Beginning Web Development, Silverlight, and ASP.NET AJAX

From Novice to Professional

Laurence Moroney

Apress®

Beginning Web Development, Silverlight, and ASP.NET AJAX: From Novice to Professional

ISBN-13 (pbk): 978-1-59059-959-4

ISBN-10 (pbk): 1-59059-959-4

ISBN-13 (electronic): 978-1-4302-0582-1

ISBN-10 (electronic): 1-4302-0582-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Kevin Goff
Technical Reviewers: Fabio Claudio Ferracchiati, Bob Lair
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Kevin Goff, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editor: Damon Larson
Associate Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Brenda Miller
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

I'd like to dedicate this book to my family: my wife, Rebecca, and my wonderful children, Claudia and Christopher. I'd also like to dedicate it to the one who has made all this possible. John 3:16

Contents at a Glance

Contents

About the Author

LAURENCE MORONEY is a senior technology evangelist at Microsoft. He specializes in Silverlight and promoting how Silverlight can be used in real-world systems to enhance the user experience. Author of many computer books and hundreds of articles, he's usually found tapping at his keyboard. Outside of his computer passions, he's big into all kinds of sports, and has been involved with professional men's and women's soccer.

About the Technical Reviewer

FABIO CLAUDIO FERRACCHIATI is a senior consultant and a senior analyst/developer using Microsoft technologies. He works for Brain Force (`www.brainforce.com`) at its Italian branch (`www.brainforce.it`). He is a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microsoft Certified Professional, and a prolific author and technical reviewer. Over the past ten years, he's written articles for Italian and international magazines and coauthored more than ten books on a variety of computer topics. You can read his LINQ blog at `www.ferracchiati.com`.

Introduction

This book is aimed at equipping you, the developer, to understand the technologies that are available to allow you to rapidly build secure, quality web experiences. Note that I use the term *experiences* and not *applications* or *sites*. That is because the user experience is the heart of the future Web.

Before you can start looking at the future, it is good to understand the current suite of web development and deployment technologies that are available to you. In Part 1 of this book, you'll look at the Microsoft stack of technologies that allow you to build web services and applications, and how you'll deploy them. It will be scenario-driven, so instead of going into depth on the various APIs, you'll get your hands dirty in a step-by-step approach to building, testing, and deploying multitier web applications. You'll look at databases and how to connect your application to them, and you'll manage these connections through the deployment process. Ultimately, in the first six chapters, you'll get a whirlwind tour of the full life cycle of application development using the .NET Framework (which always looks good on a resume!).

If you are new to ASP.NET, these six chapters will condense everything you need to know to get up and running with the framework. By the end of them, you'll have learned the technology, the tools, and the servers, and gained the know-how to deploy a multiple-tier web service–based application to the enterprise server technology from Microsoft. Even if you are experienced with ASP.NET, this is a nice refresher!

Chapter 1 will give you a tour of the history of web development, from static HTML served up from the network, through activation of servers using CGI, to activation of pages using ASP, PHP, and other technologies. It ends with a survey of the managed APIs that are available for building web applications, including J2EE, PHP, and ultimately ASP.NET.

In Chapter 2, you will look into ASP.NET in a little more detail, going through the basics of web development with this API. You'll see its architecture and how it uses the concept of *controls* to generate markup from the server. You'll see how it hangs together with the standard web technologies of HTML, JavaScript, DHTML, and more. There is a great suite of tools available to the ASP.NET developer, including the free Web Developer Express, and you'll look at how to download, install, and use this to build, deploy, and debug ASP.NET server applications. Finally, you'll survey the lifetime of an ASP.NET application, learning how the framework can provide stateful communication in an inherently stateless environment.

Chapter 3 takes you further into building ASP.NET web applications through the use of web forms. You'll look into the page processing model, postbacks, and how events are handled in web applications. You'll also start to look into data in your web applications. You'll see how to download, configure, and manage a SQL Server Express instance, and how to access the data and functionality in it from code, from UI tools, and from data binding.

Chapter 4 brings you further down the data path, looking at data binding in ASP.NET and explaining the fundamentals of the ADO.NET API. You'll look into the architecture of this flexible data framework, including data providers, and the DataSet and DataAdapter components. You'll also see how some of the data-aware controls such as the GridView are used to provide great data experiences for your users.

