Moving to ASP.NET: Web Development with VB .NET

STEVE HARRIS AND ROB MACDONALD

Apress™

Moving to ASP.NET: Web Development with VB .NET

ISBN (pbk): 1-59059-009-0

Printed and bound in the United States of America 12345678910

Technical Reviewer: Scott Brown
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager and Production Editor: Laura Cheu
Copy Editors: Christina Vaughn and Kim Wimpsett
Compositor: Impressions Book and Journal Services, Inc.
Indexer: Rebecca Plunkett
Cover Designer: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.
In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.
Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710.
Email info@apress.com or visit http://www.apress.com.

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

CHAPTER 1

Introducing ASP.NET

By now, most developers will have heard of ASP.NET and will have seen it in action. In fact, it's a pretty sure bet that if you've bought this book then you already have it installed, maybe with Visual Studio .NET, and there's a good chance you've tried a few things out. You probably already know that ASP.NET brings an object-oriented and event-driven programming model to the world of Web development and that it can dramatically simplify the structure and creation of Web applications. You might, like us, be *really* excited about the possibilities and improvements it brings, or you might just see it as a tool you can use to save a bit of time so you can get to the game earlier or spend more time with your kids. Either way, you can't afford to ignore it—ASP.NET is big news and plays a key role in Microsoft's .NET strategy.

ASP.NET solves many of the problems that currently face Web developers, and it greatly simplifies the tasks of creating, debugging, and deploying Web applications. It's radically different from its predecessors in many ways, but it shares a common heritage and background to some. It requires that you learn new skills and forget about some you already have. It'll take time to master, but that investment will be repaid many times over once you start working with it in earnest. In short, it's what many Web developers have been asking for over the past few years.

ASP.NET: The Five-Minute Guide

Okay, let's start by going (very briefly) back to basics. ASP.NET is the next stage in the evolution of Microsoft's server-side technologies for dynamically creating Web pages. It's a successor to ASP 1.0, 2.0, and 3.0 (now collectively referred to as *classic ASP*) and enables you to do everything that these older technologies could do, plus a whole lot more. Although it's different from its predecessors, it does share many classic ASP language features and supports much of the old object model, thus providing a reasonable amount of backward compatibility.

Classic ASP

Over the past few years, classic ASP has provided a convenient and effective way for developers to build dynamic and interactive Web applications. It's widely used in Internet and intranet applications, and it has found favor with developers who already have experience with other Microsoft technologies. Like every development tool, classic ASP isn't perfect, and the different versions suffer from a variety of limitations:

- VBScript, the language of choice for most ASP developers, is loosely typed, is late bound, and is interpreted rather than compiled. It offers less functionality than its "big-brother" versions of Visual Basic for Applications (VBA) and Visual Basic (VB), even in terms of fundamental requirements such as error trapping and management.

- The design and architecture of ASP applications are different from desktop applications. If we're honest, they're primitive when compared to the object-oriented designs that you can achieve with tools such as VB, Visual C++, and Visual J++.

- Continuing on the theme of architecture, one of the greatest limitations in ASP is the way it requires you to combine interface elements and code into a single ASP file. This is awkward when creating more sophisticated applications and limits code reuse and sharing.

- ASP is largely procedural, with the code within an ASP page being executed from top to bottom on each request. Modern developers are more familiar with object-oriented or event-driven models, both of which offer greater flexibility and savings in development and maintenance time.

- State management techniques in ASP are rather basic, and although they're satisfactory if you're deploying to a single server, they're completely inappropriate if you're hosting the application on a Web farm. *Web farms* consist of multiple servers, each running a copy of your Web application. With the limited state management in classic ASP, each server in the Web farm maintains its own state and is unable to share it with other servers.

- Configuration and deployment of medium- to large-scale ASP applications is cumbersome. You can copy basic content files to target servers with minimum effort, but there remains a variety of manual tasks for configuring virtual directory settings and permissions, as well as the need to register COM components and install Microsoft Transaction Server (MTS) packages and COM+ applications. The situation is further worsened because the Internet Information Server (IIS) Metabase holds IIS and ASP settings, with relatively few tools available to manage them.

- The development tools are rather immature (although usable). Visual InterDev helps developers who are prepared to accommodate its quirks and foibles, but it has the capacity to surprise the unwary user of server components, design-time controls, and so on. Some third-party tools offer improvements in a few areas, but none are perfect.

- Finally, ASP is all about server-side features. Client-side control and interaction is possible but requires manual coding from the developer. This means that pure ASP applications often require frequent server round-trips, and this in turn often compromises performance.

ASP.NET

ASP.NET is full of new features and improvements, and throughout this book we'll look at all the important ones. It might be useful to start with a checklist of what to look for, though, so you can start planning your approach to learning the tools and techniques. The following list summarizes what we think are the most significant changes and additions; however, once you've spent some time working on your own projects, you may well want to extend this list with some of your favorites:

- ASP.NET is fully integrated with the .NET Framework and with the Visual Studio .NET development environment. It's not a bolt-on addition or afterthought, and ASP.NET applications have full and unrestricted access to all of the .NET classes and features.

- ASP.NET applications are built on top of the common language runtime (CLR) and can be written in VB .NET, C#, or any other .NET-compliant language.

- ASP.NET applications are largely component-based and modularized, and almost every object, page, and HTML element can be a runtime component that can be programmed through properties, methods, and events. The currently supported languages offer full support for object-oriented development, and third-party companies deliver additional languages.•

- ASP.NET applications typically involve less code than classic ASP through the use of Web Forms, server controls, components, and other intrinsic features. Also, the architecture and structure of ASP.NET applications emphasize the separation of code from content, with interface elements held in ASPX files while programming logic is compiled into a .dll.

- ASP.NET provides browser independence, with a base level of HTML 3.2 for older browsers while taking advantage of client-side features in later browsers. ASP.NET causes the same source code to be rendered in the most appropriate form for the browser in use.•

- Powerful server-side controls provide additional functionality and rich content. Validation controls allow for automatic validating and checking of user-entered data, and data-binding features enable the display and updating of compatible data sources, including database and XML information.

- Microsoft has also made available an additional library of server controls (the Internet Explorer Web Controls) that generate rich *client-side* content for clients using Internet Explorer 5.5 or later. This content takes the form of DHTML, JavaScript, and DHTML behaviors to provide an interactive interface including tab strips, tree views, and toolbars, with much of the processing performed in client-side scripts. For clients using other browsers, these server controls render to HTML 3.2 to present a similar look and feel—though in this case any processing will be performed server-side.

- ASP.NET supports numerous caching technologies to allow efficient storage and retrieval of any kind of object or data, including XML, database query results, partial or complete pages, any part of the browser stream, images, and much more. You can associate cached items with a priority that ASP.NET uses as a guide when clearing cached items if space is a pre-

mium, so you can preserve items that are costly to rebuild at the expense of simpler items.

- ASP.NET is more crash tolerant than classic ASP, with better and tighter security management. Much of the improvement is because of the .NET environment and CLR, which provides reliable garbage collection, application isolation, thread management, resource pooling, and more. If a Web application crashes, ASP.NET restarts it when the next browser request is received.

- There are major improvements to debugging and error handling, including page- and application-level tracing. Error information can be reliably passed between pages, so that common, centralized error logging and reporting systems can be built. VB .NET supports structured error handling, with consistent reporting of errors and error information regardless of the source or cause of the error.

- ASP.NET supports easy deployment, updates and component management, and text-based configuration through XML documents. You can roll out changes to live Web servers, even while the application is running. .NET objects have no direct dependency on the registry in terms of their location and configuration, dramatically simplifying the tasks of initial deployment and updates.

- The Microsoft development team made sure that Web farms and Web gardens were supported by giving ASP.NET powerful and flexible state management, server independence across page calls and postbacks, and free-threaded components.

- ASP.NET supports creating and managing Web Services, replacing DCOM technology with a solution that is platform neutral and firewall friendly, plus incredibly easy to build, test, and deploy.

As you might imagine, we could continue this list even further, but these details should give you a good idea of what ASP.NET offers. Hopefully these points have also started to make you aware of just how different ASP.NET is from desktop development and from classic ASP Web development. If you want to make the most of these new tools and techniques, then you'll need to invest some time and effort into learning them; it's unrealistic to expect to simply "pick things up as you go." What we aim to do in this book is to give your ASP.NET career a real kick-start, not just by showing what ASP.NET offers but more importantly by showing how you'll likely use it to create real-world Web applications.

ASP.NET vs. Classic ASP

As the previous section highlighted, there are many differences between classic ASP and ASP.NET. They both seek to solve the same problems—the need to deliver flexible and efficient architectures for Web applications, but the way they achieve that goal is vastly different.

