Chapter 1

Building Blocks of AJAX-Style Web Applications

In this chapter:
The Paradigm Shift ......................................................... 4
The XmlHttpRequest Object .................................................. 9
Existing AJAX Frameworks in .NET ........................................... 19
Conclusion ................................................................. 31

Gone are the days when a Web application could be architected and implemented as a collection of related and linked pages. The advent of the so-called AJAX lifestyle is radically modifying the user’s perception of a Web application, and it is subsequently forcing developers to apply newer and richer models to the planning and implementation of modern Web applications. But what’s the AJAX lifestyle anyway?

AJAX is a relatively recent acronym, and it stands for Asynchronous JavaScript and XML. It is a sort of blanket term that indicates highly interactive and responsive Web applications. What’s the point here? Weren’t Web applications created about a decade ago specifically to be interactive, responsive, and deployed over a unique tool called the browser? So what’s new today?

The incredible success of the Internet has whetted people’s appetite beyond imagination. Over years, the users’ demand for ever more powerful and Web-exposed applications and services led architects and developers to incorporate more and more features into the server platform and the client browser. As a result, the traditional pattern of Web applications is becoming less adequate every day. A radical change in the design and programming model cannot be further delayed.

At the current state of the art, the industry needs more than just an improved and more powerful platform devised along the traditional guidelines and principles of Web applications—a true paradigm shift is required. AJAX is the incarnation of a new paradigm for the next generation of Web applications that is probably destined to last for the next decade.

From a more developer-oriented perspective, AJAX collectively refers to a set of development components, tools, and techniques for creating highly interactive Web applications. According to the AJAX paradigm, Web applications work by exchanging data rather than pages with the Web server. From a user perspective, this means that faster roundtrips occur and, more importantly, page loading and refresh is significantly reduced. A Web application tends to
look like a classic desktop Microsoft Windows application and has advanced features such as drag-and-drop and asynchronous tasks, a strongly responsive and nonflickering user interface, and other such features that minimize user frustration.

AJAX is the philosophy that has inspired a new generation of components and frameworks, each designed to target a particular platform, provide a given set of capabilities, and possibly integrate seamlessly with existing frameworks. Microsoft Atlas is the AJAX addition to the ASP.NET 2.0 platform.

Important Atlas is actually more than just an implementation of AJAX for the ASP.NET platform. As a matter of fact, AJAX inspired Atlas. However, today Atlas is a significant extension of the ASP.NET platform that brings in typical AJAX functionalities, plus new rich client capabilities. In the future, Atlas and ASP.NET 2.0 will merge into the new version of ASP.NET. In the end, AJAX and Atlas are not synonyms. Atlas, instead, implements the AJAX lifestyle for the ASP.NET platform. In doing so, Atlas goes far beyond the minimum set of typical AJAX functionalities and delivers a client framework for building Web applications that provide a rich user experience.

In this chapter, I’ll first review the basic system requirements common to all AJAX solutions and frameworks and then outline the Atlas roadmap to the next major release of Microsoft Visual Studio, codenamed “Orcas.” In the Orcas release of Visual Studio, Atlas will finally and officially integrate with ASP.NET and the rest of the Microsoft Web platform and application model.

The Paradigm Shift

We are all witnessing and contributing to an interesting and unique phenomenon—the Web is undergoing an epochal change right before our eyes as a result of our actions. As drastic as it might sound, the Web revolutionized the concept of an application. Only eight years ago, the majority of developers considered an application far too serious a thing to reduce it to an unordered mix of script and markup code. In the late 1990s, the cost of an application was sweat, blood, tears, and endless debugging sessions. There was neither honor nor fame for the “real” programmer in writing Web applications.

Note In the late 1990s, though, a number of Web sites were designed and built. Some of them grew incredibly in the next years to become pillars of the today’s world economy and even changed the way we do ordinary things. Want some examples? Google, Amazon, eBay. Nonetheless, a decade ago the guys building these and other applications were sort of avant-garde developers or smart and game amateurs.

Since then, the Web has evolved significantly. And although 10 years of Web evolution has resulted in the building of a thick layer of abstraction on the server side, it hasn’t changed the basic infrastructure—HTTP and pages.
The original infrastructure—one that was simple, ubiquitous, and effective—was the chief factor for the rapid success of the Web model of applications. The next generation of Web applications will still be based on HTTP and pages. However, the contents of pages and the capabilities of the server-side machinery will change to provide a significantly richer user experience—as rich as that of classic desktop Windows applications.

**Web Applications Today**

Today Web applications work by submitting user-filled forms to the Web server and displaying the markup returned by the Web server. The browser-to-server communication employs classic HTTP. As is widely known, HTTP is stateless, which means that each request is not related to the next and no state is automatically maintained. (The state objects we all know and use in, say, ASP.NET are nothing more than an abstraction provided by the server programming environment.)

Communication between the browser and the Web server occurs through forms. From a user’s perspective, the transition occurs through pages. Each user action that originates a new request for the server results in a brand new page (or a revamped version of the current page) being downloaded and displayed.

Let’s briefly explore this model a bit further to pinpoint its drawbacks and bring to the surface the reasons why a new model is needed today.

**Send Input via Forms**

Based on the URL typed in the address bar, the browser displays a page to the user. The page is ultimately made of HTML markup and contains one or more HTML forms. The user enters some data, and then instructs the browser to submit the form to an action URL.

The browser resolves the specified URL to an IP address and opens a socket. An HTTP packet travels over the wire to the given destination. The packet includes the form and all its fields. The request is captured by the Web server and typically forwarded to an internal module for further processing. At the end of the process, an HTTP response packet is prepared and the return value for the browser is inserted in the body.

**Get Output Through Pages**

When a request is made for, say, an .aspx resource, the Web server passes it on to ASP.NET for processing and receives the resulting HTML markup in return. The generated markup comprises all the tags of a classic HTML page, including `<html>`, `<body>`, and `<form>`. The page source is embedded in the HTTP response and tagged with a MIME type to instruct the browser how to handle it. The browser looks at the MIME type and decides what to do.

If the response contains an HTML page, the browser replaces the current contents entirely with the new chunk of markup. While the request is being processed on the server, the “old”
page is displayed to the client user. As soon as the “new” page is downloaded, the browser clears the display and renders the page.

Capabilities and Drawbacks

This model was just fine in the beginning of the Web age when pages contained little more than formatted text, hyperlinks, and some images. The success of the Web has prompted users to ask for more and more powerful features, and it has led developers and designers to create more sophisticated services and graphics. As an example, consider advertising. Today, most pages—and often even very simple pages, such as blog pages—include ad rotators that download quite a bit of stuff on the client.

