Table of Contents

Chapter 2. Web Development in Microsoft Visual Studio 2008................. 1
  Introducing Visual Studio 2008..................................................................................................................................................... 2
  Create an ASP.NET Web Site Project............................................................................................................................................ 17
  Application Deployment............................................................................................................................................................... 37
  Administering an ASP.NET Application...................................................................................................................................... 43
  Conclusion..................................................................................................................................................................................... 49
Chapter 2

Web Development in Microsoft Visual Studio 2008

In this chapter:

Introducing Visual Studio 2008 ......................................... 40
Create an ASP.NET Web Site Project ..................................... 55
Application Deployment ............................................... 75
Administering an ASP.NET Application ............................... 81
Conclusion ..................................................................... 87

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. When the application is anything different from this—for example, a Flash-powered site—it takes the serious risk of becoming an opaque box to search engines such as Google and Windows Search.

What’s the role of Microsoft ASP.NET and Visual Studio 2008 in the development of Web applications?

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 their work and help 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 2008—the successor to Visual Studio 2005. It has a lot of new features and goodies expressly designed for Web developers to overcome some of the limitations that surfaced from using previous versions.
In this chapter, we’ll review the main characteristics and features of Visual Studio 2008 as far as ASP.NET applications are concerned. We’ll see how to create and edit projects and how you can target a particular .NET platform. We’ll tour around integrated development environment (IDE) features and deployment capabilities, and also explore some new keywords in the supported languages.

Introducing Visual Studio 2008

Visual Studio is a container environment that integrates the functionality of multiple visual designers. You have a designer for building Windows applications, one for building Web sites, one for building Windows Communication Foundation (WCF) 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.

Let’s start with a quick summary of the principal characteristics of Visual Studio 2008 that have been inherited from its predecessor.

Visual Studio Highlights

The version of Visual Studio labeled with the 2005 attribute marked a milestone in the history of the product. Visual Studio 2005 matched a long list of user requirements to really provide a simpler and more friendly way to create ASP.NET applications. Here’s some highlights of features that have been maintained, if not improved, in Visual Studio 2008.

No IIS Dependency

Internet Information Services (IIS) is not strictly required for Visual Studio to work. Visual Studio 2008 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.

Note that the local Web server represents only the default option. If you open the project from an existing IIS virtual directory, Visual Studio 2008 uses IIS to run and 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.
Multiple Ways to Access Web Sites

Visual Studio 2008 supports multiple ways to open Web sites. In addition to using FrontPage Server Extensions (FPSE) to connect to a remote Web site, 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. 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, as Figure 2-2 shows.

FIGURE 2-1 The local Web server in action in Visual Studio 2008.

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.
The interaction with IIS is greatly simplified too, 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 FPSE. The IIS tab also contains buttons to create new virtual roots and applications (note the buttons in the upper right-hand corner).

![Image](image.png)

**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 2008 cannot create a new Web site via FTP.

**Project Output Build Options**

The old Visual Studio 2005 originally shipped with a project model that didn’t compile everything in the site into a single assembly, as its predecessor Visual Studio 2003 did. Instead, it was built on the modified ASP.NET 2.0 compilation model and given the ability to dynamically recognize 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 model results in a sort of dynamic compilation for code-behind classes.
There are pros and cons about the 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. A few months later, Microsoft released a Web Application Project (WAP) extension to Visual Studio 2005 to enable it to maintain projects by compiling everything into a single assembly. (WAP was later incorporated into the Visual Studio 2005 Service Pack 1.)

In Visual Studio 2008, WAP is natively part of the product, although you won’t find it available in the Web site menu. If you want to create an ASP.NET Web site based on the ASP.NET 2.0 compilation model, you choose the Web Site option from the File|New menu. If you want to create a WAP, instead, you pick up the Project option from the File|New menu and then select an ASP.NET Web Application.

Solution files (*.sln) are mandatory for WAPs but optional in the other case (ASP.NET Web sites). For ASP.NET Web sites, 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, 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 an integrated wizard. 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?)

Starting with Visual Studio 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 in the latest Visual Studio 2008.
Part I Building an ASP.NET Page

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 2008 tool can operate as a synchronization tool, which is helpful to quickly test applications in different scenarios and configurations.

**Smarter Editing with Microsoft IntelliSense**

Visual Studio 2008 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 2008 starts up and lets you edit the source code. Unlike with some previous versions, 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, including within data-binding expressions, page directives, and code inline in .aspx files. (See Figure 2-6.)