Although there are clearly differences in the implementation details, the real difference lies at the heart of ASP.NET, which delivers a truly event-driven and object-oriented development experience. What this means for you and other real-world developers is that you should be able to write less code to achieve the same objectives, which in turn should generate fewer errors and less maintenance. Organizations that have begun developing ASP.NET applications are reporting remarkable improvements in code efficiency and volume compared to older technologies. Compare some well-known sample applications such as IBuySpy (`www.ibuyspystore.com`) and Fitch and Mather (`www.fmstocks.com`), and it becomes clear that the ASP.NET solution can have as little as 25 percent of the code of its classic ASP sibling. Additionally, that code is better organized and structured and is much easier to test, debug, deploy, and maintain.

All this is great news for new developments, but what about existing classic ASP applications; how can they benefit? Well, we've found that the migration process is far from painless, and because of the new programming model, many classic ASP applications would best be rewritten from scratch rather than simply converted. As a result, classic ASP remains a necessary technology for existing installations. As time moves on, we recommend you seriously consider ASP.NET for new projects and for any significant redevelopment or enhancement of current ones, but in many cases it won't be financially viable to convert existing applications.

Fortunately, there's a simple solution; to ease the pressure of migrating from ASP to ASP.NET, both technologies can coexist on the same Web server, and even in the same application. When IIS receives a request, it uses the extension of the requested filename to determine how a request is processed; Filename.asp would be processed using ASP technologies, and Filename.aspx would be passed to ASP.NET. Chapter 10 discusses exactly how this differentiation is achieved. If you've been through previous upgrades of Microsoft's developer tools, you might feel a little suspicious, though—after all, can you *really* run two different versions of ASP on a single Web server? Well, from our experience so far, we would say that you can. It really does seem that there are no serious technical problems or difficulties, although there will be design issues arising from the differences in state management, component management, and so on.

Where we recommend caution is if you try to install Visual Studio .NET alongside Visual Studio 6. In theory this should work fine, as the two environments share few files and should have no conflicting settings. However, where

you may notice changes is in terms of the supporting components and technologies, rather than the development tools themselves. For example, Visual Studio .NET installs ADO 2.7 alongside any existing versions and upgrades your browser to Internet Explorer 6.0. Depending on how you've written your code and the features you've used, you may find these newer versions change the way your existing Visual Basic 6 applications behave. On the whole, though, the ability to have Visual Studio 6 and Visual Studio .NET installed alongside each other is positive, giving you the opportunity to build new projects in .NET while continuing to support existing ones with the original development tools.

Developing Web Applications

Many of you reading this book will have strong desktop development skills, and you'll have experience coding Windows Forms, .dll files, and .exe files. You'll be used to the idea that if you put a value into a class-level variable, that value stays there and won't be changed or destroyed except under the control of your code. You'll have used components and controls within your application because you know they can be deployed to client machines with the rest of the application. More importantly, though, you'll be familiar with the way in which events are raised and handled, allowing your code to instantaneously react to almost every user action.

Web development is different. A Web application could be comprised of many different elements, some of which are compiled into .dll files and others are deployed to the server in plain-text form. Web applications don't automatically maintain state for you, requiring that you add code to manage the persistence of values, objects, and any other data you want to keep "alive." Web applications run in a diverse and unpredictable environment, and although you have a certain degree of control over the configuration of the Web server, you have no influence over the client browser's type, version, or configuration.

Also, Web applications have traditionally been procedural rather than event driven, but this is one of the big changes for ASP.NET as it now supports a rich and powerful event model. However, ASP.NET events are generally handled on the server, so actions in the client browser are passed across the network for handling, and the result passed back to the browser. Too many round-trips can cause performance problems, and although ASP.NET provides some facilities for you to minimize and control the number of round-trips, it's up to you to write the code to do so.

Web vs. Desktop Development

To summarize the differences between these development styles, consider the following list of key Web application features:

- **Thin-client**: The Internet is a large and varied environment, and robust Web applications must be accessible from as many different client platforms and browsers as possible. For many developers the solution is to adopt a thin-client design, whereby the application returns browser-neutral HTML to the client, but this approach results in static applications that require round-trips to the server to perform any processing or updates. Contrast this with desktop development where it's usual to have thick-client technology, interactive controls and code, and the ability to access workstation features and software.

- **Rich versus reach**: An Internet developer needs to make a conscious decision to either target specific browsers (and provide a rich and interactive application) or support the widest possible set of browsers (and reach a broader audience). Desktop developers have this decision made for them—the interactive nature of a typical desktop application means that it has specific software and hardware requirements.

- **Round-trips**: Because of the thin-client nature of typical Web applications it's necessary to make a server round-trip to perform any processing, validation, or data retrieval. Each of these round-trips is expensive, however, involving measurable delays as well as the possibility of network errors because of poor Internet connections, routers, and so on. In a desktop environment, the number of server hits can be kept to a minimum through client-side caching, validation, and processing.

- **State and scalability**: You can design desktop applications using a variety of architectures, from monolithic through client-server to n-tier. However, from the point of view of building the client-side code, the developer can be sure that they can store data in memory, save values to disk files if needed, and generally write the code such that it will be used by a single user. The Web environment is different—many users will call a single Web page, often simultaneously, and therefore the code behind the page must allow this level of concurrency while still maintaining each user's information in a suitable way. Failure to design the application correctly leads to a non-scalable architecture, where the performance and reliability degrade quickly as the number of users increases.

In the ASP.NET environment, some of the new features address these problems:

- **Thin-client**: You can configure ASP.NET to generate browser-neutral HTML 3.2, with a minimal dependence on client-side features such as JavaScript support. You achieve this through Web Forms, although their default property settings mean that they're optimized for more modern browsers, in particular Microsoft Internet Explorer 5.5. It's up to you to change the properties to the settings required for your chosen audience.

- **Rich versus reach**: Certain ASP.NET features are able to adapt their behavior according to the browser in use. For example, validation controls are special server controls you add to your Web page to check that the user has entered data correctly. If a JavaScript-enabled browser (such as Internet Explorer 5.5) is detected, the validation controls will be rendered using some client-side code, but if a non-JavaScript browser is identified then the client-side code will not be generated. This adaptive behavior allows developers to take advantage of new browser technologies without compromising support for older standards.

- **Round-trips**: ASP.NET is by definition a server-side technology, so the majority of event handling and processing is on the server. However, there are times when a small amount of client-side code would prevent a server hit, such as in the previous validation control example. In many cases such as this, ASP.NET generates client-side code that minimizes or negates the need for server round-trips.

- **State and scalability**: ASP.NET eases the management of state in Web applications through numerous mechanisms. A special hidden control on each Web Form now stores page state, which is sent to and from the server transparently. This eases the creation of *postback* pages and means that such page state need not be held on the server, thus increasing scalability. Session state, which relates to a single user of the application, can now be stored in a service that is distinct from the Web server or in a SQL Server database. In both cases, you can specify a remote server to ease deployment of the application to a Web farm. You can also control caching options at page or application level, enhancing performance with increasing numbers of users.

The Visual Studio .NET development environment makes it easy for you to build these features into your project, and it makes the process of building Web applications easier than ever. In many cases it does an excellent job of hiding the underlying detail, providing developers with a set of tools similar to the traditional Windows Forms/toolbox combination present in Visual Basic.

In fact, in some ways it's almost too good at hiding these specifics and can lead unwary developers into producing Web Forms that are fully featured but incredibly inefficient. For example, ASP.NET server controls support a property called AutoPostback that causes the page containing the control to be submitted to the Web server if the control is changed or clicked. As you can imagine, incorrect use of this property is likely to result in many server round-trips, and across the Internet this will almost certainly render the application unusable.

Key Concepts for ASP.NET

By now you should have a broad idea of what ASP.NET is about, and you're probably itching to get started. Well, before we jump in and start building, there are just a few concepts to introduce. These really are important, and with a grasp of these ideas you'll find creating and understanding ASP.NET Web Applications to be a whole lot easier.

Web Application

The first concept we'll investigate is a *Web Application*. As you might imagine, a Web Application is pretty central to ASP.NET and Web development in general, so it makes a good start point. There are a number of ways of defining a Web Application, but one that works well for ASP.NET is as follows:

> A Web Application consists of all the files, pages, handlers, modules, and executable code that can be invoked or run in the scope of a given virtual directory (and its subdirectories) on a Web Application server.

If you're familiar with classic ASP then you should recognize this definition, and it's true to say that at first sight, little appears to have changed in the way that ASP.NET Web Applications run. In reality there are big differences, most of which are buried deep in the .NET Framework and supporting technologies. As a developer, you need to make sure the files and content you create are placed into the correct folder, but even that is largely automated by Visual Studio .NET.

