PART I

Web Application
Development with
Active Server Pages
CHAPTER 1

Introduction to Web Applications
This book will examine how to build Web applications using Microsoft’s SQL Server 2000 as a back-end. First, we need to embark on a journey to understand what we mean by “Web application,” and how we can use Microsoft technology to build Web applications. We will examine such topics as HTML, IIS, ASP, and n-tier development in this introductory chapter. This chapter assumes you are familiar with HTML; if not, you can refer to Appendix E.

WHAT ARE WEB APPLICATIONS?

Web applications are applications that run over the Web. If that sounds redundant, think about this: many Web sites are simply an online catalog, sometimes called “brochureware.” They are just pages of pictures and descriptions. When we talk about building applications, we are talking about building sites that do something. They allow you to enter information, and they intelligently respond to your requests. If Web applications had never been developed, we would just have Web pages that would be no more interactive than the pages of a catalog or magazine.

Web applications include all the e-commerce sites that have become the rage. These sites allow you to browse the catalog, but you can also place items in your shopping basket, choose a shipping option, and then pay for the items, all without ever picking up the phone. Many sites offer real-time inventories, so you know if the items you want are in stock. Other applications include many business-to-business applications. For example, Company A may track demographic data on consumers. Other companies can log on to Company A’s Web site and run reports to try to identify markets with their target demographic. This means these sites must have real-time access to a database and the ability not only to read the data, but also to build a page on the fly to reflect the values in the database. Building pages on the fly is at the heart of Web applications.
To build Web applications, we first need to understand how Web servers and static Web pages work. We will then move into how to build applications that are driven by the data in our SQL Server databases.

Web Servers and HTML

At their heart, Web servers can be very simple software. A client computer connected to the network requests a particular page from a particular Web server. On a Web server, a page is just a file stored on a physical drive. The server locates the page on some computer and sends a copy of the page (file) to the requesting client computer, as can be seen in Figure 1-1.

The request from the client to the Web server and the response from the server back to the client are done with a standard protocol.
of the Internet: HTTP (HyperText Transfer Protocol). HTTP takes the client request and packages it in a format that a standard Web server can understand. When the server responds, the page information is sent back to the client using the same HTTP protocol. HTTP rides on top of a well-known networking protocol, IP (Internet Protocol).

So, at the heart of their functionality, Web servers just send files to the requesting browser. And this is one of the important things to understand about static HTML (HyperText Markup Language) pages: the Web server doesn’t process them in any way. In other words, Web servers are “dumb” concerning static Web pages. However, most Web servers do have the ability to perform processing before pages are sent to the client. This processing of the page requires some form of intelligence on the Web server and is what most of this book is about.

First, however, we need to understand what happens with simple, static HTML files. If the files are sent down to the client with no processing, how are they displayed? This is the job of the browser. The two most popular browsers are Microsoft’s Internet Explorer (IE) and Netscape’s Navigator, although there are a number of others available, such as Opera, WebTV, and AOL (America OnLine). Browsers are built with the express purpose of taking the HTML and interpreting it to display the page on the screen.

HTML was originally developed so that a document’s content could be displayed on any hardware platform. This was to be achieved by marking content by meaning, and letting the browser handle display issues as they may be appropriate for that platform. If you use an application like Microsoft Word to create a document, Word stores that document in a binary format to preserve the text formatting. This binary format is not necessarily compatible with other applications on other platforms. Therefore, the creators of HTML wanted a format that could preserve text formatting but that was also portable to any platform. To do this meant that HTML needed to be all text, with no binary information.
NOTE: Many people get some credit for the concept of an HTML-like language, but Tim Berners-Lee and Robert Caillau actually created HTML. Berners-Lee and Caillau created HTML as a way to provide a platform-independent, linked information system on the various computers at CERN (European Organization for Nuclear Research), a physics research center in Geneva where they worked. In 1991, CERN created the Web that has grown into what we know today.

To accomplish this all-text requirement, HTML stores the document’s formatting in tags. These tags are not processed by the Web server, but instead are interpreted by the browser on the client computer. The browsers are the platform-specific applications, but they are all written to interpret HTML in roughly the same way, based on the standards set forth by the World Wide Web Consortium in Boston, Massachusetts (http://www.w3.org). This means that each browser will interpret the <H1> tag as a command to render the text in a larger and bolder font.

