IN THIS CHAPTER

- Getting to know web applications
- Conferring with the client
- Working with web servers

Chapter 1

Looking at How ASP.NET Works with C#

At one time, few people knew much about the Internet. However, that has all changed today. Hardly an application appears anywhere that doesn't rely on the Internet in some way. Programmers of all stripes use the web for research and communication. Providers use it for product updates and documentation. It is everywhere.

All the more reason to know how to code in the web environment. The problem is that so many so-called frameworks for development exist that it's nearly impossible to decide which to use with a reasonable methodology. You almost have to draw straws.

If you're working in a Microsoft environment and writing a simple program, it's usually best to use ASP.NET. Why? Sempf's Fourth Law: Simplicity above all. ASP.NET is a straightforward platform for web creation.

ASP.NET has its share of problems, most of which involve writing Google (or some other really big complicated program). You probably aren't writing Google, so don't worry about it. If your site gets famous, you can get some venture capital and rewrite it into some custom framework. For now, just get your site written.

That's what ASP.NET enables you to do — get the job done. With ASP.NET, you can write a good website quickly, one that can be hosted just about anywhere. No one can ask for much more than that. This chapter delves into the details of using ASP.NET with C#.

Breaking Down Web Applications

A web application is a computer program that uses a very light client interface communicating with a server over the Internet in a stateless manner. *Stateless* means that the computer browsing the site and the server providing the site don't maintain a connection. You can see this process at a high level in Figure 1-1.

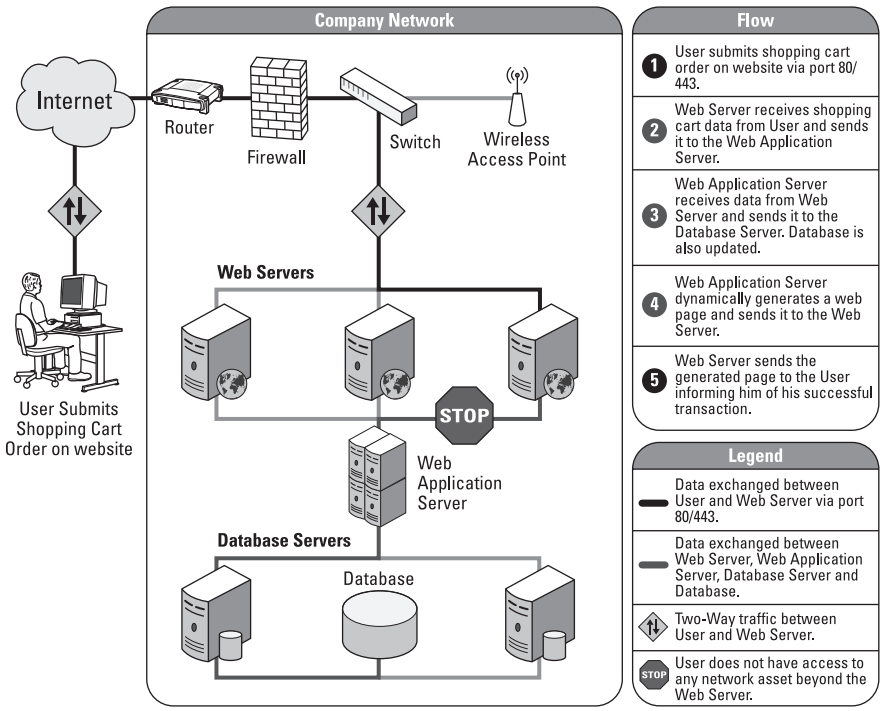

FIGURE 1-1: The web application process.

The client requests a document from the server, and the server sends the document when it gets around to it. The document, usually a combination of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript, contains standardized elements, which make up the interface of the application.

Why do you need a web application at all? If you just have a set of documents on the server and the client is just requesting one after another, what's the point of that? Of course, getting a document is a good thing, but where does the application part come in?

The answer is within web applications. Web applications are powerful because the server can construct the document on the fly whenever it receives a request. This server process is what makes web applications work (see Figure 1-2).

The light client interface — called the *web browser* or just the *browser* — provides some important functionality. It handles the user interface and can do things independently of the server.

The server is largely where it's at, though. Because the server can remember things from moment to moment, and because the client is assumed to forget everything from page to page, the server is where the long-running communications all reside.