As a result, pages are heavy and cumbersome—even though we still insist on calling them “rich” pages. Regardless of whether they’re rich or just cumbersome, these are the Web pages of today’s applications. Nobody really believes that we’re going to return to the scanty and spartan HTML pages of a decade ago.

Given the current architecture of Web applications, each user action requires a complete redraw of the page. Subsequently, richer and heavier pages render out slowly and produce a good deal of flickering. Projected to the whole set of pages in a large, portal-like application, this mechanism is just perfect for unleashing the frustrations of the poor end user.

Although a developer can build a page on the server using one of many flexible architectures (ASP.NET being one such example), from the client-side perspective Web pages were originally designed to be mostly static and unmodifiable. In the late 1990s, the introduction of Dynamic HTML first, and the advent of a World Wide Web Consortium (W3C) standard for the page document object model later, changed this basic fact. Today, the page object displayed by the browser can be modified to incorporate changes made entirely on the client-side to react to user inputs. (As we’ll see, AJAX and Atlas take this concept even further.)

Dynamic HTML is a quantum leap, but alone it is not enough yet to evolve the Web.

Web Applications Tomorrow

To minimize the impact of page redraws, primitive forms of scripted remote procedure calls (RPC) appeared around 1997. Microsoft, in particular, pioneered this field with a technology called Remote Scripting (RS).

RS employed a Java applet to pull in data from a remote URL. The URL exposed a contracted programming interface and serialized data back and forth through strings. On the client, a little JavaScript framework received data and invoked a user-defined callback to update the user interface via Dynamic HTML or similar techniques. RS worked on both Internet Explorer 4.0 and Netscape Navigator 4.0 and newer versions.

Later on, Microsoft replaced the Java applet with a COM object named XmlHttpRequest and released most of the constraints on the programming interface exposed by the remote URL.
At the same time, community efforts produced a range of similar frameworks aimed at taking RS to the next level and building a broader reach for solutions. The Java applet disappeared and was replaced by the XMLHttpRequest object.

**What’s AJAX, Anyway?**

The term AJAX was coined in 2004. It originated in the Java community and was used in reference to a range of related technologies for implementing forms of remote scripting. Today, any form of remote scripting is generally tagged with the AJAX prefix. Modern AJAX-based solutions for the Windows platform are based on the XMLHttpRequest object.

Two contrasting factors coexist in an AJAX application. On one hand, you need to serve users fresh data retrieved on the server. On the other hand, you need to integrate new data in the existing page without a full page refresh.

Browsers generally place a new request when an HTML form is submitted either via client-side script or through a user action such as a button click. When the response is ready, the browser replaces the old page with the new one. Figure 1-1 illustrates graphically the traditional way it works.

![Diagram](image)

**Figure 1-1** Browsers submit an HTML form and receive a new page to display

The chief factor that enables remote scripting is the ability to issue out-of-band HTTP requests. In this context, an out-of-band call indicates an HTTP request placed using a component that is different from the built-in module that handles the HTML form submission (that is, outside the traditional mechanism you see in Figure 1-1). The out-of-band call is triggered by an HTML page event and is served by a proxy component. In the most recent AJAX solutions, the proxy component is the XMLHttpRequest object; the proxy component was a Java applet in the very first implementation of RS.

**Update Pages via Script**

The proxy component (for example, the XMLHttpRequest object) sends a regular HTTP request and waits, either synchronously or asynchronously, for it to be fully served. When the
response data is ready, the proxy invokes a user-defined JavaScript callback to refresh any portion of the page that needs updating. Figure 1-2 provides a graphical overview of the model.

![Diagram of Out-of-band HTTP request (AJAX)](image)

**Figure 1-2** Out-of-band calls are sent through a proxy component, and a JavaScript callback is used to update any portion of the page affected by returned data

All browsers know how to replace an old page with a new page; not all of them, though, provide an object model that represents the current contents of the page. For browsers that supply an updatable object model for HTML pages, the JavaScript callback function can refresh specific portions of the old page, thus making them look updated, without a full reload.

**The Document Object Model**

The Document Object Model (DOM) is the specification that defines a platform- and language-neutral interface for accessing and updating the contents, structure, and style of HTML and XML documents. A recognized standard ratified by the W3C committee, the DOM is now supported by a growing number of browsers. The DOM provides a standard set of objects for representing the constituent elements of HTML and XML documents. All together, these objects form a standard interface for accessing and manipulating child elements of HTML pages and, more in general, XML documents.

It is essential to note that although the first working frameworks for remote scripting date back to a decade ago, the limited support browsers have had for dynamic changes in displayed documents slowed down the adoption of such technologies in the industry. Until now.

**The Role of Rich Browsers**

As shown in Figure 1-2, the AJAX model has two key requirements as far as browsers are concerned: a proxy component and an updatable DOM. For quite a long time, only high-end browsers (also known as rich, uplevel browsers) provided support for both features. In the past few years, only companies that could exercise strict control over the capabilities of the client browsers were able to choose the AJAX model for their sites. For too long, a rich browser also has meant a browser with too limited reach. And using such a browser is definitely a bad choice for most businesses.
Rich vs. Reach

Perhaps due to a rare and likely unrepeatable astral conjunction, today about 90 percent of browsers available on the market happen to have built-in support for the advanced capabilities that the AJAX model requires. Internet Explorer since version 5.0, Firefox, Netscape from version 6 and onwards, Safari 1.2 and newer releases, Opera 8.0, and a variety of mobile devices are all browsers that fully support the AJAX programming model.

For the very first time, a rich browser is not synonymous with a limited reach browser. Designing highly interactive Web applications that implement remote scripting techniques is no longer an impossible dream to chase but a concrete opportunity to seize.

Each platform and each vendor might have a particular framework and tool set to offer, but this doesn’t change the basic fact—the AJAX lifestyle is now possible on 90 percent of the browsers available today. It’s a real breakthrough and it is now possible to build and distribute applications that were not possible before.

Required Capabilities

What are exactly the capabilities required from a browser to run AJAX functionalities? As mentioned, a browser needs to provide two key capabilities: a proxy mechanism to make client code able to place out-of-band HTTP calls, and an updatable DOM.

There’s a W3C ratified standard for the updatable DOM. A W3C standard for the proxy component is currently being developed. It takes the form of the XmlHttpRequest object and is devised as an interface exposed by the browser to allow script code to perform HTTP client functionality, such as submitting form data or loading data from a remote Web site. The latest working draft is available at http://www.w3.org/TR/XMLHttpRequest.

In addition, browsers must support JavaScript and preferably cascading style sheets (CSS).

