1

Overview of AJAX

AJAX has definitely been the hot buzzword in the Web application world for the last few years. AJAX is an acronym for Asynchronous JavaScript and XML and, in Web application development, it signifies the capability to build applications that make use of the `XMLHttpRequest` object.

The creation and inclusion of the `XMLHttpRequest` object in JavaScript and the fact that most upper-level browsers support it led to the creation of the AJAX model. AJAX applications, although they have been around for a few years, gained popularity after Google released a number of notable, AJAX-enabled applications such as Google Maps and Google Suggest. These applications demonstrated to the world the real value of AJAX, and every developer that saw these applications in action immediately went out to do research in how these applications were built.

AJAX is now an out-of-the-box feature of ASP.NET 3.5, and every ASP.NET application that you build is AJAX-enabled by default. This means that you don't have to create a separate AJAX project for your Web applications as the standard projects in Visual Studio for Web application development will already be enabled to use ASP.NET AJAX. This is one of the main new features of ASP.NET 3.5 because of the power that AJAX brings to your applications. The AJAX capability has become so popular since the release of ASP.NET 3.5 that most ASP.NET applications built today make use of at least some features provided by this technology.

Why AJAX Applications Are Needed

Web applications were in a rather stagnant state for many years. The first "Web" applications were nothing more than text and some images, all represented in basic HTML tags. However, this wasn't what people wanted; they wanted more — more interactivity, a more integrated workflow, more responsiveness, and an overall richer experience.

When building applications, even today, you have to make some specific decisions that really end up dictating the capabilities and reach of your application. Probably one of the more important decisions is the choice of building the application as a "thin" client or a "thick" client.

A thick client application is a term used for applications that are either MFC (in the C++ world) or Windows Forms applications. These types of applications also provide the container along with all the container contents and workflows. A thick client application is typically a compiled executable that end users can run in the confines of their own environment, usually without any dependencies elsewhere (for example, from an upstream server).

A thin client application is the term generally used for Web applications. These types of applications are typically viewed in a browser, and your application can provide end user workflows as long as it is contained within this well-known container. A thin client application is one whose processing and rendering are controlled at a single point (the upstream server), and the results of the view are sent down as HTML to a browser to be viewed by the client.

Typically, a Web application is something that requires the end user to have Internet access in order to work with the application. On the other hand, a thick client application was once generally considered a self-contained application that worked entirely locally on the client's machine. However, in the last few years, this perception has changed as thick clients have evolved into what are termed "smart clients" and now make use of the Internet to display data and to provide workflows.

Web applications have historically been less rich and responsive than desktop applications. End users don't necessarily understand the details of how an application works, but they know that interacting with a Web site in the browser is distinctly different from using an application installed locally. Web applications are accessible from just about any browser, just about anywhere, but what these browsers present is limited by what you can do with markup and script code running in the browser.

There are definitely pros and cons in working with either type of application. The thick client application style is touted as more fluid and more responsive to an end user's actions. Thick client applications require that users perform an installation on their machine but let developers leverage the advanced mouse and graphics capabilities of the operating system that would be extremely difficult to implement in a Web browser, and also take advantage of the user's machine for tasks such as offline storage.

Conversely, the main complaint about Web applications for many years has been that every action by an end user typically takes numerous seconds and results in a jerky page refresh.

Conversely, Web applications can be updated just by changing what is running on the server, and site visitors get the latest version of that application instantaneously. However, it is much more difficult to update a desktop application, because you would have to get users to perform yet another installation or else ensure that the application has been coded to include a clever system for doing updates automatically.

Web applications are said to use a zero-deployment model, but desktop applications use a heavy deployment and configuration model. The choice is often characterized as a tradeoff between rich and reach: Desktop applications generally offer a richer user experience than what could be offered in the browser, but with a Web application you are able to reach users anywhere on any OS with almost no extra effort. Furthermore, many companies have restrictive policies in place regarding what software can be installed on employees' machines, and they often don't allow employees to have administrative access that is required to install new applications, so Web applications are the only viable option in many situations.

AJAX is the first real leap of a technology to bridge this historic wall between thick and thin. AJAX, though still working through the browser, focuses on bringing richness to Web applications by allowing for extremely interactive workflows that usually were only found in the thick client camp.

Bringing Richness to Web Applications

Years ago, having a Web presence was a distinguishing factor for companies. That is no longer the case. Now just having a Web presence is no longer enough. Companies are distinguishing themselves further through Web applications that react intuitively to customer actions and anticipate user input. This book shows you how ASP.NET AJAX addresses specific Web development challenges and paves the way for taking your Web site to another level of user experience.

The fundamental set of technologies used in the AJAX model that enable the next generation of Web applications is not entirely new. You will find that many people point to Google, Flickr, and several other services as prime examples of leveraging AJAX and its underlying technologies in unique ways. The applications have some unique features, but in reality, the underlying technologies have been around and in use for nearly a decade. Look at how Microsoft Exchange Server provided rich access to e-mail from a Web browser in the Outlook Web Access application, and the concept of ubiquitous access from a browser while leveraging a common set of browser features for a rich user experience has been around for years. In this case, users get a remarkably full-featured application with no local installation and are able to access e-mail from virtually any machine.

