CHAPTER 1

Introducing AJAX

Welcome to *Foundations of ASP.NET AJAX*. This book is intended to get you up and running with the new framework from Microsoft that allows you to build Web 2.0 applications that implement AJAX functionality. If you've been working in the field of web technology, you know AJAX is hard to avoid—and even harder to implement. Microsoft has thrown its hat into the AJAX arena by doing what it does best—giving you, the developer, a framework and the tools that allow you to build highly interactive and personalized solutions that satisfy your web-based business requirements and users' experiences more quickly and easily than previously possible.

This chapter brings you up-to-date on web application technology with a brief overview of computing history from its huge mainframe origins to today's powerful desktop PCs and the global reach provided by the World Wide Web. It's the beginning of what I hope will be an enjoyable and informative ride.

Delving into the History of Web Application Technology

After the popularity of office productivity applications exploded, and as people began using these applications daily, they required even faster and more sophisticated platforms, which caused the client to continue to evolve exponentially.

It's important to note that the more sophisticated applications were *disconnected* applications. Office productivity suites, desktop-publishing applications, games, and the like were all distributed, installed, and run on the client via a fixed medium such as a floppy disk or CD-ROM. In other words, they weren't connected in any way.

The other breed of application, which was evolving much more slowly, was the *connected* application, where a graphical front end wrapped a basic, text-based communication with a back-end server for online applications such as e-mail. CompuServe was one of the largest online providers, and despite the innovative abstraction of its simple back end to make for a more user-centric, graphical experience along the lines of the heavy desktop applications, its underlying old-school model was still apparent. Remember the old Go commands? Despite the buttons on the screen that allowed a user to enter communities, these simply issued a Go <communityname> command behind the scenes on your behalf.

Although this approach was excellent and provided a rich online experience, it had to be written and maintained specifically for each platform; so for a multiplatform experience, the vendor had to write a client application for Windows, Unix, Apple, and all other operating systems and variants.

In the early 1990s, however, a huge innovation happened: the web browser.

This innovation began the slow merger of these two application types (connected and disconnected)—a merger that still continues today. We all know the web browser by now, and it is arguably the most ubiquitous application used on modern computers, displacing solitaire and the word processor for this storied achievement!

But the web browser ultimately became much more than just a new means for abstracting the textual nature of client/server network communication. It became an abstraction on top of the operating system on which applications could be written and executed (see Figure 1-1). This was, and is, important. As long as applications are written to the specification defined by that abstraction, they should be able to run anywhere without further intervention or installation on behalf of the application developer. Of course, the browser had to be present on the system, but the value proposition of having a web browser available to the operating system was extremely important and ultimately launched many well-known legal battles.