Given that the purpose of HTML was simply to display text on the screen, preserve the formatting (boldfacing, italicizing, underlining, and so on) from one platform to another, and allow for linked information, I should point out something many people never think about: HTML is not a programming language. A programming language has certain elements, three of which are variables, loops, and decision-making structures (such as If statements). HTML does not have any of these three elements. HTML is a computer language, but its purpose is to store text-formatting information. It is not a programming language and therefore cannot perform the functions we would expect in an application. As you will see in later in this chapter, there are ways to add in some client-side script that allows our Web page to perform some processing. Realize, however, that this client-side script is going to be JavaScript or VBScript, and not HTML itself. Running code on the client requires a mix of HTML and a scripting language. Come to think of it, running code on the server
with Active Server Pages requires HTML and a scripting language. More on all of this in Chapter 2.

**NOTE:** Throughout this book, I will refer to JavaScript. Understand that JavaScript is a language created by Netscape. Microsoft created their version of it, called JScript. JavaScript and JScript are similar but not identical. In an attempt to address these differences, a third language called ECMAScript was developed, is endorsed by a standards body, and is supported on both platforms. Throughout this book, I will refer to the language as “JavaScript.” Please realize that this is just a generic term and could be relating to JavaScript, JScript, or ECMAScript.

One thing that is important to understand is that the different browsers do not always render the same code in exactly the same way. First of all, computers may be in different resolutions. A page that looks good on an 800×600 screen may not appear as desired on a screen set to 640×480. In addition, the color depth may be set differently on various machines, which can lead to colors and graphics not being displayed as expected. Also, the default font on different systems can vary, which means that the page may display differently from one machine to another.

Not only can the same code render slightly differently from one browser to the next, but browser manufacturers also add their own proprietary extensions to standard HTML. This can result in code that only works on one browser and not another. For example, the `<BGSOUND>` tag is a tag added by Microsoft to play a background sound for a Web page. This tag only works in Internet Explorer. Instead of adding tags, the browser vendors sometimes just add their own attributes to existing tags.

This brings us to another important point about how browsers work: by design, they ignore any tags or attributes they do not understand. Take a look at the following code snippet:
Welcome to Northwind

This text is formatted with the Craig style

Figure 1-2. The code snippet with the <CRAIG> tag
but HTML does not enforce good, structured coding. There are some online validation tools available, such as the World Wide Web Consortium’s HTML Validation Server (http://validator.w3.org) and Netscape’s Web Site Garage (http://Websitegarage.netscape.com).

Another important point about HTML is that once it is rendered in the browser, it cannot be updated without performing a refresh, which means going back to the server and getting the page again. In other words, the text on the page can’t change once it has been rendered. This is more evidence of the static nature of HTML. There are ways around this: the fields on a form can be updated with some client-side code, or you can turn sections of the code into programmable objects with Dynamic HTML (DHTML). However, both of these require some client-side scripting that was not originally available in HTML, and that is still not supported by all browsers.

**Client-side Scripting**

Let’s take a moment to examine client-side scripting. Remember that HTML is not a programming language and therefore cannot natively perform any processing. We can, however, include some scripting on the client that allows us to perform some action. There are a couple of considerations when you look at client-side scripting:

- Can the client browser support client-side scripting at all?
- If so, what languages does it support?

The rule of thumb here is pretty simple: VBScript is known by far more developers, is generally considered easier to program, and only runs in Internet Explorer. JavaScript works in both IE and Navigator. Obviously, your broader reach will be with JavaScript.

**NOTE:** Do not confuse JavaScript with Java Applets. Java Applets are compiled components that get downloaded and executed in a Java Virtual Machine (VM) in the client browser. JavaScript and VBScript are not compiled, but are interpreted by the client browser.
When I teach people about HTML, I cover how to create HTML forms. Forms, of course, allow the user to enter some information and send it to the server, where we presumably perform some magic on it, such as adding it to a database. A question that almost always comes up, if the students are paying attention, is: “How can I validate the form input?” The answer with straight HTML is simple: “You can’t.” Standard HTML has no way to validate data in a form field. That means that if you are asking for the users’ age, and they type “abc,” they will send “abc” to the server in the age field, and you will have to perform your validation on the server. This isn’t a bad thing, but consider what the users go through: they fill in the form, select the Submit button, and wait. The data goes across the Internet to your server, and then you perform the data validation. If a field value was incorrect, you have to build a page that tells them of the error and send that page back. Now the users get an error message and must select the Back button in the browser, return to the original form, and change the data. This whole process can take a number of seconds to perform.