While the AJAX acronym is nice, it doesn't do much to explain what is actually happening. Instead of building a Web application to be just a series of page views and postbacks, developers are using JavaScript to communicate asynchronously with the Web server and update parts of the page dynamically. This means that the Web page can dynamically adapt its appearance as the user interacts with it, and it can even post or fetch data to or from the Web server in the background. Gone are the days of the ugly postback, which clears the user's screen and breaks his concentration! Instead, you need to post back now only if you want to change to a different Web page.

> *Even that rule can be bent. Some applications are pushing this boundary and completely changing the user's view, just as though they navigated to a new page, but they do so through an asynchronous post and by changing the page content without actually navigating to a new URL.*

The AJAX acronym refers to XML as the data format being exchanged between client and server, but in reality, applications are being built that retrieve simple pieces of text, XML, and JSON (JavaScript Object Notation) (which is discussed in more detail in Chapter 4). Part of the AJAX appeal is not even covered by the acronym alone: In addition, to communicating with the server without blocking, developers are leveraging Dynamic HTML (DHTML) and Cascading Style Sheets (CSS) to create truly amazing user interfaces. JavaScript code running on the client communicates asynchronously with the server and then uses DHTML to dynamically modify the page, which supports rich animations, transitions, and updates to the content while the user continues interacting with the page. In many cases, end users will sometimes forget they are using a Web application!

Just remember that AJAX is not a single holistic entity but instead is a novel and creative way of using a combination of technologies such as the `XMLHttpRequest` object, HTML, XHTML, CSS, DOM, XML, JSON, XSLT, and JavaScript. You might be thinking of the difficulties of piecing this all together to get the Web applications you want to build. Be ready to be wowed, however, as the focus of this book is on showing you how to use the built-in technologies provided by ASP.NET 3.5 to give you this power in an easy to use manner.

Who Benefits from AJAX?

AJAX offers benefits to both end users and developers. For end users, it reduces the "rich or reach" conflict; for developers, it helps in overcoming the constraints raised by HTTP such as the dreaded page postback.

Why End Users Want AJAX Applications

Users tend to view desktop applications as a commitment. They install a program, usually from a disk pulled from a costly shrink-wrapped box. The program consumes hard disk space as well as a position in the program menu. The user may need to update the program periodically or perform an upgrade later on to get new features. If the program is proactive about updating itself, the user is confronted regularly with dialogs about accepting patches or downloads. In exchange for this investment of time, money, and energy, the user is repaid with an application that is able to leverage the operating system and machine resources. It is a rich application. It has local storage capabilities, offers quick response times, and can present a compelling and intuitive graphical user interface.

More and more applications are becoming accessible from the Web browser, where the full resources of the hardware and OS are not available, but the user commitment of a desktop application is not required. Over the years, interacting with a Web application has meant a predictable pattern for users. They click a link in the page, and the browser flashes while the user waits until the page is repainted (the dreaded page postback). This cycle is repeated over and over. The user looks at what is presented on the page, interacts with it, and clicks somewhere on the page. The browser then produces an audible click for feedback and begins to postback to the server. The screen of the Web browser flashes blank and some icon spins or flashes while the user waits for a new version of the page to be returned from the server. Many times, the new version of the page is almost exactly the same as the previous version, with only part of the page being updated. And then the cycle begins all over again. This has a sluggish feeling even when the user has a high-speed network connection and is simply unacceptable for some types of applications.

The AJAX set of technologies has changed what users expect from Web applications. JavaScript code running in the browser works to exchange data with the Web server asynchronously. There is no click sound and the browser does not flash. The request to the server is non-blocking, which means the user is able to continue viewing the page and interacting with it. The script gets the updated data from the server and modifies the page dynamically, using the DHTML coding methodology. The user is able to continue looking at the page while parts of it are updated in the background. AJAX is used to provide a more responsive experience, making Web applications behave more like desktop installations. JavaScript is used to provide a richer experience with support for drag-and-drop, modal dialogs, and seemingly instantaneous updates to various parts of the page based on user inputs.

A big part of successfully leveraging AJAX technologies is in the perceived performance increase. Users appreciate Web applications that anticipate their actions. If you also use JavaScript code in the background to pre-fetch images and data that may be needed, users can get a speedy response without the usual pause that accompanies their actions. Nobody wants to wait for data exchanges between client and server; studies have shown that a time lag between user input and subsequent UI changes can significantly reduce their productivity and give them the frustrating feeling that they are fighting the application. Users want Web applications to behave like desktop installations but without the overhead associated with an installation. As more applications employ smart caching, anticipate user actions, and provide richer UIs, the difference between Web and desktop applications is definitely becoming blurred. Expectations for Web applications are rising. The end user has now seen that it is possible to avoid the commitment of installing a desktop application and still have a rich and responsive experience.