![Figure 2-6 IntelliSense works everywhere around the source code of the page.](image)

**Visual Studio 2008—Specific New Features**

The true added value of Visual Studio 2008—and perhaps the primary reason to consider upgrades—is the boost it gives to the developer’s productivity. Lots of wizards, smart and context-sensitive popup windows, effective debuggers, and visual designers are all examples of facilities that might not necessarily make any code automatically smarter, but will certainly help developers to focus on key points while skimming over chores, repetitive tasks, and overzealous procedures.

The recipe of Visual Studio is always the same, although it’s perfected a bit more each time: improving the developer productivity using the latest technologies, supporting the newest .NET Framework, and aiming at improving the management of the entire application life cycle.
Part I Building an ASP.NET Page

Multitarget Projects

As you might have noticed, past releases of Visual Studio supported only one specific version of the .NET Framework. For years, my desktop was swarming with shortcuts to different versions of Visual Studio based on the projects I maintained at the time: Visual Studio 2002 for .NET Framework 1.0 applications, Visual Studio 2003 for .NET Framework 1.1, and Visual Studio 2005 for .NET Framework 2.0 applications. Finally, Visual Studio 2008 introduces a very cool feature called multitargeting. Quite simply, it is the loudly-demanded capability of creating applications for various versions of the .NET Framework.

Multitargeting brings two main benefits to the table. First, you no longer need to install side-by-side two or even three different versions of Visual Studio to deal with the various clients and projects you maintain. Second, you are no longer subliminally invited to upgrade to the next, super-cool version of the .NET Framework because of the new, super-cool time-saving features of the next Visual Studio. One IDE fits all .NET Frameworks, you could say. All frameworks? Really? Well, not exactly. (See Figure 2-7.)

FIGURE 2-7 Choosing the target platform for a new Visual Studio 2008 project on a Windows Vista machine.

Multitargeting in Visual Studio 2008 starts with the .NET Framework 2.0 and should continue for most of the foreseeable future versions. Significant changes in the common language runtime (CLR) that occurred between version 1.1 and 2.0 of the .NET Framework made it impossible to extend multitargeting to any older versions of the framework.

Warning The multitarget option is not available on Visual Studio 2008 Web Express Edition. You can work only on .NET 3.5 Web sites.
The nice thing about multitargeting is that any piece of the Visual Studio 2008 user interface adapts to the target platform. The toolbox of controls, compiler settings, the list of assemblies to reference, IntelliSense on objects and, of course, binaries are all specific to the selected platform.

The target platform is not a definitive choice. At any moment, in fact, you can upgrade or downgrade to any of the supported targets. You do this from the property pages of the project in the Solution Explorer, as shown in Figure 2-8.

![FIGURE 2-8 Upgrade the target .NET Framework of an existing project.](image)

Should you install Visual Studio 2008 if you’re mostly maintaining existing .NET Framework 2.0 applications? Clearly, the big news about Visual Studio 2008 is the support it offers for 3.x .NET applications. However, even from a platform-agnostic perspective it has something to offer to developers. Precisely, it’s the largely improved set of facilities for developers, with particular attention paid to Web developers. The JavaScript debugger, cascading style sheet (CSS) and master pages designer, LINQ helper tools, and IntelliSense for JavaScript are all features available independently from the target platform.

**The JavaScript Debugger**

The Visual Studio 2008 debugger has been enhanced in various ways, including the remote debugging support for Windows Vista and debugging support for LINQ queries and WCF services. Web developers, in particular, welcome the better support for script debugging. You can set a breakpoint in JavaScript code and proceed from there step by step. (See Figure 2-9.)
You have access to the content of each JavaScript variable using the same interface in use for managed objects, including watch windows and data tips. When you’re stepping through the debugger, you can navigate inside of system and AJAX script files as well as any referenced JavaScript files of your own. (See Figure 2-10.)

Note that for the script debugger to work, you must ensure that script debugging is not disabled in the browser you’re testing. For Internet Explorer, you check this setting by bringing up the Options dialog box, selecting the Advanced tab, and looking for the Disable Script Debugging (Internet Explorer) check box.
IntelliSense for JavaScript

JavaScript objects are now fully discovered and explored by IntelliSense. When you hit the dot (.) key following the name of a JavaScript variable, a small and familiar window pops up with the list of callable members on that object. (See Figure 2-11.)