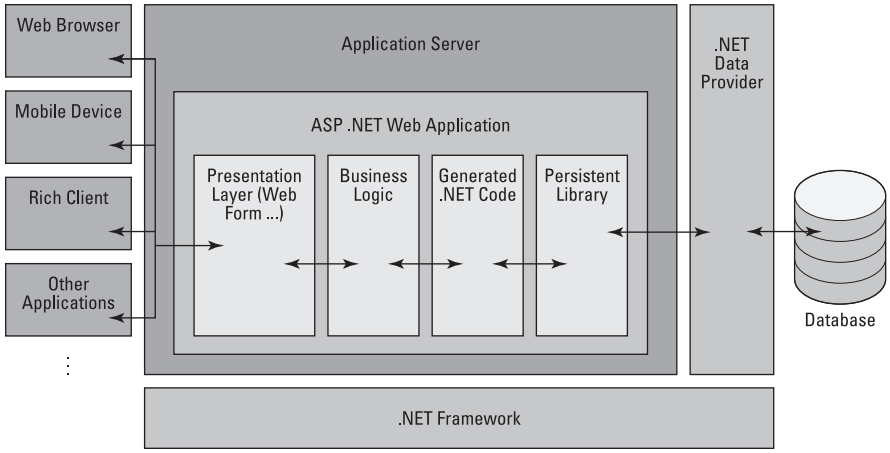

FIGURE 1-2: ASP.NET deconstructed.

ASP.NET is a library of controls and processing logic that makes constructing the server side of things much easier. You construct "server pages," but they end up transformed into the documents that the client uses to show you the user interface. ASP.NET is classified as a Web Application Framework, not a programming language.

The programming language, for your purpose, is C#. However, you can code for ASP.NET in any .NET language — it is just another set of tools in the framework.

C# is the language that you use to tell the server how to organize the ASP.NET pieces and send them to the client. The clients will never see any C# code; they won't even know that you coded it in C#.

TECHNICAL STUFF

This gets into the same discussion that the book covers several times regarding the difference between the library and the language. ASP.NET is a library of tools, and that library is language independent. You orchestrate the functionality of the parts of the library using the language.

Web development comes with a few little caveats, however. The client — which doesn't care a whit that you are coding in C# — needs some care and feeding. Web applications have no state, so you need to think a little differently. The client does a lot automatically for you in web development. The server creates some security concerns. Before you dig into the details of using C# with ASP.NET, it's important to take a tour of the details of web development and discover some things you need to keep in mind.

Questioning the Client

When you're writing a Windows Form, WPF, or console application, your platform for the client is a Windows computer (or a Linux or Mac system when using newer .NET functionality). When you're writing a web application, you have no idea what the platform for your client is. It could be a Mac, any one of the Windows operating systems, Linux, Android, iOS, or any of a huge number of other operating systems. It could be a smartphone, a tablet, or a netbook. It could be a television, Blu-ray player, gaming device, or refrigerator (really: see Figure 1-3).

FIGURE 1-3: Yes, a refrigerator.

The point is that you don't know what you're writing for, so you have a very different development experience for a web browser than you do for a Windows client. You don't know what size the screen is, you don't know what language the machine is set to, you don't know how fast the connection to the Internet is. You know nothing about your client.

You have to question the client. You can't assume much about the browser, so you have to make certain design decisions differently than you usually would.

You can depend on the browser to do some things for you that you might be used to having to do for yourself. The programming details are explained in Chapter 4 of this minibook, but you need to get a good overview now.

First, the client browser has a built-in scripting system called JavaScript, JScript, or possibly ECMAScript (these language dialects are essentially the same but have some differences). Second, the browser can tell you some subset of details about itself, its host machine, and the user that is using it.

Scripting the client

By now you get the idea that communication with the browser occurs via the network, and that the client and browser are disconnected. Although that is true, the client isn't completely out in the fog when it comes to interacting with the user.

You can send a script along with your HTML document. This script references objects on the screen — just as your C# code would — and makes things happen for the user.

Usually this script is in the JavaScript language. Why not C#? Well, C# isn't a scripting language. C# needs to be compiled, and JavaScript isn't compiled. It just sits there, in text, waiting for the browser to run it.

WARNING

The fact that it is in text means that you have to be careful not to put secure information in a script. Right-click any web page in your browser and click View Source to see that page's script code.