Why Developers Want AJAX

Often, the first question to arise when starting a new development project is what type of application it will be. Should it be a desktop application or a Web application? This is a key decision because it has historically dictated a lot about the nature of the application and the development problem space. Many developers are now choosing to build Web applications by default unless something about the application dictates that it must be a desktop install. If it must run offline or if it requires a user interface that is too complex to achieve in HTML, targeting the Web browser may be ruled out, and developers are forced to write a standalone application.

Developers have a difficult job writing modern Web applications due to the inherent World Wide Web functionality constraints imposed by the use of the Hypertext Transfer Protocol (HTTP) and the way that browsers use it. HTTP is a stateless protocol. The Web browser requests a page, possibly carrying some sort of state (a querystring or form input parameters), and the Web server processes the request and sends a response that includes HTML-rendered content. The Web server can only react to the information supplied in the current request and does not know any additional information from the request itself, such as any details about the path the user took to get to the current view.

When the response is rendered, the connection may be broken and the server will not have any information to preserve for the next request. From the server's perspective, it is simply listening for requests to come in from any browser anywhere and then reacting. The browser issues a request to the page and receives an HTML page in response. It uses the HTML it receives to render the user interface. The user interacts with the page, and, in response, the browser clears the screen and submits a new request to the server, carrying some information about user input or actions. Again, a complete HTML page is returned. The browser then presents the new version of HTML. Fundamentally, the HTTP protocol is stateless. The server gets a request and responds to it. The request carries limited information about the ongoing conversation that is happening between client and server. This can definitely be a problem.

AJAX makes this much better. AJAX breaks this pattern by updating portions of the page separately, via partial page rendering. Figure 1-1 shows a typical non-AJAX series of browser and server interactions (requests and responses). Each request results in full-page rendering. In response, the browser updates the user's entire view with the HTML that is returned.

The sequence presented here in Figure 1-1 is typical of the type of Web application that we have been living with for many years now. It has a request and response cycle that is abrupt and rather noticeable to the end user. Let it be said that, with the introduction of AJAX technologies, these types of applications are changing quickly to the new model this technology provides.

On the other hand, Figure 1-2 shows how AJAX is employed to improve the user's experience.

In this case, a request is made for the initial page rendering. From there, asynchronous requests to the server are made. An asynchronous request is a background request to send or receive data in an entirely nonvisual manner (meaning that there won't be any resulting page flickering). They are asynchronous because the user interface is not frozen during this time, and users can continue interacting with the page while the data transfer is taking place. These calls get just an incremental update for the page instead of getting an entirely new page.

JavaScript running on the client reacts to the new data and updates various portions of the page as desired. The number of requests to the server may be no different, or in some cases, there may actually be more calls to the server, but the users' perception is that the application feels more responsive. End

users are not forced to pause, even if it is only a slight pause, and wait for the server while staring at a blank browser screen. This model, while chatty in some regards, provides the fluidity you are looking for in your Web applications.

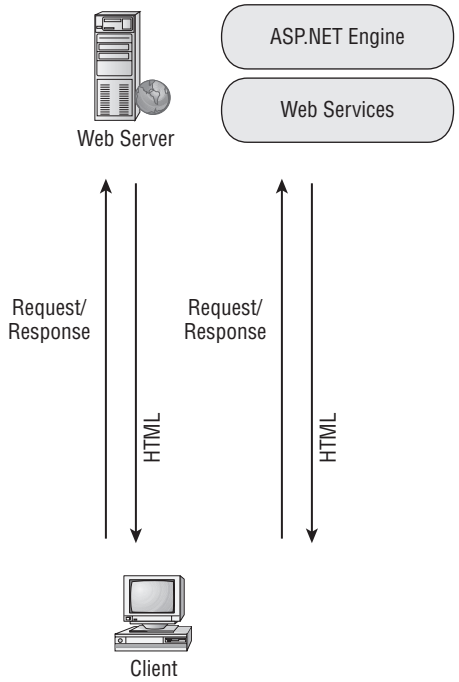

Figure 1-1

AJAX applications make use of the `XMLHttpRequest` object as an initiator and caller of the underlying data needed by the application to make the necessary changes. These requests are routed through a serialization process before being transmitted across the wire. In most cases, the output is some type of XML (such as SOAP) or JSON (for those who want an especially a tight format). The response from the server is then deserialized and provided to the JavaScript on the page, which then interacts with DHTML to render the parts of the page, outside of the normal postback process.

Technologies of AJAX

Almost a decade ago, the Microsoft Exchange Server team created an ActiveX control called `XMLHttpRequest` that could be instantiated from JavaScript and used to communicate with the server. Using the `XMLHttpRequest` object, you could send information to the server and get data back without clearing the screen and painting a completely new HTML page. JavaScript code could then manipulate the HTML dynamically on the client, avoiding the annoying flash and the wait that users associate with Web browsing. This functionality was not limited to Internet Explorer for long. Soon, other browsers included `XMLHttpRequest` objects as well. Developers could now write richer applications with their reach extending across various operating systems.