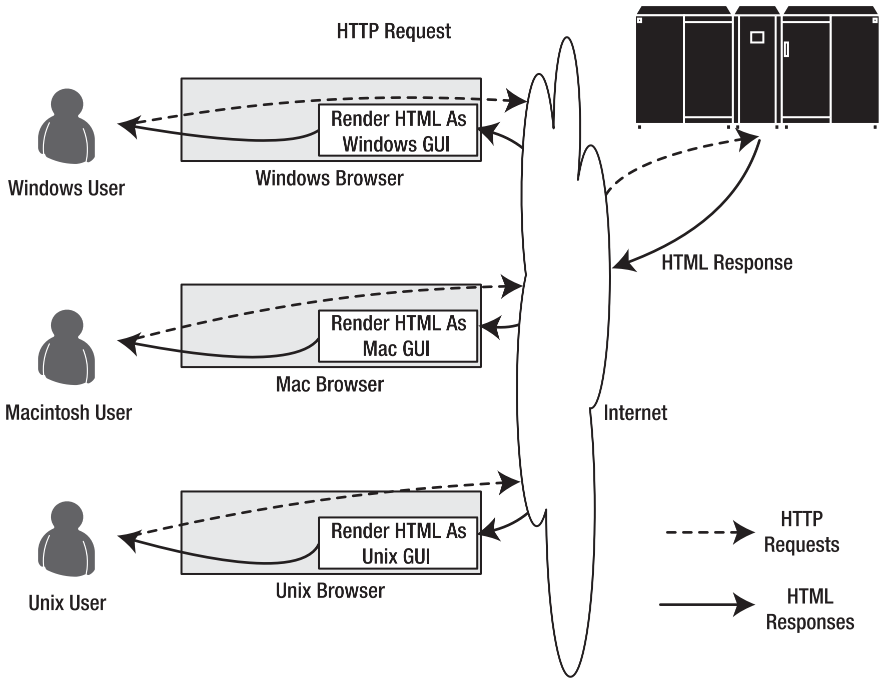


Figure 1-1. *Web browser–based request/response architecture*

Probably, the problem with this abstraction was that it was relatively simple and not originally designed or implemented for anything more complex than laying out and formatting text and graphics. I am, of course, referring to Hypertext Markup Language (HTML). This specification, implemented by a browser, meant that simple text could be placed on a web server, downloaded from a server, interpreted by a browser, and laid out in a far more pleasing way than simple green-on-black on a page, giving the user a better experience. More importantly, however, it could generate a whole new breed of application developers; all a developer had to do to create an online, connected application with a graphical experience was to generate it as HTML, and the browser would do the rest. You wouldn't need the resources of a CompuServe or an America Online to build an application that rendered the text for you! All you had to do was generate HTML, either by coding it directly or writing a server-side application (called Common Gateway Interface, usually written in the C/C++ language) that would generate it for you. Although the Internet had been around for a long time, it was just now starting to come of age.

And guess what happened? The cycle began again.

Everybody jumped on the browser bandwagon, and Common Gateway Interface (CGI) applications, running on a server and delivering content to browsers, were hot. The user experience, with the only interaction being postbacks to the server (similar to computer terminals, only prettier), soon became too limiting due to server responsiveness, huge network loads, and so on, and new technologies began to emerge to improve the user experience.

Enter Java and the applet. Java applications run on top of the Java Virtual Machine (JVM). A *Java applet* is a special kind of Java application that runs in a browser; the browser provides the JVM for the applet. In other words, the Java applet runs in a virtual machine (the JVM) on top of another virtual machine (the browser) on top of a virtual machine (the operating system) on top of a real machine (the underlying hardware). This provided a greater abstraction and introduced a new platform that developers could code to and have even richer applications running within the browser. This was important because it increased complex client-side functionality implemented in a modern, OO (object-oriented) programming language. Enhanced graphical operations (e.g., graphs), client-side processing of business rules possibly, multithreading, and so on used the same simple transport mechanisms of the Internet, but again without requiring the resources of a huge company writing their own GUI platform on which to do it. Probably, Java applets suffered from constraints; namely, to achieve a cross-platform experience, developers had to follow a lowest common denominator approach. The clearest example of this was in its support for the mouse. Apple computers supported one button, the Microsoft Windows operating system supported two, and many Unix platforms supported three. As such, Java applets could support only one button, and many Unix users found themselves two buttons short!

The Java applets run in a security sandbox and therefore cannot access local resources such as the file system or databases, and they cannot create new outbound connections to new URLs on the server (because this could be potentially dangerous). This lack of access to corporate resources led to Java spreading to the server side: server-side Java applications called *servlets* generate HTML pages dynamically and have access to enterprise resources (such as corporate databases, message queues, user information, etc.) because the servlet runs in a more secure server-side environment.

The JVM and language evolved to become a server-side implementation and a great replacement for CGI applications on the server. In addition to this, web browsers continued to evolve and became even more flexible with the introduction of the Document Object Model (DOM) and Dynamic HTML (DHTML) support. Scripting support was added to the browser with the development of JavaScript (unrelated to Java despite its name) and VBScript. To handle these scripting languages, interpreters were plugged into the browser. An extensible browser architecture proved to be a powerful feature.

Thanks to extensibility, applications such as Macromedia Flash added a new virtual machine on top of the browser, allowing for even more flexible and intense applications. The extensible browser then brought about ActiveX technology on the Windows platform, whereby Windows application functionality could be run within the browser when using Microsoft browsers (or alternative ones with a plug-in that supported ActiveX). This powerful solution enabled native functionality to be accessible from networked applications (see Figure 1-2). This got around the restrictions imposed by the security sandbox and lowest common denominator approach of the JVM, but ultimately, this led to problems in the same vein as distributing client-only applications; specifically, a heavy configuration of the desktop, was necessary to get them to work. Although this configuration could be automated to a certain degree, it resulted in two show-stopping points for many.

