Foundations of Atlas

Rapid Ajax Development with ASP.NET 2.0

Laurence Moroney

Apress®

Foundations of Atlas: Rapid Ajax Development with ASP.NET 2.0

ISBN-13 (pbk): 978-1-59059-647-0

ISBN-10 (pbk): 1-59059-647-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Keith Smith
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Nancy Sixsmith
Indexer: Michael Brinkman
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

This book is dedicated to my brother, Niall, all-round cool guy and great big brother, as well as provider of free accommodation whenever I visit Ireland; to his wife, Sandra, among the warmest and most loving people I have ever met; to his son, Sean, the local guitar god; and to his daughter, Aideen, the sweetest girl in Ireland.

Contents at a Glance

Contents

Foreword

The last year has seen a series of incredible advances in user experiences on the Web. As the Web continues to grow as a marketplace for web applications and services, the user experience has become a key differentiator in attracting and retaining users and in driving revenue and productivity. This has led to an explosion of richer, more personalized, more interactive sites that fully exploit the capabilities of the browser platform.

What is all the more remarkable about this trend is that it is built on a technology foundation that has been around for a long time and has only recently acquired the name Ajax. Microsoft pioneered both Dynamic HTML and CSS in Internet Explorer nearly a decade ago, and the current edition of the JavaScript language is several years old. And the cornerstone of Ajax—the XMLHttpRequest object, which enables more flexible communication between the browser and server—was built into Internet Explorer in 1998 to support pioneering applications such as Outlook Web Access.

So why has Ajax suddenly started to catch on now? Clearly, the technology has matured and standardized; Ajax's capabilities are available on more platforms and browsers than ever before and can now be found on the vast majority of desktop computers. The computing power of the desktop computer has also grown significantly. And, finally, the Web itself has evolved into a rich platform. These trends have, in turn, driven the need to invest in better, more differentiated experiences.

With these experiences, however, come greater challenges for the developer. It is no secret that developers often lack the frameworks, tools, and skills to be effective with DHTML and JavaScript. Differences in browsers are also a frequent cause for trouble. Finally, these applications are also harder to develop because the Ajax model is inherently asynchronous in nature. To try to address these kinds of challenges, Ajax libraries have become commonplace, but few provide a rich framework and the tools that are integrated with today's web programming models and tools.

When we began building Atlas in the summer of 2005, we set out to build an end-to-end framework and the tools that would bring unparalleled productivity to web application development and make it easy for anyone to build a rich web experience on the standards-based web client platform. At a high level, we designed Atlas with the following goals in mind:

A rich framework for web experiences: Allow anyone to develop an interactive, personalized web experience by easily wiring together controls and components. Provide a rich toolbox of built-in components, and allow developers to easily develop their own. Even developers who are familiar with scripting should benefit from patterns that enable easier manageability and reuse.

Seamless integration with the .NET programming model: Deliver an end-to-end experience that allows Atlas applications to have easy access to the richness of the programming model of ASP.NET and the .NET Framework on the server.

Choice of server-centric or client-centric application models: Allow developers who use server-centric models such as ASP.NET to easily enrich their applications with Atlas, without learning scripting or asynchronous client communications. At the same time, allow anyone to use the richness of Atlas to take full advantage of the browser.

Fully cross-platform and standards-based: We expect that developers using Atlas will want to build applications that can run on any browser. Atlas is designed to work on a wide variety of modern browsers and platforms and includes functionality that takes much of the worry out of browser compatibility. The Atlas Script framework also works with any web server.

No installation footprint: Atlas doesn't require any new client installation; it works on the browser you have on your desktop computer today. The Atlas "client" consists of a set of JavaScript files that are downloaded by the browser like any other web content.

In trying to achieve these goals, we built a free, cross-platform, standards-based framework that lets you easily build richer, more interactive, more personalized experiences on the Web.

In developing Atlas, we also took an open approach that sets a new example for how we hope to build developer tools and frameworks at Microsoft. From early in the product cycle, we began making Atlas available to the developer community as a Community Technology Preview (CTP) release—the first release was less than eight weeks after the start of the project—and have continued to release previews every four to six weeks since then. Through these CTPs, we have gathered an incredible amount of early feedback that has helped shape the product profoundly, and we are indebted to our developer community for their participation. As we continue to work toward the full release of Atlas, we have started to invite community partnership in new ways, such as by releasing the Atlas Control Toolkit, a collection of samples that we will develop in collaboration with the developer community.

Such an open development model has not come without its growing pains. In the first preview releases of Atlas, we had a raw product with little documentation and few samples. The small number of early adopters and partners who used Atlas in these early days had to dig deep to understand it and have stood by us over the months as we developed it into what it is today.