In the end, the AJAX lifestyle is possible and affordable for virtually every developer and nearly 90 percent of the Web audience, regardless of the platform. The tools required to make AJAX work are becoming as ubiquitous as HTML/XML parsers, HTTP listeners, and JavaScript processors. To paraphrase the payoff of a popular advertising campaign, I’d say that “AJAX is now.” And as far as the Windows and ASP.NET platform is concerned, AJAX takes the form of Microsoft Atlas. (But, again, ASP.NET Atlas is more than just the implementation of typical AJAX functionalities such as out-of-band calls and refreshable pages. We’ll discover this in the upcoming chapters, starting with Chapter 4, “Atlas Controls, Extenders, and Behaviors.”)

The XmlHttpRequest Object

Originally introduced with Internet Explorer 5.0, the XmlHttpRequest object is an internal object that the browser publishes to its scripting engine. In this way, the script code found in any client page—typically, JavaScript code—can invoke the object and take advantage of its functionality.
Part I   Microsoft Atlas Overview

Functionally speaking, and despite the XML in the name, the XMLHttpRequest object is nothing more than a tiny object model to place HTTP calls via script in a non browser-led way. When users click the submit button of a form, or perform any action that ends up invoking the submit method on the DOM’s form object, the browser kicks in and takes full control of the subsequent HTTP request. From the user’s perspective, the request is a black box whose only visible outcome is the new page being displayed. The client script code has no control over the placement and outcome of the request.

The XMLHttpRequest object allows script code to send HTTP requests and handle their response.

Motivation and Background

Created by Microsoft and soon adopted by Mozilla, the XMLHttpRequest object is today fully supported by the majority of Web browsers. As you’ll see in a moment, the implementation can significantly differ from one browser to the next, even though the top-level interface is nearly identical. For this reason, a W3C committee is at work with the goal of precisely documenting a minimum set of interoperable features based on existing implementations.

Note   The XMLHttpRequest object originally shipped as a separate component with Internet Explorer 5.0 back in the spring of 1999. It is a native component of all Microsoft operating systems that have shipped since. In particular, you’ll certainly find it installed on all machines that run Windows 2000, Windows XP, and newer operating systems.

The Internet Explorer Object

When the XMLHttpRequest object was first released, the Component Object Model (COM) was ruling the world at Microsoft. The extensibility model of products and applications was based on COM and implemented through COM components. In the late 1990s, it was the right and natural choice to implement this new component as a reusable automation COM object, named Microsoft.XMLHttp.

Various versions of the same component (even with slightly different names) were released in the past years, but all of them preserved the original component model—COM. Internet Explorer 6.0, for example, ships the XMLHttpRequest object in the form of a COM object. Where’s the problem?

COM objects are external components that require explicit permission to run inside of a Web browser. In particular, to run the XMLHttpRequest object and subsequently enable any AJAX functionality built on top of it, at a minimum a client machine needs to accept ActiveX components marked safe for scripting. (See Figure 1-3.)
The XMLHttpRequest object is certainly a safe component, but to enable it users need to lower their security settings and accept any other component “declared” safe for scripting that is around the Web sites they visit.

**Important** The internal implementation of XMLHttpRequest is disjointed from the implementation of any AJAX-like frameworks such as Microsoft Atlas. Under the hood, any framework ends up calling the object as exposed by, or available in, the browser.

### The Mozilla Counterpart

Mozilla adopted XMLHttpRequest immediately after its first release with Internet Explorer 5.0. However, in Mozilla-equipped browsers the XMLHttpRequest object is part of the browser’s object model and doesn’t rely on external components. Put another way, a Mozilla browser such as Firefox publishes its own XMLHttpRequest object into the scripting engine and never uses the COM component, even when the COM component is installed on the client machine and is part of the operating system. Figure 1-4 shows the different models in Internet Explorer (up to version 6.0) and Mozilla browsers.

As a result, in Mozilla browsers XMLHttpRequest looks like a native JavaScript object and can be instantiated through the classic `new` operator:

```javascript
// The object name requires XML in capital letters
var proxy = new XMLHttpRequest();
```
When the browser is Internet Explorer, the `XmlHttp` object is instantiated using the `ActiveXObject` wrapper, as shown here:

```javascript
var proxy = new ActiveXObject("Microsoft.XMLHttp");
```

Generally, AJAX-style frameworks check the current browser and then decide about the route to take.

Needless to say, as implemented in Mozilla browsers the `XmlHttp` functionality is somewhat safer, at least in the sense it doesn’t require users to change their security settings for the browser.

**XmlHttp As a Native Object in Internet Explorer 7**

Implemented as a COM component for historical reasons on Internet Explorer browsers, the `XmlHttp` object will finally become a browser object with Internet Explorer 7.0. All potential security concerns will be removed at the root, and AJAX frameworks could be updated to use the same syntax for creating the `XmlHttp` object regardless of the browser:

```javascript
var proxy = new XMLHttpRequest();
```

In addition, this change in Internet Explorer 7.0 will completely decouple AJAX-like functionality (for example, functionality found in Microsoft Atlas) from an ActiveX enabled environment.
An HTTP Object Model

I spent quite a few words on the XMLHttpRequest object and its expected behavior, but I still owe you a practical demonstration of the object’s capabilities. In this section, I’ll cover the members of the component, the actions it can perform, and details of the syntax.

As mentioned, the XML in the name of the component means little and in no way limits the capabilities of the component. In spite of the XML prefix, you can use the object as a true automation engine for executing and controlling HTTP requests, from client code generated by ASP.NET pages or the Windows shell, or Visual Basic 6.0 or C++ unmanaged applications. Using the XMLHttpRequest COM object from within .NET applications is nonsensical, as you can find similar functionality in the folds of the System.Net namespace in the .NET Framework.

**Important** If you’re going to use Microsoft Atlas or any other AJAX-like framework for building your applications, you’ll hardly hear about the XMLHttpRequest object and let alone use it in your own code. Microsoft Atlas completely encapsulates this object and shields page authors and application designers from it. You don’t need to know about XMLHttpRequest to write great Atlas applications, no matter how complex and sophisticated they are. However, knowing the fundamentals of XMLHttpRequest can lead you to a better and more thorough understanding of the platform and to more effective diagnoses of problems.

Behavior and Capabilities

The XMLHttpRequest object is designed to perform one key operation: sending a HTTP request. The request can be sent either synchronously or asynchronously. The following listing shows the programming interface of the object as it appears in the W3C working draft at the time of this writing:

```csharp
interface XMLHttpRequest {
    function onreadystatechange;
    readonly unsigned short readyState;
    void open(string method, string url);
    void open(string method, string url, bool async);
    void open(string method, string url, bool async, string user);
    void open(string method, string url, bool async,
              string user, string pswd);
    void setRequestHeader(string header, string value);
    void send(string data);
    void send(Document data);
    void abort();
    string getAllResponseHeaders();
    string getResponseHeader(string header);
    string responseText;
    Document responseXML;
    unsigned short status;
    string statusText;
};
```
Using the component is a two-step operation. First, you open a channel to the URL and specify the method (GET, POST, or other) to use and whether you want the request to execute asynchronously. Next, you set any required header and send the request. If the request is a POST, you pass to the send method the body of the request.

