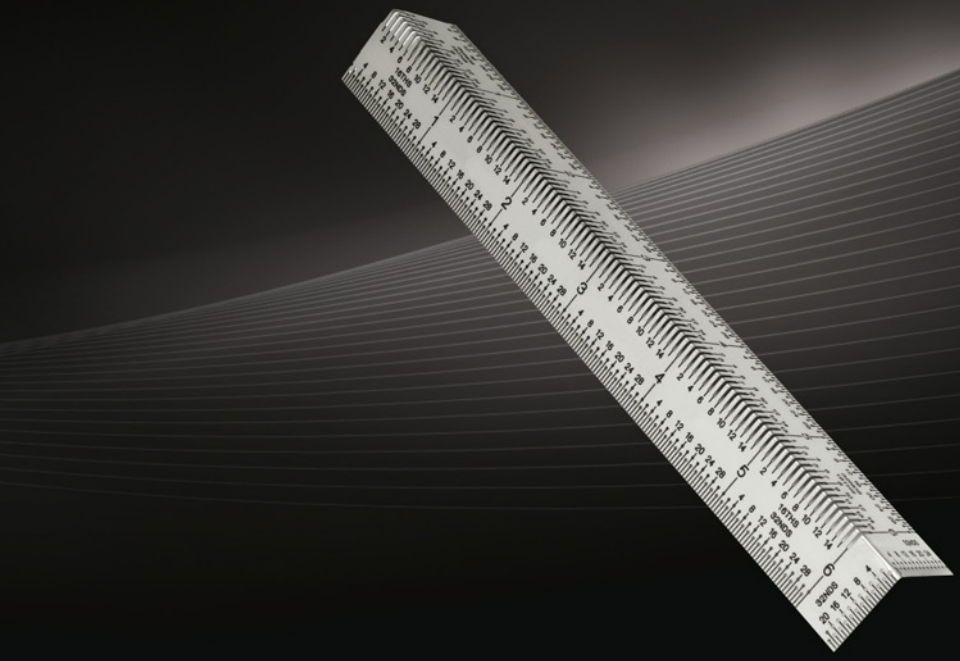
Microsoft

Microsoft® ASP.NET and AJAX: Architecting Web Applications



Dino Esposito

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008940527

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 4 3 2 1 0 9

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to msinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Developmental Editor: Lynn Finnel
Project Editor: Tracy Ball
Editorial Production: S4Carlisle Publishing Services
Technical Reviewer: Kenn Scribner; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X15-28134

To the people who help me to smile and often smile, play and laugh with me.

—Dino

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Table of Contents

Acknowledgments

A team of people helped me to assemble this book.

Ben Ryan was sneakily convinced to support the project on a colorful Las Vegas night, during an ethnic dinner during which we watched waiters coming up from and going down to the wine cellar in transparent elevators.

Lynn Finnel just didn't want to let Dino walk alone in this key project after brilliantly coordinating at least five book projects in the past.

Kenn Scribner is now Dino's official book alter ego. Kenn started working with Dino on books back in 1998 in the age of COM and the Active Template Library. How is it possible that a book with Dino's name on the cover isn't reviewed and inspired (and fixed) by Kenn's unique and broad perspective on the world of software? The extent to which Kenn can be helpful is just beyond human imagination.

Roger LeBlanc joined the team to make sure that all these geeks sitting together at the same virtual desktop could still communicate using true English syntax and semantics.

I owe you all the (non-rhetorically) monumental "Thank you" for being so kind, patient, and accurate.

—Dino

This page intentionally left blank

Introduction

This book is the Web counterpart to another recently released book I co-authored with Andrea Saltarello: *Microsoft .NET: Architecting Applications for the Enterprise* (Microsoft Press, 2008). I wrote it, in part, in response to the many architectural questions—both small questions and big ones—that I was asked repeatedly while teaching ASP.NET, AJAX, and Silverlight classes.

Everybody in the industry is committed to AJAX. Everybody understands the impact of it. Everybody recognizes the enormous power that can be derived from its employment in real-world solutions.

Very few, though, know exactly how to make it happen. There are so many variations to AJAX and so many implementations that even after you have found one that suits your needs, you are left wondering whether that is the best possible option.

The fact is that AJAX triggered a chain reaction in the world of the Web. AJAX represents a change of paradigm for Web applications. And, as the history of science proves, a paradigm shift has always had a deep impact, especially in scenarios that were previously stable and consolidated.