As an author, Laurence was one of those early partners, and his knowledge of the product clearly shows in this definitive guide to Atlas. This book gives you everything you need to get started building rich web experiences with Atlas. It covers all the concepts, starting with a clear explanation of the Ajax model and of ASP.NET. It then guides you through the Atlas framework in great breadth, exploring the client script framework, the Atlas server controls for ASP.NET, and the rich toolbox of built-in controls and components.

He covers each topic with a straightforward, step-by-step narrative, accompanied by code examples and illustrations that present concepts in a clear and practical way. A special feature is Chapter 11, which illustrates how to apply Atlas effectively to develop a rich real-world application.

I am very excited about Atlas and the potential it delivers for building rich web experiences. If you are looking for tools to unleash your creativity and help you build new, differentiated experiences on the Web, you will find Atlas and this book invaluable. I sincerely hope you have as much fun working with Atlas as we had building it.

Shanku Niyogi
Product Unit Manager, UI Framework and Services Team
Microsoft Corporation

About the Author

LAURENCE MORONEY is the director for technology evangelism at Mainsoft, the cross-platform company. He specializes in evangelizing the power of ASP.NET and Visual Studio .NET through Mainsoft technology for the development and maintenance of J2EE applications. Prior to Mainsoft, he worked as an architect for Reuters developing financial and communications applications for the Wall Street community. An experienced developer and architect, he has designed and implemented software for a variety of applications spanning the spectrum from casinos to banks to jails to airports. He's a big fan of professional sports, particularly soccer, and is still waiting for Bruce Arena to call him up for the U.S. World Cup squad.

About the Technical Reviewer

KEITH SMITH is a senior product manager at Microsoft Corporation in Redmond, Washington. Keith joined Microsoft in 1998 as a software engineer on Microsoft Visual J++ where he was responsible for the Windows Foundation Class's DHTML controls. Today he focuses on ASP.NET, Atlas, and the web development technologies and components of Visual Studio (VS) and Visual Web Developer Express Edition (VWD). Keith's ASP.NET involvement dates back to version 1.0 when he was a member of the engineering team responsible for the core ASP.NET controls and when he led the team responsible for ASP.NET performance and stress testing. More recently, Keith managed the entire VWD quality assurance engineering team. Following the release of VS 2005 and ASP.NET 2.0, Keith moved to his current product management role, in developer marketing, where he owns the overall marketing strategy and execution of creative programs that drive the adoption of Microsoft's web platform and tools.

Acknowledgments

I'd like to take this time to acknowledge some people who were amazing in helping to construct this book:

- Scott Guthrie owns the Atlas technology at Microsoft and was a terrific supporter in writing it and in getting access to the technology.
- Shanku Niyogi, the ASP.NET group program manager at Microsoft, was there to meet with me when I was shaping ideas for the construction of the book, to listen to my feedback on where the technology should be going, and to help fix bugs in my code when Atlas was at an early stage.
- Keith Smith, the product manager for ASP.NET and Atlas at Microsoft, was hands down the best technical reviewer I have ever worked with. Straight to the point, factual, and extraordinarily pleasant, he helped hone this book into what it is today.
- Simon Calvert, a developer on the Atlas project, was there to help me navigate the undocumented sections. I thanked him profusely for being the technical reviewer before realizing I had mixed him up with Keith! (Sorry, guys.)
- Wilco Bauwer is the joint-best Atlas blogger there is (`http://www.wilcob.net`).
- Nikhil Kothari is also the joint-best Atlas blogger there is (`http://www.nikhilk.net`).
- Joel Bonette, a researcher at Lousiana State University, discovered the book through my blog and, through being the first purchaser of it, strongly encouraged me through the dark and frustrating days when I couldn't make head or tail of the undocumented features.
- Ewan Buckingham, editor at Apress, is the perfect editor—hands-off and completely trusting.
- Kylie Johnston, project manager at Apress, kept me sane and having fun as I wrote.
- Kim Wimpsett, the copy editor, made me feel really good by not making too many corrections to the text.

Thank goodness this isn't the Oscars, because I just have too many people to thank; it is a book, so I can take my time to name them all!

Introduction

Ajax is fast becoming *the* buzzword in web development for 2006. It's also becoming a de facto standard for developing user-centric web applications. It's an evolutionary step in the user experience and is being used in more and more web applications from Google Local maps to Live.com to Amazon and beyond.

But how do you write Ajax applications? You've got to be a JavaScript expert and use tools that are not as sophisticated as those your C# or Java friends use. As such, it can be difficult and time-consuming to develop, debug, and maintain Ajax applications despite their innate user friendliness.