Chapter 5 takes you in a different direction, looking at Web Services and how this vital technology is implemented using ASP.NET. You'll see how to build a web service that wraps a database and exposes its contents to users in a platform-agnostic, technology-agnostic way. With Web Services, the technology that implements the service should be abstract, and you'll see how this is achieved using XML and the WS-I basic profile. You'll see how you can build your services to be consumed by applications running on other technologies, such as Java. You'll expand on some of the examples from Chapter 4, seeing how a multitier application can be built using Web Services as the data tier, and binding controls such as the GridView to them.

Part 1 of the book wraps up in Chapter 6. Here you will look at how to get your applications deployed and running using Windows Server 2003, SQL Server, and IIS 6. You'll look at how IIS serves pages up, and go through the scenario of deploying the multiple-tier application that you built in Chapter 5, moving it in a phased manner, performing unit testing on each tier. You'll also look at how to use the tools to automatically set up the virtual web sites that your application will run in.

Once you've wrapped all that up, you'll be ready to move into Part 2, which delves into the next-generation web technologies, and take an in-depth look at AJAX extensions for .NET, Windows Communication Foundation, Windows Presentation Foundation, Silverlight, and more.

PART 1

Building Web Applications

CHAPTER 1

Introduction to Web Development

To understand web development, you have to understand the Web, and to understand the Web, you have to understand the Internet that the Web is built on. This chapter will give you a brief history of the connected world, discussing first the origins of the Internet, then the origins of the Web, and finally the technologies used by developers to build applications on the Web. It will hopefully be a fun and informative ride!

The Internet and the Birth of the Web

The Internet dates back to the early development of general communication networks. At its heart, this concept of a computer network is an infrastructure that enables computers and their users to communicate with each other. There have been many types of computer networks over time, but one has grown to near ubiquity: the Internet.

Its history dates back to the 1960s and the development of networks to support the Department of Defense as well as various academic institutions. Interoperability of these different networks was a problem. In 1973, Robert E. Kahn of United States Defense Advanced Research Projects Agency (DARPA and ARPANET) and Vinton Cerf of Stanford University worked out a common "internetwork protocol" that they called the TCP/IP Internet Protocol Suite. This was both a set of standards that defined how computers would communicate as well as conventions for how the computers should be named and addressed, and thus how traffic would be routed between them.

At its core, TCP/IP follows most of the OSI (Open Systems Interconnection) model, which defines a network as an entity of seven layers:

Application layer: Provides the user interface (UI) to the network as well as the application services required by this interface, such as file access. In terms of the Internet, these application services are those typically provided by the browser, giving access to the file system to save favorites, print, and more.

Presentation layer: Translates the data from the network into something that the user can understand and vice versa, translating user input into the language used by the network to communicate. For example, when you use a browser to access a page, you type the address of that page. This address gets translated for you into an `HTTP-GET` command by the browser.

Session layer: Used to establish communication between applications running on different nodes of the network. For example, your request from the browser sets up a session that is used to communicate with the server that serves up the page. The lower levels are used to discover that server and facilitate the flow, but at this level you have a communication session storing all the data needed to manage the flow of data to the presentation tier for representation on the client.

Transport layer: Handles the task of managing message delivery and flow between nodes on the network. Networks are fundamentally unreliable because packets of data can be lost or received out of sync. Thus, a network protocol has to ensure delivery in a timely manner or trigger an action upon nondelivery. It also works to ensure that the messages are routed, assembled, and delivered correctly. For example, when using an Internet browsing session, your message is broken down into small packets as part of the TCP/IP protocol. The TCP/IP stack manages sending these packets to the server and assembling them correctly once they reach it.

Network layer: Used to standardize the way that addressing is accomplished between different linked networks. For data to flow from A to B, the locations and paths between A and B need to be known. This address awareness and routing is achieved by the network layer. There is almost never a direct connection between a client and a server. Instead, the traffic has to be routed across a number of networks through connections that involve changes of addresses between them. For example, a server might have the Internet protocol (IP) address 192.168.0.1 on its internal network, but that internal network faces the world using a different IP. A client calling the server calls the external IP, and has traffic on the web port routed to 192.168.0.1. This is all handled by the network layer.

Data link layer: Defines how the physical layer is accessed—that is, what protocols are used to talk to it, how the data is broken up into packets and frames, and how the addressing is managed on the physical layer. In the example just described, TCP/IP is the protocol. This is used on the network layer and above to manage communication between the client and the server. At this layer, the frames and routing information for TCP/IP packets are defined as to how they will run on the line as electrical signals. Network bridges operate at this layer.

