driven Web development is the tool's inability to open a single ASP.NET page outside of a project. You can open and edit an *.aspx* page, but Microsoft IntelliSense won't work on it; the same happens with other key features, such as running and debugging the page. Frankly, in this type of scenario Visual Studio .NET 2003 offers only one advantage over Notepad—HTML syntax coloring.

In Visual Studio .NET 2003, the project file is the single point of management for the constituent elements of the application. As a result, to make a file part of the project, you must explicitly add it into the project file and configure it—you can't just point at an existing virtual directory and go. The information coded in the project file counts more than the actual contents of the directory. As a result, more often than not useless files are forgotten and left around the site. Synchronizing hundreds of files in large applications is not easy; deploying projects onto other machines can be even more annoying.

This model is problematic also from the source control perspective. When managing Web projects under source control, you should perform all available source control operations using Visual Studio .NET. In addition, you shouldn't manually force a file to be under source control. All files that should be source-controlled are placed there automatically when you use the appropriate menu commands. In other words, the project file ends up being the single point of contention with source control.

Visual Studio .NET also does all that it can to force you to use a code-behind class for each page added to the project. In general, keeping code (*.cs* or *.vb* file) separated from layout (*.aspx* file) is a good and highly recommended practice. However, the Visual Studio .NET 2003 implementation of this feature injects a lot of tool-generated code in the project files. This leads to a brittle model of keeping file and control references in sync. Furthermore, the contents of a project are compiled down to single assembly, with the subsequent creation of a single contention point for shared projects, an application's domain restart on every change, and a significantly expensive (and explicitly requested) compile step for large projects.

Finally, you find no support in Visual Studio .NET 2003 for declarative resources and you must perform an explicit code-generation step for adding resources such as WSDL and XSD files.

To sum it up in one sentence: although developers successfully use Visual Studio .NET 2003 for real-world applications, the tool isn't ideal for simpler projects and still has a number of shortcomings.

Visual Studio .NET 2005 Highlights

Visual Studio .NET 2005 provides a simpler and more friendly way to create ASP.NET applications. The key improvements remedy the shortcomings detailed earlier. Let's outline these features briefly. We'll go into more detail later in the chapter as we develop a start-up project.

No IIS Dependency

IIS is no longer a strict requirement for Visual Studio .NET to work.Visual Studio .NET 2005 ships, in fact, with a local Web server that makes IIS optional, at least for quick testing and debugging purposes. Figure 2-1 shows the user interface of the embedded Web server.

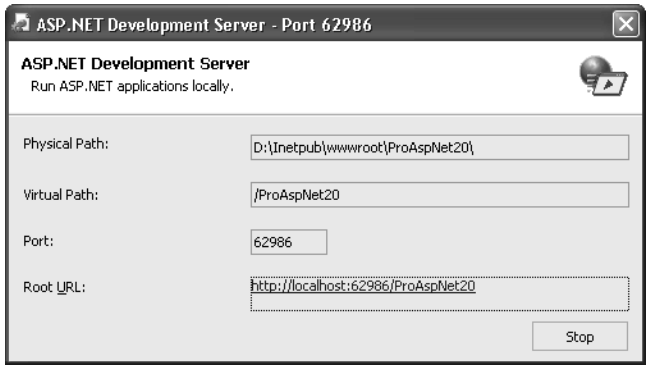


Figure 2-1 The local Web server in action in Visual Studio .NET 2005.

The embedded Web server is a revisited version of Cassini, the free mini-Web server that originally shipped with Web Matrix—a community-supported, free editor designed for ASP.NET applications. It is important to note that the local Web server represents only the default option. If you open the project from an existing IIS virtual directory, Visual Studio .NET would use IIS to test the application.

The embedded Web server is only a small piece of executable code and can't replace all the features of a full-blown Web server such as IIS. It works only with individual pages and doesn't include any of the extra features of IIS, such as the metabase.

Ways to Access Web Sites

Visual Studio .NET 2005 supports multiple ways to open Web sites. In addition to using FPSE, you can access your source files by using FTP or a direct file system path. You can also directly access the local installation of IIS, browse the existing hierarchy of virtual directories, and access existing virtual roots or create new ones. As Figure 2-2 demonstrates, you can open your Web site using a file system path or an IIS virtual directory. In the former case, the local Web server is used to test the site.

The interaction with IIS is greatly simplified, as Figure 2-3 shows. When you try to open a Web site, you are given a few options to choose from. You can locate a project by using a file

system path, using the IIS hierarchy of virtual directories (only the local IIS), using FTP, or by just typing the URL of the site configured with FrontPage Server Extensions. The IIS tab also contains buttons to create new virtual roots and applications.

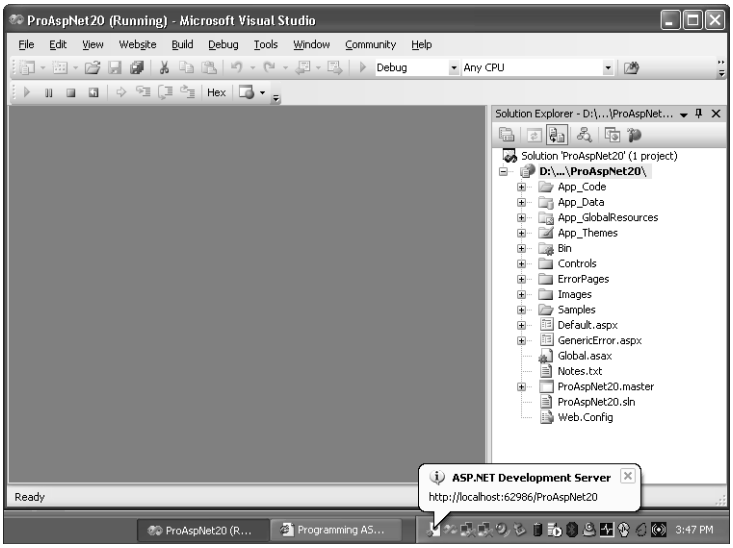


Figure 2-2 The ASP.NET application is controlled by the local Web server if the Web site is opened from a file system path.

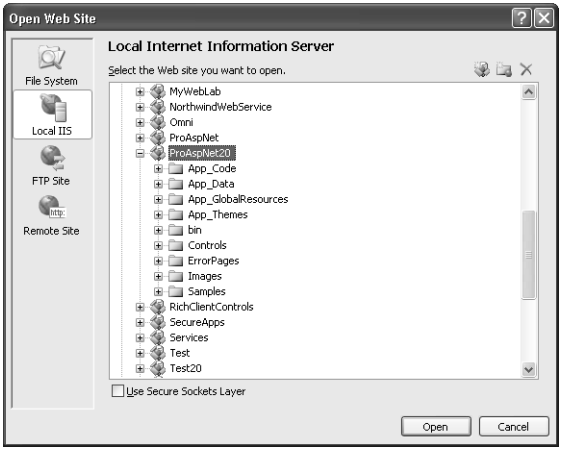


Figure 2-3 Navigating your way through the IIS hierarchy to locate an existing virtual directory to open.

Note You can open existing Web sites using the FTP protocol and then create and edit files. However, you must have access to the FTP server and read and write permissions for a particular FTP directory. The directory must already exist because Visual Studio .NET 2005 cannot create a new Web site via FTP.

Building the Project Output

Visual Studio .NET 2005 does not compile everything in the site into a single assembly, as Visual Studio .NET 2003 does. Instead, it builds on the new ASP.NET compilation model and dynamically recognizes file types based on the folder they belong to. In this way, not only are changes to *.aspx* files immediately caught, but so are those made to constituent *.cs* or *.vb* files and a variety of accessory resource files. This results in a sort of dynamic compilation for code-behind classes.

There are pros and cons about the new ASP.NET 2.0 compilation model, and some additional parameters need to be considered thoroughly before one can come to a reasonable conclusion about the model. Whatever your final assessment is, though, two facts remain set in stone. First, the ASP.NET 2.0 compilation model allows you to deploy more types of source files (for example, C# and VB.NET classes), monitors these source files for changes, and automatically recompiles. Second, this behavior is optional. If you cringe at the idea of leaving valuable C# source files on the Web potentially at the mercy of hackers, you should just stick to the old model and compile external classes into a separate assembly through an explicit compile step. Whatever your position on the matter is, ASP.NET 2.0 and Visual Studio .NET 2005 give you an alternative.

Solution files (*.sln) are supported, but they're no longer necessary for creating and managing a Web project. The root Web directory defines a Web project; you just add files to the directory and they are in the project. If a file doesn't immediately show up, you right-click on the Solution Explorer window and select Refresh Folder. Solution files are still useful to manage multiple projects, but they don't need to live in the Web directory.

Copying a Web Project

Another long-awaited feature worth a mention is the Copy Web site feature. In earlier versions of Visual Studio .NET, duplicating and synchronizing a Web project onto another machine, or simply moving it to another location within the same machine, was not a hassle-free task. Basically, it was completely up to you and to any FTP-based tool you could come up with. If your server host supported FPSE, you could go through the Visual Studio .NET 2003 integrated wizard—the Project|Copy function. Otherwise, the most viable solution was using raw File Transfer Protocol (FTP). (Moving a Web site within the same network or machine is a similar experience, except that you can use Windows Explorer.)

Sure the overall procedure was not smooth; but it was hardly a mission-impossible task because only a brute-force copy is required. But what if, with good reason, you wanted to move modified files only? Or only files that match given criteria? In these cases, you were left alone to find and copy only these files. (On the other hand, I'd say, who's better qualified than you for this kind of task?)

In Visual Studio .NET 2005, by selecting a menu item you can copy your current Web site to another local or remote location. The Copy Web Site function is a sort of integrated FTP tool that enables you to easily move files around. Figure 2-4 shows a glimpse of the feature.

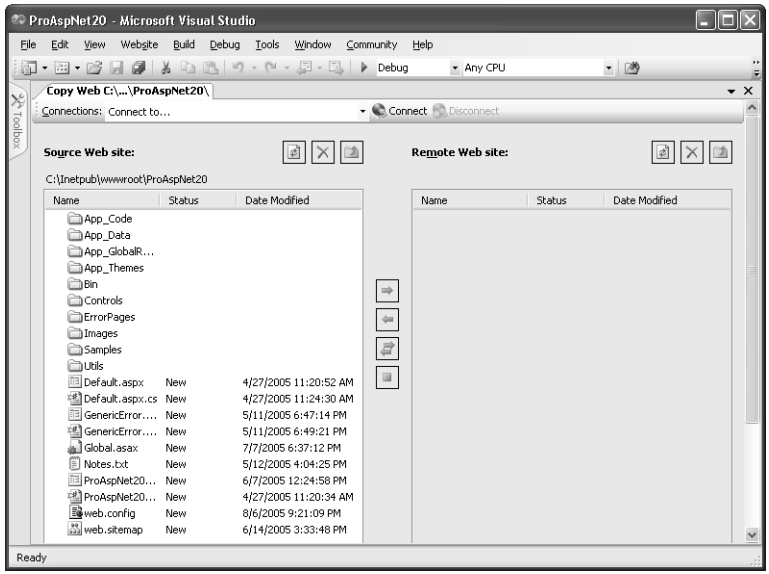


Figure 2-4 The Copy Web Site feature in action.