It's important to realize a Web Application is different from a traditional desktop application. In particular, Web Applications do not have to be comprised of a specific .exe or .dll file, and they're likely to be made up of many individual files of varying types. In fact, as you shall see later, there's no need to have a compiled .exe or .dll at all—you can create all Web Application functionality with plain-text files.

We'll return to investigate Web Applications later in this chapter in "Understanding Web Applications," but for now let's look at some other important ideas that make up ASP.NET.

Web Form

Web Forms are the most common components in Web Applications. They're the combination of the user interface and the associated logic that gets rendered as a page in the user's browser, and they're implemented in ASP.NET as .aspx files, in a similar way to the use of .asp files in classic ASP. However, where ASP.NET differs is that the associated logic for a Web Form can be written in a powerful and full-featured language such as VB .NET or C# and stored in a compiled .dll. In contrast, classic ASP relies on interpreting scripts embedded in the ASP file itself.

Each Web Form represents a separate page within an application and contains an HTML `<form>` element. Any additional tags, elements, or controls you add using Visual Studio .NET go within the `<form>`, which means that all of the content of a Web Form passes back to the Web server when the form is submitted. To make the creation of Web Forms as easy as possible, Visual Studio .NET provides you with a convenient designer that supports drag-and-drop editing and a What-You-See-Is-What-You-Get (WYSIWYG) viewer. For example, Figure 1-1 shows a simple page in the designer, consisting of labels, text boxes, an image, and a button.

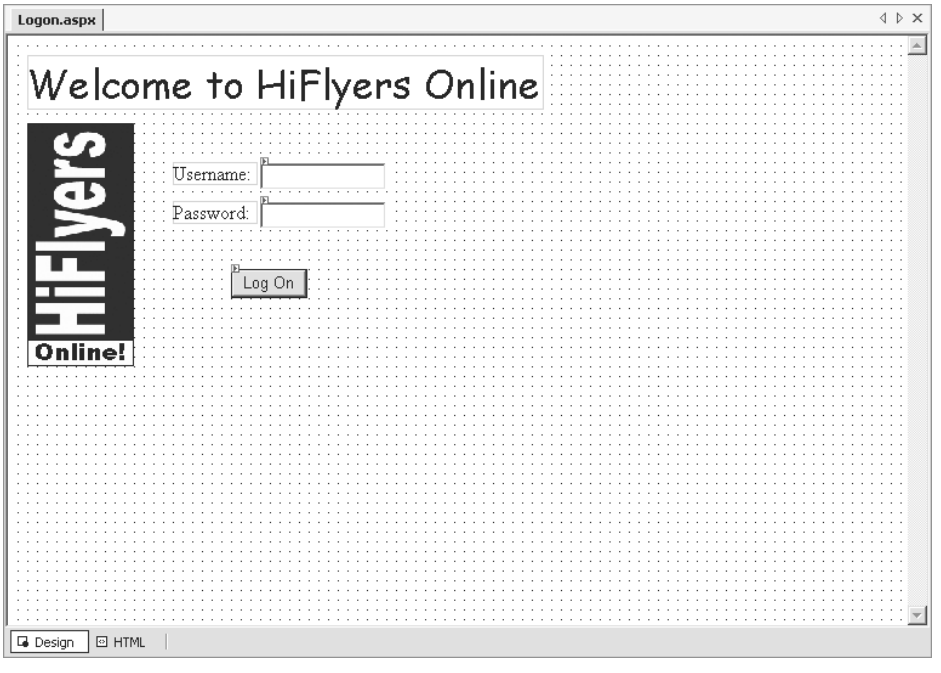

Figure 1-1. The Logon.aspx file

Some of the content added in this example is static HTML, much as you might create using Microsoft FrontPage, Macromedia Dreamweaver, or even Notepad. However, the two text boxes and the button are *server controls* that, as you'll soon see, are intelligent server-side interface objects that allow for easy interaction between your code and the Web Form. If you look closely at Figure 1-1, you'll see that the server controls have a small icon in their top-left corner; static content is not annotated in this way.

It's worth emphasizing that the designer is just a convenient tool for creating a Web Form's content. Anything added to the designer is actually converted and stored as HTML elements, and you can see this representation of the Web Form by clicking the HTML tab at the bottom of the designer. Figure 1-2 shows the HTML View of the Web Form shown in Figure 1-1.

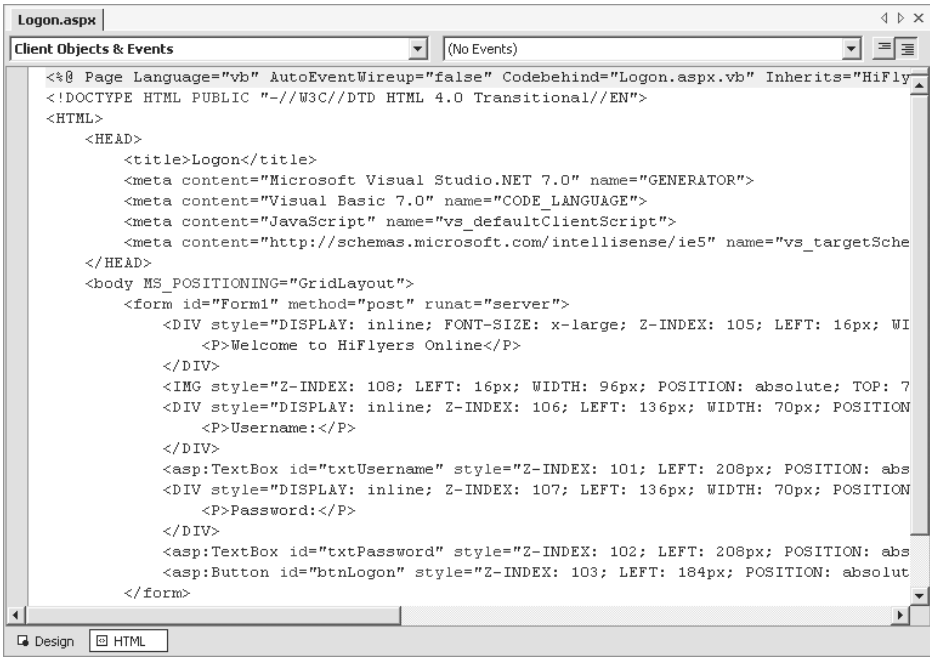

Figure 1-2. HTML View of Logon.aspx

If you're familiar with HTML then you should recognize much of this content. However, look closely at the HTML tags that define the two text boxes and button, and you'll see they have a rather non-standard format, consisting of `<asp:TextBox>` and `<asp:Button>` tags as well as a variety of non-standard attributes. Remember that these three controls are server controls—what you're seeing are the server control tags; the HTML sent to the browser by this Web

Form will be quite different, and these server control tags will be replaced with standard HTML elements.

As well as the visual content added in the designer, Web Forms will usually contain code. This may be stored within the Web Form's file itself (<filename>.aspx) or may be placed into a *code-behind* module associated with the Web Form. These modules typically have names that end in .aspx.vb for Visual Basic .NET and .aspx.cs for C# code. We'll return to the topic of code and modules in the "Understanding Web Applications" section later in this chapter.

If you're not familiar with HTML notation, or just want to brush up on your knowledge, refer to Appendix A, which provides an overview of HTML syntax and behavior. Chapter 2 returns to the topic of Web Forms in far more detail, showing how they can be created, customized, and used throughout Web Applications.

Server Control

Server controls are intelligent user interface objects you add to your Web Forms. Some server controls represent simple objects, such as text boxes, buttons, and lists, and others represent more complex structures such as grids, tables, and calendars. Server controls are able to change the way they render their output according to the client browser's capabilities. On modern browsers, they can take advantage of features such as client-side scripts and DHTML to provide a richer and more responsive interface while at the same time maintaining base-level HTML 3.2 support for older browsers. They're also interactive elements, both with the user and with your code. This enables you to manipulate a server control by setting or reading its properties and invoking its methods; at the same time, the user sees it on their screen and can use it in the same way as a regular HTML element.

Server controls can have quite different design-time and runtime appearances. For example, Figures 1-1 and 1-2 showed the Design and HTML Views for a Web Form containing text box and button server controls, but if you view the page in a Web browser and display the HTML source, it appears similar to Figure 1-3.