Microsoft is contributing to the solution for this problem with its Atlas framework. This builds on top of the best-of-breed ASP.NET technology and Visual Studio 2005 integrated development environment (IDE) to bring major productivity leaps to Ajax development.

With Atlas you can easily convert your existing ASP.NET applications to Ajax ones, and you can add sophisticated user interface elements such as drag and drop, networking, and browser compatibility layers with simple declarative programming (or, if you prefer to use JavaScript, you can do that too).

This book is a primer on this technology. It will take you through the evolution of web applications to where they are today, introduce you to Ajax, put it in context, and then take you into how to build Ajax applications quickly and simply, taking advantage of the IDE productivity and full debugging offered by Visual Studio 2005.

It's going to be a fun ride, and by the end of it, you'll be an expert in Web 2.0 and hungry to start developing for it. And who knows? You may even start shaping Web 3.0!

Who This Book Is For

This book is for anybody who is interested in developing next-generation web application interfaces that make the most of Ajax-style asynchronous functionality. Anybody who has ever coded a web page will understand the latency problems associated with postbacks and maintaining state and will be able to gain valuable new tools for their programming arsenal by reading this book.

Even if you don't know ASP.NET, C#, or Visual Basic .NET, you should still be able to understand and follow along with this book, but it would be helpful to get a good grounding in these technologies first.

How This Book Is Structured

This book starts by describing the evolution of user interface software from the earliest punch cards to the modern Ajax applications. It describes how the thin client model can be used to cut costs and complexity and tells the true story of how a thin client, Ajax-style application got the Wall Street community up and running quickly after September 11, 2001.

It then dives into what Ajax is and how it works so that if you aren't familiar with it, you can see what all the fuss is about.

Next, it introduces Atlas and how it works in the context of ASP.NET. It describes the client-side and server-side functionality of Atlas, showing each of the libraries and controls and describing how you can use them.

It is packed with sample code and figures, so you'll learn by example. The final chapter is devoted to a single example of a large-scale application that brings all the concepts together. It's very much a hands-on book.

Prerequisites

You'll need Visual Studio 2005; any edition is fine. You'll also need the Atlas binaries and add-ins to Visual Studio 2005, which can be downloaded from `http://atlas.asp.net`.

Downloading the Code

You will be able to download chapter-by-chapter source code from the Apress web site at `http://www.apress.com`. This is compressed in a single zip file for your convenience.

Contacting the Author

You can reach Laurence Moroney via `ljpm@philotic.com` or at `http://www.philotic.com/blog`.

CHAPTER 1

Introducing Ajax

Welcome to *Foundations of Atlas*. This book is intended to get you up and running with the new framework from Microsoft that allows you to build web 2.0 applications that implement Ajax functionality. If you've been working in the web field at all, you will have found Ajax hard to avoid—and even harder to implement. Microsoft has thrown its hat into the Ajax arena by doing what it does best—giving you, the developer, tools that allow you to be productive and solve your business needs.

This chapter will bring you up-to-date on web application technology from the dawn of computing history to today, putting Ajax and Atlas in context. It's the beginning of what I hope will be a fun and informative ride.

Delving into the History of Web Application Technology

The user interface has evolved along a cyclical path from a light footprint to a heavy footprint and back again. Users' requirements and demands for extra functionality drive the heavier footprint, and users' requirements and demands for an easy installation, upgrade, and maintenance drive the lighter footprint. With each iteration, the "lighter" user interfaces gain rich functionality, and the "heavier" user interfaces become easier to install, upgrade, and maintain.

The original user interfaces were probably the lightest clients of all—punch cards that were fed into a central server that in turn printed results, resulting in a simple request/response architecture (see Figure 1-1).

As computers became more sophisticated, the punch card and printer were replaced by a terminal that fed results into and rendered results from the central server (see Figure 1-2). For basic processing this was useful, but as computers' needs increased—for example, to handle the new requirements of email or network news—the terminals had to get smarter. If you remember the old terminal-based email programs such as elm or the network newsreaders, you'll remember needing three or four extra fingers to do anything useful! Architecturally, this wasn't much different from the punch card request/response architecture, but it unified the point of contact with the mainframe and changed the medium from paper to electrons. So, although the architecture did not change, the implementation did—and it was this change in implementation that was a driving factor in improving the overall user experience of applications, a fact that is still true today.