You connect to the target destination, select the desired copy mode—either Overwrite Source To Target Files, Target To Source Files, or Sync Up Source And Target Projects—and then proceed with the physical copying of files. As Figure 2-5 shows, you can copy files to and from virtual and physical folders, within or across the machine's boundaries.

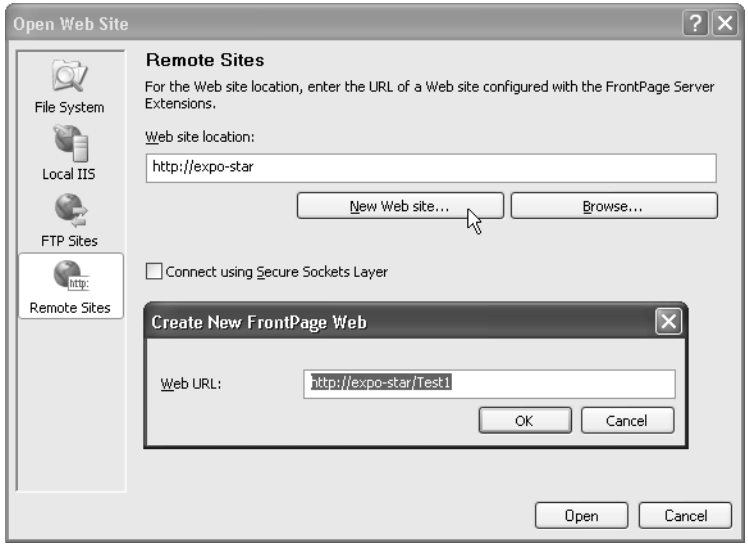


Figure 2-5 Connecting to a remote site to make a copy of the local project.

As you can see yourself, the Copy Web Site function is ideal for deployment especially in hosting environment scenarios in which you need to manage live server files. In addition, the Visual Studio .NET 2005 tool can operate as a synchronization tool, which is helpful to quickly test applications in different scenarios and configurations.

Smarter Editing with IntelliSense

Last but not least, Visual Studio .NET 2005 supports standalone file editing and doesn't require a project to edit a single file on disk. So if you double-click an *.aspx* file in Windows Explorer, Visual Studio .NET 2005 starts up and lets you edit the source code. Unlike with the previous version, IntelliSense and related syntax-coloring work effectively. The page can be viewed live in the embedded browser through the local Web server.

Note that IntelliSense now works everywhere within the source file (see Figure 2-6), including within data-binding expressions, page directives, and code inline in *.aspx* files.

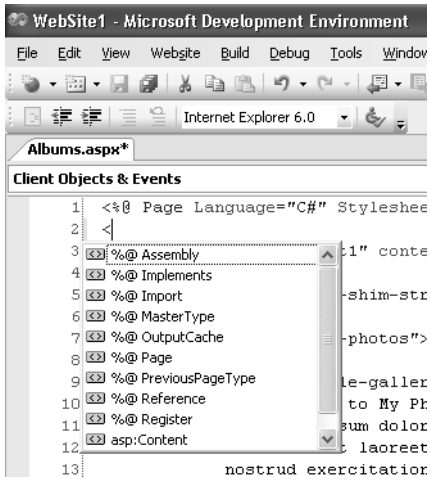


Figure 2-6 IntelliSense works everywhere around the source code of the page.

In Visual Studio .NET 2003, IntelliSense support in the HTML view of the page was hard to achieve for custom controls. Basically, you had to create an XSD file manually to describe the public interface of a control; next, you had to install that file in a particular folder and link it to the page through an *xmlns* attribute. Thankfully, you should never have to author this schema file manually in Visual Studio .NET 2005. A valid schema file is automatically generated when the page author first drops the control on the page. The schema generator does its job after looking at any metadata associated with the controls. However, no new metadata attributes are used. The schema generator grabs all that it needs out of existing attributes used for declaring expected parsing and persistence behavior for controls. (See the "Resources" section at the end of this chapter.)

Important In light of this, if you are authoring custom controls for ASP.NET 2.0 be sure to check how IntelliSense works for your control in Visual Studio .NET 2005. You can do a lot to ensure that IntelliSense works appropriately by applying the appropriate metadata to controls.

Create an ASP.NET Project

Let's go further and create a sample ASP.NET project with Visual Studio .NET 2005. You first create a new Web site by choosing the corresponding command on the File|New menu. The dialog box that appears prompts you for the type of site you want to create, as in Figure 2-7.

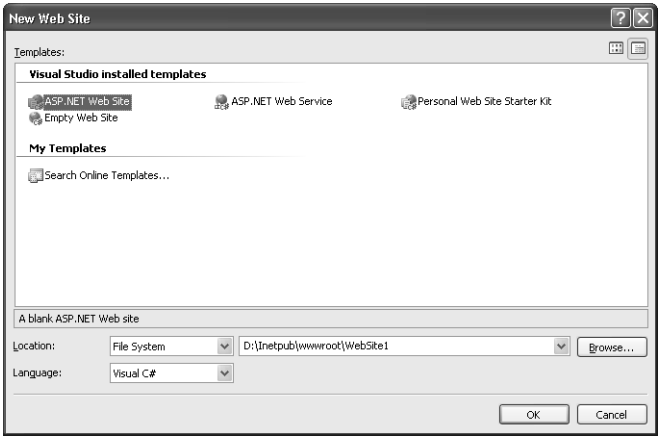


Figure 2-7 The options available for creating a new Web site with Visual Studio .NET 2005.

If you select the Web Site option, Visual Studio generates the minimum number of files for a Web site to build. Basically, it creates a default *.aspx* page and an empty Data directory. If you opt for a personal Web site, an ASP.NET starter kit is used to give you a functional Web site with several standard features built in. Let's go for a Web site. Visual Studio .NET 2005 creates a project file but doesn't use it to track all the files that form an application. The root directory of the site implicitly defines a Web project. Any file or folder added or created under the root is automatically part of the project.

Page Design Features

The ASP.NET front-end of an application can include several types of entities, the most important of which are pages. To edit a Web page, you can choose between two views—Design and Source. The Design view displays the HTML layout, lets you select and edit controls and static elements, and provides a graphical preview of the page. The Source view shows the HTML markup along with the inline code. The markup is syntax-colored and enriched by features such as IntelliSense, tips, and autocompletion.

You choose the template of the item to add to the site from the menu shown in Figure 2-8.

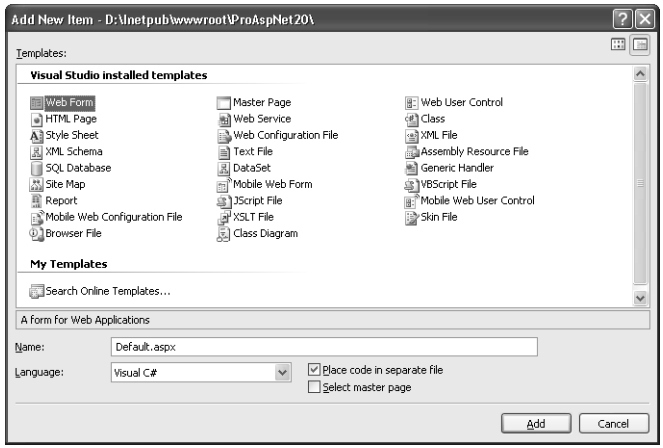


Figure 2-8 Item templates supported by Visual Studio .NET 2005.

Note the two check boxes that appear at the bottom of the window. You can choose to keep the code of the page in a separate file (similar to the code-behind model of Visual Studio .NET 2003) and can associate the current page with a master page. Master pages are a cool new feature of ASP.NET 2.0 that we'll discuss thoroughly in Chapter 6. The code-behind schema touted by Visual Studio .NET 2003 has been revised and restructured. As a result, pages built with Visual Studio .NET 2005 are not forced to use code separation (that is, the page is separated into *.aspx* and *.cs* files). Code separation is still fully supported and recommended, but it is now optional.

Before we get to add some code to build a sample page, let's review some design-time features of the page.

Master Pages

The master page is a single file that defines the template for a set of pages. Similar to an ordinary *.aspx* page, the master contains replaceable sections that are each marked with a unique ID. Pages in an application that will inherit the structure defined in the master reference the master page in their *@Page* directive or even programmatically. A page based on a master is said to be a *content page*. One master page can be bound to any number of content pages. Master pages are completely transparent to end users. When working with an application, a user sees and invokes only the URL of content pages. If a content page is requested, the ASP.NET runtime applies a different compilation algorithm and builds the dynamic class as the merge of the master and the content page.

Master pages are among the hottest new features of ASP.NET 2.0 and address one of the hottest threads in many ASP.NET 1.x newsgroups. By using master pages, a developer can create a Web site in which various physical pages share a common layout. You code the shared user interface and functionality in the master page and make the master contain named placeholders for content that the derived page will provide. The key advantage is that shared information is stored in a single place—the master page—instead of being replicated in each page.

Second, the contract between the master and content page is fixed and determined by the ASP.NET Framework. No change in the application or constituent controls can ever break the link established between master and content.

Important ASP.NET 2.0 master pages offer *one* way of building Web pages. In no way are master pages the only or preferred way of building Web sites. You should use master pages only if you need to duplicate portions of your user interface or if your application lends itself to being (re)designed in terms of master and content pages.

Content Pages

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 *.aspx* page. A content page is a special type of ASP.NET page, as it is allowed to contain only <*asp:Content*> tags. Any classic HTML tags—including client-side <*script*> and comments—are not allowed and if used raise compile errors.

The reason for this lies in the implementation of the master page feature. Because the content regions are substituted into the master page placeholders, the destination for any literal markup (that is, comments, script, other tags) would be ambiguous, because the same kind of content is also allowed in the master.

Visual Studio .NET offers a special Design view of content pages, as Figure 2-9 demonstrates. The view contains as many drawing surfaces as there are content regions in the master. At the same time, the master layout is displayed in the background grayed out to indicate that it's there but you can't access it.

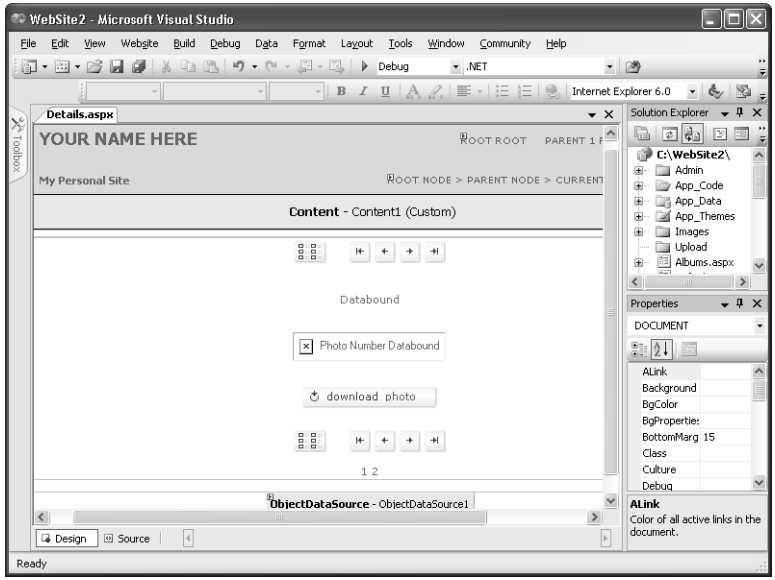


Figure 2-9 Content pages in Visual Studio .NET 2005.