The send method returns immediately in case of an asynchronous operation. You write an onreadystatechange function to check the status of the current operation and figure out when it is done.

Sending a Request

Most AJAX frameworks obtain an instance of the XMLHttpRequest object for the current browser using code that looks like the following:

```javascript
var xmlRequest, e;
try {
    xmlRequest = new XMLHttpRequest();
} catch(e) {
    try {
        xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
    } catch(e) {
    }
}
```

The code first tries to instantiate the internal XMLHttpRequest object and opts for the ActiveXObject in case of failure. As you can see, the creation of the object requires an exception to be caught when the browser is Internet Explorer. Such a code will work unchanged (and won’t require any exception) in Internet Explorer 7.0.

The open method prepares the channel for the request; no physical socket is created yet at this point, though. To execute a POST statement, you need to add the proper content-type header. The Boolean argument indicates whether the operation is asynchronous:

```javascript
xmlRequest.open("POST", url, false);
xmlRequest.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
xmlRequest.send(postData);
```

The send method opens the socket and sends the packet. In the preceding code snippet, the method returns only when the response has been fully received.

An asynchronous request requires slightly different code:

```javascript
xmlRequest.open("POST", url, true);
xmRequest.onreadystatechange = CallbackComplete;
xmlRequest.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
xmRequest.send(postData);
```
The *CallbackComplete* element is a placeholder for a JavaScript function that retrieves and processes the response generated by the request.

Note that the function assigned to the *onreadystatechange* member will be invoked whenever *readyState* changes value. Possible values for the state are the integers ranging from 0 through 4, which mean “Uninitialized”, “Open method called successfully”, “Send method called successfully”, “Receiving data”, “Response received”, respectively. The *CallbackComplete* framework-specific function will generally check that state and proceed.

**Receiving Response**

The response of the request is available in two formats—raw text and as an XML document. The *responseText* property is empty if the state is 0 through 2—that is, no data has been received yet. When the state transitions to 3 (receiving data), the property contains the data received so far, interpreted using the character encoding specified in the response. If no character encoding was specified, it employs UTF-8.

The *responseXML* property is not available until the full response has been downloaded and successfully parsed to an XML document. If the body of the response is not XML or if the parsing fails for any reason, the property returns null. It is important to note that the construction of the XML document takes place on the client once the raw HTTP response has been fully received.

**Using the *XmlHttpRequest* Object**

As mentioned, you don’t need to use the *XmlHttpRequest* object in your AJAX-based application, regardless of the framework (for example, Atlas) you end up using. For completeness, though, let’s briefly review the steps required to use the object in a sample ASP.NET 2.0 page. The same code can be used with ASP.NET 1.x as well.

**Executing an Out-of-Band Call from an ASP.NET Page**

Web pages that shoot out-of-band calls need to have one or more trigger events that, when properly handled with a piece of JavaScript code, place the request via the *XmlHttpRequest* object. Trigger events can only be HTML events captured by the browser’s DOM implementation.

The JavaScript code should initiate and control the remote URL invocation, as shown in the following code:

```javascript
<script type="text/javascript">
    function SendRequest(url, params)
    {
        // Add some parameters to the query string
        var pageUrl = url + '?outofband=true&param=' + params;
```
Part I  Microsoft Atlas Overview

    // Initialize the XMLHttpRequest object
    var xmlRequest, e;
    try {
        xmlRequest = new XMLHttpRequest();
    }
    catch(e) {
        try {
            xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
        }
        catch(e) { }
    }

    // Prepare for a POST synchronous request
    xmlRequest.open(“POST”, pageUrl, false);
    xmlRequest.setRequestHeader("Content-Type",
        “application/x-www-form-urlencoded”);
    xmlRequest.send(null);
    return xmlRequest;
</script>

The sample function accepts two strings—the URL to call and the parameter list. Note that the format of the query string is totally arbitrary and can be adapted at will in custom implementations. The URL specified by the programmer is extended to include a couple of parameters. The first parameter—named outofband in the example—is a Boolean value and indicates whether or not the request is going to be a custom callback request. By knowing this, the target page can process the request appropriately. The second parameter—named param in the example—contains the input parameters for the server-side code.

The host page looks like the following code snippet:

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Testing Out-of-band</title>
</head>
<body>
<form id="Form1" runat="server">
    <h1>Demonstrate Out-of-band Calls</h1>
    <h2>Request.Url</h2>
    <hr />

    <asp:DropDownList runat="server" ID="EmployeeList" />
    <input id="Button1" type="button" value="Go Get Data" onclick="MoreInfo()" />
    <hr />
    <span id="Msg" />
</form>
</body>
</html>
The code-behind class is shown in the following listing:

```csharp
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (IsOutOfRange())
            return;
        if (!IsPostBack)
            PopulateList();
    }

    private bool IsOutOfRange()
    {
        if (Request.QueryString["outofband"] != null)
        {
            string param = Request.QueryString["param"].ToString();
            Response.Write(ExecutePageMethod(param));
            Response.Flush();
            Response.End();
            return true;
        }
        return false;
    }

    private void PopulateList()
    {
        SqlDataAdapter adapter = new SqlDataAdapter(
            "SELECT employeedid, lastname FROM employees",
            "SERVER=(local);DATABASE=northwind;UID=...;");
        DataTable table = new DataTable();
        adapter.Fill(table);
        EmployeeList.DataTextField = "lastame";
        EmployeeList.DataValueField = "employeedid";
        EmployeeList.DataSource = table;
        EmployeeList.DataBind();
    }

    string ExecutePageMethod(string eventArgument)
    {
        return "You clicked: " + eventArgument;
    }
}
```

A couple of issues deserve more attention and explanation. The first one is the need to find out whether the request is an out-of-band call or a regular postback. Next, we need to look at the generation of the response. The IsOutOfRange method checks the outofband field in the posted form. If the outofband field is found, the request is served and terminated without going through the final part of the classic ASP.NET request life cycle—events for changed values, postback, pre-rendering, view-state serialization, rendering. An out-of-band request is therefore short-circuited to return more quickly, carrying the least data possible.
Part I  Microsoft Atlas Overview

What does the page do when an out-of-band call is placed? How does it determine the response? Most of the actual AJAX-based frameworks vary on this point, so let's say it is arbitrary. In general, you need to define a public programming interface that is invoked when an out-of-band call is made. In the sample code, I have a method with a contracted name and signature—ExecutePageMethod—whose output becomes the plain response for the request. In the sample code, the method returns and accepts a string, meaning that any input and output parameters must be serializable to a string.

```c#
string param = Request.QueryString["param"].ToString();
Response.Write(ExecutePageMethod(param));
Response.Flush();
Response.End();
```

As in the code snippet, the response for the out-of-band request is the output of the method. No other data is ever returned; and no other data except for the parameters is ever sent. In this particular implementation, there will be no view state sent and returned.