Physical layer: Defines the type of medium, the transmission method, and the rates available for the network. Some networks have broadband communication, measured in megabits per second, whereas others have narrower communication in the kilobit range or less. Because of the wide variance in bandwidth, different behavior can be expected, and applications that implement network protocols have to be aware of this. Thus, the physical layer has to be able to provide this data to the next layer up, and respond to command instructions from it.

You can see this seven-layer model, and how typical TCP/IP applications such as the web browser fit into it, in Figure 1-1.

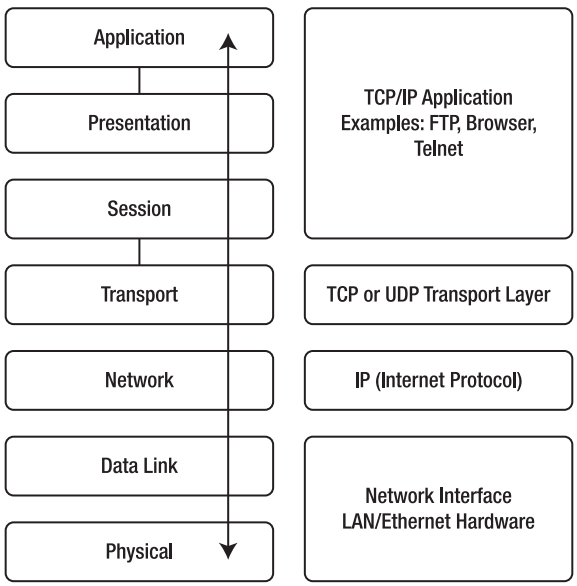

Figure 1-1. *OSI seven-layer model*

This model provided the foundation for what would become the Internet, and the Internet provided the foundation for what would become the World Wide Web.

By using TCP/IP, it became possible to build first a file transfer protocol (FTP) and ensuing application. From there, Tim Berners-Lee expanded the idea to having a "live" view of the file in an application called a "browser." Instead of just transferring a document from a distant machine to your own, the application would also render the file. To do this, the file would need to be marked up in a special way that could be understood by the browser. A natural progression to this is to allow documents to be linked to each other, so that when the user selects a link, they are taken to the document that the link refers to. This introduced the concept of *hypertext,* and thus HTML (Hypertext Markup Language) was invented.

An *HTML document* is a text document containing special markup that provides instructions to the browser for how to render the document. *Tags* such as `<H1>` and `<H2>` are used to provide styling information so that the document can be viewed as more than just text. Links to other documents are provided via another tag, `<a>` (for anchor), where a piece of text is defined as linking to another document, and the browser renders it differently (usually with a blue underline).

Once HTML was developed, the Web grew rapidly. Thanks to TCP/IP and the Internet, people could put documents on any server in the world and provide an address, called a *URL* (Universal Resource Locator), which could enable these documents to be found. These URLs could then be embedded as links in other documents and used to find those documents. Quickly, a huge network of interconnected, browsable documents was created: the World Wide Web.

Going Beyond the Static Web

This Web—a network of linked documents—was very useful, but in essence very static. Consider the scenario of a store wanting to provide links to potential customers of their current products. Their inventory changes rapidly, and static documents require people who understand the inventory and can constantly generate documents containing new details. Every time something is bought or sold by the store, these documents need to be updated. This, as you can imagine, is a time-consuming, difficult, and non-cost-effective task!

We needed some way to automatically generate documents instead of creating them manually. Also, these documents needed to be generated not in overnight batch runs, but rather upon request so that the information would always be up-to-date.

Thus, the "active" Web was born. New servers were written on the Common Gateway Interface (CGI) standard, which allowed developers to write code (usually in C) that executed in response to user requests. When a request came in for a document, this code could run, and in the case of our store scenario, that code could read a database or an inventory system for the current status and generate the results as an HTML document. This document would then be sent back to the browser. This system worked well, and was very powerful and widely used.

Maintenance of CGI applications became quite difficult, however, and CGI applications were also platform-specific, so if you had a cluster of servers, some of which were based on different technologies and/or versions of operating systems, you could end up with multiple versions of the same program to support! So, for example, if you wanted to run the same program on your cluster, but had different versions of an operating system, your code would have to be tailored for each machine.