Important Content pages can be used only in conjunction with master pages. A Web Forms page silently becomes a content page when you check the Select Master Page option in the dialog box shown in Figure 2-8.

Code-Behind Classes

When you add new Web Forms and require code and layout separation, a C# (or Visual Basic .NET) class file is created along with the *.aspx* file in the same folder. The class file is named after the *.aspx* resource simply by adding a language extension. For example, if the Web Forms is named *WebForm1.aspx*, the corresponding code-behind class is named *WebForm1.aspx.cs*. This name is just the default name, and it obeys the default naming convention. Although it is not recommended for the sake of consistency, you should feel free to rename the class file to whatever name you like.

Nothing bad can happen to your application if you make it use inline code instead of code and layout separation. Nonetheless, real-world pages need a good amount of server code, and appending all that code to the <*script*> tag of the *.aspx* file makes the file significantly hard to read, edit, and maintain. Code-behind, on the other hand, is based on the idea that each Web Forms page is bound to a separate class file that contains any code that is relevant to the page. The code-behind class ends up being the basis of the dynamically generated page class that the ASP.NET runtime creates for each requested *.aspx* resource. All the server code you need to associate to the *.aspx* resource flows into the code-behind class. The code-behind model promotes object-orientation, leads to modular code, and supports code and layout separation, allowing developers and designers to work concurrently to some extent.

Visual Studio .NET 2005 delivers an improved page model even though the overall syntax is nearly identical to that of previous versions. There are two main changes you'll notice in Visual Studio .NET 2005. First, having the code in a separate class file is now optional. (See Figure 2-8.) Second, thanks to *partial classes*—a .NET Framework–specific feature available only in version 2.0—multiple developers (and designers) can work on the same page at the same time. A *partial class* is a .NET class defined across multiple source files that the compiler sews back together.

Note You should use code-behind classes for all your project pages, except test pages you quickly arrange in the context of toy applications or to verify a given feature. Using the code-behind model along with the principle of class inheritance gives you enough programming power to create a hierarchy of classes to cut off development time and maximize code reusability.

The Toolbox of Controls

A Web Forms page is mostly made of controls—either predefined HTML and Web controls, user controls, or custom controls. Except for user controls (*.ascx* files), all the others are conveniently listed in the editor's toolbox. (See Figure 2-10.) The toolbox can be toggled on and off and is an easy way to pick up the control of choice and drop it onto the Web form via a drag-and-drop operation. The toolbox is visible only if *.aspx* resources are selected, either in Design or Source view.

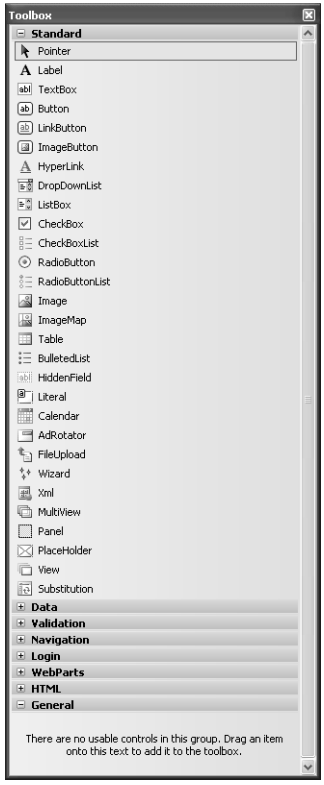


Figure 2-10 The Visual Studio .NET 2005 toolbox.

The toolbox is widely customizable and supports the addition of new controls as well as the creation of new user-defined tabs. Controls defined in a project within the current solution are automatically added to the toolbox.

Editor's Special Capabilities

The Visual Studio .NET 2005 code editor presents some interesting features that prove the team commitment to excellence and all users' satisfaction. I'd like to call your attention to four of them: check for accessibility, markup preservation, simplified tabification and indentation, and target schema validation.

A button on the HTML source editing toolbar allows you to select a few accessibility requirements and validate the page's code against them. (See Figure 2-11.)

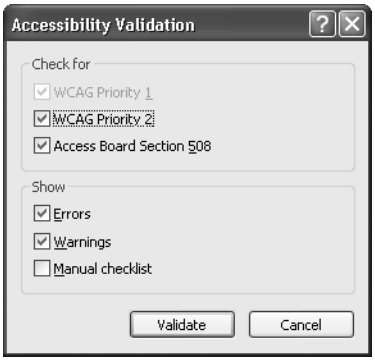


Figure 2-11 The dialog box that allows you to select options for an accessibility check.

Note how many recommendations you get even for an empty page. Among other things, you're invited to synchronize alternatives and captions with time-based multimedia tracks, and to ensure that you don't convey information using color alone.

Visual Studio .NET 2005 preserves the formatting of your HTML edits and doesn't even attempt to reformat the source as you switch between views. At the same time, it comes with powerful features for indentation and tag formatting that you can optionally turn on. The days of the Visual Studio .NET 2003 auto-formatting features kicking in on view switching are definitely gone.

In Figure 2-12, you see the list of supported client targets. Once you select a target, the whole editing process is adapted to the features of the specified device. Want a quick example? Imagine you select Netscape Navigator 4.0 (NN4) as the client target. NN4 doesn't recognize the <*iframe*> tag; instead, it sports the <*layer*> tag with nearly the same characteristics. As Figure 2-13 shows, Visual Studio .NET detects the difference and handles it correctly. IntelliSense doesn't list *iframe* but prompts you for *layer*. If you insist and type in <*iframe*> anyway, a squiggle shows up to catch your attention.

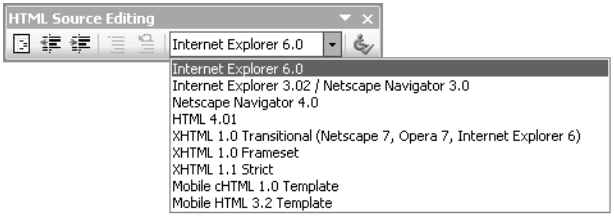


Figure 2-12 The list of client targets for which Visual Studio .NET can cross-check your markup.

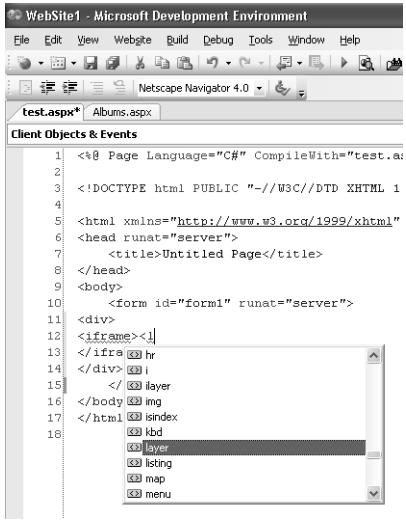


Figure 2-13 The code editor is sensitive to the selected client target schema.

> **Note** The number of ASP.NET client targets is significantly larger in Visual Studio .NET 2005 and ranges from Internet Explorer 6.0 to HTML 3.2 (covering Internet Explorer 3.x and Netscape Navigator 3.x). Other validation targets are mobile schemas (Compact HTML 1.0 and mobile HTML 3.2), Netscape 4.0, and the XHTML 1.0 Transitional schema. The latter schema covers browsers such as Netscape 7.0 and Opera 7.0.

Code Refactoring

When a *.vb* or a *.cs* file is open for editing in Visual Studio .NET 2005, a new menu appears on the top-most menu strip—the Refactor menu, shown in Figure 2-14.

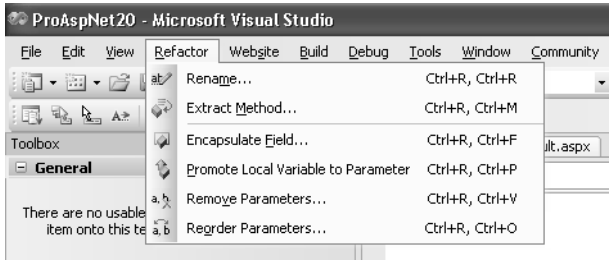


Figure 2-14 The new menu for helping developers to quickly refactor the code of classes.

As you can see, the menu provides advanced facilities for code editing. Among the other things, you can extract a block of code and transform it into a new method or rename a member all the way through. The refactor feature doesn't disappoint when it comes to managing

properties. Admittedly, one of most boring tasks when writing classes is turning a field into a property with *get* and *set* accessors. Imagine you have a field like the following one:

```
private int _counters;
```

At a certain point, you might realize that a full-featured property would be better. Instead of typing code yourself, you just select the line and refactor to encapsulate the field. Needless to say, the menu item is *Refactor|Encapsulate field*. With the power of a click, the old code becomes the following:

```
public int Counters
{
    get
    {
        return _counters;
    }

    set
    {
        _counters = value;
    }
}
```

You are free to change the public name of the property and, of course, to flesh out the bodies of the get/set accessors.

Import/Export of IDE Features

It is common for developers to move a project from one machine to another. This happens for a variety of reasons. For example, you might use multiple machines for test purposes; you continually swing between the client's site and your own office; you are an absolute workaholic who just can't spend a night home without working.

Typically, the various machines have Visual Studio .NET installed with the same set of features, and although it's not necessary, they share the same set of IDE settings. To feel comfortable with the environment, you might have developed macros, reordered menus and toolbars, added new controls to the toolbox, created new project templates, and assigned preferences for colors and fonts. This wealth of information is not easy to catalog, organize, and persist if you have to do that manually.

Figure 2-15 shows the new Import/Export Settings dialog box associated with the Tools menu. You select the IDE settings you want to persist and save them to a file. The file can be created anywhere and is given a *.vssettings* extension. In spite of the extension, it is an XML file.

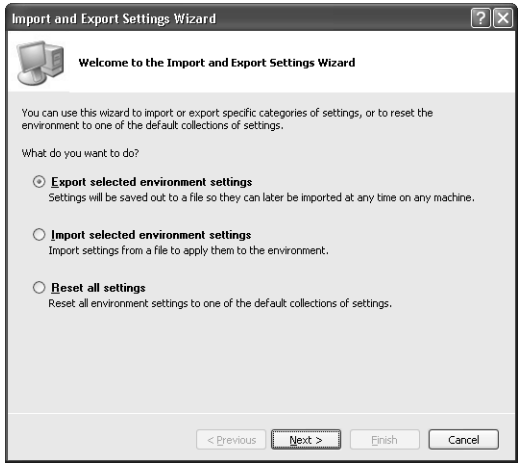


Figure 2-15 The wizard for importing and exporting IDE settings.

Adding Code to the Project

Adding code to the project mostly means that you added Web Forms to the project and now need to hook up some of the page events or events that controls in the form generate. In addition, you might want to add some classes to the project representing tailor-made functionalities not otherwise available.