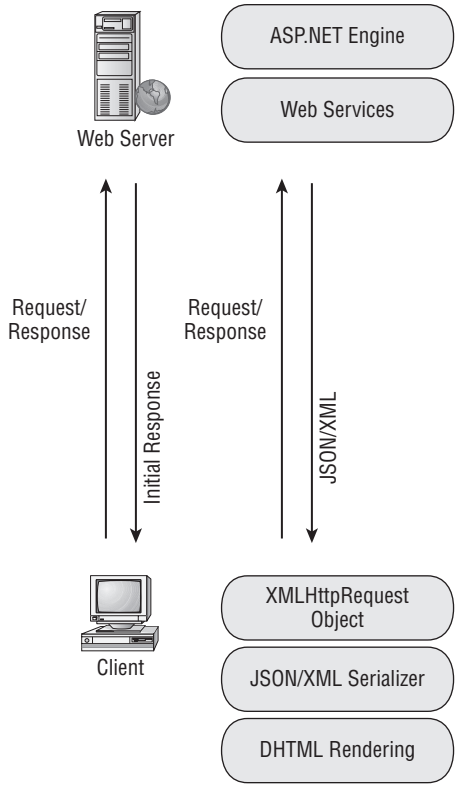

Figure 1-2

The browsers that can make use of this object include the following:

- Internet Explorer 5.0 and above (currently in version 8.0)
- Safari 3.1
- Firefox 3
- Opera 8+
- Netscape 9

The browsers also created an advanced Document Object Model (DOM) to represent the browser, the window, the page, and the HTML elements it contained. The DOM exposed events and responded to input, allowing the page to be manipulated with script. Dynamic HTML (DHTML) opened the door to writing rich interfaces hosted within the Web browser. Developers started writing hundreds and even thousands of lines of JavaScript code to make rich and compelling applications that would not require any client installation and could be accessed from any browser anywhere. Web applications began to move to a whole new level of richness. Without AJAX libraries, you would be faced with writing lots and lots of JavaScript code and debugging the sometimes subtle variations in different browsers to reach this new level of richness.

JavaScript Libraries and AJAX

Developers have had access to AJAX technologies for years, and many have been leveraging AJAX to push the limits of what can be done in the Web browser. But what is really making AJAX more compelling now are the comprehensive script libraries and integration with server technologies that make it easier to write rich Web applications and avoid the need to become an expert on the different versions of JavaScript. A JavaScript library is referenced within the HTML of a page by using the `<script>` tag:

```
<html>
    <head>
        <script src="http://www.someSite.com/someScript.js"
          type="text/javascript">
        </script>
    </head>
...
```

The script is downloaded and cached by the browser. Other pages within the application can reference the same URL for script, and the browser will not even bother to go back to the server to get it. The functionality of that script file is available for use from within the page rendered to the browser. A script library sent to the browser and then leveraged for writing a richer UI and a more responsive application is at the heart of all AJAX libraries.

The Initiator Component

To initiate calls to the back end server, you need JavaScript on the client to invoke the `XMLHttpRequest` object, which takes charge of making these out-of-bound calls. This component works to send data back and forth asynchronously. This core capability is browser-independent and allows for requests and responses to occur without interrupting the end user experience.

The JavaScript Component

AJAX technologies take advantage of the common support for JavaScript found in modern browsers. Because there is a standard that is supported across the various browsers, you can write scripts knowing that they will run. This wasn't always the case.

In the mid-1990s, Netscape and Microsoft (along with others) collaborated on a standard for a scripting language that they would support in their Web browsers. The standard is called EcmaScript. Microsoft's implementation is called JScript, but the language is generally referred to as JavaScript, as it was called in Netscape. (It has nothing to do with Java, but someone must have thought the association was useful for marketing purposes.) JavaScript program snippets are sent down to the browser along with the HTML, and they run inside the user's browser to affect how the page is processed on the client.

JavaScript is not compiled; it is interpreted. There is no static type-checking as you get in C++ and C#. You can declare a variable without needing to specify a type, and the type to which the variable refers can change at any time. This makes it easy to start programming in JavaScript, but there is inevitably a certain amount of danger in allowing the data type of a variable to change dynamically at runtime. In the following snippet, notice that the variable can reference any type without difficulty:

```
var something = 1;
something = true;
something = "a string";
```

JavaScript is a dynamic language. Types can actually be extended during program execution by other code. This means that you can write code that creates types on the fly. Because there is no enforcement of type safety, your code can receive these types as parameters or return values without any problem. This provides a great degree of flexibility and coding power.

The fundamental types in JavaScript are strings, numbers, Booleans, and functions. There is also support for objects and arrays, which are collections of the fundamental types. Some additional objects that are considered essential for many programming tasks are included in JavaScript. This includes support for regular expressions and date and time operations.