I estimate that it will take about five years to absorb the word *AJAX* (and all of its background) into the new definition of the Web. And the clock started ticking about four years ago. The time at which we say "the Web" without feeling the need to specify whether it contains AJAX or not . . . well, that time is getting closer and closer. But it is not that time yet.

Tools and programming paradigms for AJAX, which were very blurry just a few years ago, are getting sharper every day. Whether we are talking about JavaScript libraries or suites of server controls, I feel that pragmatic architectures can be identified. You find them thoroughly discussed in Chapter 3, "AJAX Architectures."

Architecting a Web application today is mostly about deciding whether to prefer the *richness* of the solution over the *reach* of the solution. Silverlight and ASP.NET AJAX are the two platforms to choose from as long as you remain in the Microsoft ecosystem. But the rich vs. reach dilemma is a general one and transcends platforms and vendors. A neat answer to that dilemma puts you on the right track to developing your next-generation Web solution.

Who This Book Is For

I believe that this book is ideal reading for any professionals involved with the ASP.NET platform and who are willing or needing to find a solution that delivers a modern and rich user experience.

Companion Content

Examples of techniques and patterns discussed in the book can be found at the following site: *http://www.microsoft.com/learning/en/us/books/12926.aspx*.

Hardware and Software Requirements

You'll need the following hardware and software to work with the companion content included with this book:

- Nearly any version of Microsoft Windows, including Vista (Home Premium Edition, Business Edition, or Ultimate Edition), Windows Server 2003 and 2008, and Windows XP Pro.
- Microsoft Visual Studio 2008 Standard Edition, Visual Studio 2008 Enterprise Edition, or Microsoft Visual C# 2008 Express Edition, and Microsoft Visual Web Developer 2008 Express Edition.
- Microsoft SQL Server 2005 Express Edition, Service Pack 2 or Microsoft SQL Server 2005, Service Pack 3, or Microsoft SQL Server 2008.
- The Northwind database of Microsoft SQL Server 2000 is used to demonstrate data-access techniques. You can obtain the Northwind database from the Microsoft Download Center (*http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en*).
- 1.6 GHz Pentium III+ processor, or faster.
- 1 GB of available, physical RAM.
- Video (800 by 600 or higher resolution) monitor with at least 256 colors.
- CD-ROM or DVD-ROM drive.
- Microsoft mouse or compatible pointing device.

Find Additional Content Online

As new or updated material becomes available that complements this book, it will be posted online on the Microsoft Press Online Developer Tools Web site. The type of material you might find includes updates to book content, articles, links to companion content, errata, sample chapters, and more. This Web site is available at *http://www.microsoft.com/learning/books/online/developer* and is updated periodically.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the companion content. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books and companion content at the following Web site:

http://www.microsoft.com/learning/support/books

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion content, or questions that are not answered by visiting the sites above, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: *Microsoft ASP.NET and AJAX: Architecting Web Applications* Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

This page intentionally left blank

Part I

The (Much Needed) Facelift for the Old Web

In this part:

This page intentionally left blank

Chapter 1

Under the Umbrella of AJAX

> *Forget what we think we know about the limitations of the Web, and begin to imagine a wider, richer range of possibilities.*
>
> *—Jesse James Garrett*

In 2007, more or less at the same time I was proudly showcasing my hot new book on ASP.NET AJAX, an old friend of mine started investigating the features of AJAX and the still largely unknown Silverlight platform. He had just been given the task of planning and coordinating a huge migration project within his company.

He spent about ten years building, maintaining, and progressively enhancing a vertical application that had won an industry award and was aimed at some special categories of professionals, such as lawyers and public accountants. At some point, his company had been acquired by a larger group and the old application had to be integrated into an existing Web platform.

With the whole company about to abruptly switch from a desktop mindset to a Web paradigm, my friend was trying to be reasonably thoughtful. He was looking for the best available tools of the current Web paradigm to minimize the pain and costs of migration while delivering an effective, desktop-like experience to the existing users. With all the buzz and hype around AJAX (and that fancy new thing known as Silverlight), his efforts seemed to be a matter of prudence and the fruit of an innate "try before you buy" attitude.

I met my friend at TechEd 2007, where I was giving a couple of presentations on the subject of ASP.NET AJAX. To my greatest surprise, at the end of my last session he came by and whispered apologetically, "Sorry Dino, but is that all of it?" He was aware that his question might sound insolent or silly and that it undermined the beautiful story I had just told the audience.