Filling a Web Forms page is easy and intuitive. You open the form in layout mode and drag and drop controls from the toolbox onto it. Next, you move elements around and configure their properties. If needed, you can switch to the Source view and manually type the HTML markup the way you want it to be.

A pleasant surprise for many developers is that you can drag and drop controls from the toolbox directly into the Source view; instead of viewing the control graphically rendered, you see the corresponding markup code. Similarly, you can edit the properties of a server control by selecting it in the Design view or highlighting the related HTML in the Source view. In addition, each control deployed on the form can have its own design-time user interface through which you can configure properties for the run time.

Defining Event Handlers

Adding code to a Web Form page means handling some page's or control's events. How do you write an event handler for a particular page element? To try it out, place a button on a form and double-click. Visual Studio switches to the Source view and creates an empty event handler for the control's default event. For a button control, it is the *Click* event. The code you get looks similar to the following:

```
void Button1_Click(object sender, EventArgs e)
{
    ...
}
```

The HTML markup is automatically modified to contain an additional *OnClick* attribute:

```
<asp:button runat="server" id="Button1"
    text="Click"
    OnClick="Button1_Click" />
```

Notice that event binding is always done declaratively in the body of the *.aspx* page. Unlike its predecessor, Visual Studio .NET 2005 doesn't inject automatically generated code in the page for event wireup. Recall that in Visual Studio .NET 2003, double-clicking a button adds the following (C#) code to the code-behind class:

```
// VS.NET injects this code in the code-behind class of a page
// when you double-click a button to handle its default event
Button1.Click += new EventHandler(this.Button1_Click);
```

The code would obviously be different if Visual Basic .NET is your language of choice.

If you're dealing with a code-behind page, the event handler is defined in the code-behind class instead of being placed inline.

When you double-click on a control or on the body of the page, a handler for the default page or control event is generated. What if you need to write a handler for another event? You select the desired control and click on the Events icon in the Properties window. You get a view like that in Figure 2-16 and pick up the event you need.

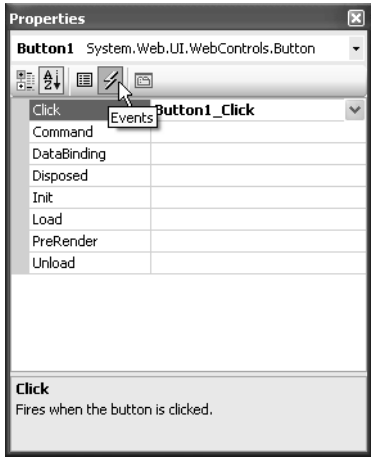


Figure 2-16 The Events view in the Properties window.

Writing Helper Classes

Writing helper classes is as easy as adding a new class to the project, as shown in Figure 2-17.

The class file can define any number of classes, even partial class definitions, and will actually be compiled to an assembly. Where should you deploy this class file in your project? You have

two options: either you create an additional project to generate a DLL component library or you drop the class file in a special folder below the application's virtual root—the *App_Code* folder.

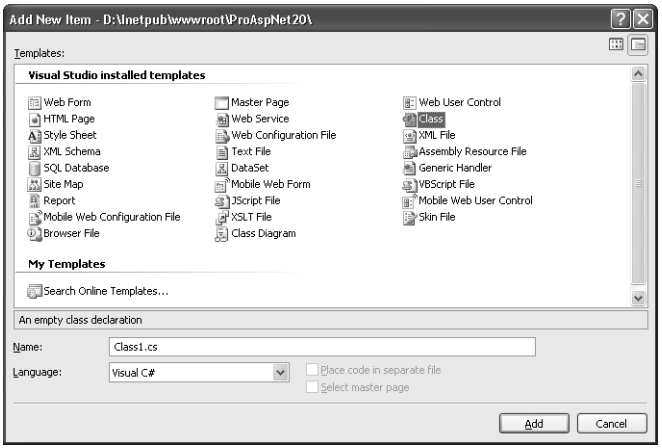


Figure 2-17 Adding a new class to an ASP.NET project.

In the former case, you add another project to the solution by using the File|Add menu. From the list of available projects, you pick up a Class Library project and then add any class files to it. When you're done, you reference the library project in the Web site project and go. Pleasantly enough, IntelliSense will just detect new classes and work as expected.

What's the *App_Code* folder, then? It is an application's subdirectory that has a special meaning to the ASP.NET runtime. The *App_Code* folder is designed to contain reusable components that are automatically compiled and linked to the page code. The folder stores source class files (*.vb* or *.cs*) that the ASP.NET runtime engine dynamically compiles to an assembly upon execution. Created in a predefined path visible to all pages in the site, the resulting assembly is updated whenever any of the source files are updated. It is important to note that any file copied to the *App_Code* folder is deployed as source code on the production box. (I'll say more about special ASP.NET directories in the next section.)

Building a Sample Shared Class

To experience the advantages of reusable source components, let's design a page that makes use of a nontrivial component that would be annoying to insert inline in each page that needs it. The page looks like the one in Figure 2-18.

Many products and services available over the Web require a strong password. The definition of a "strong password" is specific to the service, but normally it addresses a password at least eight characters long with at least one character from each of the following groups: uppercase, lowercase, digits, and special characters. We'll use that definition here. The sample page you

will build asks the user for the desired length of the password and suggests one built according to the rules just mentioned. You create a new file named *StrongPassword.cs* and place it in the purposedly created *App_Code* subdirectory. The class outline is shown here:

```
public class StrongPassword
{
    public StrongPassword()
    {...}

    public string Generate()
    {...}
    public string Generate(int passwordLength)
    {...}
}
```

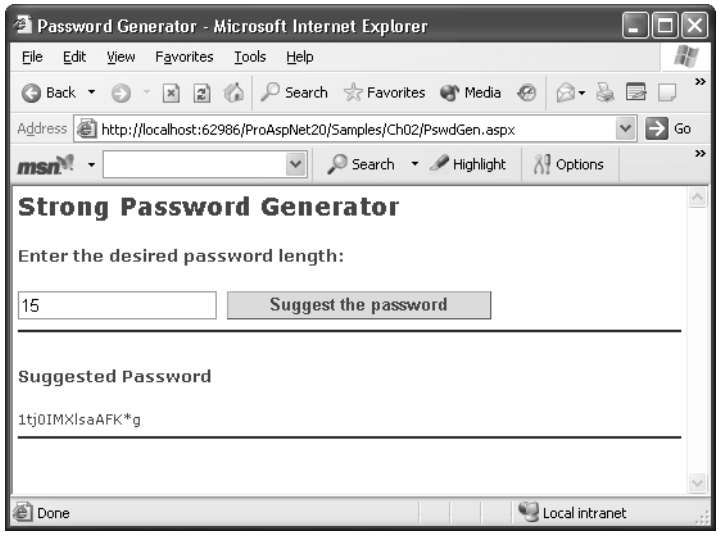


Figure 2-18 The PswdGen.aspx page to generate a new "strong" password of the specified length.

The class features one method—*Generate*—that will actually generate a new strong password. Of course, the definition of a "strong password" is arbitrary. Once placed in the *App_Code* directory, this class is compiled on demand and made available to all pages. In the sample page, the code to generate and validate a password becomes simpler and more readable:

```
void buttonGenerate_Click(Object sender, System.EventArgs e)
{
    // Gets the desired length of the password and ensures
    // it is really expressed as a number. (This is a simple but
    // effective pattern to prevent code/data injection.)
    int pswdLen = 8;
    bool result = Int32.TryParse(PswdLength.Text, out pswdLen);

    // Create and display the new password
    StrongPassword pswd = new StrongPassword();
    labelPassword.Text = pswd.Generate(pswdLen);
}
```

Figure 2-18 shows the page in action. Note that the same functionality can also be achieved by placing the code inline or packing the *StrongPassword* class in a separate assembly.

A Look at the *web.config* File

The behavior of an ASP.NET application is affected by the settings defined in various configuration files—*machine.config* and *web.config*. The *machine.config* file contains default and machine-specific values for all supported settings. Machine settings are normally controlled by the system administrator, and applications should never be given write access to it. An application can override most default values stored in the *machine.config* file by creating one or more *web.config* files.

At a minimum, an application creates a *web.config* file in its root folder. The *web.config* file is a subset of *machine.config*, written according to the same XML schema. Although *web.config* allows you to override some of the default settings, you cannot override all settings defined in *machine.config*.

If the application contains child directories, it can define a *web.config* file for each folder. The scope of each configuration file is determined in a hierarchical, top-down manner. The settings actually applied to a page are determined by the sum of the changes that the various *web.config* files on the way from *machine.config* to the page's directory carry. Any *web.config* file can locally extend, restrict, and override any type of settings defined at an upper level. If no configuration file exists in an application folder, the settings valid at the upper level are applied.

Visual Studio .NET usually generates a default *web.config* file for you. The *web.config* file is not strictly necessary for an application to run. Without a *web.config* file, though, you can't debug the application.

ASP.NET Reserved Folders

ASP.NET uses a number of special directories below the application root to maintain application content and data. In ASP.NET 1.x, only the Bin directory was used. ASP.NET 2.0 introduces seven additional protected directories. None of these directories are automatically created by ASP.NET 2.0 or Visual Studio .NET 2005, nor are the directories necessarily required to exist. Each directory needs to be created either manually by developers or on demand through Visual Studio .NET when a feature that requires it is enabled.

Additional Application Directories

Table 2-1 lists all the additional directories you can take advantage of. Note that the directories will be there only if they are required by your specific application. Don't be too worried about the number of new directories (that is, seven) you can potentially have. A reasonable estimate would be that only two or three (out of seven) additional directories will be present in an average ASP.NET application.

Table 2-1 Special Reserved Directories in ASP.NET Applications

Directory Name	Intended Goal
Bin	Contains all precompiled assemblies needed by the application.
App_Browsers	Contains browser capabilities information.
App_Code	Contains source class files (*.vb* or *.cs*) used by pages. All the files must be in the same language; you can't have both C# and VB.NET files in the folder.
App_Data	Contains data files for the application. This can include XML files and Access databases to store personalization data.
App_GlobalResources	Contains *.resx* resource files global to the application.
App_LocalResources	Contains all *.resx* resource files that are specific to a particular page.
App_Themes	Contains the definition of the themes supported by the application. (I'll say more about themes in Chapter 6.)
App_WebReferences	Contains *.wsdl* files linking Web services to the application.

The content in all the directories listed in Table 2-1 won't be accessible via HTTP requests to the server. The only exception is the content of the *App_Themes* folder.

Important The names of these folders aren't customizable. The reason lies in the way the ISAPI filter in charge of blocking HTTP requests to these folders work. For performance reasons, the ISAPI filter can't just access the *web.config* file to read about directory names to look for. That would require the filter to parse the XML file on *any* request and, as you can easily imagine, would be a major performance hit. Alternately, the names of the directories could have been written in the registry, which would make for a much faster and affordable access. Unfortunately, a registry-based approach would break XCopy deployment and introduce a major breaking change in the ASP.NET architecture. (See the "Application Deployment" section for more information on XCopy deployment.)