But when there is a problem, there is also opportunity. And where there is opportunity, there is innovation. One of these opportunities was moving toward a *managed cross-platform code* approach in which a high-level language such as Java could be used

to build an application that generates dynamic pages. Because Java is a cross-platform language, the platform on which the code ran no longer mattered, and server-side Java, called *servlets*, became an effective replacement for CGI.

But the problem of generating HTML still existed. In these applications, string management, or `printf` statements, were used to write HTML, leading to ugly and onerous code. In another approach, HTML defined the output, and special extension tags instructed the server to do something when it reached those tags and fill in the placeholders. The code would look like the pseudocode here:

```
<h3>We have <% nQuantity %> widgets in stock. </h3>
```

The `<% %>` contains code that will be executed by the server. In this case, it is a value calculated on the fly as part of the session. When it's evaluated, the result is injected into the HTML and returned to the browser.

This is the underpinning of technologies such as classic ASP, which runs on Internet Information Server (IIS) and uses a Microsoft Visual Basic–like language between the tags. A similar architecture is used by Personal Hypertext Processor (PHP), which runs on its own interpreter that can be an IIS or other web server extension, and uses its own C++-like language. There are many other examples as well, such as Java Server Pages (JSP), which uses an approach where the HTML is not written out using code, but instead contains tags that are interpreted and replaced at runtime with calculated values; or Ruby, an object-oriented scripting language that works well for generating web content.

The opportunity was there for a best-of-both-worlds approach. And here is where ASP.NET arrived to fill the gap.

The Arrival of ASP.NET

ASP.NET was the result of developers and architects sitting back and thinking about the direction in which web development had taken to date. In some ways, developers had been painted into a corner by rapid innovation and were now in a nonoptimal development environment.

ASP.NET was designed to get around a number of issues regarding how to develop web applications at the time. At the same time, it began spurring innovation of new types of applications that previously might not have been possible.

First, it was designed to be a code-friendly environment using sophisticated object-oriented methodology that allowed for rapid code development and reuse, as opposed to the scripting-like environment used previously.

Second, it was designed to be a multiple-language single runtime, allowing developers from different backgrounds to use it with minimal retraining. For the Visual Basic folk, Visual Basic .NET was available, and for those used to more traditional object-oriented languages such as C++ or Java, a new language—C#—was introduced.

Third, the concept of *web services* was identified as being vital for the future of the Web, because they are a device-agnostic, technology-agnostic means for sharing data across the multi-platform Internet. ASP.NET was designed to make the complicated process of creating, exposing, and consuming web services as simple as possible.

Finally, performance of the Web depends not only on the speed of the network, but also on the speed of the application serving you. Absolute performance, defined as the overall speed of the application, is difficult enough, but application performance under different user loads implemented concurrently across multiple servers is more of a trick. ASP.NET was designed with optimizations for this in mind, including a compiled code model, where all the source code is turned into native machine language ahead of time, instead of an interpreted one, where all the source code is turned into native machine language step by step as it executes. It also includes a scalable data access mode, a way to keep state between client and server, data caching, and much more.

Summary

This chapter has given you a very brief background on what the Internet is, how the Web fits into the Internet, and how web application development has evolved to this point. It has also introduced you to the ASP.NET technology.

In this book, you'll look at ASP.NET in the .NET Framework and how it is used to build the web applications and services of today and tomorrow. In Part 1, you'll learn about the framework for building traditional web applications. Then, in Part 2, you'll move on to looking at how innovations for technologies such as Ajax and Windows Presentation Foundation (WPF) allow you to start improving the overall user experience. You'll also look at the development frameworks of tomorrow to learn how you can take the Web into the next phase of its evolution—that is, toward the next-generation Web, where the user experience is at the heart of everything you do.

CHAPTER 2

Basics of Web Development with ASP.NET

In Chapter 1, we looked at the history of web development technologies, culminating in smart server-oriented code that generates client-side markup as well as script that gets rendered by the browser for your users. The Microsoft technology that achieves this is ASP.NET, where *ASP* stands for Active Server Pages, and *.NET* is Microsoft's contemporary software runtime environment. In this chapter, you'll get an overview of what ASP.NET is and how it works. You'll first look at the tools that are available to build ASP.NET applications, from the free Microsoft Visual Studio Express tools to the various professional Visual Studio 2008–based packages. You'll see how to build an ASP.NET application with these tools before digging into the architecture of ASP.NET so that you can understand how it all fits together.