My presentation had been about how a new age of prosperity and success was about to begin for all Web developers and architects. It included the success story of how one of the building blocks of Web 2.0 came along. I told the fantastic story of how the Web was, all of a sudden, about to offer the same set of functionality as the desktop.

Unfortunately, the Web is not the desktop.

And it will never be like that, no matter which moderating suffix you attach to the word *desktop*. You can label the Web as *desktop over HTTP* or *browser-hosted desktop* or even *desktop in the cloud*. It is, and will always be, a pure marketing gimmick.

What Web Do We Want?

My friend got it right quite quickly. The Web is the Web, with its pros and cons. Using AJAX (or even Silverlight) as a shortcut or, worse yet, as a magic wand to simplify development—from developing new commercial Web sites to performing complex enterprise migration projects—is just incorrect. And it's potentially a deadly decision with regard to the assets of a company.

My friend, who looked at AJAX with a totally unbiased mind, had the farsightedness to clearly and quickly see that AJAX was something important for Web-related development but that it was not the easy fix that many people were enthusiastically depicting it to be. (And to some extent, it's still being depicted that way today, two years later.)

In light of this, the following equation is not realistic:

```
desktop = Web + AJAX
```

It doesn't work outside the dreams of some IT managers.

Although my friend had perceived the key facts about AJAX, his insight didn't solve his primary concern. By figuring out AJAX quickly, though, he was able to focus his brainstorming in the right direction and center his thoughts around the right questions.

So what are the right questions to ask about AJAX?

It's All About User Experience

As I see things, there's just one key question, and a number of more technical and in-depth questions spread out from this question later. The fundamental question is, "What Web do we want?"

Admittedly, the question implies we are not entirely happy with today's Web and are looking for a different type of Web. At the end of the day, what we all want from the Web is a much better *user experience*, in the broadest possible meaning of both the term *user* and the term *experience*.

So what does *user experience* mean to various people?

User Experience for Dummies

Jesse James Garrett has made it into the history books as the man who coined the now ubiquitous and universal acronym *AJAX*, back in 2005. (To read the full story, pay a visit to *http://www.adaptivepath.com/ideas/essays/archives/000385.php*.)

For readers who might have spent the last three years in a remote rainforest with no connectivity at all, I'll spell out the acronym here—Asynchronous JavaScript And XML. (Later in the chapter, I'll comment on the role and importance of each part of the acronym.)

Jesse James Garrett, however, is neither a software architect nor a Web developer. He calls himself an experience designer, and he's also the author of a widely referenced book on Web design titled *The Elements of User Experience* (New Riders Press, 2002). In short, Jesse James Garrett is probably one of the most qualified people in the world to give us a concise and comprehensible definition of *user experience (UE)*.

Understanding the whys and wherefores of UE is the first step to understanding what Web we want and how to go forward and look for valid technologies to employ in its implementation.

The concept of UE is made of many disparate parts. Creating a positive user experience for a Web site involves enabling end users to use the site for their own purposes, which might include business, work, personal activities and interests, and entertainment. UE is about how a Web site (or, generally, any system) is perceived, learned about, and finally used; and how a user feels about that.

A brilliant team of developers, architects, and designers might be able to serve up a set of Web functions that meet the original strategic intent. But they might fail to provide a consistent and pleasant user experience. Using Garrett's wording, the concept of a good user experience sounds like this:

> *A site that really works fulfills your strategic objectives while meeting the needs of your users. Even the best content and the most sophisticated technology won't help you balance those goals without a cohesive, consistent user experience to support it.*

A superior UE springs from a powerful mix of usability, data and work flows (often referred to as *information architecture*), appealing graphics, and interaction model. As you can see, there's no code or software architecture involved at this level. Software comes later or is developed in parallel to figuring out how to implement these characteristics. For sure, it takes the overall Web development thing to another dimension.

User Experience for the Poor Web User

Let's set aside these concepts from the field of *experience design* as applied to the Web and focus on the software side of the *new* Web. For the purposes of this book, we'll happily assume that someone on the development team has valuable ideas they want to instill in the otherwise foggy minds of the team's members.

For the end user, the next WWW (short for the *Web We Want*) is centered on providing the user with a high-quality, first-class experience when passing through your site. Whatever that means. Figuring out what that means is the job of developers, designers, and UE people.