It works for both native objects (for example, the `Date` object in Figure 2-11) and user-defined JavaScript objects, including ASP.NET AJAX JavaScript objects.

```javascript
var d = new Date();
var current = new Date();
d.setDate(current.getFullYear());
if (current > d)
    output = 0.1000
else
    output = 0.0000

function get 
    return output

var itemprop =...
```

FIGURE 2-11  IntelliSense on JavaScript objects.

CSS Designer

One of the most pleasant surprises that Web developers will find in Visual Studio 2008 is its significantly improved HTML and CSS designer. When an ASP.NET page is being edited, a Format menu appears on the menu bar that contains a Manage Styles item. The subsequent tool window shows all the cascading style sheets and their corresponding rules for the page you are currently editing. It is required that the page be in design mode or at least in a combined view of markup and source (split view). (See Figure 2-12.)

When you select an HTML element or ASP.NET server control, the CSS designer window shows you all the settings currently applied to it. The designer also gives you a chance to modify any of the values using the CSS property grid.
New Language Features

In Visual Studio 2008, the C# compiler supports a richer language with some interesting new features. Some are time-saving features that basically move onto the compiler the burden of creating some required code; some are just new capabilities added to the language and backed by the underlying .NET Framework.

I’m illustrating the new language features with respect to C#. It should be noted, though, that analogous features also exist in Visual Basic .NET.

Extension Methods

Extension methods are a way to extend the contract of a class by adding new methods without creating a derived class and without resorting to a related set of partial classes. Extension methods add a duck-typing flavor to a strongly typed, compiled environment such as the
The expression duck-typing refers to a programming style typical of many dynamic languages (for example, Python, Ruby, JavaScript) where you pay more attention to the actual set of methods and properties exposed by an object rather than its inheritance from a particular class. The expression duck-typing, in fact, descends from the following test: *If it walks like a duck and quacks like a duck, then call it a duck.* In a duck-typed language, you can call a given method on an object of any type. If the object lacks the method, you then get a run-time error.

In practice, you use extension methods to add helpful methods to a given class, regardless of the original contract exposed by the class and known to perspective callers. If you have no access to the source code of the class and still want to extend it, in Visual Studio 2008 you define an extension. Extension methods can be defined for any class, including native classes of the .NET Framework. Here's how to extend the `System.String` class with new methods such as `IsDate` and `ToDate`:

```csharp
public static class StringExtensions
{
    public static bool IsDate(this string content)
    {
        DateTime date;
        bool result = DateTime.TryParse(content, out date);
        return result;
    }

    public static DateTime ToDate(this string content)
    {
        DateTime date;
        bool result = DateTime.TryParse(content, out date);
        if (result)
            return date;
        else
            return DateTime.MinValue;
    }
}
```

An extension method is defined as a static method on a static class. The binding between the method (say, `IsDate`) and type (say, `System.String`) is established through the keyword `this` in the method's prototype:

```csharp
public static bool IsDate(this string content)
```

The type that follows the keyword `this` is treated as the type to extend. Figure 2-13 shows IntelliSense in action on the extended `System.String` type.
Figure 2-13: IntelliSense shows dynamic extensions to the String type.

The following code snippet illustrates how you can use these new methods in your code:

```csharp
void Button1_Click(object sender, EventArgs e)
{
    string content = TextBox1.Text;
    if (content.IsDate())
    {
        DateTime date = content.ToDateTime();
        Label1.Text = date.AddDays(1).ToString("dd MMMM yyyy");
    }
    else
    {
        Label1.Text = "Not a valid date.";
    }
}
```

Extension methods are checked at compile time and can also be applied to any parent class or interface in the .NET Framework.

Note: Extension methods can be used to implement a feature that looks very similar to mixins.