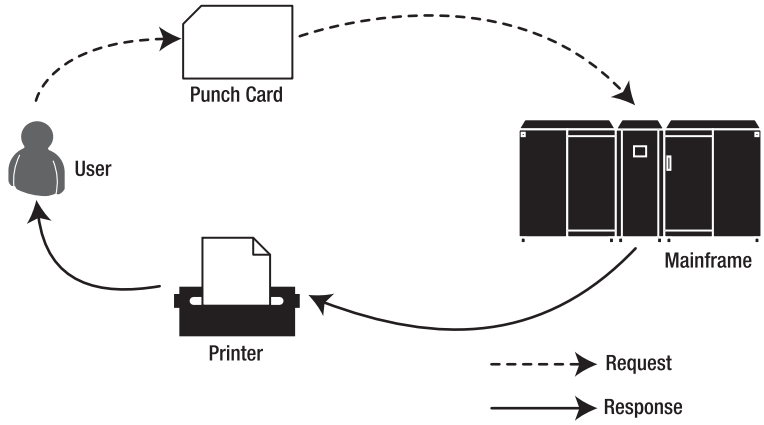

Figure 1-1. *Punch card request/response architecture*

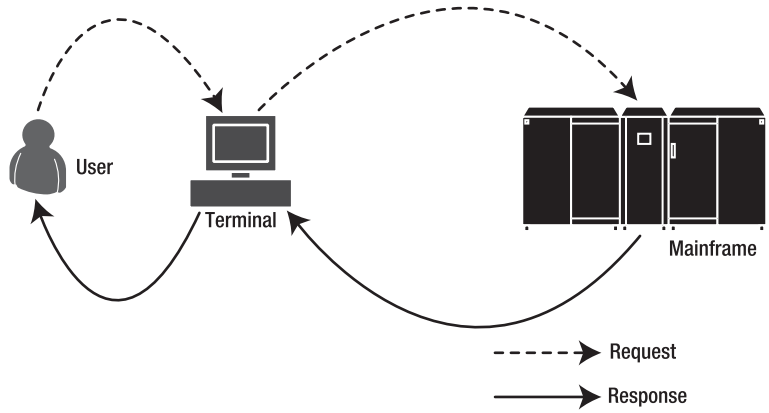

Figure 1-2. *Terminal-based request/response architecture*

With the advent of the personal computer, much of the old server functionality wasn't necessary anymore. This was because the main use of servers at that time was for functions that could easily take place on a personal computer, such as calculations or analyses (see Figure 1-3). Functionality that required connectivity, such as email and network newsreaders, could still be achieved on a personal computer through terminal emulation.

Then someone had the bright idea of using the power of the personal computer to enhance the online experience, taking it away from the green-and-black terminal and allowing features such as email, news, and graphics to appear in full four-color glory (see Figure 1-4). The personal computer flourished in power, and the early personal computers gave way to much more powerful machines with better graphics, faster processing chips, more memory, and persistent storage through hard drives.

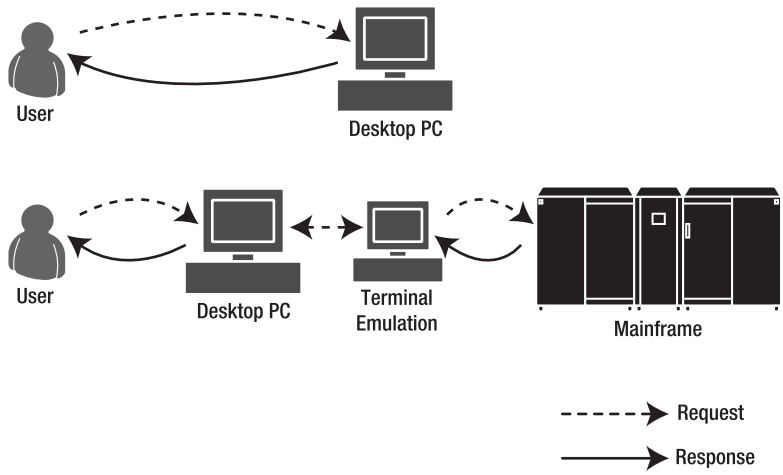

Figure 1-3. *Personal computer request/response online and offline*

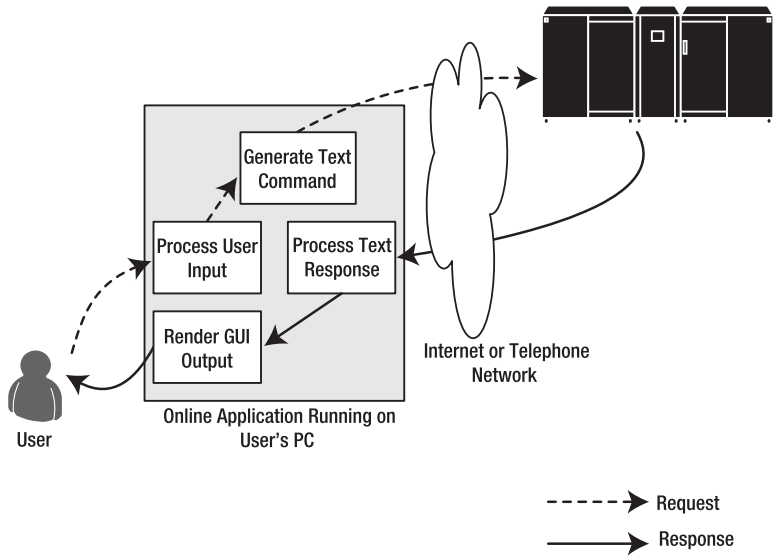

Figure 1-4. *Request/response architecture of graphical user interface (GUI) application talking to mainframe*