The contents of many folders listed in Table 2-1 are compiled to a dynamic assembly when the request is processed for the first time. This is the case for themes, code, resources, and Web references. (See the "Resources" section for more information on the ASP.NET 2.0 compilation model.)

The *App_Code* Directory

As mentioned, you can use the server *App_Code* directory to group your helper and business classes. You deploy them as source files, and the ASP.NET runtime ensures that classes will be automatically compiled on demand. Furthermore, any changes to these files will be detected, causing the involved classes to recompile. The resulting assembly is automatically referenced in the application and shared between all pages participating in the site.

You should put only components into the *App_Code* directory. Do not put pages, Web user controls, or other noncode files containing noncode elements into the subdirectory. The

resulting assembly has application scope and is created in the *Temporary ASP.NET Files* folder—well outside the Web application space.

> **Note** If you're worried about deploying valuable C# or VB.NET source files to the Web server, bear in mind that any (repeat, *any*) access to the *App_Code* folder conducted via HTTP is monitored and blocked by the aforementioned ASP.NET ISAPI filter.

Note that all class files in the *App_Code* folder must be written in the same language—be it Visual Basic .NET or C#—because they are all compiled to a single assembly and processed by a single compiler. To use different languages, you must organize your class files in folders and add some entries to the configuration file to tell build system to create distinct assemblies—one per language.

Here's an example. Suppose you have two files named source.cs and source.vb. Because they're written in different languages, they can't stay together in the *App_Code* folder. You can then create two subfolders—say, *App_Code/VB* and *App_Code/CS*—and move the files to the subfolder that matches the language. Next you can add the following entries to the *web.config* file:

```
<configuration>
<system.web>
<compilation>
    <codeSubDirectories>
        <add directoryName="VB" />
        <add directoryName="CS" />
    </codeSubDirectories>
</compilation>
</system.web>
</configuration>
```

Note that the <*codeSubDirectories*> section is valid only if it is set in the *web.config* file in the application root. Each section instructs the build system to create a distinct assembly. This means that all the files in the specified directory must be written in the same language, but different directories can target different languages.

> **Note** The *App_Code* directory can also contain XSD files, like those generated for typed *DataSets*. An XSD file represents the strongly typed schema of a table of data. In the .NET Framework 1.1, a typed *DataSet* must be manually created using the *xsd.exe* tool. In ASP.NET 2.0, all you have to do is drop the source XSD file in the *App_Code* folder.

The Resource Directories

A localizable Web page uses resources instead of hard-coded text to flesh out the user interface of contained controls. Once a resource assembly is linked to the application, ASP.NET can select the correct property at run time according to the user's language and culture. In

ASP.NET 1.x, developers had to create satellite assemblies manually. ASP.NET 2.0, on the other hand, creates resource assemblies parsing and compiling any resource files found in the two supported folders—*App_LocalResources* and *App_GlobalResources*.

A local resource is a resource file specific to a page. A simple naming convention binds the file to the page. If the page is named *sample.aspx*, its corresponding resource file is *sample.aspx.resx*. To be precise, this resource file is language neutral and has no culture defined. To create a resource assembly for a specific culture, say Italian, you need to name the resource file as follows: *sample.aspx.it.resx*. Generally, the *it* string should be replaced with any other equivalent string that identifies a culture, such as *fr* for French or *en* for English. Figure 2-19 shows the a sample local resource folder.

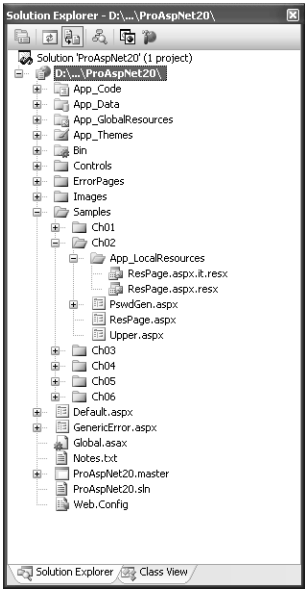


Figure 2-19 The local resource directory for the *respage.aspx* page.

Local resources provide a sort of implicit localization where the page itself automatically ensures that each contained control is mapped to a particular entry in the *.resx* file. Here's how a simple page changes once you add support for local resources.

```
<%@ Page Language="C#" meta:resourcekey="PageResource1" UICulture="auto" %>

<html>
<head id="Head1" runat="server">
    <title>Pro ASP.NET (Ch 02)</title>
</head>
<body>
<h1>
   <asp:Label runat="server" id="H1" meta:resourcekey="LabelResource1" />
</h1>
<form id="Form1" runat="server">
```

```
    <asp:Button ID="btn" Runat="server" meta:resourcekey="BtnResource1" />
</form>
</body>
</html>
```

The page itself and each constituent control are given a resource key. The *.resx* file contains entries in the form *ResourceKey.PropertyName*. For example, the *Text* property of the button is implicitly bound to the *BtnResource1.Text* entry in the *.resx* file. You don't have to write a single line of code for this mapping to take place. You are only requested to populate the resource files as outlined. The *UICulture* attribute set to *auto* tells the ASP.NET runtime to use the current browser's language setting to select the right set of resources.

Tip To quickly test a page against different languages, you open the Internet Explorer Tools menu and click the Languages button. Next, you add the language of choice to the list box of supported languages and move the language of choice to the first position in the list. Click OK and exit. From now on, Internet Explorer will be sending the selected language ID with each served request.

Figure 2-20 shows how the same page looks when different languages are set.

Figure 2-20 The *respage.aspx* file in English and Italian.

Implicit localization works automatically, meaning that you don't need to specify how to read information about each property from a resource file. However, at times you need more direct control over how properties are set. For this, you turn to global resources. When you choose to add a resource file to the application, Visual Studio .NET creates the *App_GlobalResources* directory and places a new *.resx* file in it. You can rename this file at will and fill it with strings, images, sounds, and whatever else is suitable to you. (See Figure 2-21.)

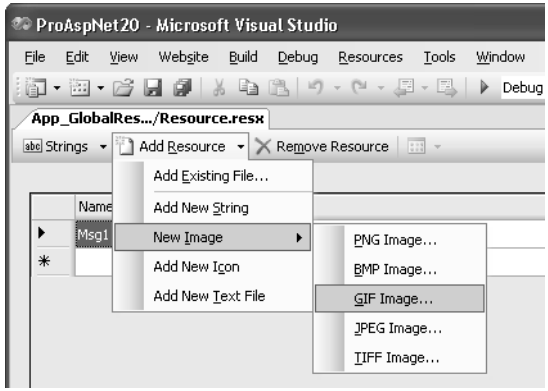


Figure 2-21 The Visual Studio .NET Resource editor in action.

Within the page or controls code, you reference resources using an expression, as in the following code:

```
<asp:Label Runat="server" Text="<%$ Resources:Resource, Msg1 %>" />
```

Resources is the namespace of the object, whereas *Resource* is the name of the *.resx* file that contains the resources. Finally, *Msg1* is the name of the entry to use. Explicit localization is useful when you have large bodies of text or custom messages you want to localize.

> **Note** The resulting resource assembly for the neutral culture has application scope and is therefore referenced from other assemblies generated in the application. All types defined in the resource assemblies belong to the *Resources* namespace and are static objects.

Linked Web Services

When you add a reference to a Web service, the *.wsdl* file for the Web service is downloaded and copied to the *App_WebReferences* directory. At runtime, any WSDL file found in the Web reference directory is dynamically compiled in a C# or Visual Basic .NET proxy class in much the same way as business classes in the *App_Code* directory are processed.

Note that in ASP.NET 1.x, you have to reference the Web service and have Visual Studio .NET 2003 to generate the proxy class explicitly. If you can obtain a WSDL file in other ways (that is, you don't download it through the Add Web Reference Wizard), you can add it manually to the *App_WebReferences* directory.

Available Themes

The *App_Themes* directory defines one or more themes for controls. A *theme* is a set of skins and associated files such as style sheets and images that can be used within an application to give a consistent user interface to controls. In the *App_Themes* directory, each theme

occupies a single subdirectory, which has the same name as the theme. All related files are stored in this directory.

When a theme is loaded, the contents of the theme directory are parsed and compiled into a class that inherits from a common base class. Any theme defined outside the *App_Themes* directory is ignored by the ASP.NET build system.

Build the ASP.NET Project

To build and run the ASP.NET application, you simply click the Start button on the toolbar (or press F5) and wait for a browser window to pop up. In Visual Studio .NET 2005, no compile step takes place to incorporate code-behind and helper classes. All the dynamically generated assemblies, and all precompiled assemblies deployed into the *Bin* folder, are linked to the application as used to visit pages.

Once any needed assemblies have been successfully built, Visual Studio .NET autoattaches to the ASP.NET run-time process—typically, *w3wp.exe*—for debugging purposes. Next, it opens the start page of the application.

Important The ASP.NET run-time process might differ based on the process model in use. The process model is configurable; the default model, though, depends on the underlying operating systems and Web server settings. If your Web server runs Windows 2000 Server, or perhaps any version of Windows XP, the run-time process is *aspnet_wp.exe*. It runs under a weak user account named ASPNET and is designed to interact with IIS 5.x. If you run Windows 2003 Server and IIS 6.0 and didn't change the default process model, the run-time process is *w3wp.exe*, which is the standard worker process of IIS 6.0. The *w3wp.exe* process runs under the NETWORK SERVICE account. This process doesn't know anything about ASP.NET, but its behavior is aptly customized by a version-specific copy of the ASP.NET ISAPI extension. Under IIS 6.0, you can even switch back to the IIS 5 process model. If you do so, though, you lose a lot in performance.

When you build the project, Visual Studio .NET 2005 might complain about a missing *web.config* file, which is necessary if you want to debug the code. If you just want to run the page without debugging it, click the Run button in the message box you get. Otherwise, you let Visual Studio generate a proper *web.config* file for you. If you create your own *web.config* file, make sure it contains the following string to enable debugging:

```
<compilation debug="true" />
```

Once this is done, you can commence your debugging session.

Debugging Features

As long as you compiled the project in debug mode (which is the default indeed), you can set a few breakpoints in the sources and step into the code, as shown in Figure 2-22.

```
12 public partial class PswdGen_aspx
13 {
14     void buttonGenerate_Click(Object sender, System.EventArgs e)
15     {
16         // Get the password length
17         int pswdLen = 8;
18         bool result = Int32.TryParse(PswdLength.Text, out pswdLen);
19
20         StrongPassword pswd = new StrongPassword();
21         labelPassword.Text = pswd.Generate(pswdLen);
22     }
23 }
24
```

Figure 2-22 Stepping into the code of the page by using the Visual Studio .NET integrated debugger.

The Debug menu is a little richer in Visual Studio .NET 2005 than it was in the previous version. You now have more choices as far as exceptions and breakpoints are concerned. In particular, you can configure the IDE so that it automatically breaks when an exception is thrown. The feature can be fine-tuned to let you focus on any exceptions, any exceptions in a specified set, or all exceptions not handled by the current application.

Breakpoints can be set at an absolute particular location or in a more relative way when the execution reaches a given function. Braveheart debuggers also have the chance to break the code when the memory at a specified address changes.