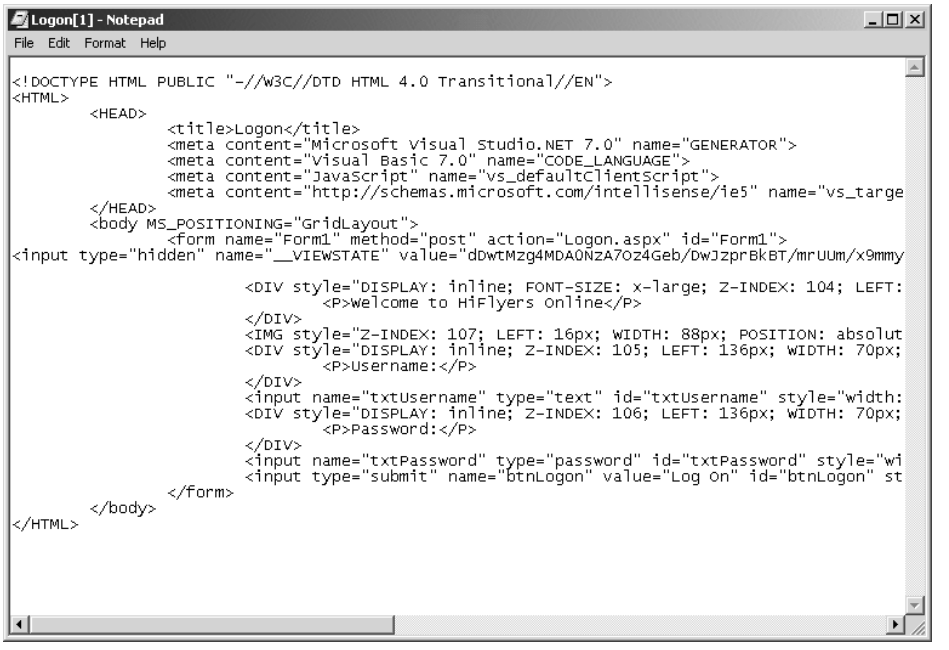

Figure 1-3. Browser-side HTML for Logon.aspx

You can see that the `<asp:TextBox>` and `<asp:Button>` tags have been translated to regular HTML `<input>` tags and that the `runat="server"` attribute has gone. Also, although the *id* attribute has been maintained, a matching *name* attribute has been added in the HTML sent to the browser. These changes were made within ASP.NET and were controlled by the logic within the controls themselves. There are other differences, too, including that the second textbox has been rendered to an input element of type `password` as this had a `TextMode="password"` setting in the source file.

However, the really interesting thing about server controls is that, from the point of view of code on the server, the controls are simply programmable objects with rich sets of properties, methods, and events. They're not HTML tags nor elements, and they're not textual definitions that have to be generated by "cookie-cutter" code. They're objects. This enables you to take a completely new approach to Web development and lifts the barrier on structured coding, code reuse, and many other often-requested features.

Postback

Postback is the term given to the process that occurs when a Web Form is submitted. Submission occurs when the user clicks one of the buttons on a Web Form or when some other action causes a request to be sent to the Web server.

The definitive thing about postbacks—that is, the thing that makes them different from other submissions and requests—is that the Web Form is *submitted to itself*. In other words, the code used to process the request and create the next Web page is the same code used to create the current Web page.

The use of postbacks in this way enables ASP.NET to simplify page processing because it ensures that all the logic for handling the Web Form request (which is used when the postback occurs) has direct access to the objects that define its interface. This is the same approach used for Visual Basic desktop development, where each Windows Form contains the user interface objects and the associated code that is executed when events occur for those objects. In fact, as you'll see in the next section, ASP.NET also adopts the concept of events (known as *server events*) that are raised during the postback process.

Postbacks are the default mechanism used by Web Forms in ASP.NET and occur because the `<form>` element within the Web Form has no `action` attribute defined for it. For example, look back at the HTML View of the Web Form in Figure 1-2 and notice that the only attributes are `id, method,` and `runat`. Now, look back at Figure 1-3 and examine the `<form>` element sent to the browser. When ASP.NET processed the Web Form, it replaced the design-time attributes with valid HTML settings including an `action="Logon.aspx"` attribute to cause the postback.

If for some reason we didn't want a postback to occur, but instead wanted to submit the content of the Logon form to another page for processing, then at design-time we could define an action attribute that referred to the required page. At runtime ASP.NET will simply pass this through to the browser unmodified.

Server Event

The final concept we'll introduce at this stage is the *server event*. Server events are closely allied to Web Forms and server controls, and indeed these two types of object are the source for many server events. As the term implies, server events are notifications sent to your server-side code from ASP.NET objects, and these events correspond to phases in the page-processing cycle or to actions initiated by the user. Irrespective of the event's cause, when it triggers on the server, your code can respond to it by way of event procedures.

For example, if you return to Visual Studio .NET and double-click the button on Logon.aspx, you're presented with an empty event procedure for the button's click event. Any code you add to this procedure will be executed when the user

clicks the button in the browser. Figure 1-4 shows an extract of code to perform simple verification of the details entered into the username and password text boxes.

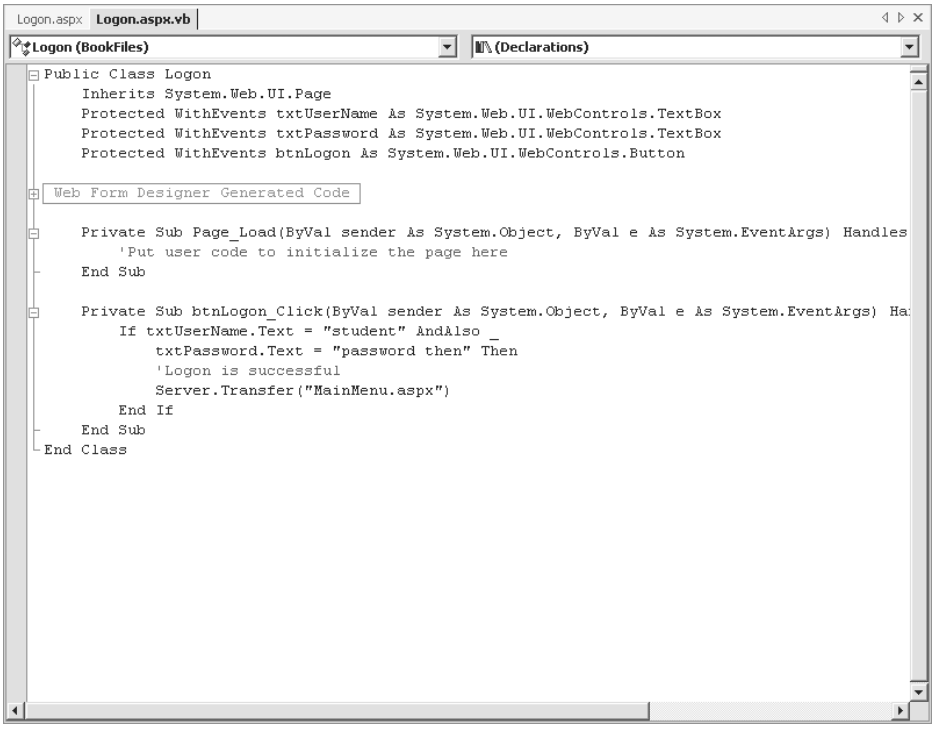

Figure 1-4. Code for the click event

You can see that the controls are referenced as objects, and that the code simply reads the Text property of each. In this respect it's similar to code you may write for desktop applications, but compared to classic ASP it's a revolutionary change.

We'll examine how these events are raised in much more detail in later chapters, but for now you should remember that although the source of the event was an action in the browser, the effect of the event is to run code on the server.

Understanding Web Applications

It's worth spending a little more time investigating Web Applications at this stage, but rather than just letting us describe what a Web Application looks like and where it's stored, why don't you go ahead and create one?

Start by loading Visual Studio .NET. Unless you've configured it otherwise, you should see a Start Page similar to that shown in Figure 1-5.