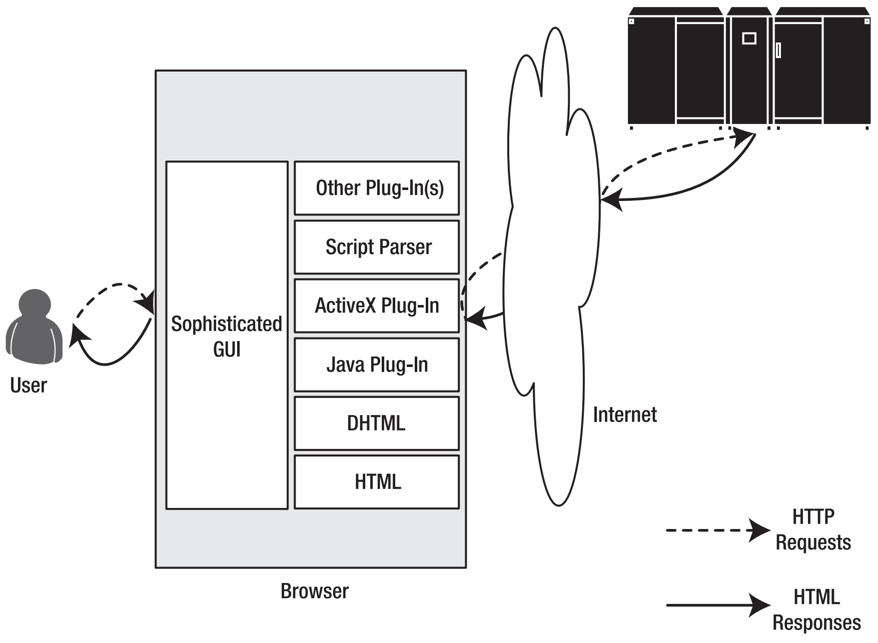


Figure 1-2. *Sophisticated browser architecture*

First, it didn't always work, as the nature of the configuration, changing the Windows registry, often failed—or worse, broke other applications. ActiveX controls were rarely self-contained and usually installed runtime support files. Different versions of these support files could easily be installed on top of each other—a common occurrence leading to broken applications (called DLL Hell).

The second problem was security. A user's computer, when connected to the Internet, could effectively allow code, written by anybody, to run. The ActiveX technology was fully native, not restricted by the Java or HTML sandboxes (more about these in a moment); therefore, users could innocently go to a web page that downloaded an ActiveX control and wrought havoc or stole vital information from their systems. As such, many users refused to use them, and many corporate administrators even disallowed them from use within the enterprise. The virtual nature of Java and HTML—where applications and pages were coded to work on a specific virtual machine—offered better security; these machines couldn't do anything malicious and, therefore, applications written to run on them couldn't either. Users were effectively safe, although limited in the scope of what they could do.

At the end of the 1990s, Microsoft unveiled the successor to ActiveX (among others) in its .NET Framework. This framework would form Microsoft's strategic positioning for many years to come. Like Java, it provided a virtual machine (the Common Language Runtime [CLR]) on which applications would run. These applications could do only what the CLR allowed and were called *managed* applications. The .NET Framework was much more sophisticated than the JVM, allowing for desktop and server-side web applications with differing levels of functionality (depending on which was used). This was part of "managing" the code. With the .NET Framework came a new language, C#, but this wasn't the only language that could be used with .NET because it was a multilanguage, single-runtime platform that provided great flexibility.

The .NET Framework was revolutionary because it united the client-application experience and connected-application experience with a common runtime that ActiveX had tried but ultimately failed to accomplish. Because the same platform was used to write both types of applications, the result was that the user experience would be similar across both (see Figure 1-3). Coupled with the emergence of Extensible Markup Language (XML), a language similar to HTML but specialized for handling data instead of presentation, web application development was finally coming of age.