With this steadily exponential increase in computing power at the desktop, applications became more sophisticated, complex, and functional than anything before on centralized mainframe supercomputers. Full GUIs soon became the norm. Microsoft Windows appeared, following Apple, Atari, and other GUI-focused computers. Soon after, office productivity applications exploded onto the scene; and as people began using these applications daily, they required even faster and more sophisticated platforms, and the client continued to evolve exponentially.

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

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

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

But in the early 1990s, a huge innovation happened: the web browser.

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

But the web browser ultimately became much more than just a new means for abstracting the textual nature of the client/server network. It became an abstraction on top of the operating system on which applications could be written and executed (see Figure 1-5). This was, and is, important. As long as applications are written to the specification defined by that abstraction, they should be able to run anywhere without further intervention or installation on behalf of the application developer. Of course, the browser would have to be present on the system, but the value proposition of having a web browser available to the operating system was extremely important and ultimately launched many well-known legal battles.

The problem, of course, with this abstraction was that it was relatively simple and not originally designed or implemented for anything more complex than laying out and formatting text and graphics. I am, of course, referring to Hypertext Markup Language (HTML). This specification, implemented by a browser, meant that simple text could be placed on a server, transferred from a server, interpreted by a browser, and laid out in a far more pleasing way than simple green-on-black on a page, giving the user a better experience. More important, however, it could generate a whole new breed of application developers; all a developer had to do for an online, connected application to have a graphical experience was to generate it as HTML, and the browser would do the rest. You wouldn't need the resources of a CompuServe or an America Online to build an application that rendered the text for you! All you had to do was generate HTML, either by coding it directly or writing a server-side application (in C) that would generate it for you. Although the Internet had been around for a long time, only at this point was it really being born.

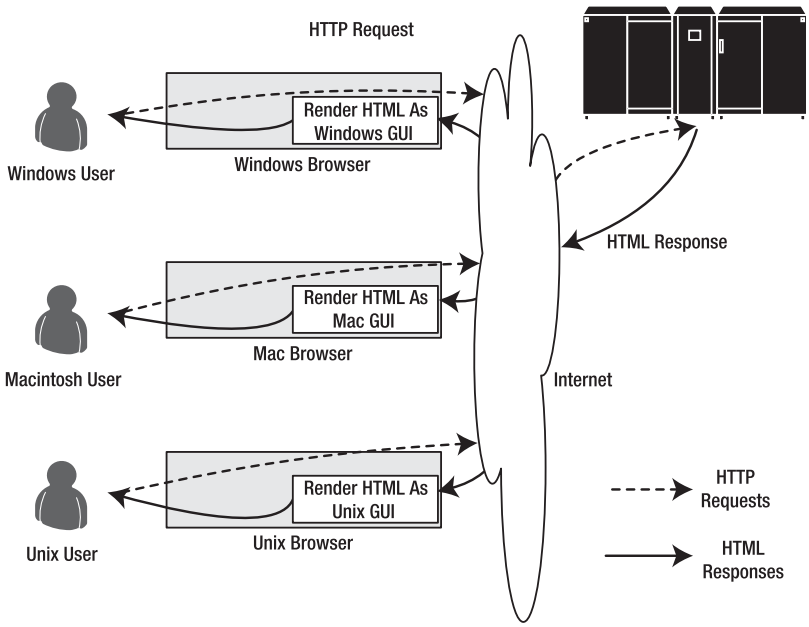

Figure 1-5. *Web browser–based request/response architecture*

And guess what happened? The cycle began again.

Everybody jumped on the browser bandwagon, and Common Gateway Interface (CGI) applications, written on a server and delivered to a browser, were hot. The user experience, with the only interaction being postbacks to the server (in much a similar vein to terminals, only prettier), soon became too limiting, and new technologies began to emerge to improve the user experience.