---

**Important**  Although you'll probably never get to write any such code, be aware that thus far I just provided a minimal but effective description of the underlying mechanism common to most frameworks that supply AJAX-like functionality. Each framework (for example, Atlas, AJAX.NET) encapsulates a good number of details and add new services and capabilities. At its core, though, this is how AJAX libraries work.

---

**Displaying Results**

One more step is missing—what happens on the client once the response for the out-of-band call is received? The following snippet shows a piece of client code that, when attached to a button, fires the out-of-band call and refreshes the user interface:

```javascript
function MoreInfo() {
    var empID = document.getElementById("EmployeeList").value;
    var xml = SendRequest("default.aspx", empID);

    // Update the UI
    var label = document.getElementById("Msg");
    label.innerHTML = xml.responseText;
}
```

According to the code, whenever the user clicks the button a request is sent at the following URL. Note that 3 in the sample URL indicates the ID of the requested employee. (See Figure 1-5.)

```
default.aspx?outofband=true&param=3
```
Displaying results correctly on most browsers can be tricky. Internet Explorer, in fact, supports a number of nonstandard shortcuts in the DOM that just don't work with other browsers. The most common snag is retrieving references to HTML elements using the `document.getElementById` method instead of the direct name of the element. For example, the following code works on Internet Explorer but not Firefox and other Mozilla-equipped browsers:

```javascript
// Msg is the ID of a <span> tag.
// This code works only with IE
Msg.innerHTML = xml.responseText;
```

In summary, cross-browser JavaScript code is required to update the currently displayed page on the client. At the same time, a number of assumptions must be made on the server to come up with a working and effective environment. For this reason, frameworks are the only reasonable way of implementing AJAX functionalities. Different frameworks, though, might provide a different programming interface on top of an engine that uses the same common set of parts.

### Existing AJAX Frameworks in .NET

Today, quite a few APIs exist to implement AJAX functionality in ASP.NET, and one of these API is already integrated into ASP.NET 2.0. Other APIs come from third-party vendors or take form from open-source projects. In the final part of the chapter, I'll briefly look at some of these API. It is essential to note, though, that as long as ASP.NET is your development environment, the most reasonable choice you can make is Microsoft Atlas.

Although it's also available as a separate add-on for ASP.NET 2.0, Microsoft Atlas is expected to be part of one of the next versions of ASP.NET—code-named “Orcas.” Atlas is definitely the next big step in the evolution of ASP.NET. Atlas originates from and significantly expands a
couple of AJAX-based APIs available for ASP.NET—ASP.NET Script Callback and AJAX.NET (now BorgWorX AJAX.NET).

At the time of this writing, other companies (for example, Telerik and ComponentArt) have integrated AJAX functionality into their existing suite of controls for rapid and rich Web development. Telerik in particular also released a service pack for its suite that makes most components compatible with Atlas.

The rest of the chapter outlines the way to use Atlas as it is coming out for the ASP.NET platform. I’ll first present ASP.NET Script Callback, a native part of ASP.NET 2.0 and measure it against the BorgWorX AJAX.NET library, an open-source library. It should be clear by then that to write great applications that provide users with a rich experience you need a framework designed for the AJAX lifestyle—and it’s even better if this AJAX framework is well integrated with the underlying platform.

If you write code using ASP.NET components, Microsoft Atlas is where you should land. If you rely on the services of an ASP.NET-powered suite of controls, you should expect to find in the next release of the suite integrated support for Atlas functionality. Either way, there’s a lot of Atlas in the future of Web development on the ASP.NET platform.

**ASP.NET Script Callbacks**

ASP.NET 2.0 contains a native API, named ASP.NET Script Callback, to implement out-of-band calls to the same URL of the current page. This API makes the out-of-band request look like a special-case page request. It transmits the view state along with original input fields. A few additional input fields are inserted in the body of the request to carry extra information. Once on the server, the request passes through the regular pipeline of HTTP modules and raises the expected sequence of server-side events up to the pre-rendering stage.

Just before the pre-rendering stage, the page method is executed, the return value is serialized to a string, and then the string is returned to the client. No rendering phase ever occurs, and the view state is not updated and serialized back.

**The Client-Side Code**

An ASP.NET out-of-band call begins with the execution of a system-provided JavaScript function. This function is generally bound to the handler of a page-level event such as a client button click or a change event in a drop-down list. To illustrate the process, let’s consider a page with a button that retrieves additional information about the selected employee.

Imagine a Web page that shows information about a specific employee. Users pick up a user name from a list and click a button to find out more about the user. For the out-of-band mechanism to work, you must make sure that the button is not a submit button; if it is, a regular postback occurs with a full refresh of the page. Let’s work with the following markup:

```html
<asp:dropdownlist id="cboEmployees" runat="server"
DataTextField="Lastname" DataValueField="employeeid" />
```
The user picks up an element from the list and clicks the button to get more details. The button's `onclick` event handler is added dynamically as the page is processed on the server. The `Page_Load` event is the right place for all this to happen:

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        // Populate the drop-down list
        cboEmployees.DataSource = GetEmployeeList();
        cboEmployees.DataBind();

        // Ask ASP.NET to build a string with the Javascript
        // function to call in order to start the out-of-band call
        string rpc = ClientScript.GetCallbackEventReference(
            this,
            "document.forms[0].elements['cboEmployees'].value",
            "UpdatePage",
            "null",
            "null",
            false);

        // Trigger the out-of-band call through the button
        string js = String.Format("javascript:{0}", rpc);
        buttonTrigger.Attributes["onclick"] = js;
    }
}

private Collection<Employee> GetEmployeeList()
{
    //
}
```

ASP.NET Script Callback provides its own JavaScript API to wrap any needed calls to `XmlHttp-Request`. As a developer, you are not required to know about this API in detail. As a developer, instead, you should uniquely focus on the programming interface of the `GetCallbackEventReference` method of the `Page.ClientScript` object. You provide this method with the parameters to use in the call, and it returns to you a string with the JavaScript source code to embed in the page.

When the page is served to the user, the button that will trigger the out-of-band call is bound to the following script:

```html
<input name="buttonTrigger" type="button" id="buttonTrigger"
    value="More Info"
    onclick="javascript:WebForm_DoCallback('__Page',
        document.forms[0].elements['cboEmployees'].value,
        UpdatePage,
        null, null, false)" />
```
Where’s `WebForm_DoCallBack` defined? The script is stored in the resources of the `system.web` assembly and is linked to the page through a dynamically generated script-include statement:

```html
<script src="/ProAspNetAdv/WebResource.axd?d=...&t=" type="text/javascript"></script>
```

In the page served to the user, the preceding element appears immediately after the opening `<form>` tag. Needless to say, `WebForm_DoCallBack` calls the `XmlHttpRequest` object and manages the HTTP request.

**The `GetCallbackEventReference` Method**

The arguments passed to the `GetCallbackEventReference` method determine how the underlying callback machinery is invoked and works:

```csharp
public string GetCallbackEventReference(
    Control target,
    string argument,
    string clientCallback,
    string context,
    string clientErrorCallback,
    bool useAsync)
```

The role of each parameter is detailed in Table 1-1.

**Table 1-1 Parameters of `GetCallbackEventReference`**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>target</code></td>
<td>Indicates the server-side object that will handle the call. The object can be the ASP.NET page or any of the server controls defined in the page. The target object must implement the <code>ICallbackEventHandler</code> interface. If the target object doesn’t implement the required interface, an exception is thrown.</td>
</tr>
<tr>
<td><code>argument</code></td>
<td>Indicates the input parameters for the method to be invoked on the server in response to the call. If multiple values are to be passed, it is up to developers to pack all of them into a single string. The contents of this parameter are passed verbatim to the server method serving the request.</td>
</tr>
<tr>
<td><code>clientCallback</code></td>
<td>Indicates the name of the client-side JavaScript function that will receive the return value of the server call. Whatever the server method returns—even a string—is passed verbatim to this JavaScript function. If this function is not defined or available on the client page, a script error occurs.</td>
</tr>
<tr>
<td><code>context</code></td>
<td>Client-side script that is evaluated on the client prior to initiating the call-back. The result of the script is passed back to the client-side event handler.</td>
</tr>
<tr>
<td><code>clientErrorCallback</code></td>
<td>The name of the client-side JavaScript function that receives the result when an error occurs in the server-side method. In this case, the <code>clientCallback</code> is not called.</td>
</tr>
<tr>
<td><code>useAsync</code></td>
<td>If set to <code>true</code>, the server call is performed asynchronously; otherwise, it occurs synchronously. It is <code>false</code> by default.</td>
</tr>
</tbody>
</table>
Armed with this background information, let’s return to the code that we briefly looked at earlier:

```csharp
// Called from within Page_Load
string rpc = ClientScript.GetCallbackEventReference(
    this,
    "document.forms[0].elements['cboEmployees'].value",
    "UpdatePage",
    "null",
    "null",
    false);

The target of this call is the page, meaning that the code-behind class of the page must implement ICallbackEventHandler. The argument passed to the methods of the interface is the result of the specified JavaScript expression:

document.forms[0].elements['cboEmployees'].value

Note that server control properties are not guaranteed to contain fresh values updated with the latest changes occurred on the client. To make sure that up-to-date values are passed from the client to the server, you must specify a JavaScript expression that retrieves those values explicitly.

The ICallbackEventHandler Interface

If correctly set up, the client code of a page that uses ASP.NET Script Callback has a system-provided piece of JavaScript code bound to an HTML event. This JavaScript code uses XMLHttpRequest internally to place an out-of-band call. The JavaScript code also references another piece of JavaScript used to update the page with the results generated on the server. But what happens on the server when the secondary request is made? Which page method is ever executed?

ASP.NET Script Callback defines an interface—the ICallbackEventHandler interface—that any server object being the target of an out-of-band call can implement. The target of the out-of-band call can be either the page or any of its child controls. The interface has two methods, as shown here:

```csharp
public interface ICallbackEventHandler
{
    string GetCallbackResult();
    void RaiseCallbackEvent(string eventArgument);
}
```

The execution of an out-of-band call is divided into two steps: preparation and results generation. The RaiseCallbackEvent method is invoked first to prepare the remote code execution. The GetCallbackResult method is invoked later in the request life cycle when it is time for the ASP.NET runtime to generate the response for the browser. Conventionally, in the RaiseCallbackEvent method you cache the argument in an internal member and execute any significant code in GetCallbackResult. This is just a common practice though. The following listing shows
how to retrieve and serve information to the client:

```csharp
private int _empID;
void ICallbackEventHandler.RaiseCallbackEvent(string argument)
{
    Int32.TryParse(argument, out _empID);
}

string ICallbackEventHandler.GetCallbackResult()
{
    // Get more info about the specified employee
    EmployeeInfo emp = GetEmployeeDetails(_empID);

    // Prepare the response for the client: a comma-separated string.
    // The format of the return string is up to you. This format must
    // be known to the Javascript callback
    string[] buf = new string[6];
    buf[0] = emp.ID.ToString();
    buf[1] = emp.FirstName;
    buf[2] = emp.LastName;
    buf[3] = emp.Title;
    _results = String.Join(",", buf);
    return;
}
```

The data to be returned must be represented as a string. If the original data is a managed object, you will serialize it to a string. On the client, then, the string will be appropriately parsed to extract any significant information.

**Putting It All Together: ASP.NET Script Callbacks**

The `GetCallbackResult` method is expected to return a string that represents the response of the request. The returned string is passed to a JavaScript function that will update the page. A sample client callback is shown here:

```javascript
<script language="javascript">
function UpdateEmployeeInfo(response, context)
{
    // Deserialize the response to an array of strings
    var o = response.split(',');

    // Get references to page elements to update
    var e_ID = document.getElementById("e_ID");
    var e_FName = document.getElementById("e_FName");
    var e_LName = document.getElementById("e_LName");
    var e_Title = document.getElementById("e_Title");
    var e_Country = document.getElementById("e_Country");
    var e_Notes = document.getElementById("e_Notes");

    // Update the page elements with data from the server
    e_ID.innerHTML = o[0];
    e_FName.innerHTML = o[1];
</script>
```
e_LName.innerHTML = o[2];
e_Title.innerHTML = o[3];
e_Country.innerHTML = o[4];
e_Notes.innerHTML = o[5];
</script>

The page author is responsible for having a similar function in the client page.

**Key Facts**

All in all, the programming interface of ASP.NET Script Callback is a bit clumsy, and although it shields developers from a lot of internal details it still requires good JavaScript skills and is articulated in a bunch of boilerplate server code. You need server code to bind HTML elements to client-side event handlers, and you need ad-hoc server code to publish a programming interface that is callable from the client.

Each request carries with it a copy of the original view state and rebuilds the last known good state on the server. In other words, the original value of all input fields in the currently displayed page (regardless of any changes entered before the out-of-band call is made) are sent to the server along with any parameters for the server method.