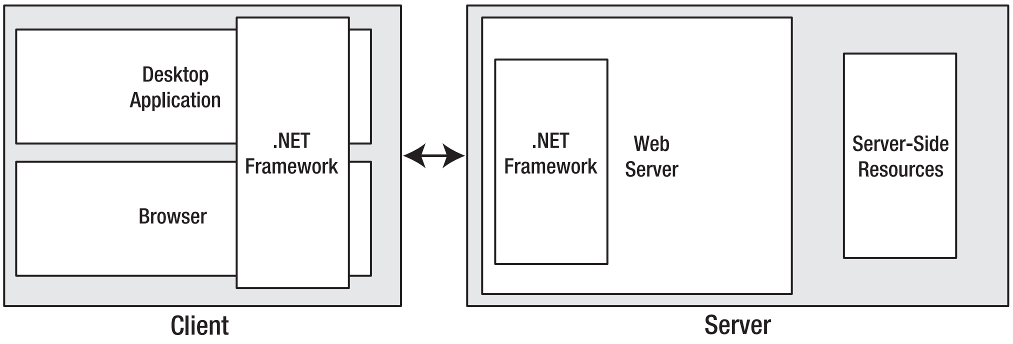


Figure 1-3. *The .NET Framework provides consistent browser, desktop, and server application programming interfaces (APIs).*

Thus, the pendulum has swung back toward the thin client/fat server approach. Ironically, the thin client is probably fatter than the original servers because it's an operating system that can support a browser that is extended to support XML (through parsers), scripting (through interpreters), and other plug-ins, as well as Java and .NET virtual machines! With all these runtime elements available to developers and a consistent server-side API (through the .NET Framework or server-side Java), rich, high-performing applications built using the client/server model are now possible.

Thin Client Applications Save the Day

In the summer of 2001, I had my first "wow" experience with the power of what could be done with a browser-based interface using scripting, DHTML, and asynchronous XML. I was working for a product development group in a large financial services company in New York and was invited by one of their Chief Technical Office (CTO) teams to take a look at their new prototype of a zero-footprint technology for delivering financial information, both streaming and static. They claimed they could stream news, quotes, and charts to a browser with no installation necessary at the desktop, and they could do it in such a manner that it met all the requirements of a typical client. In those days, the biggest support problems were in the installation, maintenance, and support of heavy Component Object Model (COM) desktop applications, and this would wipe them all out in a single blow.

Naturally I was skeptical, but I went to see it anyway. It was a prototype, but it worked. And it largely preserved the user experience that you'd expect from a heavier application with drag-and-drop functionality; streaming updates to news, quotes, and charts; and advanced visualization of data. If anything, it was almost superior to the heavy desktops we were using!

And, it was all built in DHTML, JavaScript, DHTML behaviors, and a lot of server-side functionality using Microsoft-based server products. It was pretty revolutionary.

In fact, it was too revolutionary—and it was too hard for management to take a risk on it because it was so beyond their understanding of how applications *should* work and how the market would accept it. (To be fair, part of their decision was based on my report of concerns about how well the streaming part would scale, but that was nothing that couldn't be fixed!)

But then something terrible happened: September 11, 2001. On that fateful day, a group of individuals turned airliners into missiles, crashing into the World Trade Center and the Pentagon, and killing thousands of people. Part of all this destruction was the loss of many data distribution centers that our company ran for the Wall Street community. With the country having a "get-up-and-running" attitude and wanting the attack to have as little impact on day-to-day affairs as possible, the pressure was on our company to start providing news, quotes, charts, and all the other information that traders needed to get the stock market up and running. The effort to build new data centers and switch the Wall Street users over to them by having staff reconfigure each desktop one by one would take weeks.

The CTO group, with its zero-footprint implementation, ran a T3 line to the machines in the lab that was hosting the application, opening them to the Internet; set up a Domain Name System (DNS) server; and were off and running in a matter of hours. Any trader—from anywhere—could open Internet Explorer, point it at a URL, and start working…no technical expertise required!

Thanks to an innovative use of technology, a business need was met—and that is what our business is all about. Thanks to this experience, and what that group did, I was hooked. I realized the future again belonged to the thin client, and massive opportunities existed for developers and companies that could successfully exploit it.