A lot of the script that your pages need will be generated by ASP.NET. This book is about C#, not JavaScript, so it focuses on the server features, not the client features. For now, you should just know that "things are happening" on the client side, however.

Getting information back from the client

When a browser makes a request to a server, a lot of useful information about the client gets sent along with the page name that is requested. You can reference this in the backwardly named `ServerVariables` collection.

This is about as good as learning about the client gets. These details will, however, help you get the most out of your communication with the client. Here are some examples of the values available:

- `AUTH_USER`: The logged-in user.
- `REMOTE_USER`: The same as `AUTH_USER`.
- `CONTENT_LENGTH`: Size of the request. This is useful if a file is uploaded.
- `HTTP_USER_AGENT`: Which browser the client is using.
- `REMOTE_ADDR`: The IP number of the client computer.
- `QUERY_STRING`: All the stuff after the question mark in the request. Check out the Address bar of a browser in many web applications to see what this term means.
- `ALL_HTTP`: Every *header variable* (request details made available to the server) sent by the client.

In total, the header usually contains 63 variables. The most commonly accessed of them are in the `System.Web.HttpRequest` class. You'll also find the collection of `ServerVariables` there. You can find out more on getting to that class in Chapter 4 of this minibook. Search MSDN for ServerVariables to get the complete list.

Understanding the weaknesses of the browser

As you can imagine, the browser-as-client model has a few weaknesses. At this point in the book, you should know that the more layers you stack on, the more problems you'll encounter. Not even letting the programmer know what computer the user has is a problem as well. Here are a few of these weaknesses:

- **Changing window sizes:** The most basic difficulty in using a browser as a client is changing window sizes. In a Windows program, you have at least some control over how large the window is. If it gets too small, you can change it programmatically. In a web application, however, you have very little control over the window size. For all you know, the user could be using a cellphone, right? That's a small window!

Because of this, every form you develop using ASP.NET has to be size-agnostic to the best of your ability. Especially when your form is destined for the public Internet, you just can't make assumptions about the size of the user's screen.

- **Sporadic communication:** When the client wants something from the server, the client requests it from the server. It is the server's responsibility to get the client what it wants and then wait. And wait. And wait some more.

  Even if the server wants an acknowledgement from the client, it might never get one. The user might have closed the browser.

  Communication from the client to the server is sporadic. Secure transactions are nearly impossible because the server can't be sure that the client will communicate any information in return. Because of the loosely coupled nature of the Internet, you just can't make any assumptions about communication time.

- **Distrusted input:** Browser documents are sent as a package of text and images (and sounds and fonts and Flash movies, if needed) to the client from the server. The requests are sent back to the server from the client in a similar way: textually.

  With all this text, you would think that some clients would find a way to fake a request. Oh wait, they have. Lots of them. And it is easy, and free.

  Look at Fiddler (`http://www.fiddlertool.com`), a free tool that lets you completely alter the requests sent from a browser (see Figure 1-4). Fiddler gives you the capability to view a request (including the results from a form, even a login form), alter the text directly, and then send it on its merry way.

  Security risk? You betcha. You need to distrust every character sent to you from the client. Validate everything in the server code. Chapters 2 and 3 of this minibook discuss this issue.

- **Random implementations:** When you're writing a WPF application in C#, you know that the application will be running on Windows. You might not know which edition, but you have a good idea about how it will work. When you write an ASP.NET system, on the other hand, you know that the client might or might not follow a set of standards, but that's about it. You don't know how the browser will behave.

  The more significant impact of this is in positioning, as it turns out. The technology that specified most of the positioning and styling in browsers, called Cascading Style Sheets (CSS), is rife with misimplementation.

  That means that not only do you not know how your application will behave out there, but also that it might not look right, either! Yea for the web! Seriously, scripting is another piece of the puzzle that isn't the same across browsers. ECMAScript is another standard, and it is implemented differently by different browser companies.