Using Visual Studio

Before we go into the architecture of ASP.NET, it's a good idea to create a simple ASP.NET application that can be used to demonstrate its architectural principles. This application is a multiple-tier application that consumes a back-end web service to expose a stock quote to the user. The examples in this chapter are built using Visual Web Developer Express (VWDE), downloadable for free from Microsoft at `http://msdn.microsoft.com/vstudio/express/vwd/`. (For those who are not familiar with Visual Studio, we will look into the different versions available in a little more detail later in this chapter.) This section assumes that you aren't familiar with building ASP.NET applications, so we will follow the process in small steps.

Creating the Application

VWDE enables you to create web site applications that run on either the Visual Studio 2005 Web Server (also known as Cassini), which comes with Visual Studio, or from Internet Information Services (IIS). The first example will use Cassini.

■ **Note** When running Visual Studio in Windows Vista, if you want to debug your application you should run it as an administrator.

To begin creating the application, launch Visual Studio or VWDE, select the File menu, and then select New Web Site (see Figure 2-1).

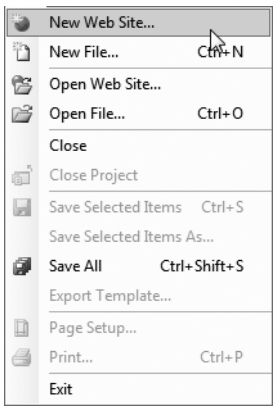

Figure 2-1. *Creating a new web site*

The New Web Site dialog box appears, in which you can select the type of web site application that you want to create. The available default options are as follows:

ASP.NET Web Site: This is a web site that uses ASP.NET to execute. It comes with a sample file called `Default.aspx`, which is an ASP.NET Web Forms application. (This is the type of web site application you'll create in this chapter; web forms are discussed in detail in Chapter 3.)

ASP.NET Web Service: A web service is a device- and platform-agnostic implementation of business logic. It does not have a UI, but instead is called using an XML vocabulary called Simple Object Access Protocol (SOAP). Web services are the underpinnings of the Service-Oriented Architecture (SOA) and methodologies of the Web. You'll be looking into this important technology in Chapter 5.

Personal Web Site Starter Kit: This kit includes a template for everything that you can use to create a simple personal-use web site that hosts pages, pictures, and more.

Empty Web Site: This option creates the stubs that you need to build a web site, including all the required ASP.NET configuration files, but no sample pages or web forms.

The Location drop-down list is important in this dialog box. As you can see in Figure 2-2, the options available are File System, HTTP, and FTP.

The *File System* option creates a directory on your hard drive for your web application and treats that directory like a web site. This option uses the embedded Visual Studio 2005 web server to run your application (that is to say, it uses Cassini as the web server rather than a full web server, easing administration and debugging).

The *HTTP* option is used to connect to an IIS instance. If you choose to create your web application on a remote server, that server will need to have IIS installed and running, as well as Front Page (Server) Extensions (FPE). You may also need to administer IIS in some cases to enable remote debugging (be sure to see an IIS administrator for help with this!). To run your new web application under IIS on your local machine, make sure IIS is installed and running (FPE will have been installed for you when you installed Visual Studio), and then specify `http://localhost` as the server. This will create a new directory under `C:\Inetpub\wwwroot` for your application, and IIS will serve it from there.

The *FTP* option enables you to use FTP to connect to a server. This server can be running IIS or another web server. However, to use ASP.NET applications as opposed to simple HTML pages, the server must be running IIS. If you use an alternative server, such as Apache, you can still create a blank web site and standard HTML pages by using VWDE and deploy those pages to Apache by using the FTP option.

In the following section, you'll learn about the integrated development environment (IDE) and use it to edit your web site application. But first, you need to create a new web site on your file system. To do this, select File System from the Location drop-down list (as shown in Figure 2-2) and select your preferred language. The examples in this book will be using C#, but if you prefer Visual Basic, converting between the two is fairly straightforward. Next, set a location for where you want the web site to be stored, as well as the desired name for your site. For example, Figure 2-2 shows the path as `C:\WebNextBook\Chapter2Example1`, so this will create a new web site called Chapter2-Example1 in the `C:\WebNextBook` directory. When you click OK, Visual Studio will create the web site and launch the IDE, which enables you to create and edit your application's various web form pages and support files.