Enter Java and the applet. Java, a virtual machine on top of a virtual machine (the browser) on top of a virtual machine (the operating system) on top of a real machine (the underlying hardware), gave a greater abstraction, and it introduced a new platform that developers could code to and have even richer applications running within the browser. This was important, because it accelerated what could be done within a browser, delivered using the simple transport mechanisms of the Internet but again without requiring the resources of a huge company writing your own GUI platform on which to do it. Of course, it suffered from constraints; namely, to achieve a cross-platform experience, developers had to follow a lowest common denominator approach. The clearest example of this was in its support for the mouse. The Apple operating systems supported one button, the Microsoft Windows–based ones supported two, and many Unix platforms supported three. As such, Java applets could support only one button, and many Unix users found themselves two buttons short!

The Java virtual machine and language evolved to become a server-side implementation and a great replacement for C on the server. In addition to this, HTML continued to evolve, allowing for more flexibility, and its big brother, Dynamic HTML (DHTML), was born. In addition, scripting was added to the platform (at the browser level), with JavaScript (unrelated to Java despite the name) and VBScript being born. To handle these scripting languages, parsers were bolted onto the browser, and the extensible browser architecture proved to be a powerful addition.

Thanks to extensibility, applications such as Macromedia Flash added a new virtual machine on top of the browser, allowing for even more flexible and intense applications. The extensible browser then brought about ActiveX technology on the Windows platform, whereby native application functionality could be run within the browser when using Microsoft browsers (or alternative ones with a plug-in that supported ActiveX). This was a powerful solution, because it enabled native functionality to be accessible from networked applications (see Figure 1-6). This got around the restrictions imposed by the security sandbox and lowest common denominator approach of the Java virtual machine, but this ultimately led to problems in the same vein as distributing client-only applications; specifically, a heavy configuration of the desktop was necessary to get them to work. Although this configuration could be automated to a certain degree, it gave two show-stopping points for many.

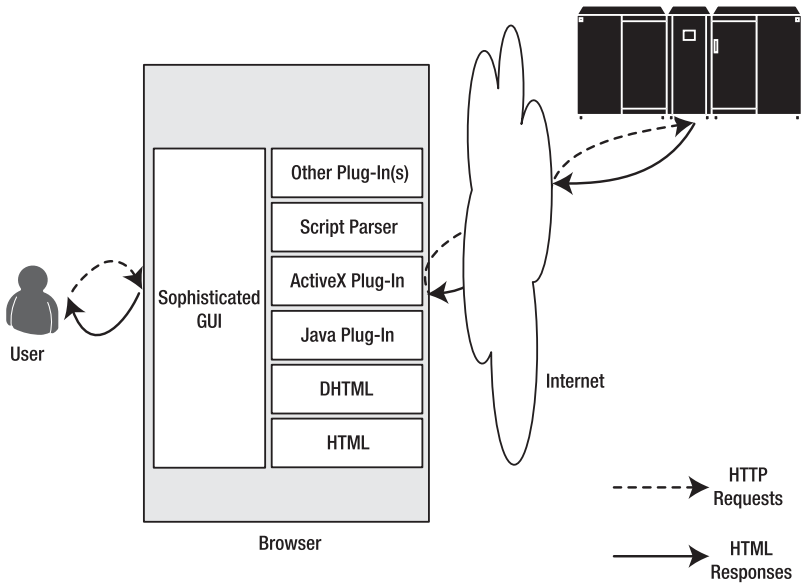

Figure 1-6. *Sophisticated browser architecture*

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

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

and therefore applications written to run on them also couldn't. The user was effectively safe, though limited in the scope of what they could do.

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

The .NET Framework was revolutionary because it united the client-application experience and connected-application experience across a unified runtime, which ActiveX tried but ultimately failed to do. Because the same platform could be written for both, the result was that the user experience would be similar across both (see Figure 1-7). Coupled with the emergence of Extensible Markup Language (XML), a language similar to HTML but specialized for handling data instead of presentation, web application development was finally coming of age.

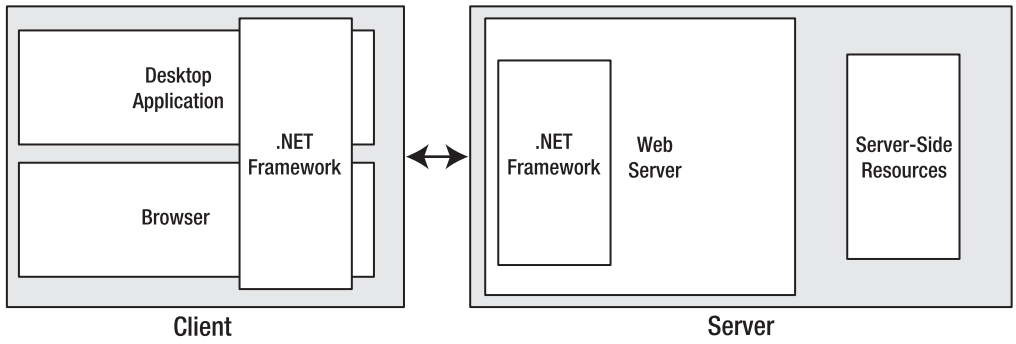

Figure 1-7. *The .NET Framework provides consistent browser, desktop, and server application programming interfaces (APIs).*