Watch windows feature a richer user interface centered around the concept of visualizers. A visualizer is a popup window designed to present certain types of data in a more friendly and readable manner—XML, text, or *DataSets*. (See Figure 2-23.)

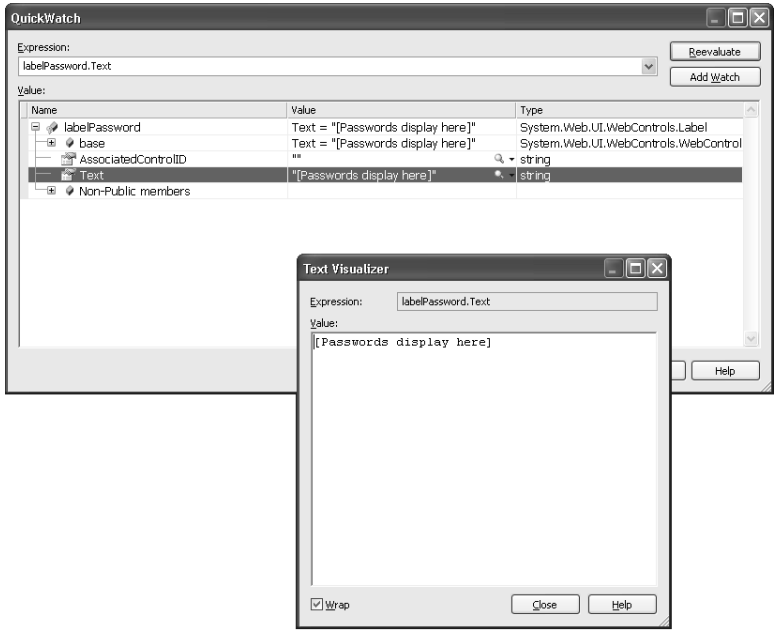


Figure 2-23 The text visualizer invoked from the quick-watch window during a debug session.

Visualizers are also active from within code tip windows. A code tip is the made-to-measure ToolTip that previews the value of variables during a debug session. (See Figure 2-24.)

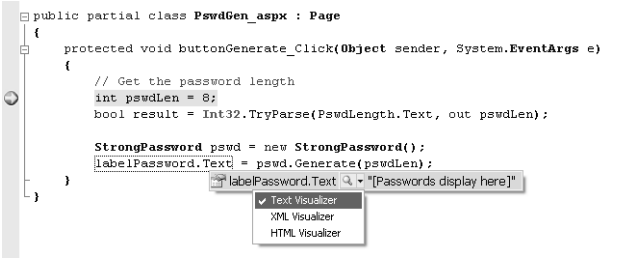


Figure 2-24 Invoking a visualizer from within a code tip.

Visualizers are defined for a few types of data. Personally, I just love the *DataSet* visualizer. Checking what's in a *DataSet* instance couldn't be easier.

Testing the Application

As mentioned, there are two ways of testing pages in Visual Studio .NET 2005. You can have pages served by IIS (if installed) or by the embedded local Web server. By default, Visual Studio .NET uses IIS if you open the project as a Web site and indicate its URL; otherwise, it defaults to the local Web server. An important point to consider about the embedded Web server concerns the security context. Under IIS, an ASP.NET application is served by a worker process running under the *real* account defined for the application—typically, a highly restricted account such as ASP.NET or NETWORK SERVICE.

In contrast, the embedded Web server takes the security token of the currently logged-on user—that is, you. This means that if the developer is currently logged on as an administrator—a much more common scenario than it should be—the application receives administrative privileges. The problem here is not a risk of being attacked; the real problem is that you are actually testing the application in a scenario significantly different from the real one. Things that work great under the local Web server might fail miserably under IIS.

For simple applications that only read and run ASP.NET pages, this problem is not that relevant. However, the results of your testing under the local server will become less reliable if you access files other than Web pages, files located on other machines, files in the Windows registry, or files on a local or remote database. In all these cases, you must make sure that the real ASP.NET account has sufficient permissions to work with those resources.

The bottom line is that even though you can use the local Web server to test pages, it sometimes doesn't offer a realistic test scenario.

Application Deployment

Installing a .NET application in general, and an ASP.NET application in particular, is a matter of performing an *XCopy*—that is, a recursive copy of all the files—to the target folder on the target machine. Aside from some inevitable emphasis and promotion, the *XCopy deployment* expression, which is often used to describe setup procedures in .NET, communicates the gist of .NET deployment: you don't need to do much more than copy files. In particular, there's no need to play with the registry, no components to set up, and no local files to create. Or, at least, nothing of the kind is needed just because the .NET Framework mandates it.

XCopy Deployment

The deployment of a Web application is a task that can be accomplished in various ways depending on the context. As far as copy is concerned, you can use any of the following: FTP transfer, any server management tools providing forms of smart replication on a remote site, or an MSI installer application. In Visual Studio .NET 2005, you can even use the Copy Web Site function that we discussed earlier in this chapter.

Each option has pros and cons, and the best fit can only be found once you know exactly the runtime host scenario and the purpose of the application is clearly delineated. Be aware that if you're going to deploy the application on an ISP host, you might be forced to play by the rules (read, "use the tools") that your host has set. If you're going to deliver a front end for an existing system to a variety of servers, you might perhaps find it easier to create a setup project. On the other hand, FTP is great for general maintenance and for applying quick fixes. Ad hoc tools, on the other hand, could give you automatic sync-up features. Guess what? Choosing the right technique is strictly application-specific and is ultimately left to you.

Copying Files

FTP gives you a lot of freedom, and it lets you modify and replace individual files. It doesn't represent a solution that is automatic, however: whatever you need to do must be accomplished manually. Assuming that you have gained full access to the remote site, using FTP is not much different than using Windows Explorer in the local network. I believe that with the Copy Web Site functionality of Visual Studio .NET 2005 in place, the need for raw FTP access is going to lessen. If nothing else, the new Copy Web Site function operates as an integrated FTP-like tool to access remote locations.

The new copy function also provides synchronization capabilities too. It is not like the set of features that a specifically designed server management tool would supply, but it can certainly work well through a number of realistic situations. At the end of the day, a site replication tool doesn't do much more than merely transfer files from end to end. Its plusses are the user interface, and the intelligence, built around and atop this basic capability. So a replication tool maintains a database of files with timestamps, attributes, properties and can sync up versions of the site in a rather automated way, minimizing the work on your end.

Building a Setup Project

Another common scenario involves using an out-of-the-box installer file. Deploying a Web application this way is a two-step operation. First, create and configure the virtual directory; next, copy the needed files. Visual Studio .NET makes creating a Web setup application a snap. You just create a new type of project—a Web Setup Project—select the files to copy, and build the project. Figure 2-25 shows the user interface of the setup project.

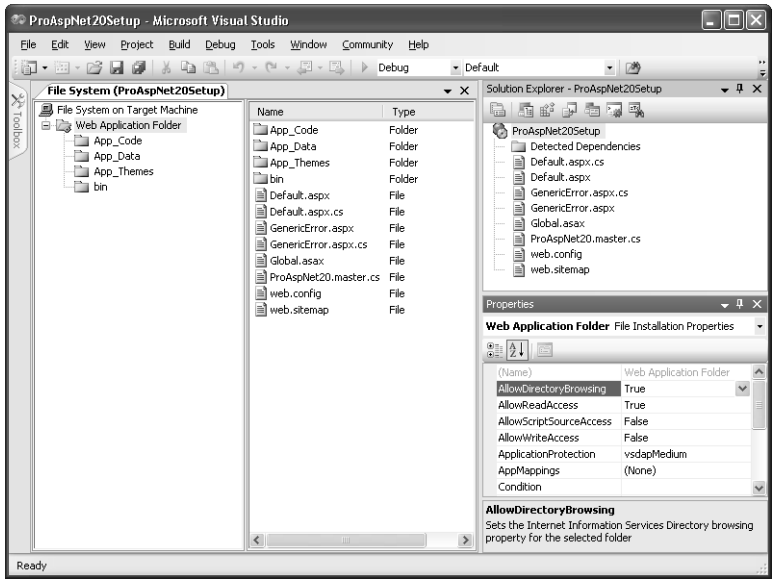


Figure 2-25 Creating a Web setup project.

The Web Application Folder node represents the virtual directory of the new application on the target machine. The Properties box lets you configure the settings of the new virtual directory. For example, the *AllowDirectoryBrowsing* property lets you assign browsing permission to the IIS virtual folder you will create. You can also control the virtual directory name, application execute permissions, level of isolation, and default page. The *Bin* subfolder is automatically created, but you can ask the setup to create and populate as many subfolders as you need.

When you build the project, you obtain a Windows Installer *.msi* file that constitutes the setup to ship to your clients. The default installer supports repairing and uninstalling the application. The setup you obtain in this way—which is the simplest you can get—does not contain the .NET Framework, which must be installed on the target machine or explicitly included in the setup project itself.

What Else Do You Need to Do?

One of the coolest features of .NET assemblies is that they are self-describing components. An application that wants to know about the internal characteristics of an assembly has only to ask! The .NET reflection of an API is the programming interface by which a client

can interrogate an assembly. This fact eliminates the need of using the registry (or any other sort of centralized repository) to track paths and attributes of binary components. Another pleasant effect of the assembly's structure is that side-by-side execution is now a snap, and ASP.NET applications take advantage of it on a regular basis. In practice, whenever you update a page, two versions of the "same" assembly live side by side for awhile without interference and conflict.

So the XCopy deployment model rocks. Is there something more you need to do to finalize the deployment of your application? Sure there is. Let's detail some additional tasks.

If you use read/write files (XML files, configuration files, Access databases), you need to grant proper writing permission to the application. Likewise, if your application or control generates temporary files, you need to make accommodations for a proper folder with proper writing permissions. These tasks must be accomplished in one way or another before the application goes live. Note that in an ISP scenario you are normally given an isolated disk subtree with full write permissions granted to the ASP.NET account. You must design your applications to be flexible enough to support a configurable path for all their temporary files.

Note We're not saying anything specific about database configuration here. We're simply assuming that all required databases are in place, properly working, and entirely configured. If this is not the case, you might want to add this task to the list too. The same holds true for any remote application and network services you might need, including Web services and COM+ components.

Configuring the Runtime Environment

Another aspect to consider is runtime configuration. When you develop the ASP.NET code, you test it on a machine with its own *machine.config* file. When you deploy the application on a production box, you might not be able to restore the same settings. One possible reason is that the administrator does not want you to modify the current settings because they proved to be great for other applications. (This is especially true in an ISP host scenario.)

You can work around the issue by simply replicating any needed *machine.config* settings to the application's *web.config*. However, if you are deploying your code to a service provider, you might find that many *machine.config* settings have been locked down and cannot be overridden. In this case, you should ask (or more exactly, beg) the administrator to let you tweak the server's configuration in a way that suits you without hurting other applications. This normally entails creating an application-specific <*location*> section in the server's *machine.config* file.

Deploying an ASP.NET application in a Web-farm scenario poses a few extra configuration issues you must be aware of. All *machine.config* files in the Web farm must be synchronized to

the value of a few attributes. You can achieve this in two ways, the simplest of which is packing all attribute values in the application's *web.config*. This approach greatly simplifies the deployment because you only have to run the setup on all machines and no other changes are needed. If any of the required sections are locked down (once more, this is likely to happen in an ISP scenario), you find yourself in the situation described previously, that of begging the administrator to create a new <*location*> section for you.