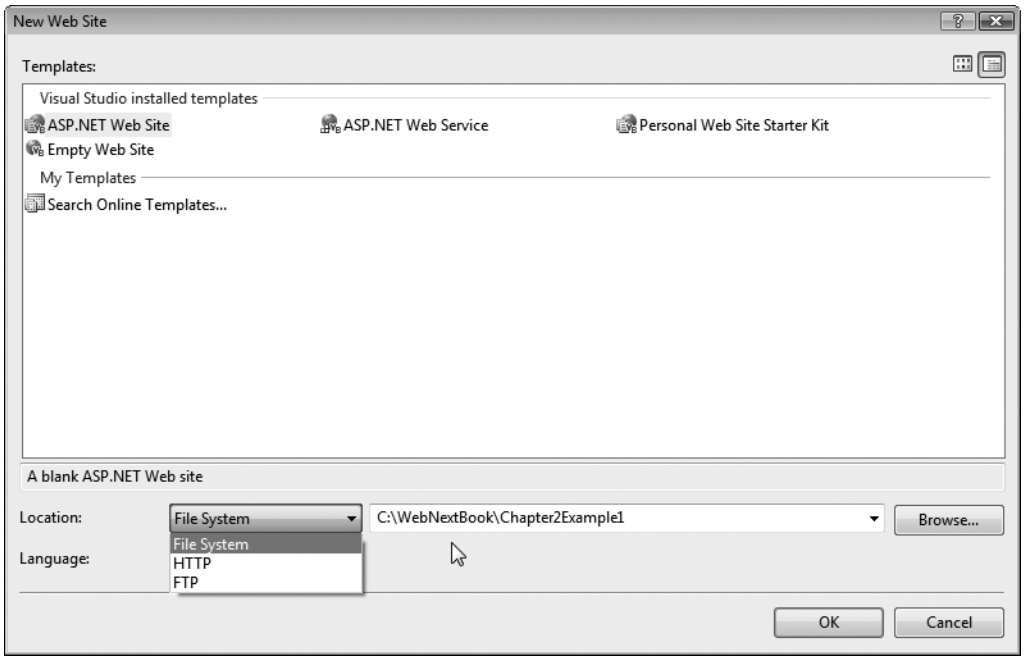

Figure 2-2. *New Web Site dialog box*

Exploring the IDE

Once you've created the web site, the IDE will launch and you will be able to edit your new web site, as shown in Figure 2-3.

VWDE provides you with a fully featured IDE that enables you to edit and maintain your web application. The IDE is divided into several distinct areas; we'll take a brief tour of them now.

The Toolbox

On the left side of the screen is the Toolbox, which enables you to drag and drop controls onto your web application. Controls can be either ASP.NET server controls, which wrap basic HTML controls but provide for a greatly enhanced programming interface; or standard controls, which represent basic HTML controls, such as buttons, input text controls, and lists, but without the enhanced programming interface (their programming interface mimics the basic control itself). Both types of controls are rendered into raw HTML when the page is compiled, which occurs either beforehand if you like, or when the page is first accessed on the server.