AJAX Enters the Picture

AJAX, which stands for Asynchronous JavaScript and XML or Asynchronous Java and XML (depending on who you ask), is a technique that has received a lot of attention recently because it has been used with great success by companies such as Amazon and Google. The key word here is *asynchronous* because, despite all the great technologies available in the browser for delivering and running applications, the ultimate model of the browser is still the synchronous request/response model. This means that when an operation occurs in the web page, the browser sends a request to the server waiting for its response. For example, clicking the Checkout button within an HTML page of an e-commerce application consists of calling the web server to process the order and waiting for its response. As such, duplicating the quick refresh and frequent updates provided by desktop applications is hard to achieve. The typical web application involves a refresh cycle where a postback is sent to the server, and the response from the server is re-rendered. In other words, the server returns a complete page of HTML to be rendered by the browser, which looks kind of clunky compared to desktop apps. This is a drawback to this type of architecture because the round-trip to and from the server is expensive in user time and bandwidth cost, particularly for applications that require intensive updates.

What is interesting about the AJAX approach is that there is really nothing new about it. The core technology—the `XMLHttpRequest` object—has been around since 1999 with Internet Explorer, when it was implemented as an ActiveX plug-in. This is a standard JavaScript object recognized by contemporary browsers, which provides the asynchronous postback capabilities upon which AJAX applications rely. More recently, it has been added to the Mozilla Firefox, Opera, and Safari browsers, increasing its ubiquity, and has been covered in a World Wide Web Consortium (W3C) specification (DOM Load and Save). With the high popularity of web applications that use the `XMLHttpRequest` object, such as Google Local, Flickr, and Amazon A9, it is fast becoming a de facto standard.

The nice part about the `XMLHttpRequest` object is that it doesn't require any proprietary or additional software or hardware to enable richer applications. The functionality is built right into the browser. As such, it is server agnostic. Except for needing to make some minor changes to your browser security settings, you can use it straightaway, leveraging coding styles and languages you already know.

To see an example of how it works, refer to Google Local (see Figure 1-4). As you use the mouse to drag the map around the screen, the sections of the map that were previously hidden come into view quickly; this is because they were cached on your initial viewing of the map. Now, as you are looking at a new section (by dragging the mouse), the sections bordering the current one are downloading in the background, as are the relevant satellite photographs for the section of map you are viewing.