Note The <*location*> section can be used in both *machine.config* and *web.config* to limit Web settings to the specified application path. In a deployment scenario, the section assumes particular importance in the *machine.config* file and subsequently requires administrative privileges. The <*location*> section is normally used in a *web.config* file in case of a deployment with a main application and a bunch of subapplications.

Site Precompilation

As mentioned, dynamically created assemblies are placed in an internal folder managed by the ASP.NET runtime. Unless source files are modified, the compilation step occurs only once per page—when the page is first requested. Although in many cases the additional overhead is no big deal, removing it still is a form of optimization. Site precompilation consists of deploying the whole site functionality through assemblies. A precompiled application is still made up of source files, but all pages and resources are fictitiously accessed before deployment and compiled to assemblies. The dynamically created assemblies are then packaged and installed to the target machine. As you can see, site precompilation also saves you from deploying valuable source files, thus preserving the intellectual property.

Important Source files, like C# classes or WSDL scripts, are protected against HTTP access. However, they are at the mercy of a hacker in the case of a successful exploitation that allows the attacker to take control of the Web directories.

Site precompilation was possible in ASP.NET 1.x, but in version 2.0 it has the rank of a system tool, fully supported by the framework. In summary, site precompilation offers two main advantages:

- Requests to the site do not cause any delay because the pages and code are compiled to assemblies.
- Sites can be deployed without any source code, thus preserving and protecting the intellectual property of the solutions implemented.

Precompilation can take two forms: in-place precompilation and deployment precompilation.

Note To protect intellectual property, you can also consider obfuscation in addition to site precompilation. *Obfuscation* is a technique that nondestructively changes names in the assembly metadata, thus preventing potential code-crackers from scanning your files for sensitive strings. Obfuscation does not affect the way the code runs, except that it compacts the executable, making it load a bit faster. If decompiled, an obfuscated assembly generates a much less readable intermediate code. Although applicable to all .NET applications, there is nothing wrong with obfuscating your ASP.NET assemblies in case of hacker access for the very same reasons. Visual Studio .NET 2005 provides the community edition of a commercial tool—Dotfuscator.

In-Place Precompilation

In-place precompilation allows a developer or a site administrator to access each page in the application as if it were being used by end users. This means each page is compiled as if for ordinary use. The site is fully compiled before entering production, and no user will experience a first-hit compilation delay, as in version 1.x. In-place precompilation takes place after the site is deployed, but before it goes public. To precompile a site in-place, you use the following command, where */proaspnet20* indicates the virtual folder of the application:

```
aspnet_compiler –v /proaspnet20
```

If you precompile the site again, the compiler skips pages that are up to date and only new or changed files are processed and those with dependencies on new or changed files. Because of this compiler optimization, it is practical to compile the site after even minor updates.

Precompilation is essentially a batch compilation that generates all needed assemblies in the fixed ASP.NET directory on the server machine. If any file fails compilation, precompilation will fail on the application.

Precompilation for Deployment

Precompilation for deployment generates a file representation of the site made of assemblies, static files, and configuration files—a sort of manifest. This representation is generated on a target machine and can also be packaged as MSI and then copied and installed to a production machine. This form of precompilation doesn't require source code to be left on the target machine.

Precompilation for deployment also requires the use of the *aspnet_compiler* command-line tool:

```
aspnet_compiler –m metabasePath
                -c virtualPath
                -p physicalPath
                targetPath
```

The role of each parameter is explained in Table 2-2.

Table 2-2 Parameters of the *aspnet_compiler* Tool

Parameter	Description
metabasePath	An optional parameter that indicates the full IIS metabase path of the application
virtualPath	A required parameter that indicates the virtual path of the application
physicalPath	An optional parameter that indicates the physical path of the application
targetPath	An optional parameter that indicates the destination path for the compiled application

If no target path is specified, the precompilation takes place in the virtual path of the application, and source files are therefore preserved. If a different target is specified, only assemblies are copied, and the new application runs with no source file in the production environment. The following command line precompiles ProAspNet20 to the specified disk path:

```
aspnet_compiler –v /ProAspNet20 c:\ServerPath
```

Static files such as images, *web.config*, and HTML pages are not compiled—they are just copied to the target destination.

Warning If you don't want to deploy HTML pages as clear text, rename them to *.aspx* and compile them. A similar approach can be used for image files. Note, however, that if you hide images and HTML pages behind ASP.NET extensions, you lose in performance because IIS is used to process static files more efficiently than ASP.NET.

Precompilation for deployment comes in two slightly different forms—with or without support for updates. Sites packaged for deployment only are not sensitive to file changes. When a change is required, you modify the original files, recompile the whole site, and redeploy the new layout. The only exception is the site configuration; you can update *web.config* on the production server without having to recompile the site.

Sites precompiled for deployment and update are made of assemblies obtained from all files that normally produce assemblies, such as class and resource files. The compiler, though, doesn't touch *.aspx* page files and simply copies them as part of the final layout. In this way, you are allowed to make limited changes to the ASP.NET pages after compiling them. For example, you can change the position of controls or settings regarding colors, fonts, and other visual parameters. You can also add new controls to existing pages, as long as they do not require event handlers or other code.

In no case could new pages be added to a precompiled site without recompiling it from scratch.

Of the two approaches to precompilation for deployment, the former clearly provides the greatest degree of protection for pages and the best performance at startup. The option that provides for limited updates still requires some further compilation when the site runs the first time. In the end, opting for the deployment and update in ASP.NET 2.0 is nearly identical to the compilation and deployment model of ASP.NET 1.1, where *.aspx* files are deployed in source and all classes (including code-behind classes) are compiled to assemblies.

Administering an ASP.NET Application

In addition to working pages, well-done graphics, and back-end services and components, a real-world Web application also requires a set of administrative tools to manage users, security, and configuration. In most cases, these tools consist of a passable and quickly arranged user interface built around a bunch of database tables; application developers are ultimately responsible for building them. To save time, these tools are often created as Windows Forms applications. If the application is properly designed, some business and data access objects created for the site can be reused. Are these external and additional applications always necessary?

While an ad hoc set of utility applications might be desired in some cases, having an integrated, rich, and further customizable tool built into Visual Studio .NET would probably be helpful and sufficient in many cases. In Visual Studio .NET 2005, you find available a whole Web application to administer various aspects of the site. The application, known as the Web Site Administration Tool (WSAT), is available through the Web site menu (or the Solution Explorer toolbar) and is extensively based on the ASP.NET provider model.

The Web Site Administration Tool

Figure 2-26 presents the administration tool in its full splendor. The tool is articulated in four blocks (plus the home), each covering a particular area of administration—membership, user profiles, application settings, and providers.

As mentioned, WSAT is a distinct application that the ASP.NET 2.0 setup installs with full source. You find it under the *ASP.NETWebAdminFiles* directory, below the ASP.NET build installation path. This path is:

```
%WINDOWS%\Microsoft.NET\Framework\[version]\CONFIG\Browsers
```

You can also run the tool from outside Visual Studio .NET 2005. In this case, though, you must indicate a parameter to select the application to configure. Here's the complete URL to type in the browser's address bar for an application named ProAspNet20.

```
http://localhost:XXXX/asp.netwebadminfiles/default.aspx?applicationUrl=/ProAspNet20
```

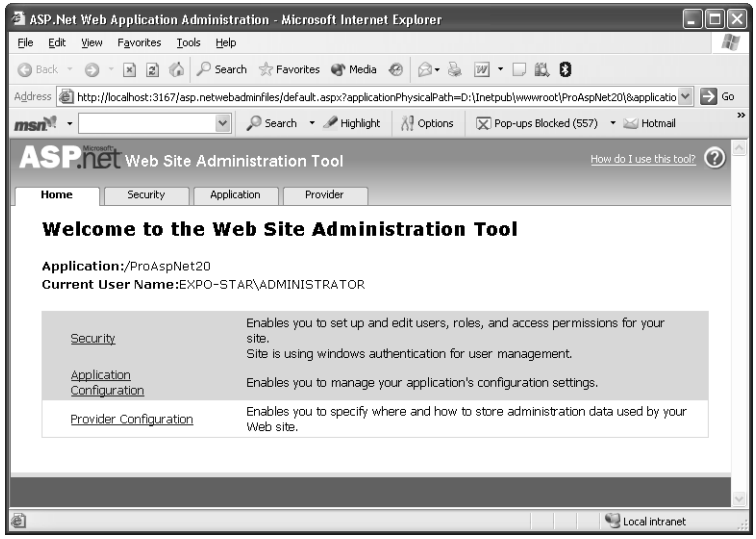


Figure 2-26 The Visual Studio .NET 2005 ASP.NET Administration Tool.

The *XXXX* indicates the port used by the local Web server. The WSAT application, in fact, is not publicly exposed through IIS for obvious security reasons. Table 2-3 details what you can expect to do with the tool.

Table 2-3 Classes of Settings Defined Through WSAT

Configuration Tab	Description
Security	Enables you to set up and edit users, roles, and access permissions for your site
Application	Enables you to manage your application's configuration settings, such as debugging and SMTP options
Provider	Enables you to select the provider to use for each ASP.NET feature that supports providers

Membership and Role Management

The Security tab of WSAT lets you manage all the security settings for your application. You can choose the authentication method, set up users and passwords, create roles and groups of users, and create rules for controlling access to specific parts of your application. A wizard will guide you throught the steps needed to set up individual users and roles. By default, membership and roles information are stored in a local SQL Server database (aspnetdb.mdf) stored in the *App_Data* folder of your Web site. If you want to store user information in a different storage medium, use the Provider tab to select a different provider.

In ASP.NET 1.1, it is fairly common to have a custom database store credentials for authorized users. The point is that this database must be filled out at some time; in addition, the site administrator must be able to manage users and especially roles. In ASP.NET 1.1, you have a

few options: charge your developers with this additional task, be charged by external consultants with this extra cost, or buy a third-party product. If you can find the product that suits you to perfection in terms of functionalities and costs, you're probably better off buying this instead of building code yourself. With homebrew code, you end up with a smaller set of features, often renounce the implementation of important security guidelines (for example, force password change every *n* days), and usually spend at least as much money, if not more, for a system with less capabilities and likely less reliability.

On the other hand, a WSAT-like tool doesn't sound like a mission-impossible task. However, it is the kind of cost that you might cut out of your budget. Finding a WSAT-like tool integrated in the development environment sounds like the perfect fit. It lets you accomplish basic administration tasks at no extra cost; and if you need more features, you can always turn to third-party products or, because you have the source code, you can inject your own extensions quite seamlessly.

Application Settings Management

Sometimes ASP.NET applications consume information (UI settings, favorites, general preferences, and connection strings) that you don't want to hard-code into pages. While applications can work out their own solutions for keeping data as configurable as possible (for example, databases or XML files), still the *<appSettings>* section in the *web.config* file provides an easy way out. The *<appSettings>* section, in fact, is specifically designed to store application-specific settings that can be expressed in a simple name/value fashion. The WSAT Application tab provides a convenient way to edit this section and create or edit entries.