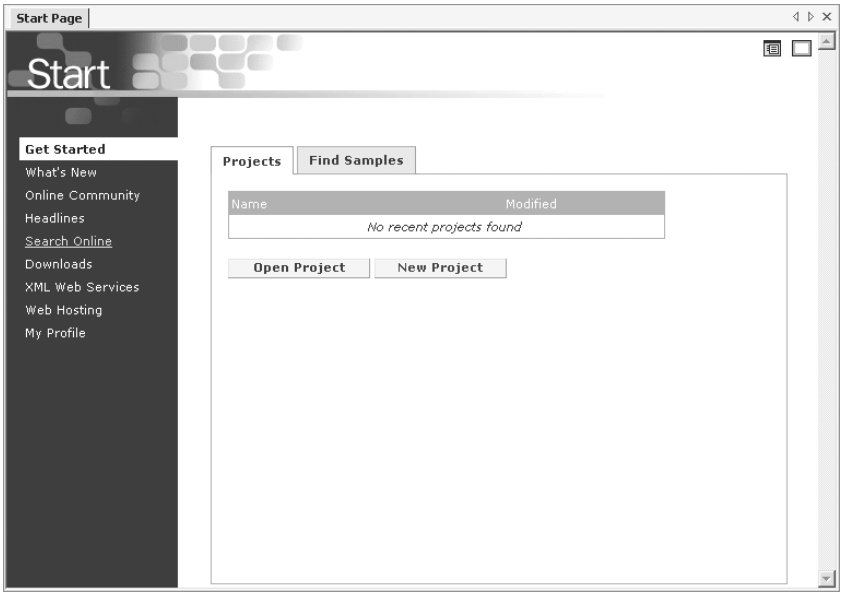

Figure 1-5. Visual Studio .NET Start Page

Click **New Project**, select **Visual Basic Projects** for the project type, and then select the **ASP.NET Web Application** icon in the Templates pane. Finally, enter the Location of the application as `http://localhost/FirstApplication`, at which point the dialog box looks like Figure 1-6, and then click **OK**.

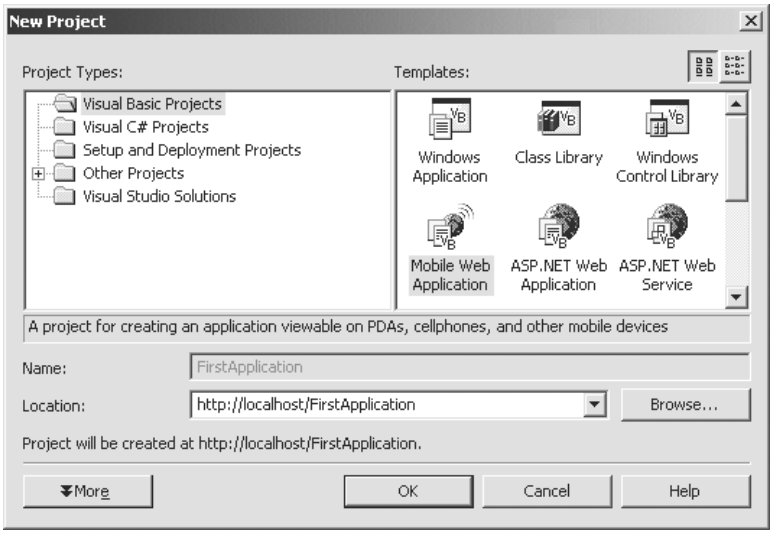

Figure 1-6. Creating your first Web Application

Once the project has been created, Visual Studio .NET shows a summary of its content in the Solution Explorer. At the moment you should see that it contains the following files:

- AssemblyInfo.vb
- FirstApplication.vsdisco
- Global.asax
- Styles.css
- Web.config
- WebForm1.aspx

It should also contain a References folder containing the .NET assemblies currently referenced from the project, much as the References dialog box in Visual Basic 6.0 lists the COM components currently referenced.

This is merely a summary of the project's content because the default configuration of Visual Studio .NET hides many files from you. If you want to see all of the files in the project, select **Project ➢ Show All Files** from the menu, and you'll see additional elements, mostly child elements of the existing files. Figure 1-7 shows the Solution Explorer with all of the files displayed.

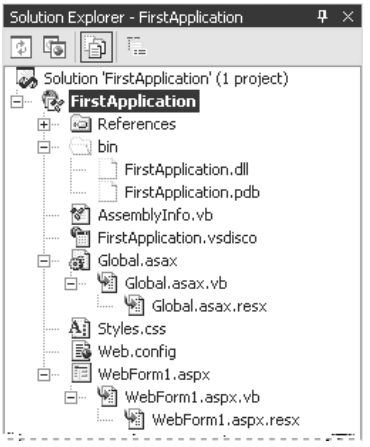

Figure 1-7. Complete project contents

Okay, so this is the Visual Studio .NET view; what about when you run it? Well, if you try and run the application right now (by selecting **Debug ➢ Start**), then it'll look pretty plain. In fact, it'll look empty because you've not created anything on the Web Form to be displayed in the browser. Although there's not much to see at this stage, it's actually quite useful to go and have a look at the project files from the point of view of IIS because this is ultimately the software that hosts the Web Application.

Load Windows Explorer and navigate to the main IIS root folder. This is usually C:\Inetpub\wwwroot but could be different on your PC. You'll see a **FirstApplication** folder, which in turn contains the files shown in Figure 1-8.

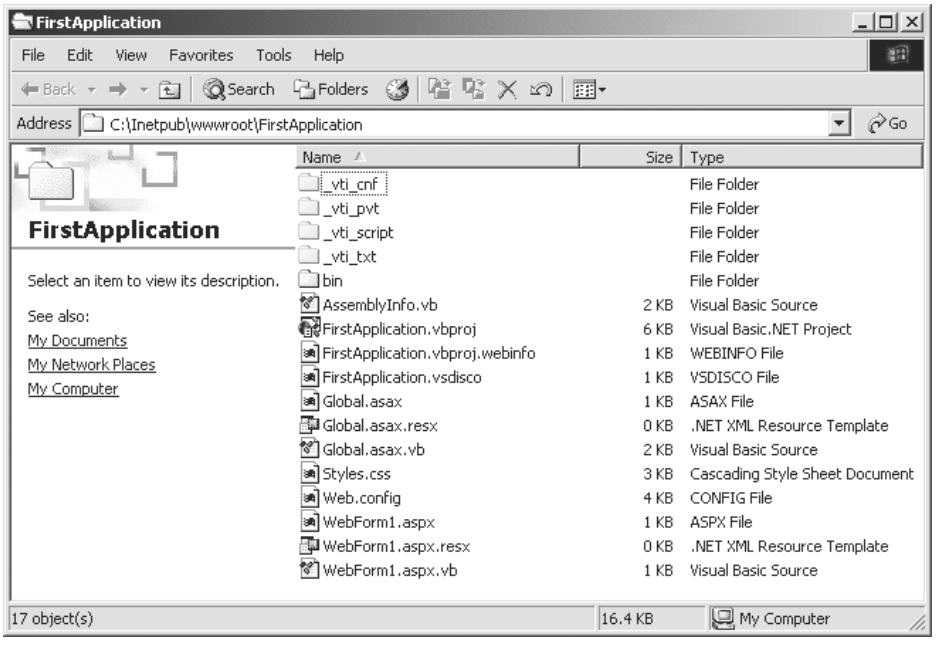

Figure 1-8. Project contents through Windows Explorer

You can see that all of the files shown in the Solution Explorer are present, plus some additional folders (which are flagged with a hidden attribute) and the project file itself. Any files added to the Web project in Visual Studio .NET will be copied to this folder as well.

NOTE *This is the only copy of your project and its content that is maintained by Visual Studio .NET. Make regular back-ups of this folder.*

Additional Content

The files that Visual Studio .NET creates within a Web Application are only a starting point. You'll most likely need to create additional Web Forms, classes, controls, and Web Services at some stage (although not necessarily in the same application), and you'll probably want to bring in existing content files such as HTML and XML documents. In truth, the list of possible content files is endless because your Web Application can contain any valid file type that the operating

system supports. However, most applications use a small subset of the possible file types, with the following being the most common:

- **.asp files**: Classic ASP files, which can be run side by side with ASP.NET applications, even within the same virtual directory. This eases migration and upgrades by allowing conversion to be performed gradually. However, classic ASP files will not have access to any of the new .NET features and will be handled by the standard asp.dll handler.

- **.aspx files**: ASP.NET Web Forms, which form the user interface of a Web Application. They're often associated with .aspx.vb and .aspx.resx files, which are used within the development environment to hold code and resource information respectively.

- **.asmx files**: ASP.NET Web Services, which are components that can be called over the network by other applications to perform specific functions. Web Services are one of the replacement technologies for DCOM, and they're designed to be Internet and firewall friendly. As with Web Forms, they're usually associated with .asmx.vb and .asmx.resx files.

- **Global.asax**: ASP.NET version of global.asa, which contains application-level event handlers, definitions, and objects.

- **.htm, .html, .css**: Traditional HTML files and style sheets.

- **.xml**: XML documents, which can be processed by .NET applications (see Chapter 9) or passed straight to the browser for client-side manipulation.

- **.gif, .jpeg**: Image files and graphics, often maintained in their own \Images directory, although this is a preference rather than a requirement.