With client-side scripting, however, we could validate some of the data at the client before it is sent to the server. This gives users immediate feedback on the data and keeps them on the same page so there is no need to select the Back button after receiving an error message. Not all data can be validated on the client, obviously. We may need access to a database to validate certain values, and we do not usually have direct database access from the client, although direct database access from the client is possible, as we will see in Chapter 12.

To see how client-side scripting can be used to validate form input, the following code shows an oversimplified Web page with a two-field form. If the data entered in both fields is valid, we will send it to the server. If not, we will notify the users of the problem and allow them to fix it. The checks will be simple, too: in the e-mail field, we will just make sure there is an “@” symbol somewhere in the field. In the date of birth field, we will simply make sure what they type is a valid date. The code is then responsible for submitting the form. To see the form in action, see Figure 1-3. Here, we do not actually submit the form, because we don’t yet have anything on the
server to handle the data. Notice too that the code is written in VBScript, which means this example will only work in IE.

<HTML>
<H1>Form Validation Example</H1>
<FORM name="frmSignUp" method="post">
    Enter your e-mail address: <INPUT name="email"><br>
    Enter your date of birth: <INPUT name="dob"><br>
    <INPUT type="button" name="cmdSubmit" value="Submit">
</FORM>
<SCRIPT language="VBScript">
    <!--
    Sub cmdSubmit_onClick()
        If InStr(document.frmSignUp.email.value,"@")=0 Then
            alert "E-mail address must contain the @ symbol."
        Else
            If IsDate(document.frmSignUp.dob.value) then
                alert "Data is fine, form will be submitted"
            Else
                alert "Date of birth is not a valid date format." & " Please re-enter."
            End If
        End If
    End Sub
    -->
</SCRIPT>
</HTML>

Clients can also be enhanced using other techniques. For example, we have ActiveX Controls and Java Applets. While these technologies are quite different, they have a similar end result: the client interface can do things you can’t normally do with HTML. For now, the biggest difference between the two is that ActiveX Controls only run in IE. Java Applets work in both IE and Navigator, although not all Applets work equally well in both browsers.

ActiveX Controls are popular because they pervade the Windows world and they are easy to create in Visual Basic and Visual C++, among other tools. ActiveX Controls, however, depend on the
The controls are downloaded and registered on the client machine, where they stay installed from then on. That way, when users visit the site again in the future, the control is already present on their local machine and does not have to be downloaded again. The control has all the capabilities you might associate with a Windows application: it is event driven, can access the machine’s hardware, can read/write to the hard drive, and so on. This ability to access the computer’s hard drive has led some people to worry about the security implications of ActiveX Controls. It is true that malicious ActiveX Controls could cause damage to a user’s machine. There are some measures taken to try to prevent this, such as the Authenticode technology. We will examine Authenticode in more detail when we discuss security in Chapter 15.

Figure 1-3. The client-side script validates the e-mail address before submitting the form. Here, the code catches that the e-mail address that was entered is not valid.
Java Applets can be event driven and give the user a richer interface, but they cannot access the hardware as can ActiveX Controls. This makes many people feel more secure using Java Applets. In addition, they have a broader reach because they are supported by the current versions of both IE and Navigator. There are many Java Applets available, such as stock tickers, countdown timers, and calendars.

Server-side Scripting

Now that we’ve talked about how Web servers simply send streams of static HTML to a browser to be interpreted, let’s examine the concept of Web servers that can build pages dynamically. The reasons for this are obvious: you may want to build pages that reflect data from a database, the results of a search, or time-sensitive information. Without the capabilities provided to us by Web servers that build pages on the fly, we would not have online shopping carts, inventories, or any other Web-based application.

The first Web servers that allowed us to build pages dynamically used CGI (Common Gateway Interchange). CGI is still very common today, and Microsoft’s Internet Information Services (IIS) can use CGI as well as Active Server Pages, which we will use throughout this book. CGI is not a programming language. Instead, CGI is a technology that allows us to call compiled programs or scripts on the Web server. These programs or scripts can accept user input and then build pages dynamically based on those inputs. CGI is often seen on UNIX-based Web servers, and many of the CGI programs are Perl scripts (identifiable because they typically end with “.pl”).

As previously mentioned, IIS can use CGI, but IIS uses another technology called Active Server Pages (ASP). ASP enables us to mix scripting right into the HTML page. This means that, unlike when using CGI, we are not starting a separate server process when we build a page dynamically. ASP also allows us to work with portions of the final HTML as we interlace the script directly into it. This approach has been so popular that there are now Java Server Pages that work much the same way.