As you can see in Figure 2-27, you can use the Application tab also to set debugging/tracing options and manage SMTP settings. In particular, mail settings determine how your Web application sends e-mail. If your e-mail server requires you to log on before you can send messages, you'll use the page to specify the type of authentication that the server requires, and if necessary, any required credentials.

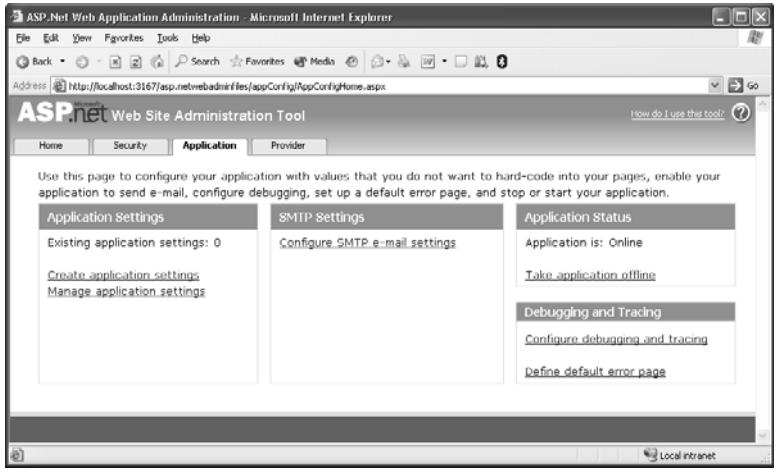


Figure 2-27 The Application tab in the Web Site Administration Tool.

The Application tab also contains a page for you to set error pages to show for particular HTTP errors.

Selecting and Configuring Providers

Profile and membership information require a persistent storage medium for user-specific data. The ASP.NET 2.0 provider model (discussed in Chapter 1) supplies a plug-in mechanism for you to choose the right support for storing data. ASP.NET 2.0 installs predefined providers for membership, roles, and personalization based on SQL Server local files. Extensibility, though, is the awesome feature of providers, and it gives you a way to write and plug in your own providers.

If you want to change the default provider for a particular feature, you use the Provider tab. You also use the same page to register a new provider, as in Figure 2-28.

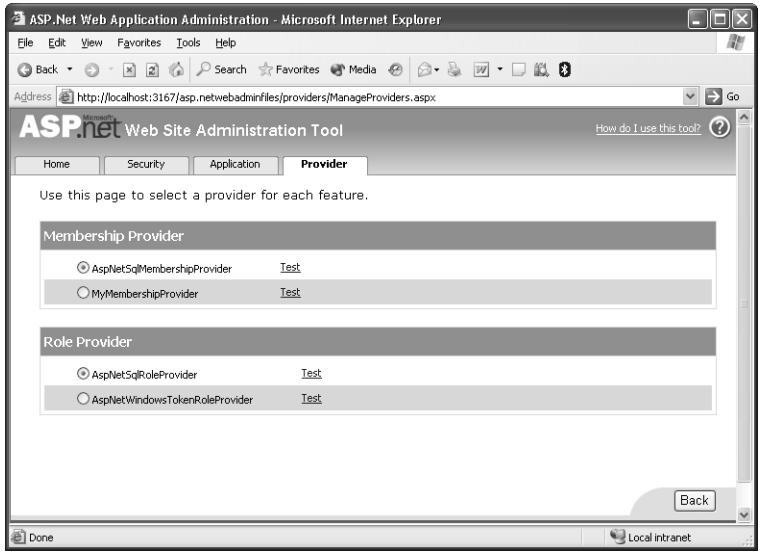


Figure 2-28 The Provider tab in the Web Site Administration Tool.

Editing ASP.NET Configuration Files

WSAT is mostly an administrative tool and, although it allows you to edit certain areas of the configuration files, you can't just consider it to be a *web.config* editor. Visual Studio .NET 2005 has improved the text editor that takes care of *web.config* files and has made it offer full IntelliSense support. Even though IntelliSense helps quite a bit, editing *web.config* through the IDE still requires a lot of tapping on the keyboard and typing many angle brackets on your own. Where else can you turn to edit *web.config* files more seamlessly?

A Visual Editor for *web.config* Files

ASP.NET 2.0 provides an interactive tool for configuring the runtime environment and ultimately editing the *web.config* file. The tool is an extension (that is, a custom property page) to the IIS Microsoft Management Console (MMC) snap-in. As a result, a new property page (named ASP.NET) is added to each Web directory node. (See Figure 2-29.)

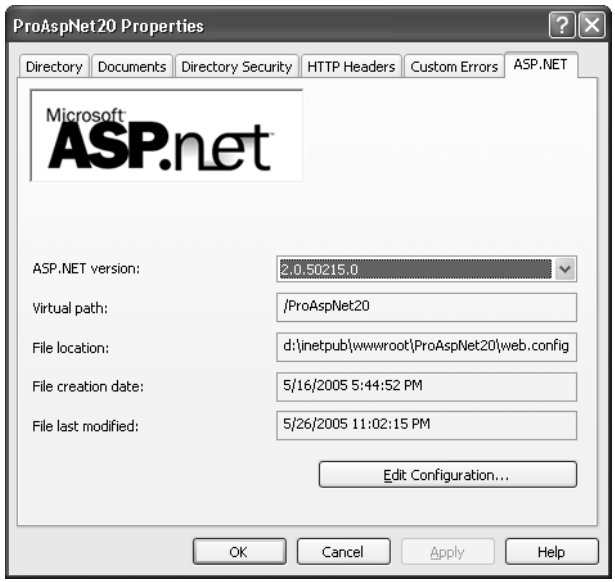


Figure 2-29 The ASP.NET MMC snap-in.

To reach the aforementioned property page, you open the IIS MMC snap-in (from the Control Panel) and select the desired Web application in the specified Web site. Next, you right-click to see the properties of this application and select the tab named ASP.NET. At this point, you should get what's presented in Figure 2-29. To start the *web.config* editor, click the Edit Configuration button. At this point, you get a new set of property pages that together supply an interactive user interface for editing the *web.config* files. The code behind this ASP.NET administrative tool leverages the new configuration API that allows you to read and write the contents of *.config* files. Figure 2-30 shows how to configure session state management for the current Web application.

The editor lets you edit virtually everything you might ever need to change in a *web.config* file. Any changes you enter are saved to a *web.config* file in the current directory—be it the application's root or the subdirectory from where you clicked. In other words, if you want to create or edit the *web.config* file of a subdirectory, locate that directory in the IIS snap-in tree, right-click to turn the editor on, and edit the local configuration.

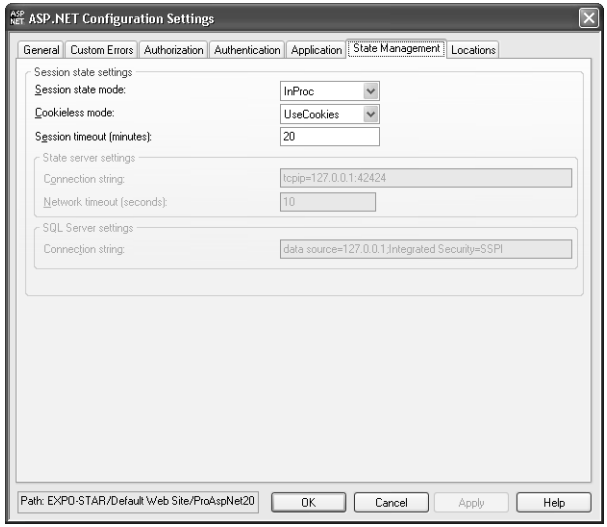


Figure 2-30 The bolted-on visual editor for *web.config* files.

When to Use the Configuration Editor

The snap-in tool works only with *web.config* files located on the local machine. You can use it to edit the *web.config* file you're concurrently editing through Visual Studio .NET, provided that you have a local installation of IIS and that you have opened the project through IIS. Aside from that, the *web.config* editor mostly remains a server-side, administrator-level tool to tweak, rather than edit/create, the *web.config* files of a site on a staging, or even production, box.

Note I love the snap-in *web.config* editor and would like to have it integrated with Visual Studio .NET, at the very minimum as an external tool. Unfortunately, there's no way to bring up the IIS MMC snap-in in a specific application using a command line. For future versions of IIS (starting with IIS 7), there are plans to provide a richer administration UI that can be integrated directly into the Visual Studio .NET shell. Let's wait and see.

Conclusion

Visual Studio .NET 2005 is the made-to-measure tool to build ASP.NET applications. Built to integrate the functionality of multiple visual designers in a common container environment, Visual Studio .NET is capable of providing a unique editing experience to Web and Windows developers. In situations where Microsoft came up short with its predecessor—Visual Studio .NET 2003—the newest version rocks. It's impressive to see how strong points in the new version overcome the shortcomings in the previous version.

In this chapter, we traversed the main phases of Web application development through Visual Studio .NET 2005—the page design, maintenance, and evolution of a Web project; and the deployment and administration of the final set of pages, files, and assemblies. The goal of this chapter was to provide the details of ASP.NET development with Visual Studio .NET 2005—what's great, what's been improved, what's new, and what you should know. If you successfully worked with previous version of Visual Studio .NET, you'll just fall in love with this one. If you had complaints about Visual Studio .NET 2003, you'll be pleased to see that most of the problems you complained about have been resolved.

I also spent some time discussing themes such as deployment and administration. Both are essential steps in finalizing a project, but both steps are often overlooked and often end up forcing developers and customers to swallow bitter pills. In some cases, deployment and administration require ad hoc tools; in as many other cases, though, a small handful of applications can let developers and administrators do their work smoothly and effectively. The point of contrast discussed in the chapter was who writes what and at what cost? Visual Studio .NET 2005 makes it easy to agree on the following points: essential tasks are cost-free, well done, and all included in the product.

Just the Facts

- IIS is no longer a strict requirement for developing ASP.NET applications, as Visual Studio .NET 2005 incorporates a local, mini Web server to be used only for testing during the development cycle.
- Visual Studio .NET 2005 supports multiple ways to open Web sites. In addition to using FPSE, you can access your source files by using FTP, IIS, and even the file system path.
- Visual Studio .NET 2005 supports standalone file editing and doesn't require a project to edit a single file on disk.
- An ASP.NET 2.0 application can be made of folders that receive special treatment from the ASP.NET runtime—for example, App_Code for classes, App_Themes for themes, and App_GlobalResources for satellite assemblies.
- Even though you can use the local Web server to test pages, be aware that it doesn't offer a realistic test scenario (such as having different accounts, different settings, and so forth). Don't rely on it to determine conclusively that your application works as expected.
- ASP.NET supports two forms of site precompilation: in-place precompilation and deployment precompilation.
- In-place precompilation applies to deployed applications and simply precompiles all pages to save the first-hit compilation delay.
- Precompilation for deployment creates a file representation of the site made of assemblies and static files. This representation can be generated on any machine, and it can be packaged to MSI and deployed.
- Precompilation for deployment doesn't leave source files on the production server, thus it preserves your intellectual property