- **.config**: XML documents that manage .NET specific settings. The project will contain a Web.config file in the virtual root, but each subdirectory can have its own Web.config file to override specific settings. There's also a global Machine.config file that maintains machine-wide settings; you can find this file under the folder C:\WINNT\Microsoft.NET\Framework\v1.0.XXXX\CONFIG rather than within any single application.

- **\bin directory**: Contains .NET assemblies and compiled code required by the Web Application. If you use Visual Studio .NET to build Web Applications, there will be a .dll file with the same name as the project that contains the compiled code for the application.

To add any of these files types to an application, you need only to place them into the Web Application's virtual directory. Subject to permissions and configuration settings (see Chapters 10 and 11 for a full discussion of these issues), the files will then be accessible from the client browser.

However, content added in this way will *not* automatically become a part of the Visual Studio .NET project. To add a file to the project you can select **Project ➢ Add Existing Item** and then browse for the names of the files to be added. If necessary, Visual Studio .NET will copy them to the virtual directory and then add them to the list of files shown in the Solution Explorer.

The benefit of adding files to the project becomes clear when you need to deploy the application because a Web Setup Project can be used to automatically deploy *all* of the project content. Web Setup Projects are the .NET equivalent of tools such as the Package and Deployment Wizard and the Visual Studio Installer; you'll see how to create them in Chapter 10, which discusses the processes of packaging and deploying Web Applications. You may also find that other management and development tasks are also eased, as you will have the full capabilities of the Visual Studio .NET development environment available to you.

Virtual Directories and ASP.NET

IIS configures the FirstApplication folder that contains the Web Application as a virtual directory (see Appendix A for more information if you're not sure what this means). This happens when the Web Application is first created and enables IIS to apply a variety of configuration parameters to the application independently of any other sites or applications running on the same server. To see the virtual directory configuration, load the Internet Services Manager utility, found in Administrative Tools, and then expand the **ComputerName** and **Default Web Site** nodes to display the list of folders, virtual directories, and applications. Click on **FirstDirectory** to display the content. Figure 1-9 shows how this may appear, although the list of virtual directories on your computer will contain different entries.

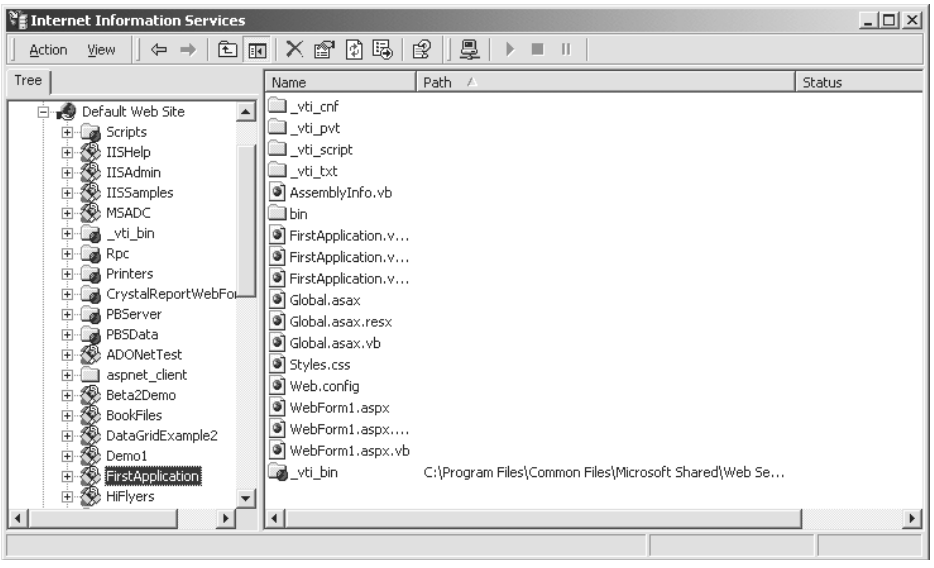

Figure 1-9. Project contents shown through Internet Services Manager

As you can see, the content of the virtual directory matches that shown in Windows Explorer, but Internet Services Manager also allows you to view the properties of the virtual Web directory. Right-click the **FirstApplication** entry, select **Properties** from the context menu, and you'll see the dialog box shown in Figure 1-10.

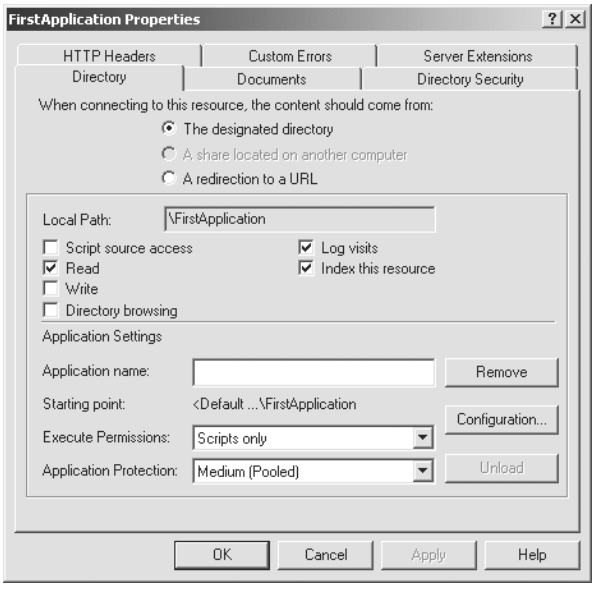

Figure 1-10. Properties for the FirstApplication virtual directory

Clearly, you can define many settings, some of which we'll return to later. For now, notice that the Directory tab contains basic permissions and application settings. If you switch to the Documents tab, you'll see that it defines the names of the default files that IIS will search for when a user browses to this application. Visual Studio .NET defined all of these settings when it created the virtual directory, during the initial creation of the project.

Web Application Content

In our previous definition of a Web Application we stated, "a Web Application consists of all the files, pages, handlers, modules, and executable code that can..." How does this compare with the Visual Studio .NET view of a Web Application? Well, you can clearly see that all the content added to the application was placed in a single virtual directory, and when we delve further into the architecture of .NET you'll see that ASP.NET functionality contained within HTTP Handlers and HTTP Modules are also executed within the scope of the application.

However, this does not mean you *must* create all content within Visual Studio .NET. To illustrate this, we'll use Notepad to create an additional file in our virtual directory:

1. If Visual Studio .NET is open, close it completely, so you can be sure it doesn't play any role in what follows.

2. Load Notepad, and open the Webform1.aspx file in the FirstApplication directory. Add a heading to identify the page between the `<form>` and `</form>` tags:

```
<form id="Form1" method="post" runat="server">
    <h1>This is WebForm1.aspx</h1>
</form>
```

3. Save Webform1.aspx, then create a new blank file in Notepad, and enter the following:

```
<%@ Page Language="vb" %>
<html>
  <body>
    <form id="Form1" method="post" runat="server">
        <h1>Welcome to FirstApplication</h1>
        <asp:button id="btnNavigate" runat="server" text="Navigate"/>
    </form>
  </body>
</html>
```

4. Save this file as Default.aspx in the FirstApplication directory. Make sure that Notepad doesn't add its own .txt file extension.

5. Load your browser, and navigate to `http://LocalHost/FirstApplication`. You should see the Default.aspx page because this name is configured as one of the default documents that IIS recognizes. However, although the button is displayed, it doesn't do anything yet.

6. Return to the source code of Default.aspx in Notepad, and add the following at the bottom of the file:

```
<script runat="server">
    Protected Sub btnNavigate_Click(Sender as Object, E as EventArgs)
        Server.Transfer("webform1.aspx")
    End Sub
</script>
```

   This code defines an event-handling routine, the purpose of which is to transfer control to another Web Form (Webform1.aspx) when a user clicks the Navigate button. We've chosen to do this with the `Server.Transfer` method, although we could also have used `Response.Redirect`. Chapter 12 examines navigation techniques and methods, and describes the relative merits and disadvantages of each approach. The signature of this procedure is important, as all .NET event handlers are expected to accept two parameters.

   The first parameter (**Sender**) is a reference to the object that raised the event. You might think this is redundant because we've already decided that this handler will be associated with events from the Navigate button; however, as we shall see in Chapter 4, you can define event-handling routines to be associated with multiple controls, and so the Sender parameter provides an easy way to identify which of these controls has raised the event.

   The second parameter (**E**) is a reference to an object that provides additional information about the event. In the case of a click event there's no useful additional information, but for events such as ItemClick (in a ListBox) or ItemCommand (in a DataGrid) the E parameter includes details of which item or row has been selected or activated.