Overall, a mixin is a sort of interface with some implemented methods. A class that implements a mixin includes, but does not inherit, all the members on the mixin's interface. Latest versions of C# and Visual Basic .NET don't natively support mixins, even though instructing the compilers to produce the code for it didn't appear to be a huge effort for the developers of the latest versions of C# and Visual Basic .NET. With extension methods, you can simulate mixins in the latest versions of C# and Visual Basic .NET by creating groups of methods and adding them to an existing class without the mechanism of partial classes or inheritance. Why are mixins superior or just preferable to an abstract class? You implement a mixin; you inherit an abstract class. By using mixins, you import the benefit of an existing piece of code without employing a base class.
Automatic Properties

Automatic properties is a feature that instructs the compiler to automatically add a default implementation for the get/set methods of a class property. The following code is valid in an interface in previous versions of C#. It is perfectly legal, instead, in a class compiled with the newest C# compiler:

```csharp
public string CompanyName { get; set; }
```

The compiler automatically expands the code as shown here:

```csharp
private string companyName;
public string CompanyName
{
    get { return companyName; }
    set { companyName = value; }
}
```

It is worth noting that automatically generated get/set properties are not equivalent to public fields. From a metadata perspective, properties and fields are quite different entities. The code generated for each is different. The key thing that is going on here is that you delegate the creation of some repetitive code to the compiler. Automatic properties are clearly not an option when you need to do more than just store a value in the get/set methods. Using automatic properties is not a non-return option: at any later time, you can always come back and provide your own get/set methods that contain any logic you need.

Object Initializers

Object initializers are another great piece of syntactic sugar that might speed up the creation of the code needed to initialize an object. Instead of going through a potentially long list of assignment instructions, you can code as shown here:

```csharp
Person person = new Person { FirstName="John", LastName="Doe", Age=24 };  
```

The idea can be extended to collections, as in the following code snippet:

```csharp
List<Person> friends = new List<Person> {
    new Person { FirstName="Nancy", LastName="Davolio", Age=28 },
    new Person { FirstName="Andrew", LastName="Fuller", Age=35 },
    ...
};
```

Compared to the syntax required in Visual Studio 2005, the savings is pretty clear and, for large procedures, can easily add up to dozens of lines of code.
Type Inference and Anonymous Types

The var keyword is another interesting new entry in the latest version of C#. Used to qualify a variable, it doesn’t indicate a late-bound reference, as its rather popular JavaScript counterpart does. Instead, it merely indicates that the developer doesn’t know the type of the variable at the time he’s writing the code. However, the type won’t be determined at run time (late-binding); instead, the compiler will infer the type from the expression assigned to the var variable. For this reason, an initial value assignment is required to avoid a compiler error. When the var keyword is used, a strongly typed reference is always generated.

The var keyword enables another cool feature of the latest C# language—anonymous types. Quite simply, an anonymous type is an unnamed type that you define using the same object initializer syntax mentioned earlier:

```csharp
var person = new { FirstName="Nancy", LastName="Davolio", Age=28 };
```

For the CLR, anonymous and named types are exactly the same entity. Anonymous types can be used in a variety of scenarios, but they have been introduced primarily to support LINQ queries.

LINQ Operators

As I’ll cover in more detail in Chapter 10, both C# and Visual Basic .NET feature a set of SQL-like language operators specifically designed to back queries. Currently, developers have to use a different application programming interface (API) to retrieve data from different repositories. It will be SQL for relational databases, XQuery and XPath for XML documents, and methods on some interface (for example, ICollection) for collections and arrays. The main goal of LINQ is unifying this model by providing an ad hoc framework. Nicely enough, this query framework is wired to some new keywords in C# and Visual Basic. The following code snippet gives you a valuable glimpse of LINQ in action:

```csharp
int[] fiboNumbers = new int[] {0,1,1,2,3,5,8,13,21,34};
var data = from n in fiboNumbers
           where n % 2 == 0
           select n;
```

As weird as it might seem, this is C# code that compiles in Visual Studio 2008. The new keywords are from, where, and select. Their meaning is really close to the meaning that analogous keywords have in SQL. Translated to a human language, the query assigns to the resulting variable all elements in the queryable object (a collection, in this case) that match the specified condition (all even numbers in this case). We’ll return to LINQ queries and syntax in Chapter 10.
Create an ASP.NET Web Site Project

Let's go further and create a sample ASP.NET Web site project with Visual Studio 2008. 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

If you select the Web Site option, Visual Studio generates the minimum number of files for building a Web site. Basically, it creates a default .aspx page and an empty App_Data directory. 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 three views—Design, Split, 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. Finally, the Split view splits the screen real estate in two and provides both design and source views.

You choose the template of the item to add to the site from the menu shown in Figure 2-14.

FIGURE 2-14 Item templates supported by Visual Studio 2008.

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 and to associate the current page with a master page. Master pages are a feature of ASP.NET that we'll discuss thoroughly in Chapter 6.

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.

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 master pages are 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.