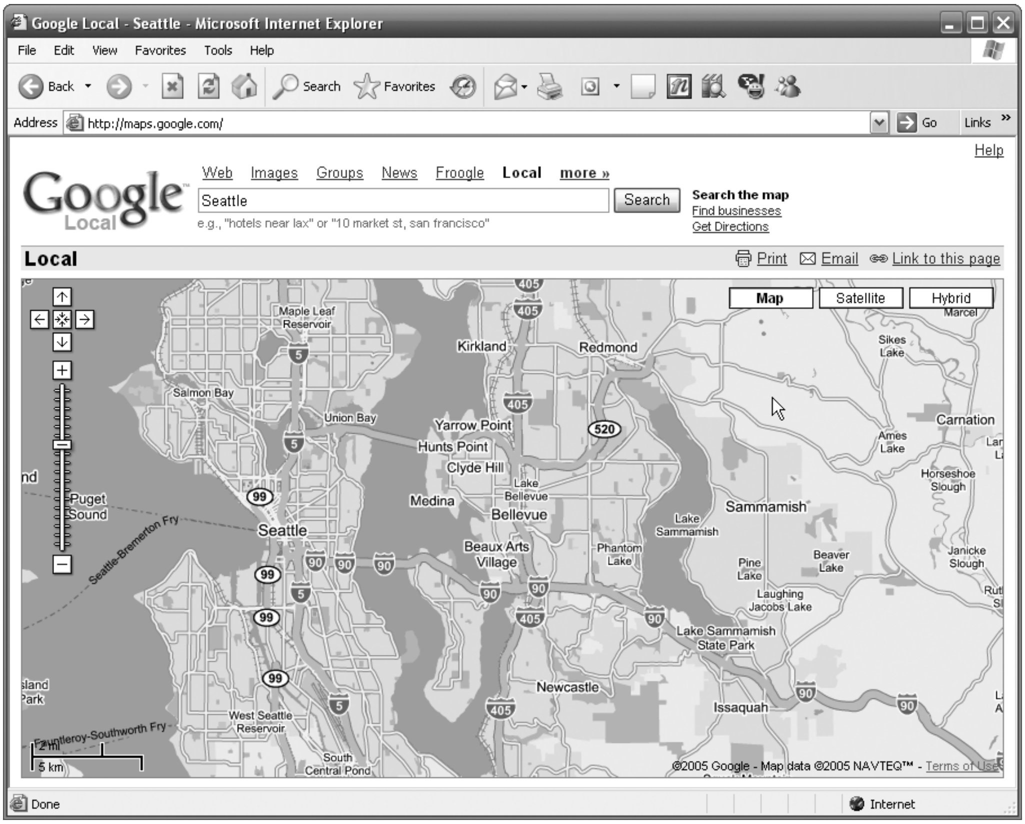

Google Local - Seattle - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://maps.google.com/
Go Links
Help
Web Images Groups News Froogle Local more »
Seattle
Search
e.g., "hotels near lax" or "10 market st, san francisco"
Search the map
Find businesses
Get Directions
Local
Print Email Link to this page
Map Satellite Hybrid
Seattle
Kirkland
Redmond
Bellevue
Sammamish
Issaquah
Newcastle
©2005 Google - Map data ©2005 NAVTEQ™ - Terms of Use
Done
Internet

Figure 1-4. *Google Local uses AJAX extensively.*

This background downloading, using the `XMLHttpRequest` object, makes using Google Local such a smooth and rewarding experience. Remember, nothing is *new* here; it's just that having the `XMLHttpRequest` object built into the browser that can do this asynchronously makes it possible to develop applications like this.

Note For full details on how to develop in AJAX, check out *Foundations of AJAX* (Apress, 2005).

You will be looking at AJAX from a high level in this book and delving more deeply into how Microsoft ASP.NET AJAX will allow you to quickly and easily build AJAX-enabled applications.

Using the `XMLHttpRequest` Object

As mentioned, the `XMLHttpRequest` object is the heart of AJAX. This object sends requests to the server and processes the responses from it. In versions of Internet Explorer prior to IE7, it is implemented using ActiveX, whereas in other browsers, such as Mozilla Firefox, Safari, Opera, and Internet Explorer 7, it is a native JavaScript object. Unfortunately, because of these differences, you need to write JavaScript code that inspects the browser type and creates an instance of it using the correct technology.

Thankfully, this process is a little simpler than the spaghetti code you may remember having to write when using JavaScript functions that heavily used DOM, which had to work across browsers:

```
var xmlHttp;
function createXMLHttpRequest()
{
    if (window.ActiveXObject)
    {
         xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest)
    {
         xmlHttp = new XMLHttpRequest();
    }
}
```

In this case, the code is simple. If the browser doesn't support ActiveX objects, the `window.ActiveXObject` property will be null, and, therefore, the `xmlHttp` variable will be set to a new instance of the native JavaScript `XMLHttpRequest` object; otherwise, a new instance of the `Microsoft.XMLHTTP` ActiveX Object will be created.

Now that you have an `XMLHttpRequest` object at your beck and call, you can start playing with its methods and properties. Some of the more common methods you can use are discussed in the next few paragraphs.

The `open` method initializes your request by setting up the call to your server. It takes two required arguments (the Hypertext Transfer Protocol [HTTP] command such as `GET`, `POST`, or `PUT`, and the URL of the resource you are calling) and three optional arguments (a boolean indicating whether you want the call to be asynchronous, which defaults to true, and strings for the username and password if required by the server for security). It returns void.

```
xmlHttp.open("GET" , "theURL" , true , "MyUserName" , "MyPassword");
```

The send method issues the request to the server. It is passed a single parameter containing the relevant content. Had the original request been declared as asynchronous (using the boolean flag mentioned earlier), the method would immediately return; otherwise, this method would block until the synchronous response was received. The content parameter (which is optional) can be a DOM object, an input stream, or a string.

```
xmlHttp.send("Hello Server");
```

The setRequestHeader method takes two parameters: a string for the header and a string for the value. It sets the specified HTTP header value with the supplied string.

```
xmlHttp.setRequestHeader("Referrer","AGreatBook");
```

The getAllResponseHeaders method returns a string containing the complete set of response headers from the XMLHttpRequest object after the HTTP response has come back and containing their associated values. Examples of HTTP headers are "Content-Length" and "Date". This is a complement to the getResponseHeader method, which takes a parameter representing the name of the specific header you are interested in. The method returns the value of the header as a string.

```
var strCL;
strCL = xmlHttp.getResponseHeader("Content-Length");
```

In addition to supporting these methods, the XMLHttpRequest object supports a number of properties, as listed in Table 1-1.