Any out-of-band calls are processed as a regular postback request up to the pre-rendering stage, meaning that all standard server events are fired: Init, Load, LoadComplete, and so on. Before the pre-rendering stage, the callback is prepared and executed shortly after. The requests ends immediately after the server method executes. The view state is not updated to reflect the state of the page after the out-of-band call and, subsequently, it is not sent back to the client.

The advantage of using ASP.NET Script Callback is that it is a native part of ASP.NET and can be easily encapsulated in server controls. For example, the TreeView control in ASP.NET 2.0 uses script callbacks to expand its nodes.

ASP.NET Script Callback is not free of significant issues, however. In particular, the server method is constrained to a fixed signature and can only take and return a string. Sure, you can place any contents in the string, but the serialization and deserialization of custom objects to the string is something you must take care of entirely on your own. In addition, a page based on ASP.NET Script Callback can have only one endpoint for remote calls. This means that if a client page needs to place two distinct calls to the same remote page you have to implement a switch in the implementation of the ICallbackEventHandler interface to interpret which method was intended to be executed.

**The AJAX.NET Library**

ASP.NET Script Callback is a low-level API that provides developers with an extremely thin abstraction layer that doesn’t do much more than auto-generate the script code required to trigger the call. To effectively implement out-of-band calls in application-wide scenarios, a library is required that hides all the nitty-gritty details of HTTP communication and exposes additional and higher-level controls and services.
AJAX.NET is a popular open-source library that adds a bit of abstraction over the XmlHttp-Request machinery. To use the library, you need to tweak your web.config file to register an ad-hoc page HTTP handler. The handler provides additional services, such as object serialization. In particular, the handler guarantees that any managed .NET object that is returned by a server method will be serialized to a dynamically created JavaScript object to be seamlessly used on the client.

Let’s see how to rewrite the previous example using the AJAX.NET library.

**Note**  The AJAX.NET library was first developed for ASP.NET 1.x by Michael Schwarz and then updated to ASP.NET 2.0. The URL to check out is http://www.ajaxpro.info. Lately, the development of the library has been stopped and resumed as an open-source project run by BorgWorX (http://www.borgworx.net).

### Setting Up the Library

To use the AJAX.NET library, you need the AJAX.dll assembly, which is available in the global assembly cache (GAC) or in the Bin folder of the application. Next, you need to add an HTTP handler entry in the web.config file:

```xml
<httpHandlers>
  <add verb="POST,GET" path="AJAX/* .ashx"
       type="AJAX.PageHandlerFactory, AJAX" />
</httpHandlers>
```

The goal of the entry is to make sure all requests for URLs that match the pattern AJAX/* .ashx are forwarded to the AJAX.NET-specific page handler.

Note that you don’t need to create an AJAX subdirectory in the application to contain all pages callable via AJAX.NET. The library makes use of a custom HTTP handler (the AJAX.Page-HandlerFactory component) to hook up requests for resources with a given name. The client infrastructure of AJAX.NET guarantees that any URL invoked from the client has a compliant format.

The HTTP handler automatically injects in the response stream any client code that is required to trigger out-of-band calls. From a developer’s perspective, all that is needed to set up the AJAX.NET library in an application is adding the section just shown to the web.config file.

### The Client Code

An AJAX.NET client page contains some JavaScript code that triggers the out-of-band call. Imagine the same page we considered earlier for ASP.NET Script Callback. The page contains
a button to retrieve additional information on the selected employee. The button (again, not a submit button) is associated with code as follows:

```html
<script type="text/javascript">
function GetCustomerDetail()
{
    var customerId = document.getElementById("customerId");
    var response = TestPage.GetCustomerById(customerId.value);
    var oCustomer = response.value;

    if (oCustomer.Type == "UnknownCustomer")
    {
        alert("Customer not found");
    }
    else
    {
        var fn = document.getElementById("firstName");
        var ln = document.getElementById("lastName");
        fn.innerHTML = oCustomer.FirstName;
        ln.innerHTML = oCustomer.LastName;
    }
}
</script>
```

As you can see, the JavaScript function makes a call to an apparently missing client object (the TestPage object), and a JavaScript object is used to process the return value of the call (the oCustomer variable). Where’s the TestPage.GetCustomerById method defined? And where is the prototype of the oCustomer variable defined? The answer is in the code-behind class.

**The Server Code**

The code-behind class of an AJAX.NET page meets a couple of requirements. First, it provides one or more public methods on the page class decorated with the [AJAXMethod] attribute. Second, the Page_Load event registers one or more types with the AJAX infrastructure.

```csharp
public partial class TestPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Utility.RegisterTypeForAJAX(typeof(TestPage));
        Utility.RegisterTypeForAJAX(typeof(Samples.Customer));
        Utility.RegisterTypeForAJAX(typeof(Samples.UnknownCustomer));
    }

    [AJAXMethod()]
    public Customer GetCustomerById(string customerId)
    {
        // Place here any code to retrieve customer
        // information from a database
        switch (customerId)
        {
            case "1":
                return new Customer("John", "Doe");
        }
    }
}
```
case "3":
    return new Customer("Jim", "Idontknowthisguy");
default:
    return new UnknownCustomer();
}
}

The server page is rendered on the client through a JavaScript object having the same name as the page class—TestPage, in this case. Any server method decorated with the [AJAXMethod] attribute is defined as a method on this JavaScript object. The body of this method is automatically generated to place an out-of-band call to the corresponding server method using XmlHttpRequest.

The server method can return any object that is registered with the AJAX infrastructure and is in no way limited to returning strings as in ASP.NET Script Callback. Each object registered with AJAX is rendered as a JavaScript object whose definition is then linked to the client page.

As a result, you can make remote calls to as many server-side methods as you want and use your own objects to carry information in a sort of strongly typed HTTP call.

**Putting It All Together: The AJAX.NET Library**

Setting up an AJAX.NET solution is simple. First, you attach the AJAX.NET hook to your application by making changes in the web.config file. Next, you write inline JavaScript code to control the out-of-band call. Each call targets a JavaScript object that represents the publicly callable interface of the server ASP.NET page. When a page is first requested, the AJAX.NET page handler automatically injects some script code that defines a JavaScript “page object.” In this object, each method references a corresponding server method.

The return value of the server method is wrapped by an AJAX-defined JavaScript object. To extract the return value, you call the value property on the provided object. The return value of the server method is a JavaScript object that mimics the public interface of the data type returned by the server method. In other words, the server method returns a .NET type and the client code receives a corresponding JavaScript object. Serialization and deserialization occur care of the AJAX.NET infrastructure. This process is known as JSON serialization, where JSON stands for JavaScript Object Notation.

**Key Facts**

AJAX.NET has some advantages over the ASP.NET Script Callback API. It uses an attribute to mark server methods that can be called from the client. This means that you have the greatest flexibility when it comes to defining the server public interface callable from the client.