You can use the plus operator on strings in JavaScript to concatenate them:

```
var theEnd = "THE END.";
var result = "Beginning, " + "middle, and " + theEnd;
```

In this example, the result variable is now the string: `"Beginning, middle, and THE END"`.

JavaScript interpreters use the IEEE floating-point standard for storing numbers. Ignoring the gory details, you can assume that for most programming tasks you won't have any trouble.

The Boolean type in JavaScript is about what you would expect it to be but maybe not exactly so. The Boolean represents whether or not an expression is true, but it uses the C-style convention using integer values 0 and 1.

Variables can exist in JavaScript without having a value, and a variable may simply be undefined, which can produce unexpected results. In this snippet of JavaScript, three variables are declared, and all of these comparisons are designed to return a true value.

```
<script type="text/javascript">
   var one = 1;
   var zero = 0;
   var undefinedVar;

   if(one) {
      alert("1 is true");
   }

   if(!zero) {
      alert("0 is false");
   }

   if(!undefinedVar) {

      // this test tells us that "undefinedVar" either contains 0,
      // or is really undefined: both of which equate to false
      alert("undefinedVar is false");
   }

   if(one != zero) {
      alert("one and zero are not the same");
   }
</script>
```

You can check specifically to see if a variable has been defined like this:

```
if( typeof(undefinedVar ) == "undefined" ) {
   alert("undefinedVar is undefined");
}
```

Variables can also have a null value, which is not the same thing as being undefined, because a null value does constitute a value.

Functions are also real types in JavaScript. They can accept arguments and return values. Functions can be passed to other functions and can be created dynamically by other script code.

Here are two equivalent definitions for a function called `Add()` that will take two variables and return the result of applying the plus operator. Notice that this did not state that it takes two numbers. Remember, JavaScript variables do not have a defined type, so I could just as easily pass two strings and concatenate them with my `Add()` function.

```
<script type="text/javascript">
   function Add(x, y) {
      return x + y;
   }

   var AddAgain = function(x, y) { return x + y; }
</script>
```

Once either of these styles is used to create a function, it can be called from that scope and any nested scope to perform the addition. There is no advantage to one of these forms over the other. You can simply choose to use the syntax that you prefer.

```
<script type="text/javascript">
   var result = Add(36, 24);
   alert(result);  //displays 60

   var stringResult = Add("Hello ", "there.");
   alert(stringResult); //displays "Hello there."
</script>
```

Objects and arrays are just collections of other types. Array types do not require that the values they hold be named. Instead, you can access them by index. The values held in an object are referenced by field or property names. Objects can also hold functions (which can be accessor functions to give public visibility to local variables), which lets you create data structures that represent entities in JavaScript code. Missing from this sort of object-oriented programming is a concept of type inheritance. Although, there are things like the Microsoft AJAX Library, which provides a set of classes and recommended patterns for achieving inheritance in JavaScript, making it more natural to switch between JavaScript and other high-level languages. The following code example includes a definition for an Album object that holds and returns the artist and album title. An array is then used to store information about several albums.

```
<script type="text/javascript">
   // define an object named Album - note that this object is typeless
   Album = function(title, artist) {
      var _title = title;
      var _artist = artist;
```

```
        this.get_title = function() { return _title; }
        this.get_artist = function() { return _artist; }
    }

    // create object instances by calling the constructor
    var albumA = new Album("Rift", "Phish");
    var albumB = new Album("A Picture of Nectar", "Phish");

    // create an array to hold the instances (also typeless)
    var albumArray = new Array();

    albumArray[0] = albumA;
    albumArray[1] = albumB;

    // iterate over the array to show the album titles
    for(var i = 0; i < albumArray.length; i++) {
        alert((albumArray[i]).get_title()); // call get_title accessor
    }
</script>
```

The Web Services Component

The fundamental concept of Web services is powerful and continues to evolve and advance. The original Simple Object Access Protocol (SOAP) standard is the use of the HTTP protocol to pass XML-formatted data to the server from a client and receive XML-formatted results in return. This can be from within a Web browser using the `XMLHttpRequest` object or directly from a desktop application or another server. Before Web services became widely adopted, it was not uncommon for developers to programmatically request a page as an HTML document and extract the desired data from it, a technique known as screen-scraping. This causes all sorts of frustrations as sites are continually updated and the screen-scraping clients must try to keep up by modifying their parsing code to adapt to the new HTML that the target site is rendering.

This resulted in frustration, because sites that presented data using HTML visual pages were prone to modifying those pages, and this would break the screen-scraping program, which expected to see the data in the original format. Web services were created as a nonvisual way to transfer data over the Web, and they are the natural way to isolate remote method calls from the presentation layer. Now, instead of screen-scraping, you are able to call a Web service and have XML-formatted data returned that is easily consumed by a program.

By passing plain-text data formatted as XML and by eliminating the visual elements, data passed in Web services is much easier to parse than HTML. Moreover, since XML can contain an embedded schema, code can inspect the schema and use it to determine the structure and types used in the data. You can extend the schema passed with the data being returned without worrying that consuming applications will be broken, and therefore XML readers can be somewhat tolerant of modifications that would have certainly caused a screen-scraper a great deal of grief!