Note This is a sort of psychological note. For about ten years, the poor Web user navigated to a site to get some sort of information related to personal or business interests—documents, reports, charts, prices, best prices, timetables, account balances, various types of news, live scores, itineraries, guides, essays, and so on.

For part of this decade, the poor Web user felt lucky and happy to draw something of value from the bottomless well of the Internet. At some point, though, while the well remained largely bottomless, other critical resources began to be scarce—bandwidth and, more importantly, patience.

The poor Web user could accept slow responses when he had enthusiasm for this new thing. But when the exciting new thing turned to a commodity, the enthusiasm vanished and the poor Web user began to wonder if there could be a better way to accomplish the same tasks. Now if there's no better way, he feels unhappy and starts looking around for smarter and more cutting-edge competitors.

User Experience for Developers

In raw developer terms, a *high-quality user experience* means essentially a more responsive application that can better deal with network latency. Web users today are more sophisticated than they were ten years ago and demand higher performance and responsiveness regardless of the latency and bandwidth hurdles you, as a developer, might have to overcome.

However, you can't change the laws of physics. It still takes electrons a certain amount of time to move from one place to another. Therefore, developers work to optimize the server-side code and logic to tweak every ounce of performance and scalability from that code. But often even this isn't enough. That's when developers look to other tricks, perhaps even very new tricks that require rethinking how Web applications interact with the user.

The tricks mostly involve asking the application to do more work on the browser's side—even in the background, when the browser is idle—by sending and requesting much less data over the wire and by repainting smaller areas of the page. In summary, we will accept more roundtrips, but each carrying only a small chunk of data and only if the communication is performed in the background and results in partial page refreshes instead of complete page browser reloads.

From a developer's perspective, this is not a small step at all. It's not merely a smart form of optimization. Instead, it's a huge jump that changes our understanding of the foundation of the Web as we know it today.

A more responsive application is also more interactive from the user's perspective. It shows animations, visual effects, and sharp graphics that change quickly and smoothly to reflect the state of the page. This aspect of the new Web is an enhancement aimed at improving the experience. However, it doesn't have much to do with hard-core, server-side development and the information architecture.

It is rather more about having enhanced graphics and layout, which are more the purview of the Web designer and artist than the implementing developer. In the end, though, some browser-side code trickery will still be necessary to give the user the impression that the page is more responsive and easier to interact with. Building a nice user experience is a team effort—design plus browser-side script.

How can developers implement more responsive and interactive Web pages? Again, let me answer this question with another question. What is a Web page made of? HTML and JavaScript. These are the pillars of Web pages as we know and write them today. Any tricks you come up with will necessarily be applied here.

User Experience for Managers

Written four years ago, Garrett's aforementioned excellent essay on AJAX is starting to look a bit—yes, let's say it—outdated. The paper discusses incontrovertible facts about the mechanics of the Web with and without AJAX. But it also contains an introduction and a conclusion that are a bit misleading when read today, four years later. (As a rule of thumb, I consider five years of software progress as the logical equivalent to a geological era. So four years are definitely a lot of time.)

Managers might sometimes read through moderately technical stuff like what you find in Garrett's work, but it's very hard for them to read between the lines and grasp the implications of a technical description. What remains in their mind is that the interactivity and responsiveness gap between desktop and Web applications is now closing thanks to AJAX.

AJAX is a big innovation and a revolutionary change for the Web. However, it's not free and often costs you quite a bit in terms of resources.

Managers see the user experience as mashups and cool features. Building a mashup, though, is not like querying a database table on a local or remote server. Using mashups makes well-designed information architecture more essential than ever. It makes software architecture slightly different and raises a whole bunch of new development issues.

Making Web sites appealing and easy to navigate is more possible with AJAX than without it. But AJAX is not magic; it will never give you a desktop platform over HTTP. And, finally, there's the matter of tradeoffs and making (ideally, correct) decisions.

Origins of the Web

The Web We Want is a Web that can deliver a much better user experience. As a Web developer or architect, your role is to increase the responsiveness of pages and the interactivity of most features. The former will likely require some architectural work on code and information; the latter just requires more script code to be put to work.

The final destination for this book is to take you to the recommended architectural changes needed to get the Web We Want.

AJAX is a revolution. Great, but why? What is wrong with the old Web?