7. Modify the tag for the `<asp:button>` by adding a definition of the `OnClick` handler. This will read as follows:

```
<asp:button id="btnNavigate" runat="server" text="Navigate"
                    onclick="btnNavigate_Click"/>
```

We need to add this additional attribute to ensure that ASP.NET associates our server-side event procedure with the control.

8. In the Web browser, navigate to `http://LocalHost/FirstApplication` once again. Click the button on the page, and you'll be redirected to the WebForm1.aspx page. Return to the Default.aspx page and view its source—you'll see that there's no client-side script, demonstrating that the event handler we added is executed only on the server.

Although you can create content using Notepad or other text editors, in most cases it would be inappropriate to do so. In this example, the Default.aspx file contains both the visual elements for the page as well as the code that handles the events. This approach can be problematic in the long term, increasing maintenance requirements and minimizing the chance of reusing code. One of the key features of ASP.NET is its ability to separate code from content, and this is emphasized when you build applications using the Visual Studio .NET tools.

Another point worth noting is that if you return to Visual Studio .NET and view the content of the project in the Solution Explorer, the Default.aspx file will either be gray (if Show All Files is selected) or be hidden (if the option is off). This is because you have not added the Default.aspx file to the ASP.NET project, even though you added it to the application's virtual directory. Figure 1-11 shows the view when Show All Files has been selected.

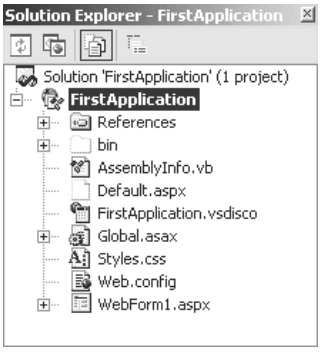

Figure 1-11. Solution Explorer showing all files

As mentioned earlier, you may find it easier to manage Web Applications if *all* of the content is included in the Visual Studio .NET project. Because Default.aspx is already present in the application's virtual directory, the easiest

way to add it to the project is to right-click on the name in the Solution Explorer and select **Include In Project**. You'll be warned that no class file exists for the Web Form, but you should specify No when asked if one should be created. Our Web Form contains code embedded in the .aspx file, whereas Visual Studio .NET expects it to be in a code-behind module. There's nothing wrong with this approach, it's just not what the development environment is expecting.

Understanding Web Services

The type of application you've seen up until this point is perhaps more accurately described as a *Web Forms application* because it uses Web Forms to create a visible user interface that can be displayed in a browser. However, Web Applications can contain other types of component as well, and one of these is a *Web Service*.

Web Services present a programmatic interface rather than a visible one, and users usually access them from other applications (including Web Applications and desktop applications) rather than from a browser. You build Web Services using .asmx files rather than .aspx files, but the two types of file can freely coexist within the same project. Web Services and Web Forms share many features, and you'll find that their coding structures and style are similar.

We'll discuss Web Services in much more detail in Chapter 14, and you'll get plenty of opportunities to try creating your own. For now, let's add a simple Web Service to the FirstApplication project created previously:

1. If necessary, load Visual Studio .NET and open the FirstApplication project.

2. Select **Project ➢ Add Web Service** and set the Name for the new component to `Forecast.asmx`.

3. You'll be presented with a blank designer, which you can close as this Web Service will be entirely code-based.

4. Select Forecast.aspx in the Solution Explorer, then display the code for the component by selecting **View ➢ Code** or pressing the F7 function key. It'll appear as shown in Figure 1-12.

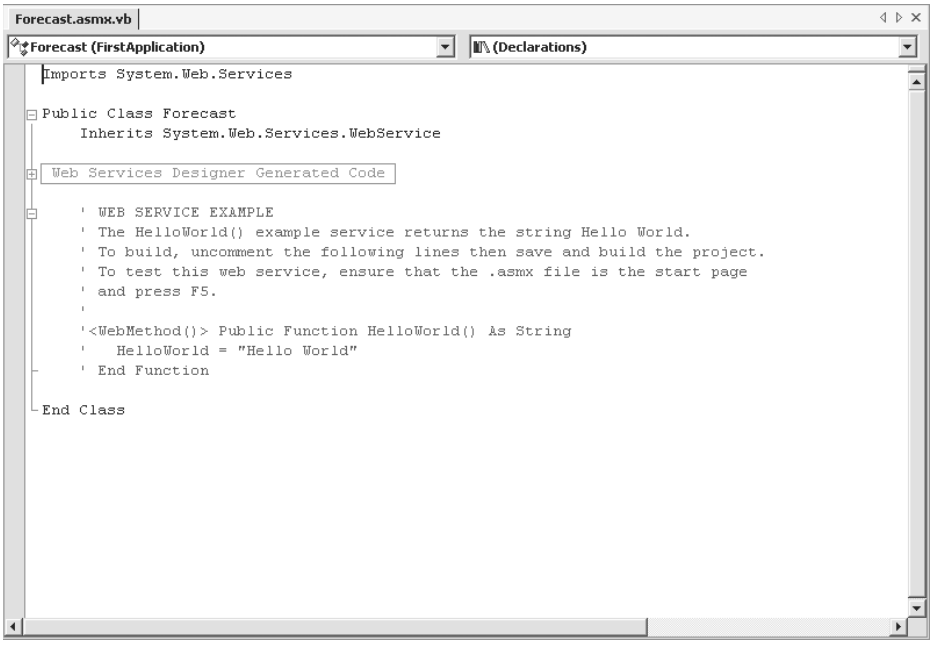

Figure 1-12. Default content of a Web Service file

5. Delete all of the commented (green) code, and replace it with the following:

```
<WebMethod()> Public Function GetUKWeather() As String
    Dim intRandom As New Random()
    Select Case intRandom.Next(3)
        Case 0
            Return "It is cloudy"
        Case 1
            Return "It is raining"
        Case 2
            Return "It is raining hard"
        Case 3
            Return "It is raining very hard"
    End Select
End Function
```

6. Save the file, then right-click in the Solution Explorer and choose **Build And Browse**. Ordinarily you would use a separate client application to call the Web Service, but in this case you don't yet have one. Fortunately,

ASP.NET helps out by creating a browser-based interface that you can use for testing. Figure 1-13 shows how this appears in the browser window.

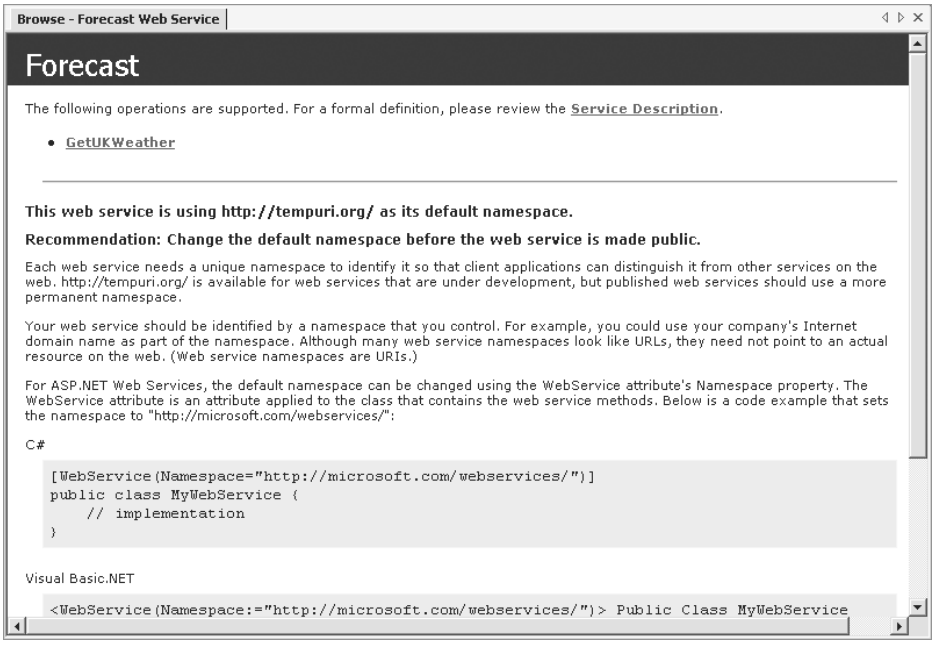

Figure 1-13. Testing the Web Service

7. Click the **GetUKWeather** hyperlink to display the next page, and then click the **Invoke** button to test the Web Service. A separate browser window opens to show the result, as shown in Figure 1-14.