Because ASPs are executed on the server, the choice of the scripting language is up to the developer. Remember that with
client-side scripting, we have to know if the browser supports a scripting language, and if so, which one(s). With server-side scripting, we are not concerned about the capabilities of the client browser; we process all the code on the server and just send straight HTML to the client machine. This means that the client browser does not have to support any scripting language at all. In fact, if the clients view the source of the document in their browser, it looks like static HTML; they do not see any of the code that was used to create the page.

ASP's allow us to have a much more complex application environment than we could get with just HTML and client-side scripting. For example, we might need to access some data, or even programs, on a mainframe. These programs could crunch a significant amount of data and send us the results. We might then put this data into a database to further manipulate it before producing the final page that will be sent to the client. Such an application is shown in Figure 1-4. The abilities to run programs on

![Figure 1-4. A complex Web application](image-url)
the mainframe, add and query data from a database, and build a page with the results are beyond the capabilities of standard HTML and client-side script. Intelligent Web servers that support server-side scripting, however, allow us to build these rich applications without a concern for the capabilities of the client browser.

If you’ve ever worked in a mainframe environment, this is all sounding very familiar. When computers were first created, they were huge machines that took up hundreds or thousands of square feet. Everything ran on that one computer, and people accessed it through dumb terminals. All these terminals could do was display information on the screen and accept user input. Around 1991, client/server began capturing the hearts and minds of the IT (information technology) world. This “power to the people” movement said that the power should be in the PCs sitting on each person’s desktop. The data was stored in a central location (the server) and moved down to the PC (the client) where it was processed and displayed. The only thing that was shared was the data, with some business logic possibly being implemented in stored procedures on the database. This was the birth of distributed processing. This client/server architecture, which we now recognize as two-tier or fat-client, had some major advantages over traditional, monolithic systems. The display could be much richer and include things like mouse support, menus, buttons, graphs, animations, colors, and so on. No longer did the users have to type arcane command names and wait to have a report print on green-bar paper. Instead, they could select a few buttons and get the results back quickly, on screen, with graphs and colors and all other sorts of bells and whistles (which we call “features” when talking to management).

Client/server took us 180 degrees from the monolithic, mainframe environment into a distributed environment. While this was a boon for users, it led to some real IT headaches. Anyone who developed client/server applications in this period remembers the horrors of distribution: DLL conflicts, INI files, and ODBC data source names. These distribution issues led to maintenance issues as well. You were very concerned with the capabilities of the clients’ machines: Did they have enough RAM? Did they have enough disk space? How would they connect to the server? Was the connection fast enough? All of these issues led IT shops to long for the days of monolithic applications with nothing more than dumb-terminal access.
Enter Web applications. Web applications took IT right back to the monolithic structure they had known and loved: all the functionality was on the Web server. The browser was nothing more than a dumb terminal. It allowed the user to enter text (via HTML forms) and it displayed text (via HTML). That was all a browser had to do, so it was in many respects a dumb terminal interface. Nothing had to be distributed to the client. As long as the clients had a browser capable of handling standard HTML, they were ideal.
candidates for using Web applications. In addition, connectivity for remote users no longer had to mean dedicated dial-up lines and modems. Remote users could now use any ISP, anywhere in the world, and connect to the application with ease. Figure 1-5 shows the cycle applications have gone through.

Before you monolithically-minded developers get too comfortable, realize that we are again moving to the distributed paradigm with technologies like Dynamic HTML (DHTML). DHTML is very browser specific at the moment, which means you are once again concerned about the client browser. In addition, some of the technology, such as Microsoft’s data binding from the client, does require certain components to exist on the client machine. These enhancements lead us right back to the reach-versus-richness debate: the richer the user interface, the less reach it has. In other words, if we add all the bells and whistles, oops, features that are available to IE, the application won’t work anymore with Netscape, Opera, and other browsers. If we keep it as just straight HTML, it will work with any browser, but won’t have as rich a user interface. There is no right answer to the reach-versus-richness debate; it all depends on your client’s needs. In fact, almost all questions in Web development can be answered with two words: “It depends.”