Nested Master Pages

Master pages can be nested, meaning that you can have one master page reference another page as its master. For example, in a large site an overall master page can define the look of the site and then content areas can be created using distinct masters that each partner or content manager can populate and personalize. Another good example is when you have sites that users can customize and choose, say, between a single-column or two-column layout. In this case, you can have a global master to create the look and feel and then a child master for the layout of choice.

Nested master pages are supported since ASP.NET 2.0. The catch is that Visual Studio 2005 didn’t support the design of nested masters very well. Thankfully, this capability changes radically with Visual Studio 2008, where nested master pages are fully supported in the page designer.
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, or other tags) would be ambiguous, because the same kind of content is also allowed in the master.

Visual Studio offers a special Design view of content pages, as Figure 2-15 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 2-15 Content pages in Visual Studio 2008.](image)

**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 2-14.
Code-Behind Classes

When you add new Web Forms and require code and layout separation, a C# (or Visual Basic) 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 harder 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.

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 2-16.) The toolbox can be toggled on and off, and it 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 in whatever view mode.

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 2008 code editor presents some interesting features that prove the development team’s commitment to excellence and creating satisfaction among all users. I’d like to call your attention to a few of them: markup preservation, improved tab insertion logic and indentation, and target schema validation.

Visual Studio 2008 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.

In Figure 2-17, 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 2-18 shows, Visual Studio detects the difference and handles it correctly. IntelliSense doesn’t list `iframe` but prompts you for `layer`. If you are insistent and type in `<iframe>`, anyway, a squiggle shows up to catch your attention.

**FIGURE 2-18** The code editor is sensitive to the selected client target schema.

### Code Refactoring

When a `.vb` or a `.cs` file is open for editing in Visual Studio 2008, a new menu appears on the top-most menu strip—the Refactor menu, shown in Figure 2-19.

**FIGURE 2-19** The new menu for helping developers quickly refactor the code of classes. This menu has fewer items in the Visual Studio 2008 Express Edition.

As you can see, the menu provides advanced facilities for code editing. Among 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:

```plaintext
private int _counters;
```

```csharp
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:

```csharp
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 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 2-20 shows the new Import And Export Settings Wizard 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.
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 that represent 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. (The Split view offers both options at the same time.) 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.
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:

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

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

```html
<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. 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 2-21 and pick up the event you need.

![Figure 2-21 The Events view in the Properties window.](image)

### Writing Helper Classes

Writing helper classes is as easy as adding a new class to the project. 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...
Part I Building an ASP.NET Page

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.

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 2-22.

FIGURE 2-22 The PswdGen.aspx page to generate a new “strong” password of the specified length.

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 refers to 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 purposely created `App_Code` subdirectory. The class outline is shown here:

```csharp
public class StrongPassword
{
    public StrongPassword()
    {
    }
    public string Generate()
    {
    }
    public string Generate(int passwordLength)
    {
    }
}
```

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:

```csharp
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.)
    string text = PswdLength.Text;
    int pswdLen = 8;
    if (text.IsInt32())
        pswdLen = text.ToInt32();
    // Use extension methods to do the job
    StrongPassword pswd = new StrongPassword();
    labelPassword.Text = pswd.Generate(pswdLen);
}
```

Figure 2-22 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 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 Protected 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. Starting with version 2.0, ASP.NET introduces a few additional protected directories. None of these directories are automatically created by ASP.NET or Visual Studio 2008, nor are the directories necessarily required to exist. Each directory needs to be created either manually by developers or on demand through Visual Studio 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 is that only two or three (out of seven) additional directories will be present in an average ASP.NET application.

**TABLE 2-1 Special Protected Directories in ASP.NET Applications**

<table>
<thead>
<tr>
<th>Directory Name</th>
<th>Intended Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin</td>
<td>Contains all precompiled assemblies needed by the application.</td>
</tr>
<tr>
<td>App_Browsers</td>
<td>Contains browser capability information.</td>
</tr>
<tr>
<td>App_Code</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Directory Name Intended Goal

<table>
<thead>
<tr>
<th>Directory Name</th>
<th>Intended Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>App_Data</td>
<td>Contains data files for the application. These can include XML files and Microsoft Access databases to store personalization data.</td>
</tr>
<tr>
<td>App_GlobalResources</td>
<td>Contains .resx resource files global to the application.</td>
</tr>
<tr>
<td>App_LocalResources</td>
<td>Contains all .resx resource files that are specific to a particular page.</td>
</tr>
<tr>
<td>App_Themes</td>
<td>Contains the definition of the themes supported by the application. (I'll say more about themes in Chapter 6.)</td>
</tr>
<tr>
<td>App_WebReferences</td>
<td>Contains .wsdl files linking Web services to the application.</td>
</tr>
</tbody>
</table>

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 is related to the way the ISAPI filter in charge of blocking HTTP requests to these folders works. 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 can be written in the registry, which would make for much faster and more 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 "The Resource Directories" section for more information on the ASP.NET 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.
Part I Building an ASP.NET Page

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, or in a child 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 subfolders and add some entries to the configuration file to tell the 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:

```xml
<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. Starting with ASP.NET 2.0, all you have to do is drop the source XSD file in the App_Code folder.
Chapter 2. Web Development in Microsoft Visual Studio 2008

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. Starting with version 2.0, ASP.NET 2.0 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 2-23 shows a sample local resource folder.

![FIGURE 2-23 The local resource directory for the respage.aspx page.](image-url)
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:

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

<title>Localized Resources</title>
<body>
<form id="Form1" runat="server">
<h1>
<asp:Label runat="server" id="Label1" meta:resourcekey="Label1_ResourceID" />
</h1>
<asp:Button ID="btn" runat="server" meta:resourcekey="Button1_ResourceID" />
</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 Button1_ResourceID.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 this point on, Internet Explorer sends the selected language ID with each served request.

Figure 2-24 shows how the same page looks when different languages are set.

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.
FIGURE 2-24 The testres.aspx file in Italian and English.

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

```<asp:Label runat="server" ID="Label2" Text="<%= Resources:Core35, Label2_ResourceID %>
" /></asp:Label>
```

**Note** The resulting resource assembly for the neutral culture has application scope and is therefore referenced from other assemblies generated in the application.

**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.

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 2008, no compile step takes place to incorporate code-behind and helper classes (unless you opt for the WAP project option). All the dynamically generated assemblies, and all precompiled assemblies deployed into the Bin folder, are linked to the application used to visit pages.
Once any needed assemblies have been successfully built, Visual Studio auto-attaches 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 for whatever reason, 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 don'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 terms of performance. The default settings for IIS 6.0 and Windows 2003 Server apply to the newest Windows 2008 Server and IIS 7.0, and even to Windows Vista.

When you build the project, Visual Studio 2008 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:

```xml
<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, indeed, the default), you can set a few breakpoints in your source files and step into the code, as shown in Figure 2-26.

![Figure 2-26](image-url)

The Debug menu offers you 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 2-27.)

![Figure 2-27 The text visualizer invoked from the quick-watch window during a debug session.](image1)

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 2-28.)

![Figure 2-28 Invoking a visualizer from within a code tip.](image2)

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

There are two ways of testing pages in Visual Studio 2008. You can have pages served by IIS (if installed) or by the embedded local Web server. By default, Visual Studio 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 becomes 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. Often used to describe setup procedures in .NET, the XCopy deployment expression 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 2008, 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 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 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, and properties and it 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 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.

You create a Web application folder to represent 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, the level of isolation, and the 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 to use 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 file write permission to the application. Likewise, if your application or control generates temporary files, you need to make accommodations for a proper folder with proper file write 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 pair of `machine.config` and system `web.config` files. 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.)
Part I Building an ASP.NET Page

You can work around the issue by simply replicating any needed settings to the application's root `web.config`. However, if you are deploying your code to a service provider, you might find that many system 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 `web.config` file.

Deploying an ASP.NET application in a Web-farm scenario poses a few extra configuration issues you must be aware of. All configuration 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 on 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 starting with version 2.0 it reached 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 2008 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 `/core35` indicates the virtual folder of the application:

```bash
aspnet_compiler -v /core35
```

If you recompile 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...
Part I Building an ASP.NET Page

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 supported parameter is explained in Table 2-2.

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-aptca</code></td>
<td>If this switch is specified, compiled assemblies will allow partially trusted callers.</td>
</tr>
<tr>
<td><code>-c</code></td>
<td>If this switch is specified, the precompiled application is fully rebuilt.</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>If this switch is specified, the debug information is emitted during compilation.</td>
</tr>
<tr>
<td><code>-deleyesign</code></td>
<td>If this switch is specified, compiled assemblies are not fully signed when created.</td>
</tr>
<tr>
<td><code>-errorstack</code></td>
<td>Shows extra debugging information.</td>
</tr>
<tr>
<td><code>-m</code></td>
<td>Indicates the full IIS metabase path of the application.</td>
</tr>
<tr>
<td><code>-f</code></td>
<td>Indicates that the target directory will be overwritten if it already exists and existing contents will be lost.</td>
</tr>
<tr>
<td><code>-fixednames</code></td>
<td>If this switch is specified, the compiled assemblies will be given fixed names.</td>
</tr>
<tr>
<td><code>-keycontainer</code></td>
<td>Indicates the name of the key container for strong names.</td>
</tr>
<tr>
<td><code>-keyfile</code></td>
<td>Indicates the physical path to the key file for strong names.</td>
</tr>
<tr>
<td><code>-p</code></td>
<td>Indicates the physical path of the application to be compiled. If this switch is missing, the IIS metabase is used to locate the application. This switch must be combined with <code>-v</code>.</td>
</tr>
<tr>
<td><code>-u</code></td>
<td>If this switch is specified, it indicates that the precompiled application is updatable.</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Indicates the virtual path of the application to be compiled. If no virtual path is specified, the application is assumed to be in the default site: W3SVC/1/Root.</td>
</tr>
</tbody>
</table>

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 Core35 to the specified disk path:

```
aspnet_compiler -v /Core35 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 terms of 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 can 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 is nearly identical to the compilation and deployment model of ASP.NET 1.1, where .aspx files are deployed in the 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
Part I Building an ASP.NET Page

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?

Although 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 would probably be helpful and sufficient in many cases. In Visual Studio 2008, 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 2-29 presents the administration tool in its full splendor. The tool is articulated in three blocks (plus the home), each covering a particular area of administration—membership, user profiles, application settings, and providers.

![The Visual Studio 2008 ASP.NET Administration Tool](image)

**FIGURE 2-29** The Visual Studio 2008 ASP.NET Administration Tool.

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

C:\WINDOWS\Microsoft.NET\Framework\[version]
You can also run the tool from outside Visual Studio 2008. 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 Core35:

http://localhost:XXXX/asp.netwebadminfiles/default.aspx

?applicationUrl=/Core35

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>
<thead>
<tr>
<th>Configuration Tab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>Enables you to set up and edit users, roles, and access permissions for your site</td>
</tr>
<tr>
<td>Application</td>
<td>Enables you to manage your application’s configuration settings, such as debugging and SMTP options</td>
</tr>
<tr>
<td>Provider</td>
<td>Enables you to select the provider to use for each ASP.NET feature that supports providers</td>
</tr>
</tbody>
</table>

**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 through 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 was fairly common to have a custom database store credentials for authorized users. The point is that this database had to be filled out at some time; in addition, the site administrator had to be able to manage users, 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 homebrewed 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 fewer 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.

![Application Settings Management](image)

**FIGURE 2-30** The Application tab in the Web Site Administration Tool.

As you can see in Figure 2-30, 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.
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 provider model (discussed in Chapter 1) supplies a plug-in mechanism for you to choose the right support for storing data. ASP.NET 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 shown in Figure 2-31.

![Figure 2-31](image)

**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 truly consider it to be a web.config editor. Visual Studio 2008 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

Starting with version 2.0, ASP.NET 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 2-32.)

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 2-32. 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.

If you use IIS 7.0, you get editing applets right in the IIS Manager, as shown in Figure 2-33.

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.
Conclusion

Visual Studio 2008 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 is capable of providing a unique editing experience to Web and Windows developers.

In this chapter, we traversed the main phases of Web application development through Visual Studio 2008—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 2008—what’s great, what’s been improved, what’s new, and what you should know. 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.

**FIGURE 2-33** The bolted-on visual editor for the web.config file in IIS 7.0.
Just the Facts

- IIS is no longer a strict requirement for developing ASP.NET applications, as Visual Studio 2008 incorporates a local, mini Web server to be used only for testing during the development cycle.
- Visual Studio 2008 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 2008 supports multitargeting and can create and edit applications written against any version of the .NET Framework, starting with version 2.0.
- 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 (different account, different settings). Don’t rely on it to determine 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.
- Visual Studio 2008 provides richer support for IntelliSense and debugging in the context of client JavaScript code and offers new capabilities as far as programming languages (C#, Visual Basic .NET) are concerned.