The schema for data can be extended without requiring all consumers to be updated. Consumers can easily get the parts of the XML document they wish to process and disregard the rest. This has progressed beyond simple XML formats. Unlike previous implementations of Web services, you can now define Web service contracts to be built to employ arbitrary encoding and utilize any one of a number of wire protocols. What drives the Web service concept is the ability to access data easily from various

applications in a loosely coupled way, and the new Microsoft Windows Communication Foundation (WCF) takes this concept to a completely new level, allowing the contract to specify wire protocols, deployment strategies, and logging infrastructure, along with providing support for transactions.

ASP.NET AJAX provides a set of JavaScript proxy objects to access some new Web services built into ASP .NET. Profile information, membership services, and role management can be easily accessed from the client. Developers don't need to create their own infrastructure to support these fundamental application services but can include a few lines of code to take advantage of server resources from JavaScript code running in the browser, thereby dramatically extending the reach of ASP.NET to include both the client and the server. Moreover, because the JavaScript libraries are designed to be easy to use by developers already familiar with server-side .NET programming, all of this extra functionality comes in a friendly package that is easy to leverage.

The Dynamic HTML Component

Dynamic HTML is not a freestanding technology. It is the use of a set of technologies in a specific way. HTML is returned to the browser following a Web server request. The browser then renders the page, and the user is able to view it. The browser also exposes the DOM that represents the structure of the HTML being displayed. The DOM can be accessed from JavaScript embedded in, or referenced by, the page. The appearance of the HTML is affected by applying Cascading Style Sheets (CSS), which control colors, fonts, position, visibility, and more. You can bind JavaScript code to events that the browser will raise when users perform certain actions, such as hovering over a particular element or entering text in a textbox. The JavaScript code can update text or manipulate the CSS settings for elements within the page. It can also communicate with the server to expand the dynamic nature of the page even further. The user will see a dynamically changing user interface that responds to his actions in real time, which will greatly enhance his overall experience, thereby increasing his productivity and satisfaction with the application. This one piece of the pie is the one that changes the underlying page without the page refresh.

All these pieces together, in the end, provide you the tools that you need to build state-of-the-art applications for today's Web world.

AJAX Libraries

ASP.NET AJAX is the focus of this book. ASP.NET AJAX is an AJAX library that provides you with the tools and components that you need to tie all the varying aforementioned technologies together. Although this book is about ASP.NET AJAX in particular, there are many third-party AJAX libraries that can be used with ASP.NET, although not all of them were specifically designed for it.

Some of these AJAX libraries are mainly focused on providing JavaScript libraries for use from within the browser to make manipulation of the browser DOM easier. Others include some level of server functionality for use within ASP.NET pages (where server controls will render on the client side).

This section briefly highlights some of what these libraries offer. The ASP.NET AJAX Framework can coexist with script and controls from other libraries, although given the dynamic nature of the JavaScript language, it is possible to extend types so that they conflict with each other. Mixing and matching libraries might work just fine for many uses, but you might find conflicts in other cases. So

if you decide to move forward on this route, make sure that you are aware of the potential problems. Some of the available libraries include the following:

- **Ajax.NET Professional**: Michael Schwartz developed Ajax.NET Professional as a tool primarily used to simplify the data transport mechanism that enables a client JavaScript routine to communicate with a server program. The server code is simple to use. You merely need to register the control in your page and decorate some code-behind methods with attributes to designate which ones can be called from the client. Then you can leverage the script library to make the calls and pass data. This is intended for developers who are well versed in DHTML, and there are not many prebuilt visual controls. This is a lightweight solution with very little overhead in terms of bytes transferred and processing cycles needed on the client and server. The source code is available, and the package is free (`www.ajaxpro.info`).
- **Anthem.NET**: Anthem.NET is a SourceForge project where users are able to download the sources to the project. It targets ASP.NET 1.1 and ASP.NET 2.0. It has a set of server controls that use their underlying JavaScript library to communicate with the server. They provide the ability to access the state of controls on the page during an asynchronous callback. At the time of writing, the Anthem.NET Web page (`http://anthem-dot-net.sourceforge.net`) points out that the Anthem.NET user needs to be an experienced ASP.NET developer to get the most out of it. However, this is generally easier to use than Ajax.NET Professional, especially for developers who are not well versed in DHTML. This project is similar to ASP.NET AJAX in many ways but is not as comprehensive.
- **DoJo**: The DoJo toolkit can be found at `www.dojotoolkit.com`. It is a client-side library for AJAX development without ties to any server technology. DoJo has a type system for JavaScript and a function for binding script to events from JavaScript objects or DHTML elements. One of its strengths is rich support for dynamic script loading. You can specify dependencies and ordering in the way that scripts are retrieved and processed.
- **Prototype**: The Prototype script library is available at `www.prototypejs.org`. It does not target any server technology for integration. It has a type system for scripting in a more object-oriented way, along with some shortcut syntaxes for dealing with JavaScript arrays as well as accessing and manipulating HTML elements on the page. Prototype provides networking functionality and a method for automatically updating an HTML element with the results of an HTTP request when given a URL. The Prototype library also has functions for associating script objects and methods with DOM objects and events. The library is focused on simplifying tasks that can be cumbersome and tedious. It does not provide much help for producing a richer user interface but puts forth the building blocks for an improved Web-scripting experience.
- **Script.aculo.us**: The Script.aculo.us library can be found at the Web site of the same name, `http://script.aculo.us`. Their tagline is "it's about the user interface, baby!" which accurately describes their focus. `http://script.aculo.us` is built on top of the Prototype script library and picks up where it stops. It includes functionality for adding drag-and-drop support to an application. It has a lot of effects code for fading, shrinking, moving, and otherwise animating DOM elements. `http://script.aculo.us` also has a slider control and a library for manipulating lists of elements.
- **Rico**: The Rico library also builds on top of the Prototype script library. It has support for adding drag-and-drop behavior to browser DOM elements. It also has some controls to bind JavaScript objects to DOM elements for manipulating data. Rico has constructs for revealing and hiding portions of a page using an accordion style. It also has animation, sizing, and fading effects prebuilt for easier use. These UI helpers are available at `www.openrico.org`.