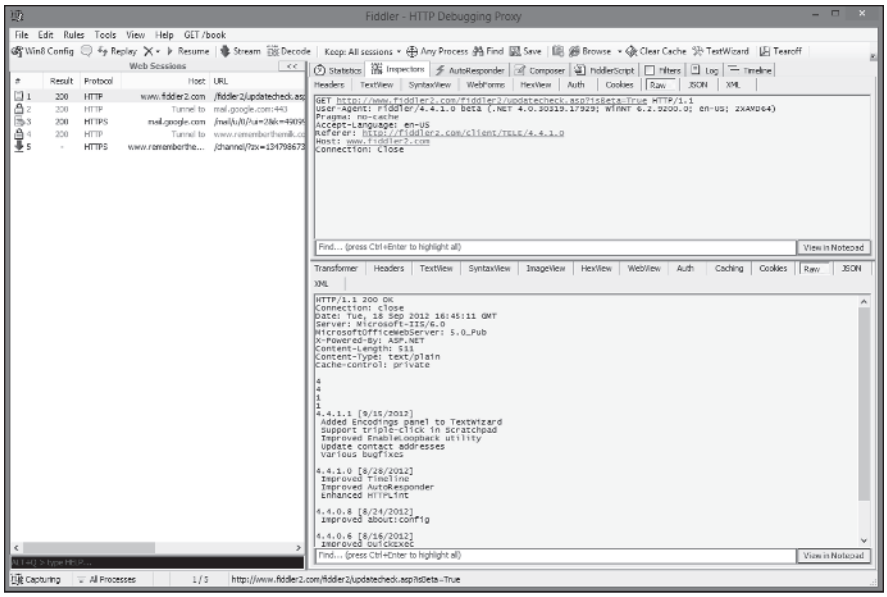

FIGURE 1-4: Fiddler.

The only way around all this is to test, test, test. If all else fails, take the simplest road. And don't get tremendously picky about a pixel here and there.

Dealing with Web Servers

The other side of this client/server equation is the server. For those of you writing in ASP.NET, the server will be Internet Information Server (IIS) most of the time.

TIP

Other servers also implement ASP.NET in one fashion or another, and you may see them more often because Microsoft is now promoting ASP.NET on other systems. For example, you can see how to configure a Linux system (with Nginx) to use ASP.NET at `https://docs.microsoft.com/en-us/aspnet/core/publishing/linuxproduction`. In fact, you don't absolutely have to have IIS; you can use other products, such as Apache (see `https://wiki.archlinux.org/index.php/ASP.NET_with_Apache` for details).

The role of the web server is to accept requests from clients, do whatever processing is required, and then pass back a browser-viewable page. This means that the ASP.NET code you write will be turned into HTML and JavaScript by the web server. You have less control over what is happening than you might think.

Getting a PostBack (Hint: It's not a returned package)

A *PostBack* isn't a returned package from the post office. It is how ASP.NET handles communication between the client and the server so that you don't have to.

The first time users request a page in ASP.NET in a given session, they usually type the URL into the address bar, or click a link like `http://blog.johnmuellerbooks.com`. The next page loaded, however, is a carefully controlled communication with the server, called a PostBack.

A PostBack is a JavaScript function used in place of the built-in POST function to send information about the data on the page back to the server. It looks like this to the client:

```
javascript:WebForm_DoPostBackWithOptions(new WebForm_PostBackOptions(&quot;ctl00
   $cphAdmin$btnSave&quot;, &quot;&quot;, true, &quot;&quot;, &quot;&quot;,
   false, false))
```

Quite a change from a URL, huh? To understand why things work as they do, you need to go back in time.

Looking back to how things used to be

Back in the day, web servers supported `GET`s and `POST`s. What's more, they still support `GET`s and `POST`s.

A `GET` is a request for a page, using just the URL. You can send data in the URI (after a question mark in the link of text) but that's all you are sending — nothing from the page itself goes back. A `GET` request looks like this:

```
GET /fiddler2/updatecheck.asp?isBeta=False HTTP/1.1
User-Agent: Fiddler/2.2.4.6 (.NET 2.0.50727.4918; Microsoft Windows NT
   6.1.7100.0)
Pragma: no-cache
Host: www.fiddler2.com
Connection: Close
```

A POST sends values from a form on the page. The form can be invisible, but it has to be there. POSTs are a lot bigger than GETs and have a set format that is hard to navigate at times. Here is an example POST:

```
POST /feedbackAction.asp HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml, image/
   gif, image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel,
   application/vnd.ms-powerpoint, application/msword,
   application/x-shockwave-flash, */*
Referer: http://www.grovecity.com/feedback.asp
Accept-Language: en-US
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64;
   Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR
   3.0.30729; Media Center PC 6.0; .NET CLR 1.1.4322; InfoPath.2;
   OfficeLiveConnector.1.3; OfficeLivePatch.0.0)
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Host: www.grovecity.com
Content-Length: 69
Connection: Keep-Alive
Pragma: no-cache
Cookie: ASPSESSIONIDAATQCRCA=MJEFOIFACGPONMOBADPIDLKH

user=Bill&email=bill@sempf.net&subject=Test&comment=This+is+a+POST%21
```

All the header variables are there, and then the contents of the form are all linked together at the end of the request.