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

Thin Client Applications Arriving to Save the Day

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

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

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

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

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

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

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

Ajax Entering the Picture

Ajax, which once stood for Asynchronous Java and XML, is a technique that has received a lot of attention recently because it has been used to great success by companies such as Amazon and Google. The key word here is *asynchronous*, because, despite all the great technologies available in the browser for delivering and running applications, the ultimate model of the browser is still the synchronous request/response model. As such, the refreshing and updating of applications that the user is used to is hard to achieve. The typical web application involves a refresh cycle where a postback is sent to the server and the response from the server is rerendered. This is a drawback of this type of architecture, because the round-trip to and from the server is expensive in user time and bandwidth cost, particularly for applications that require intensive updates.

What is interesting about the Ajax approach is that nothing is really new about it. The core technology—the XMLHttpRequest object—has been around since 1999 with Microsoft Internet Explorer, when it was implemented as an ActiveX plug-in. More recently, it has been added to the Mozilla and Safari browsers, increasing its ubiquity, and has been covered in a World Wide Web Consortium (W3C) specification (DOM Load and Save). With the popularity of web applications that use XMLHttpRequest such as Google Local, Flickr, and Amazon A9, XMLHttpRequest is fast becoming a de facto standard.

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

To see an example of how it works, refer to Google Local (see Figure 1-8). As you use the mouse to drag the map around the screen, the new sections of the map that were previously hidden come into view quickly; this is because they were cached on your first viewing of the map. Now, as you are looking at a new section (by having dragged the mouse), the sections bordering the current one are downloading in the background, as are the relevant satellite photographs for the section of map you are viewing.

It is this background downloading, using the XMLHttpRequest object, that makes using Google Local such a smooth and rewarding experience. Remember, nothing is *new* here; it's just that having an object built into the browser that can do this asynchronously makes it easier to develop applications like this. For full details on how to develop in Ajax, check out *Foundations of Ajax* (Apress, 2005).