Although there are many options out there for ASP.NET developers, you will find that the toolkit defined in this book is the most integrated with ASP.NET and will be one of the more robust options in your arsenal.

Creating a Simple Web Page with AJAX

Although this book focuses on the power of using AJAX with ASP.NET 3.5 and the AJAX capabilities this framework brings to AJAX development, it is also interesting to see how you could build a simple AJAX-enabled HTML page without the use of any of the aforementioned frameworks.

To show this in action, create an ASP.NET solution within Visual Studio 2008. The first step you can take is to build the Web service that will return a value to be loaded up through the AJAX process. The code (in C#) of the Web service is presented here in Listing 1-1.

Listing 1-1: The Web service to communicate with the AJAX-enabled Web page

```
using System.Web.Services;

/// <summary>
/// Summary description for WebService
/// </summary>
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class WebService : System.Web.Services.WebService {

    [WebMethod]
    public string HelloWorld() {
        return "Hello Ajax World!";
    }
}
```

This is a simple C# Web service and as you can see from the single WebMethod, it only returns a string with no required parameters on the input. The AJAX page that you create for this example works with this Web service, and calls the Web service using the HTTP-GET protocol. By default, ASP.NET disallows HTTP-GET and you have to enable this capability in the `web.config` file in order for this example to work. Listing 1-2 shows the additions you need to make to this file.

Listing 1-2: Changing the web.config file in order to allow HTTP-GET

```
<system.web>
   <webServices>
      <protocols>
         <add name="HttpGet" />
      </protocols>
   </webServices>
</system.web>
```

Including the `<add>` element within the `<protocols>` element enables the HTTP-GET protocol so that your Web page will be able to make the AJAX calls to it. Finally, the HTML page that will make the AJAX call is represented here in Listing 1-3.

Listing 1-3: The HTML page that will make the AJAX calls to the Web service

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Basic Ajax Page</title>

    <script type="text/javascript" language="javascript">
       function InitiationFunction()
       {
        var xmlHttp;

        try
        {
            // Try this for non-Microsoft browsers
            xmlHttp = new XMLHttpRequest();
        }
        catch (e)
        {
            // These are utilized for Microsoft IE browsers
            try
            {
                xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
            }
            catch (e)
            {
                try
                {
                    xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
                }
                catch (e)
                {
                    alert("Your browser does not support AJAX!");
                    return false;
                }
            }
        }

    xmlHttp.onreadystatechange = function()
    {
      if(xmlHttp.readyState == 4 && xmlHttp.status == 200)
      {
        response  = xmlHttp.responseXML;
        document.forms[0].statement.value =
           response.getElementsByTagName("string")[0].firstChild.nodeValue;
      }
    }

    xmlHttp.open("GET","/BasicAjax/WebService.asmx/HelloWorld?",true);
    xmlHttp.send(null);
    }
    </script>

</head>
<body>
```

Continued

Listing 1-3: The HTML page that will make the AJAX calls to the Web service *(continued)*

```
    <form>
      <div>
          Statement from server:
          <input type="text" name="statement" />
          <br />
          <br />
          <input id="Button1" type="button" value="Button"
           onclick="InitiationFunction()" />
      </div>
      </form>
  </body>
  </html>
```

From this bit of code, you can see that the HTML page creates an instance of the `XMLHttpRequest` object. From there, a JavaScript function is assigned to the `onreadystatechange` attribute. Whenever the state of the `XMLHttpRequest` instance changes, the function is invoked. Since, for this example, you are interested only in changing the page after the page is actually loaded, you are able to specify when the actual function is utilized by checking the `readystate` attribute of the `XMLHttpRequest` object.