Before we leave this discussion and take a closer look at Microsoft’s offerings in this area, keep in mind the following points:

- Standard HTML is only for the display of data and the acceptance of user input.
- Client-side script is usually used for form validation.
- Client-side VBScript is only supported by Internet Explorer.
- Client-side JavaScript is supported by both IE and Navigator.
- Dynamic HTML is a mix of scripting and HTML that is often browser specific.
- Server-side script is code that is executed by the Web server.
- Server-side script does not rely on capabilities of the client browser.
MICROSOFT’S IIS AND ASP

Microsoft’s Web server is called Internet Information Services (IIS). IIS is built into the Windows NT and Windows 2000 operating systems. A subset of IIS, called Personal Web Server, is available for Windows 95 and Windows 98. IIS can act as a simple Web server, serving up static HTML pages like any other Web server. However, IIS has many other features that make it a popular platform on which to develop powerful Web applications. IIS enables such functionality as built-in state management, integration with the Windows 2000 security model, objects to ease Web application creation, and server-side scripting with Active Server Pages.

State management is important for developing Web applications. When you log on to a network or log in to a mainframe, there is a process that verifies you and then maintains an open session for you while you are logged in. On some systems, a period of inactivity will cause you to be logged out, which simply destroys your session on the network and removes your permissions from all network resources. This is an example of a stateful connection, in that it knows when you are and when you are not connected, and can maintain information about you while you are connected.

HTTP, on the other hand, is a stateless protocol. When a user requests a page from a Web server, the HTTP request is sent to the server, and the Web server sends the file back to the client. Once the file is sent to the client, the HTTP connection is closed. The users now have the HTTP document in their browser, so they do not care if the connection is closed. Now when the user requests the next page, the request goes to the server and the server sees it as a brand-new request. In other words, the Web server sees each request as a new request.

To understand the challenges of only having a stateless connection, imagine an online shopping cart. You are browsing around, looking at items, and find one you want to order. You choose the item and the system replies by saying that item is in your shopping cart. You now select the Check Out button, and it takes you to the checkout area. But your item is not in the shopping cart! In a stateless environment, that
is exactly what would happen. A stateless application could not remember from one page to another whether something had been ordered. HTTP, then, presents a challenge when it comes to developing applications. Applications almost always require some form of state management, so developers have to find ways to handle state in what winds up being a disconnected environment. One of the strengths about IIS is that it can automatically maintain state for us even though we are using standard HTTP. The developer doesn’t have to worry about creating cookies or passing hidden variables. We will examine how this IIS state management works in Chapter 4.

IIS is the hosting environment for the Active Server Pages engine. ASPs are files with a mix of HTML and a scripting language, and the file is processed on the server before the data is sent to the client. ASPs are a powerful way to create Web applications and can handle a number of different scripting languages. ASPs can easily call COM components that provide added functionality, including components that provide database access.

IIS provides a number of built-in objects that make building Web applications easier. These objects can be called from any scripting language used in an ASP. For example, there is an object that makes it simple to read the data sent from a client through an HTML form. Another object makes it simple to call COM components that allow us to extend the functionality of a Web application as far as it can go. We will examine the available IIS objects in Chapter 4.

When it comes to Active Server Pages, the ASP engine can handle a variety of scripting languages. In fact, any available interpreter for any language can be plugged into the engine. ASP natively supports VBScript and JavaScript. There are interpreters available for PerlScript, Python, and Rexx. This language-neutral position is made available because the Web server is handling the processing of the scripting.

When the ASP engine finishes processing all the server-side script, the result that is sent back to the client is just standard HTML. The script is mixed inline with the HTML, and the page is processed within IIS. This means that, unlike when using CGI, a separate process is not started to build a dynamic page.

ASP is often called a “compile-free” technology. This is somewhat misleading and confuses many developers. The ASPs are not compiled
by the developer. Instead, the files are merely placed into the appropriate directory on the Web server. When the client requests an ASP, it is interpreted by the ASP engine and then cached in memory in a compiled state. The page is cached for improved performance, since this means it does not have to be interpreted every time it is requested.

Caching the page in memory is a great performance-booster, but since this is a compile-free environment, how does it know when the page has changed? The simple answer is, IIS checks on every call. Say a page, called Order.asp, is cached in memory in its compiled state. When you, as the developer, make a change to Order.asp and save it, the old version is still what is cached in memory. However, when the next user calls Order.asp, IIS performs a check to see if the file on the disk has changed. Normally it has not, so it uses the version in memory. In this case, however, the file has changed, so the ASP engine goes through the interpretation process on the new file, and then stores the new compiled version in the cache. This way, the user is always getting the latest version of the ASP without you doing any more than just saving the file.