From the past to the PostBack

Having to handle the header had a lot of problems when it came to the sophisticated web forms that ASP.NET provides, so Microsoft used JavaScript to short-circuit the process. It created a special method that would be on every page that ASP.NET generates. This method formats the information in order to better manage the communication with the server. A PostBack looks like this:

```
POST /admin/Pages/Add_entry.aspx?id=e387aab8-1292-4c75-985f-5f8e5db3089a
   HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml, image/
   gif, image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel,
   application/vnd.ms-powerpoint, application/msword,
   application/x-shockwave-flash, */*
Referer: http://www.sempf.net/admin/Pages/Add_entry.aspx?id=e387ccb8-1292-4c75-
   985f-5f8e5db3089a
Accept-Language: en-US
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; WOW64;
   Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR
   3.0.30729; Media Center PC 6.0; .NET CLR 1.1.4322; InfoPath.2;
   OfficeLiveConnector.1.3; OfficeLivePatch.0.0)
Content-Type: multipart/form-data; boundary=---------------------------
   7d922c323b0016
Accept-Encoding: gzip, deflate
Host: www.sempf.net
Content-Length: 9399
Connection: Keep-Alive
Pragma: no-cache

-----------------------------7d922c323b0016
Content-Disposition: form-data; name="__LASTFOCUS"


-----------------------------7d922c323b0016
Content-Disposition: form-data; name="ctl00$cphAdmin$txtTitle"

C# 7.0 at CONDG
-----------------------------7d922c323b0016
Content-Disposition: form-data; name="ctl00$cphAdmin$ddlAuthor"

Admin
-----------------------------7d922c323b0016
Content-Disposition: form-data; name="ctl00$cphAdmin$txtDate"

2017-09-08 22:21
-----------------------------7d922c323b0016
Content-Disposition: form-data; name="ctl00$cphAdmin$txtContent$TinyMCE1$txtCont
   ent"

<p>The sky is blue!</p>
-----------------------------7d922c323b0016
Content-Disposition: form-data; name="ctl00$cphAdmin$txtUploadImage";
   filename=""
Content-Type: application/octet-stream


-----------------------------7d922c323b0016
Content-Disposition: form-data; name="ctl00$cphAdmin$txtUploadFile"; filename=""
Content-Type: application/octet-stream


-----------------------------7d922c323b0016
Content-Disposition: form-data; name="ctl00$cphAdmin$txtSlug"

C-40-at-CONDG
-----------------------------7d922c323b0016
```

Notice the nice layout of the data, with the form field name, and the data and everything all nice to read? Fortunately, you don't have to worry about any of this. ASP.NET gets it for you.

When you create an event handler for an object — a button, for instance — a PostBack will be used for that communication. This is how ASP.NET changes your interface with the server and client. It makes web programming look like Windows programming, at least as far as events are concerned.

It's a matter of state

The other major piece of the puzzle in web server usage is state, which means that the server should know the state of the application at any given time. Web servers don't do a very good job of remembering state because of one of the problems with web browsers: inconsistency of communications. Because the server doesn't know from one minute to the next whether you're going to send anything, it can't depend on getting it.

For this reason, the server tries to remember certain things about your session by putting values in the form that it can use later. The server uses a *ViewState* value for that purpose. The `ViewState` is an encrypted string that the server can use to remember who you are from POST to POST.

Even with these values, maintaining session state has its problems. For instance, in a secure application, you might need to maintain a transaction for a database function. Doing so is nearly impossible in a web application, because you can't be sure that you will get the return acknowledgment. Nearly all the data processing therefore happens on the server. Learn more about data processing in Online Chapter 3, which you can find by going to `www.dummies.com`, searching this book's title, and locating the Downloads tab on the page that appears.