```
if(xmlHttp.readyState == 4 && xmlHttp.status == 200)
{

}
```

In this case, the `readystate` is checked to see if it equals a value of `4` and if the request status is equal to 200, a normal request was received. If the `status` attribute is equal to something other than `200`, a server-side error most likely occurred. A `readystate` value of `4` means that the page is loaded. The possible values of the `readystate` attribute include `0`, `1`, `2`, `3`, and `4`. A value of `0` (which is the default value) means that the object is uninitialized. A value of `1` means that the `open()` method was called successfully, a value of `2` means that the request was sent but that no response was yet received, a value of `3` means that the message is being received, and a value of `4` (again) means that the response is fully loaded.

The `open()` method called by the `XMLHttpRequest` object is quite simple, as it is an HTTP-GET request made to the specific URL of your service.

When you run this page and click the button found on the page, you are presented with results illustrated in Figure 1-3.

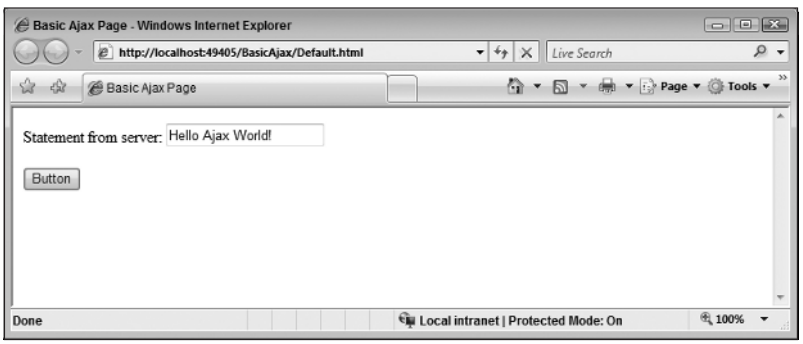

Figure 1-3

The Power of AJAX

Without the advanced use of JavaScript running in the browser, Web applications have their logic running on the server. This means many page refreshes for potentially small updates to the user's view. With AJAX, much of the logic surrounding user interactions can be moved to the client. This presents its own set of challenges. Some examples of using AJAX include streaming large datasets, managed entirely in JavaScript, to the browser. While JavaScript is a powerful language, the debugging facilities and options for error handling are very limited. Putting complex application logic on the client can take a lot of time, effort, and patience. AJAX as a whole allows you to naturally migrate some parts of the application processing to the client, while leveraging partial page rendering to let the server actually control some aspects of the page view.

Some Web sites make an application run entirely from a single page request, where JavaScript and AJAX will do a great deal of work. This presents some tough challenges. Users generally expect that the Back button will take them to the state of the application they were just viewing, but with AJAX applications, this is not necessarily the case. The client may be sending some information to the server for persistent state management (perhaps in server memory or a database), but this requires extra code and special attention to error handling and recovery.

The richest, most maintainable applications seem to be those that balance client and server resources to provide quick response times, easy access to server resources, and a minimum of blocking operations while new page views are fetched.

ASP.NET AJAX itself provides a mix of client and server programming features. The Microsoft AJAX library is aimed at client development. It provides a type system for an object-oriented approach to JavaScript development. It makes it easy to register code to respond to events. It provides useful functions to simplify common tasks like finding elements on the page, attaching event handlers, and accessing the server. The server features include functionality for managing JavaScript code to be sent to the client, declaring regions of the page to be updated asynchronously, creating timers for continuous updates, and accessing ASP.NET services such as user profile data and authentication.

Summary

The Web has evolved over the last decade from providing a static presence to being the default choice for developers writing applications. With Web applications, you get reach without having to deal with the deployment and servicing issues that accompany desktop applications. But the bar continues to move higher for Web applications as users come to expect more. AJAX technologies are driving Web applications to rival rich desktop apps. You can use the results of asynchronous communication with the Web server to update portions of the page without forcing the user to stop his or her work and wait for the page to post back and be repainted. Dynamic HTML allows you to create a rich GUI with transitions and animations leveraging CSS for colors, fonts, positioning, and more.

ASP.NET AJAX includes the Microsoft AJAX library, which makes writing browser-based JavaScript easier and simplifies many common client programming tasks. It is easy to attach code to DOM events, write JavaScript in an object-oriented way, and access the server for persistent storage, authentication, and updated data.

ASP.NET AJAX also includes extensions to version 2.0 of the .NET Framework that can greatly improve your Web application. There is built-in support for returning data in the JSON format that is easily consumed by JavaScript in the browser.

In this book, you will see how the client and server features of ASP.NET AJAX make it easier to push the limits of what you can do with a Web application! You learn how to update portions of a page asynchronously and how to manage the scripts that are used in the browser. You will find out how to use the networking facilities, with a dive into accessing ASP.NET services such as authentication and profile storage. You will get a closer look at the JavaScript language and how the Microsoft AJAX library builds on the language to simplify programming tasks. You will also see what ASP.NET AJAX offers for adding a richer UI to Web applications and look at how to debug and test Web applications.