Let's begin by looking at the limitations imposed by the origins of the Web and take our first step toward understanding why the recommended architectural changes are necessary to shift to the new, more responsive Web.

The First Cry

The Web as we know it today was prototyped in the early 1990s at CERN, the European Organization for Nuclear Research. (The acronym originates from the French name of the organization.)

Scientists at CERN worked on the concept of hypertext and arranged an ad hoc markup language for expressing interlinked text-based documents and a communication protocol for retrieving such documents. Needless to say, the markup language is HTML and the communication protocol is HTTP. HTTP in particular works on top of a Transmission Control Protocol (TCP) connection occurring over port 80 by default.

The first experiment of connecting two machines over HTTP took place in the summer of 1991. Less than two years later, the CERN waived any copyrights on it, thus officially starting the era of the World Wide Web.

A lot has happened since. We had, for instance, the browser wars. This refers to a period in the late 1990s when basically each new browser release was made to edge out competitors by developing custom extensions to the markup and building in-house technologies to improve the programmability of sites and, only as a side effect, the user experience. Standardization via the World Wide Web Consortium (W3C) committees helped to have an official specification for some Web features such as CSS and HTML Document Object Model (DOM). To be effective, though, a Web standard must be widely supported by actual browsers. It took years before all major browsers aligned to support a common set of features (often only in the realm of standard specifications) that was powerful enough to begin a new era for the Web—the AJAX era.

The Mechanics of the Web

The Web is based on a request/response model that involves a client browser and the Web server. This is shown in Figure 1-1.

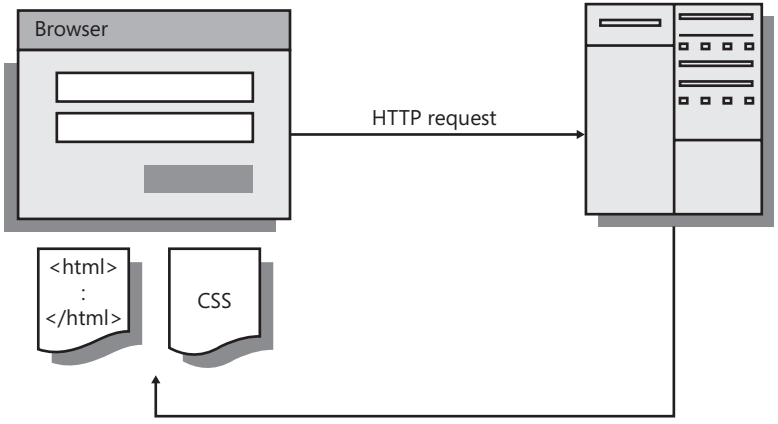


FIGURE 1-1 The traditional Web application model

According to this model, a continuous action originates a sort of stop-and-go pattern. The user interacts with the page and at some point sends a request back to the server. As the server processes the request, the user waits. Next, a new page is displayed to the user that requires some more work. The work produces a new request to the server, after which the user waits—over and over again.

In HTML, the user starts a request by hitting a submit button. Today, the standard implementation of the HTML Document Object Model also requires a script-based method, but that was not the case in the beginning. The browser interprets a submit click as the order of submitting the content of the host form to the specified action URL. Next, the browser freezes the user interface (UI) until a new HTML page is received.

In the classic Web model, the browser implements a request by sending out an HTML form and receiving a brand new HTML page.

The Original Purpose of HTML and HTTP

HTML and HTTP were created at CERN to serve a well-defined purpose: improving the flow of information across the network and sharing documents more easily using the hypertext model. A document created using HTML can contain links to other documents in the same network—for example, documents referenced in the bibliography of a scientific paper.

After they were released for public use in 1993, HTML and HTTP gained the incredible success we all know. An army of developers were able to use HTML and HTTP to build millions of pages in richer and richer Web sites. Since then, HTTP and HTML in particular have been squeezed to extract every single fragment of functionality.

Quite paradoxically, the Web was originally created to serve as an internal tool in a relatively small community of people—at least compared to today's communities. It turned out, instead, to be a monster that changed our personal lives and our businesses.

Paradox of the Web

The use of the adverb *paradoxically* is deliberate. So what is the paradox of the Web? In 15 years, developers and designers have been able to build the World Wide Web as we know it today using extremely simple tools that were not specifically designed for the job.

This process has given rise to two opposing forces. One is the force of progress, which wants the Web to become more powerful every day, with new features and applications. The other is the resistance from the limitations of the building blocks of the Web, which are not really designed to support the current workload.