In addition, you can register server types for use on the client, which provides for a strongly typed data transfer. The AJAX.NET infrastructure serializes .NET types to JavaScript objects and vice versa.

AJAX.NET hooks up and replaces the standard request processing mechanism of ASP.NET—the page handler. As a result, you won’t receive classic ASP.NET server events such as Init,
Load, and postback. At the same time, you won’t have the view state automatically transmitted with each out-of-band request. An AJAX.NET request, though, is still processed by the ASP.NET HTTP runtime, meaning that the request is still subject to the modules registered with the HTTP pipeline, including session state management, role, and authentication. This is because the request is consumed and processed by the AJAX.NET HttpHandler, so the rules that govern that processing within ASP.NET still apply.

**AJAX-Based Frameworks**

Two elements clearly emerge as far as the AJAX programming style is concerned and qualify as the next big thing in Web development. First, for application developers, it is key to have a programming model that is not too far from what they are currently familiar with. Developers should be able to move to AJAX with a short learning curve. This move is possible only if the new framework is an extension of an existing framework. But secondarily, at the same time the framework must provide rich and effective out-of-the-box components and services that minimize the impact on developers regarding such complex and objective issues as scripting, cross-browser scripting, server-side programming models, and data serialization.

Instead of using stand-alone libraries such as ASP.NET Script Callback and AJAX.NET (at least for the original project), developers should opt for ASP.NET frameworks and control suites that provide rich AJAX support.

Presented at the 2005 Professional Developer’s Conference, Microsoft Atlas is an add-on for the ASP.NET 2.0 platform and will be one of the pillars for the next version of ASP.NET (“Orcas”). Atlas takes the form of a new family of server controls that, by taking advantage of new runtime services, can be programmed on the client and the server in a totally integrated way. We’ll examine the architecture and programming model of Atlas in the rest of the book. It is instructive, though, to take a quick look at a couple of commercial products that provide AJAX-style functionality and will likely integrate Atlas capabilities in the future.

**ComponentArt’s Web.UI**

Part of ComponentArt’s Web.UI product, which contains a suite of ASP.NET controls, the CallBack component adds AJAX functionality to the other components in the Web.UI suite. In addition, it can also enable virtually any ASP.NET control to expose AJAX capabilities.

Similar to a native component of the Atlas framework (the UpdatePanel that we’ll review in detail in Chapter 3, “Updatable Page Fragments”), the CallBack component can either bypass the standard page life cycle and execute server-side logic more quickly or maintain the latest state of all ASP.NET controls contained in the page through the view state.

The CallBack component addresses a server-centric programming model but doesn’t currently provide built-in additional capabilities for developing client-centric solutions, such as invoking a Web service, a page method, or client-side data binding. Future releases of the
Web UI for ASP.NET suite, though, will provide full support for all the Atlas features. For more information, visit http://www.componentart.com.

**Telerik r.a.d.ajax**

The Telerik r.a.d.ajax control is a new patent-pending AJAX framework that allows you to AJAX-enable any ASP.NET application in a codeless fashion. The purpose of the product is to eliminate the complexities of building JavaScript-intensive AJAX application so that developers can take advantage of this new technology with no additional learning curve to climb. Complexities are eliminated by encapsulating the AJAX engine and all surrounding logic into ASP.NET components, which can be configured visually with convenient builders in Visual Studio 2005. As a result, developers can simply write regular postback-based applications and turn them into AJAX-enabled ones without writing any JavaScript or server-side code.

The core advantage of the product is the patent-pending technology that can automatically replace the postback functions of selected page elements with AJAX callbacks. As a result, the control being “ajax-ified” starts making callbacks without any developer intervention—no JavaScript and no manual invoking of XMLHttpRequest. This technology can AJAX-enable any standard ASP.NET controls, the Telerik UI controls for ASP.NET (r.a.d.controls suite), as well as most third-party controls. (Find out more at http://www.telerik.com/)

The Telerik r.a.d.ajax framework features the following components:

- **AJAX Panel**—A universal AJAX-enabling container, which causes any controls placed within it to start making callbacks instead of postbacks
- **AJAX Manager**—A component that allows you to configure at design-time which page elements need to initiate callbacks and which elements need to be respectively updated
- **AJAX Timer**—A component that performs AJAX callbacks at specified time intervals

In addition, you will find built-in AJAX support in all Telerik UI components. For example, all data-intensive components from the Telerik r.a.d.controls suite support AJAX internally to retrieve data on demand for almost real-time performance. The AJAX mode can be switched on with a single property.

The Telerik engine completely preserves the life cycle of the ASP.NET page, which is imperative for the proper operation of your application. The view state, event validation, and client-side scripts are also preserved as if a normal postback takes place. All form values are automatically sent to the server for processing.

The Telerik r.a.d.ajax framework is an end-to-end solution that allows even beginner ASP.NET developers to build sophisticated AJAX-enabled Web applications similar to Microsoft Outlook Web Access.
Conclusion

Most attentive developers in the community have been developing around interactive Web technologies since the late 1990s. Various technologies (for example, Microsoft Remote Scripting and open-source and commercial variations) followed without forming a critical mass of acceptance and use. Or perhaps the mass was big enough already, but we were all waiting for the spark of a killer application. Another factor that slowed down the adoption of more advanced client techniques was the lack of cross-browser support for these techniques.

Today, the situation is radically different from what it was only three or four years ago. Now, about 90 percent of the available browsers support all the minimal requirements for implementing interactive Web applications, known as AJAX applications. In addition, the W3C is standardizing the XMLHttpRequest object that is the brains behind most of the existing platforms for AJAX. The next generation of Web applications will be based on a different mechanism: it will no longer be, or not just be, forms posted in a change of pages, but individual requests for data and dynamic updates to displayed pages.

As a server technology aimed at the creation of Web pages, ASP.NET takes advantage of the opportunity for providing this much desired functionality. Script callbacks were the first attempt to offer an API for building AJAX-style pages. Modeled after the classic postback event, callbacks are sometimes unnecessarily heavy and inflexible. AJAX.NET—an open-source project—is an even better solution that provides a number of additional services, but it’s based on an opposite vision of the interaction. AJAX.NET is chatty, where script callbacks are chunky. AJAX.NET is much more flexible, lightweight, and direct in its programming model than ASP.NET remote scripting. Both technologies have two main drawbacks—the lack of a rich client object model to blur the distinction between the DOM of the various browsers, and the neat separation between the client and server aspects of programming.

Microsoft Atlas shows the way ahead for AJAX applications as far as the ASP.NET platform is concerned. Atlas integrates the AJAX lifestyle into the existing application model of ASP.NET, resulting in a familiar programming model with greatly improved and richer functionalities.