Table 1-1. *The Standard Set of Properties for* XMLHttpRequest

Property	**Description**
onreadystatechange	Specifies the name of the JavaScript function that the XMLHttpRequest object should call whenever the state of the XMLHttpRequest object changes
readyState	The current state of the request (0=uninitialized, 1=loading, 2=loaded, 3=interactive, and 4=complete)
responseText	The response from the server as a string
responseXML	The response from the server as XML
status	The HTTP status code returned by the server (for example, "404" for Not Found or "200" for OK)
statusText	The text version of the HTTP status code (for example, "Not Found")

Using Visual Studio 2005

Throughout this book, you'll be using Visual Studio 2005 to develop AJAX applications using ASP.NET AJAX. Several editions of this application are available to satisfy different needs.

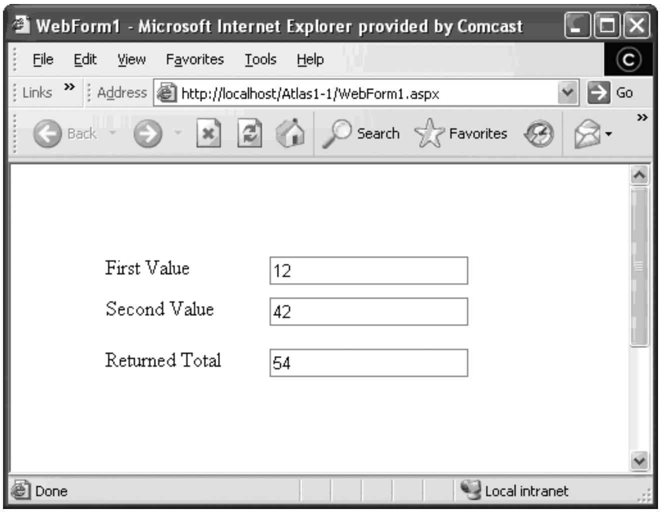
You can download the free edition, Visual Web Developer 2005 Express, from the Microsoft Developer Network (`http://msdn.microsoft.com/vstudio/express/vwd`). From this page, you can also navigate to the downloads for the other Express editions, including ones for C#, VB .NET, Visual J#, and C++ development.

You can use any edition of Visual Studio 2005, including Standard, Professional, or one of the flavors of Team Edition, to build and run the samples included in this book.

If you are following along with the figures in this book, you'll see they have been captured on a development system that uses the Visual Studio 2005 Team Edition for Software Developers.

Seeing a Simple Example in Action

Understanding how this technology all fits together is best shown using a simple example. In this case, suppose you have a client application that uses JavaScript and an `XMLHttpRequest` object to issue a server request to perform the simple addition of two integers. As the user types the values into the text boxes on the client, the page calls the server to have it add the two values and return a result, which it displays in a third text box. You can see the application in action in Figure 1-5.


WebForm1 - Microsoft Internet Explorer provided by Comcast
File Edit View Favorites Tools Help
Links Address http://localhost/Atlas1-1/WebForm1.aspx Go
Back Search Favorites
Done
Local intranet

First Value 12
Second Value 42
Returned Total 54

Figure 1-5. *The AJAX addition client*

To create this client, start Visual Studio 2005, create a new web site, edit the page Default.aspx, and set its content to be the same as Listing 1-1.