Active Server Pages are part of the IIS environment. However, a third-party product called Chili!Soft ASP (http://www.chilisoft.com) allows non-IIS servers to run ASPs. For example, a Lotus Domino or Netscape Enterprise Server could be the Web server used, but still allow ASPs to run. There is even a version for the popular Unix-based Apache Web server. This product allows ASP to extend beyond just the Microsoft platform, potentially making it a viable solution in mixed environments.

N-TIER WEB APPS

If you think back to the discussion on Web applications being similar to monolithic, mainframe applications, you’ll remember that I talked about two-tier client/server applications instead of the more powerful n-tier solutions. N-tier, or three-tier, solutions involve pulling the business logic out of the user interface. You now have three logical layers: the presentation layer, the business layer, and the data layer. The presentation layer contains just the user interface (UI) components,
such as the forms of your application. You don’t want business logic at this level; you want to have just enough code to display the data and accept user input (sound familiar?). This approach is often called a “thin client” application.

The client is thin because it doesn’t do much other than display data and accept some user input. Most of the work is done in the business layer. Here, you create a set of reusable components that implement your business rules. These components can then be used by any number of applications that need to perform the same functions.

The data layer sometimes includes just the database, and sometimes a set of components that lie between the business components and the database. This is a design decision that should be based on many factors. We will examine some of these factors later. Whether or not any components reside in this layer, the database may have some business logic implemented in stored procedures and triggers. This often leads to better performance at the cost of some ease of changing the back-end database in the future.

When we talk about components in the business or data layers, we are talking about COM components. COM is Microsoft’s technology for allowing one application to create objects out of a separately compiled piece of code. There are a number of factors necessary for a COM component, and COM does a lot more than what I’ve just mentioned. But for now, let’s just understand that COM will allow our front-end to call the business components we create.

Let’s look at an example of an n-tier application. Hospitals keep patient information according to a specific data structure. Patients have some form of Patient ID, a name, an address, one or more insurance providers, one or more physicians looking after them, and so on. Each hospital may have a completely separate set of systems in-house to handle patients. Each one of these systems would need to look at the structure of a patient in exactly the same way. So, if the Patient ID were an alphanumeric value of ten characters, each system handling patients in any way would need this definition coded into it.

Now comes the great problem of coding this way: what if the Patient
ID format were to change? With traditional systems, each application would have to be modified, tested, and placed back into production. Almost without fail, one system or another gets forgotten, and we now have systems that are out of sync with each other.

The answer to this is to build a Patient component. You build just one Patient component and store it physically in a central location. Every system that works with patients simply accesses the Patient component. The applications create objects from the component, and each object is identical in structure. If the Patient ID format needs to change, you can (in an ideal world) simply modify the component and be done. The applications using this component will automatically pick up the new structure the next time they create an object from the component.

If all of this is new to you, don’t worry; we will examine n-tier application development in more detail in Chapters 13 and 14. For now, understand this important point: n-tier development allows us to create components that enforce our business rules, and our applications just call those components. Changes in business rules can be made independent of the presentation layer, which allows our applications to handle changing business conditions more easily and more quickly.

N-tier applications sound great, and they can be great, in both traditional Windows applications (Visual Basic, Visual C++, and so on) and in Web applications. IIS allows our ASPs to call COM components that implement our business rules for us. There is no reason why our Patient component mentioned earlier could not be used from a Web front-end. There are many reasons to build our Web applications using n-tier technology, and we will examine many of those throughout this book. We will describe how the use of components can make our applications more scalable, perform better, and support heterogeneous database transactions.

With n-tier Web applications, almost everything is under the control of the IT department: the ASPs, components, and database all reside in the computer room, safely under the control of the IT department. The only piece of the application on the client’s machine is the browser. This gives IT the same control they enjoyed with
monolithic systems and means that issues such as backups, distribution, and security are maintained within a department that (hopefully) has existing policies and resources to make sure things run smoothly.

**SUMMARY**

The World Wide Web is a popular platform for developing Web applications because the Web provides a number of important benefits: access from almost anywhere in the world, no distribution needed, and control by the IT department. To build Web applications, we need some method of dynamically creating pages, and Microsoft's Internet Information Services delivers with its support of the Active Server Pages engine.

Building large, robust Web applications is possible thanks to server-side scripting. We will use ASPs to access reusable COM components and SQL Server. We will examine how to make the application more scalable and perform better by use of COM+. If you are concerned that we haven’t talked enough about databases yet, don’t worry. We need to build some simple ASPs first to make sure we know how they work, but our Web application will end up being database driven.