Figure 1-14. Results from the Web Service

Notice that the results display in XML notation, which is the format in which all Web Service information is transferred. The reason Web Services return information in the form of XML is that XML is a completely language- and platform-independent way of representing data that can be passed across the Internet, so

virtually every developer in the world can call your Web Service. Typically, developers will use client tools to call Web Services, which completely hide the fact that the Web Service uses XML at all. For example, a .NET client program can create a *Web reference* to your Web Service, after which it can call the `GetUKWeather` method simply using code such as the following:

```
Dim objSvc As New MyServerName.Forecast()
lblWeather.Text = objSvc.GetUKWeather
```

This example assumes that the machine hosting the Web Service is called `MyServerName`.

NOTE *You should remember that Web Services are simply components that can be placed into a Web Application and that they're created, executed, and managed in a similar way to Web Forms. There are some major differences in terms of their design and planning, but the features available to these two component types are almost identical.*

Chapter 14 examines the specific details of creating Web Service applications and Web Service clients. However, remember that there's a lot of shared technology between Web Form applications and Web Service applications, so the content of most other chapters in the book is equally applicable.

Introducing ASP.NET Intrinsic Objects

Visual elements and components, such as Web Forms and server controls, provide many powerful features, and you'll use at least some of them in almost every application. They play a similar role to that of Windows Forms and controls in desktop applications, but as with desktop applications it's also often necessary to get "under the surface" of Web Applications.

In the case of ASP.NET, *under the surface* means using the classes provided within the .NET Framework, most of which are common to all .NET application types. However, ASP.NET has a set of specialized objects you use for interacting with a Web server, as well as manipulating the information received from and sent to the client browser. These are known as ASP.NET *intrinsic objects* and are available to every Web Form and Web Service element in a project.

Strictly speaking, the intrinsic objects are exposed as properties of a number of classes, including the System.Web.UI.Page and the System.Web.Services.WebService classes, which are the base classes for Web

Forms and Web Services, respectively. Because your Web Form or Web Service derives from one of these classes, it inherits all of the class properties, methods, and events. We'll see more about this inheritance relationship in Chapter 2. (If the concept of inheritance is new to you, you may find it useful to review Appendix C first.)

There are many intrinsic objects, of which the following sections describe only the most commonly used. We've included these objects in this introductory chapter to make you aware of their presence, as we'll be using some of them in examples and code fragments throughout the book. In fact, you've already used the Server object when you called the `Server.Transfer` method in the earlier exercise. This emphasizes the fact that even for the most trivial of ASP.NET applications, there's a good chance you'll need to use some of this functionality.

Application

The `Application` object is an instance of the HTTPApplicationState class, and its main purpose is to allow you to store *state* (information) in a Web Application such that it:

- Is persistent across page requests and user sessions
- Is shared between all concurrent users of a Web Application

Chapter 12 discusses state management in detail, including a thorough coverage of how you use and control the Application object.

Cache

The `Cache` object is an instance of the Cache class. It's also provided to allow state management; however, it works differently than the Application object:

- Cache state is persistent across page requests and user sessions but can be associated with dependencies that cause the data to become invalidated under certain circumstances, such as when a time period elapses or a file on disk changes.
- The Cache object is automatically thread-safe, whereas the Application object requires that your code make explicit calls to the `Lock` and `Unlock` methods.

We'll leave further discussion of the Cache object to Chapter 12, where we'll compare and contrast the different options available for state management.

Error

The `Error` object is an instance of the Exception class, and it represents the *first* error that occurred (if any) during the processing of the current request. As we will see in Chapter 10, there are multiple phases to the request processing cycle, many of which occur before any of your code can run. The Error object provides a way for you to determine if those phases were error-free, or if some problem occurred that you now have to handle.

The Error object is not exposed directly as a property of the Page or WebService classes, but instead must be referenced through the page's Context property. For example:

```
strFirstExcep = "Exception caused by " + Context.Error.Source
```

Request

The `Request` object is an instance of the HTTPRequest class and is created by ASP.NET to enable your code to read information passed from the client browser to the server when the page request was made. This enables you to do a number of key things:

- Read data entered by the user into an HTML form on the page that generated the request.
- Read data from the *querystring* defined for the request. The querystring is the sequence of characters that is appended to the URL part of the requested page address, and the querystring and URL are separated with a *?* character.
- Read cookie values from the information passed by the browser. Note that if the browser does not pass the cookie values to the server, your code cannot read them; you have access only to the information sent from the browser.
- Read the value of *server variables* sent with the request or generated from it. Server variables provide additional information about a request, such as what browser is in use, what the user's IP and hostname are.

Note that ASP.NET performs some of these tasks automatically. For example, the browser type and version is automatically identified and used to determine what client features are rendered, and the content of form data is read automatically during the postback process. However, the Request object is still extremely important.

Response

The `Response` object is an instance of the HTTPResponse class, and it performs a complementary process to the Request object. Where Request enables your code to obtain information from the client browser, Response enables you to send information back. There are many tasks where the Response object will be used, including the following:

- Sending textual and binary data to the browser
- Controlling how page content is transmitted and defining whether buffering is enabled to prevent pages being drawn piecemeal in the browser
- Controlling page caching, at the server and browser levels
- Specifying additional HTTP headers
- Controlling navigation

The Response object is used extensively in most ASP.NET applications, as it allows fine control as well as dynamic content generation.

Server

The `Server` object is an instance of the HttpServerUtility class. This class provides properties and methods that assist in processing page requests. Typical tasks the Server object can perform include:

- Application-wide exception handling.
- Encoding of strings into valid HTML and URL notations. This includes substitution of special characters with their HTML equivalents.

- Controlling page execution, processing, and navigation.
- Mapping logical file names to physical locations.

Although not used as much as Request and Response, Server is still an important object and allows code to interact more readily with Web clients without the need for custom translations and mappings.

`Session`

The `Session` object is an instance of the HTTPSessionState class, and it's used for session and state management. If Session state is enabled (see Chapter 12), each user is allocated a session ID when they first access the application. This session ID is usually stored in a transient cookie and passed to the server with each request. The server uses the session ID to track each individual user and to allow storage of user-specific state. ASP.NET also uses the session ID to track which users are continuing to use the application and therefore to timeout or expire the sessions of inactive users.

Transient cookies are typically stored in-memory within the browser process, so when the browser is closed the session ID is lost. If the user subsequently opens another browser and views the same application, they'll receive a different session ID. Also, if a user opens two browsers on the same computer and navigates to the same application in each, each browser will be allocated a different session ID, and so the application believes there are two distinct users.

If cookies are disabled on the client browser, ASP.NET instead inserts the session ID into the URL of the response sent to the browser—this is known as *munging* the URL. When subsequent requests are made from this page, the munged URL is passed back to the server and the session ID retrieved. In this way, ASP.NET is not dependent on cookies for session support, although the cookie-based approach is neater.

As with the Application object, the main purpose of Session is to allow you to store state in a Web Application such that it:

- Is persistent across page requests
- Is correctly released and destroyed when the session terminates
- Is unique to a single user session within a Web Application

Chapter 12 discusses state management in detail, including a thorough coverage of how the Session object is used and controlled.

Trace

The Trace object is an instance of the TraceContext class, and it provides methods and properties that enable you to write custom entries to the trace log that ASP.NET can generate for your application.

As you'll see in Chapter 10, you can enable tracing for the entire application or for specific pages, and by default it'll record details relating to request details, server events, form and query string content, cookies, and much more. By using the Trace object you're able to supplement this default output with your own messages, including details of any exceptions that have occurred in your code.

User

The User object is an instance of either a GenericPrincipal or WindowsPrincipal class, depending on the current security configuration of the Web Application. The main purpose of the User object is to provide a mechanism for determining the security permissions and privileges of the user making the request.

If you make use of the User object for checking security, you're said to be implementing a *programmatic security* policy; it's a technique that provides great flexibility and control, as you can perform security checks at any level in the application—when displaying a page, when rendering a control, or when responding to a server event.

The alternative to programmatic security is to implement *declarative security*, where the security settings are defined in the Web.config file. This approach is more granular, or coarse, because declarative security settings can be applied only at the folder or file level, rather than on a method-specific basis.

Chapter 11 deals with the topic of security and examines how declarative and programmatic techniques can be applied to ASP.NET applications.

Summary

Right now, ASP.NET has to be *the* reason for switching to the .NET environment. ASP.NET applications are by definition centralized onto a single Web server or Web farm, and if you plan it properly, you can install the .NET Framework and components with minimal disruption and interruption.

Developers of ASP.NET applications have the most to gain, with a much simplified event-driven programming model, powerful server controls, practical data binding, and comprehensive full-featured languages such as VB .NET and C#. System administrators and tech-support personnel are catered to with easy text-based configuration and management, no-touch deployment and upgrades, and an independence from the registry and COM.

Index

I

M

N

T

W

X