Listing 1-1. *Creating Your First AJAX Application*

```
<%@ Page language="C#" CodeFile="Default.aspx.cs" AutoEventWireup="false"
Inherits="_Default" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
  <title>WebForm1</title>
  <script language="javascript">
    var xmlHttp;

    function createXMLHttpRequest() {
        if (window.ActiveXObject) {
            xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
        }
        else if (window.XMLHttpRequest) {
            xmlHttp = new XMLHttpRequest();
        }
    }

    function updateTotal() {
        frm = document.forms[0];
        url="Default2.aspx?A=" + frm.elements['A'].value +
              "&B=" + frm.elements['B'].value;
        xmlHttp.open("GET",url,true);
        xmlHttp.onreadystatechange=doUpdate;
        xmlHttp.send();
        return false;
    }

    function doUpdate() {
        if (xmlHttp.readyState==4 && xmlHttp.status == 200) {
                document.forms[0].elements['TOT'].value=xmlHttp.responseText;
        }

}
  </script>
  </HEAD>
  <body onload="createXMLHttpRequest();">
        <form>
        <TABLE height="143" cellSpacing="0" cellPadding="0"
                      width="300" border="0" >
                <TR vAlign="top">
                        <TD height="32">First Value</TD>
                        <TD><INPUT type="text" id="A" value="0"
                                              onkeyup="updateTotal();"></TD>
                </TR>
                <TR vAlign="top">
                        <TD height="32">Second Value</TD>
                        <TD><INPUT type="text" id="B" value="0"
                                              onkeyup="updateTotal();"></TD>
                </TR>
                <TR vAlign="top">
                        <TD height="23">Returned Total</TD>
                        <TD><INPUT type="text" id="TOT" value="0"></TD>
                </TR>
        </TABLE>
        </form>
    </body>
</HTML>
```

When the web page loads, the `createXMLHttpRequest` function is called (as a result of setting the `onload` event handler in the body tag) to create the `XMLHttpRequest` object. After that, whenever a key is pressed in the A or B text boxes, the `updateTotal` function is called (by trapping the `onkeyup` event on the two text boxes).

The `updateTotal` function takes the values of A and B from their form elements and uses them to build the URL to *Default2.aspx*, which will look something like `Default2.aspx?A=8&B=3`. It then calls the `open` method on `XMLHttpRequest`, passing it this URL and indicating that this will be an asynchronous process. Next, it specifies the `doUpdate` function to handle the `readystate` changes on the `XMLHttpRequest` object.

To get this application to work, add a new C# web form to the project, and leave the default name of *Default2.aspx*. In the page designer, delete all of the HTML so that the page contains just the ASPX `Page` directive:

```
<%@ Page language="C#"
    CodeFile=”Default2.aspx.cs”
    AutoEventWireup="true"
    Inherits="Default2" %>
```

Then add the following code to the C# code file's `Page_Load` method (you can add it by double-clicking the *Default.aspx* page when it is shown in the design window of Visual Studio 2005):

```
int a = 0;
int b = 0;
if (Request.QueryString["A"] != null)
{
        a = Convert.ToInt16(Request.QueryString["A"].ToString());
}
if (Request.QueryString["B"] != null)
{
        b = Convert.ToInt16(Request.QueryString["B"].ToString());
}

Response.Write(a+b);
```

This handles the asynchronous request from the page *Default.aspx*, getting the values of A and B, and writing the sum to the response buffer. When the `XMLHttpRequest` object receives the response from *Default2.aspx,* it calls the `doUpdate` function, which checks to see if the value of the `readyState` property is equal to “4,” indicating that the request has been completed. If the value is equal to “4,” the function updates the `INPUT` field named `TOT` with the value returned by *Default2.aspx,* which is stored in the `XMLHttpRequest` object's `responseText` property.

Summary

In this chapter, you were given a brief history of the methodologies of building user interfaces that send data to servers for processing and the constantly swinging pendulum from thin client to fat client. You were brought up-to-date on what the newest trend in this development is—web-based thin clients with rich functionality—thanks to the asynchrony delivered by the `XMLHttpRequest` object, which is the core of AJAX. You then built a simple example that demonstrated how it works. This example was very basic and barely scratched the surface of what can be done with AJAX. However, it demonstrated one of the drawbacks of using this methodology; namely, that it requires a lot of scripting. JavaScript, although powerful, is tedious to write and onerous to debug and manage when compared to languages such as C#, VB .NET, and Java. As such, the application benefits you receive by using an AJAX approach may be more than offset by the application development getting bogged down in thousands (or more) lines of JavaScript.

With this problem in mind, Microsoft integrated the power of AJAX with the productivity of ASP.NET 2.0 and Visual Studio 2005 to develop ASP.NET AJAX.
In the next chapter, you'll be introduced to the wonderful world of ASP.NET AJAX. You will look at its architecture, learn how it allows you to use Visual Studio 2005 and ASP.NET 2.0 server controls to generate client-side code, and see how this can give you the best of AJAX while avoiding the worst of it.