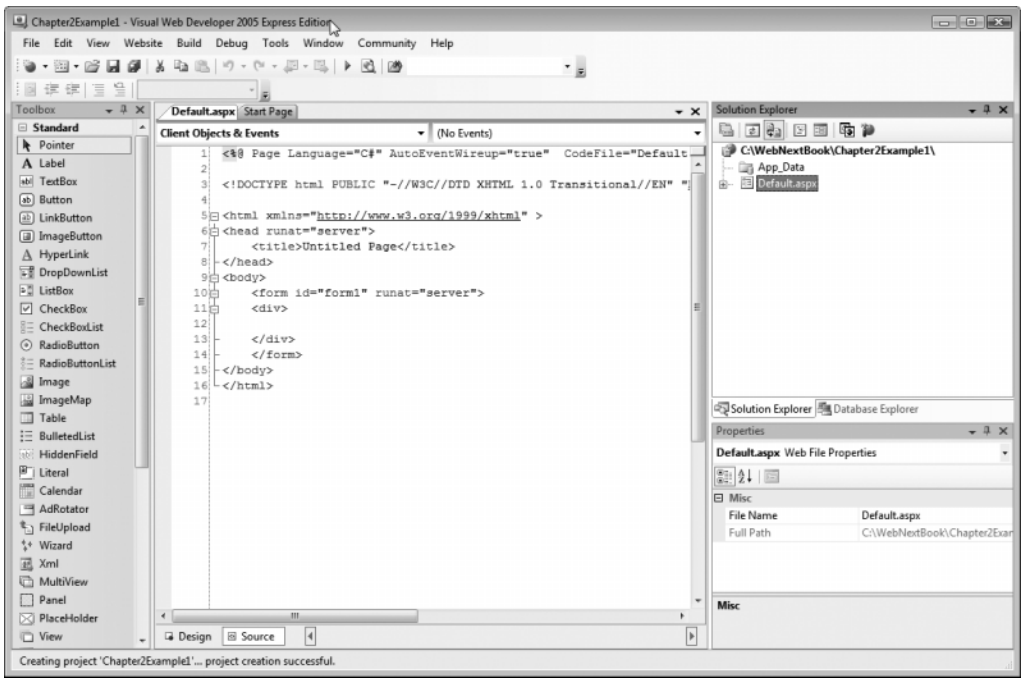

Figure 2-3. *The VWDE IDE*

Figure 2-4 shows the Toolbox with each group of controls hidden in collapsed tree control nodes commonly referred to as "tabs" (even though this isn't a Tab control!).

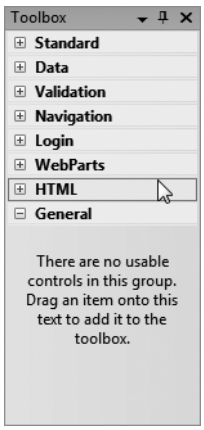

Figure 2-4. *The Toolbox*

The Toolbox gathers all your controls into logical groupings, accessed by tabs. Following is the default set of tabs:

Standard: Contains the standard suite of ASP.NET server controls

Data: Contains the suite of ASP.NET server controls that can be bound to data

Validation: Contains the suite of ASP.NET server controls that can be used to validate user input

Navigation: Contains the suite of ASP.NET server controls that can be used to build navigation on a web site

Login: Contains the ASP.NET server controls for login and user management

WebParts: Contains server controls used to build SharePoint WebParts applications

HTML: Contains standard, generic HTML controls

General: Empty by default, but can be used as a scratch pad for your favorite controls

You aren't limited to this group of tabs or the controls that they contain. You can right-click the Toolbox to add or remove tabs and to add or remove controls from a tab.

■ **Note** Did you know that you can put code snippets in the Toolbox, too? Simply highlight them, and then drag and drop them onto the Toolbox!

It's pretty common for people to use a third-party or add-on control that isn't part of the default control suite. To add a new tab, right-click a blank area in the Toolbox and select Add Tab to create a new tab for your control (see Figure 2-5). A new tab with a text box placeholder appears. Type a name for the new tab (e.g., My Controls) and press Enter. You'll now have a new, empty tab on which you can place your controls.

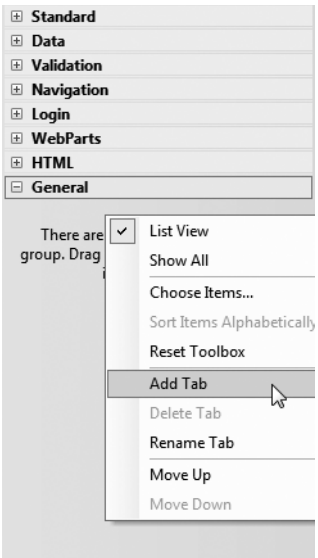

Figure 2-5. *Adding a new tab*

The next step is to add controls to the new tab. To do this, right-click in the blank area on the tab and select Choose Items from the context menu (see Figure 2-6).

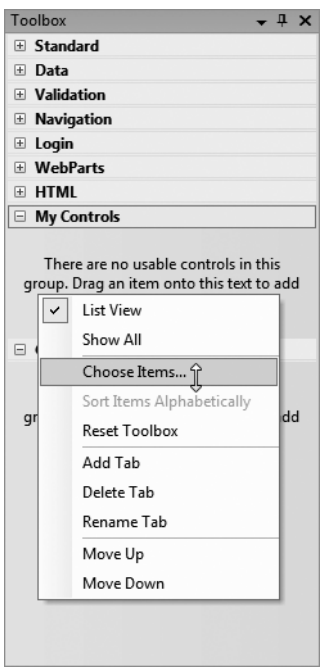

Figure 2-6. *Adding a new control to your tab*