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

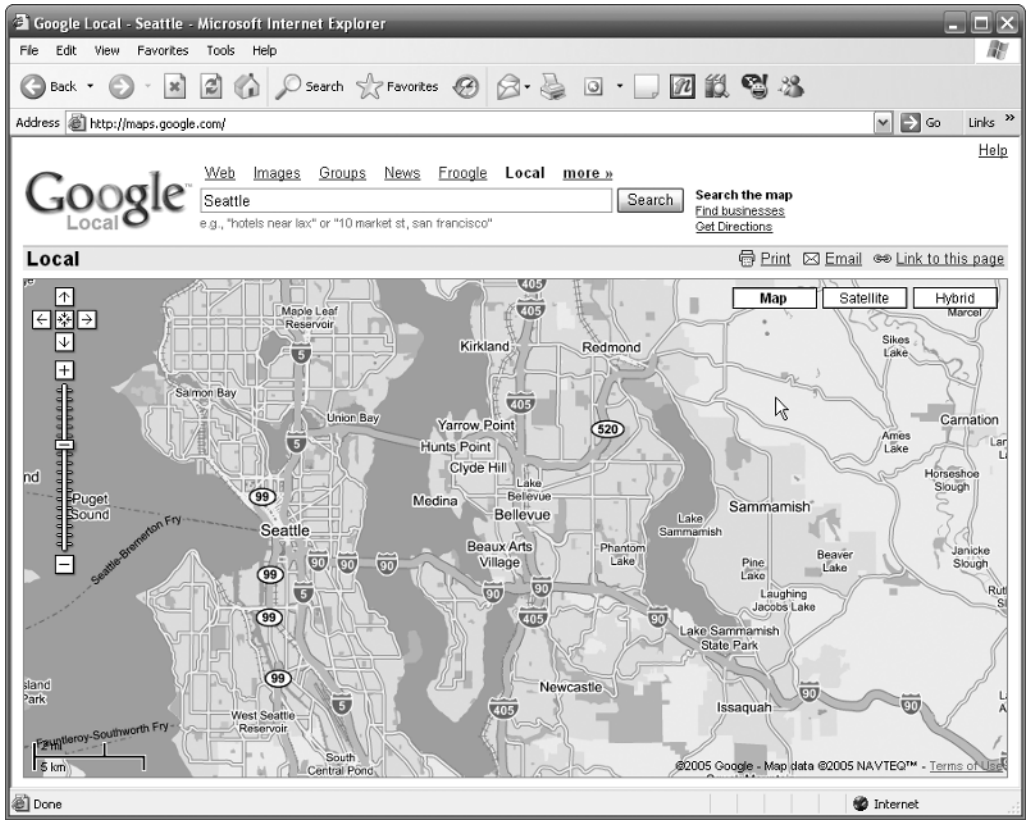

Figure 1-8. *Google Local uses Ajax extensively.*

Using the XMLHttpRequest Object

As mentioned, the XMLHttpRequest object is the heart of Ajax. This object sends requests to the server and processes the responses from it. In current versions of Internet Explorer, it is implemented using ActiveX, whereas in other browsers, such as Firefox, Safari, and Opera, it is a native JavaScript object. Unfortunately, because of these differences, your JavaScript code has to be coded to inspect the browser type and create an instance of it using the correct technology. (In Internet Explorer 7, Microsoft will be supporting XMLHttpRequest as a native JavaScript object.)

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

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

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

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

The open method sets up the call to your server to initialize your request. It takes two required arguments (the Http command of GET, POST, or PUT and the URL of the resource you are calling) and three optional arguments (a Boolean indicating whether you want the call to be asynchronous, which defaults to true, and strings for the username and password if the server requires these for security). It returns void.

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

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

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

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

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

The getAllResponseHeaders method returns a string containing the complete set of response headers from the XMLHttpRequest object. Examples of this include the HTTP headers of Content-Length and Date, with their appropriate values. This is accompanied by the getResponseHeader method, which takes a parameter representing the name of the header you want to query, and its value is returned as a string.

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

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

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

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

Using Visual Studio 2005

Throughout this book you'll be using Visual Studio 2005 for developing Ajax applications using the Atlas extensions for ASP.NET 2.0. Several editions of this application are applicable to different tasks.

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

You may also use any of the other editions of Visual Studio 2005, including Standard, Professional, or Team System, to use and build the samples included in this book.

If you are following along with the figures in this book, you'll see these have been captured on a development system that uses the Team System edition of Visual Studio 2005.

Seeing a Simple Example in Action

Understanding how this technology all hangs together is best shown using a simple example. In this case, say you have a client application that uses JavaScript and an XMLHttpRequest object that calls a server to perform the simple addition of two integers. As the user types the values into the text boxes on the client, the page calls the server to have it to add the two values and return a result that it displays in a third text box. You can see the application in action in Figure 1-9.

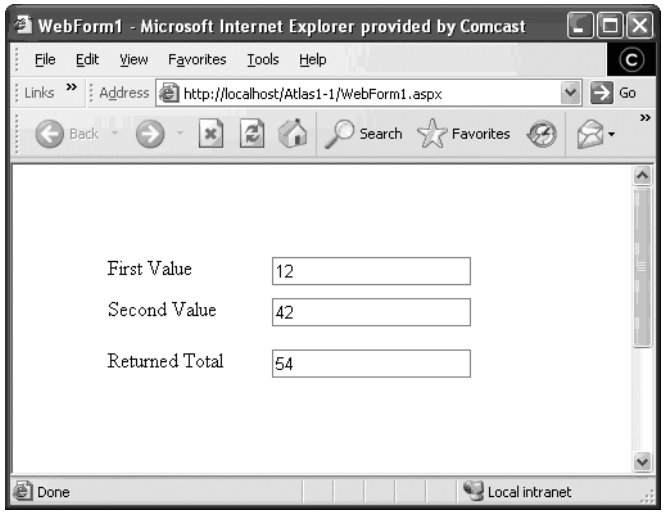

Figure 1-9. *The Ajax addition client*

To create this client, start a new Visual Studio 2005 web site, and edit the default WebForm1.aspx content to match Listing 1-1.

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

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs" AutoEventWireup="false"
Inherits="Atlas1_1.WebForm1" %>

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

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

```
    function updateTotal() {
        frm = document.forms[0];
        url="WebForm2.aspx?A=" + frm.elements['A'].value +
              "&B=" + frm.elements['B'].value;
        xmlHttp.open("GET",url,true);
        xmlHttp.onreadystatechange=doUpdate;
        xmlHttp.send();
        return false;
    }

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

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

When the web page loads, the createXMLHttpRequest function is called (from onload= in the body tag) to initialize the object. After that, whenever a key is pressed in the A or B text boxes, the updateTOT function is called (by trapping the onkeyup event).

This function then takes the values of A and B from their form elements and uses them to build the URL to WebForm2.aspx, which will look something like WebForm2.aspx?A=8&B=3. It then calls the open method on XMLHttpRequest, passing it this URL and indicating that this