The paradox lies in the fact that rebuilding the Web entirely is completely unrealistic. We need to improve it significantly, but without changing its (now inadequate) columns.

The Sturdy, Old Columns That Hold Up the Web

The Web wasn't designed for many of the purposes we use it for today. In particular, it wasn't designed to do any publishing. It wasn't specifically aimed at building the presentation layer for any distributed systems. Supporting multimedia content and rich graphics was certainly not a priority.

More importantly, it was not designed to secure its content. HTTP is an extremely simple and efficient protocol, but it's not *technologically* secure. What about HTTPS, then? HTTPS is essentially an extra layer of cryptography applied at the gate when the packet leaves or reaches the computer. HTTPS protects the message but doesn't help much with authentication and authorization. What about client certificates? Well, they certainly work. But like HTTPS, client certificates are a feature bolted onto the native (and unsecure) HTTP protocol.

Why was HTTP designed this way?

In 1991, the whole theme of Web security was unimaginable. Web security started to be a serious issue only after the bold success of the World Wide Web made it worthwhile for hackers to plan their attacks. Once we started sending money over the Web, with the associated personal information, then and only then did it make it worthwhile for malicious hackers. Before that, hacking was more a college prank than anything.

Born as a tool to manage HTTP connections and parse HTML pages, the browser became an increasingly powerful tool step by step with the rapid increase in the number of Web sites around the world.

One of the first enhancements that browsers made to the syntax of HTML was the support for a programming language—JavaScript. The first browser to deploy a JavaScript engine was Netscape Navigator 2.0 in December 1995. JavaScript was introduced to give authors of Web-deployed documents the ability to incorporate some logic and action in HTML pages.

Later on, other features were added, such as cookies, the Document Object Model (DOM) for publicly exposing in a programmable way the content being displayed, and cascading style sheets (CSS) to quickly style elements of the page. In the heat of the "war of the browsers," multiple browsers offered the same features with each using its own syntax and model. By the end of the last century, it was clear that serious Web programming couldn't be planned or actually done without common worldwide standards.

The W3C committees made it happen. As a result of their efforts, we have standard HTML and a standard JavaScript language. These are the pillars of today's Web. And for the purposes of today, they are tottering pillars.

Pillars Can't Be Changed

A pillar is not something you can replace without possibly causing the building to collapse. You can fix it or make it stronger, but you can't replace it. This is the situation we currently face with the pillars of the World Wide Web: HTML and JavaScript.

These twin supports for the Web are common and popular. Revolutionary changes to either of them would seriously affect activity on the Internet. Existing applications wouldn't be touched, but new browsers would be needed to run applications based on the modified pillars. The whole world of users would split in two—those who can change browsers and those who can't or don't want to change browsers. For the Web, which owes its popularity to being accessible to all, this is a nightmare scenario.

The Web grew too quickly to allow people to consider the adequacy or limitations of its pillars. Or, put another way, people found it easier to push the Web to the maximum instead of planning for an infrastructure with more capabilities. On the other hand, the Web is public and since 1993 it has not been the intellectual property of any company or organization. Changes to it are possible, but only if they're in compliance with accepted and recognized standards.

Note Despite the *Java* prefix in its name, the JavaScript language has very little to do with the popular Java language. JavaScript was designed to look like a simpler Java for nonexpert page authors—hence, the name. JavaScript is an interpreted, dynamic-binding, and weakly typed language with first-class functions. It has some light flavors of object orientation, it's not compiled and, maybe more importantly, it's subject to the browser's implementation.

Created to add action to Web pages, and kept simple on purpose, the JavaScript language perfectly met initial expectations for it, but it failed to exceed those expectations. That's why JavaScript is currently a pain in the neck for Web developers. But we can't replace it without breaking widely agreed upon and stable standards. This is a big part of the Web paradox.

The Biggest Benefit of AJAX

What users want is a better experience, and not all Web applications and sites offer that. For this reason, the world of the Web is moving toward AJAX.

AJAX is definitely a plus for the Web.

AJAX capabilities address the user's experience in the broadest sense—by providing a continuous feel, flicker-free updates, interface facilities, mashups, live data, and so on. AJAX is the way that's available to us to reinforce the tottering pillars safely and making them more stable.

AJAX is the only significant plus we can afford. This limitation is not merely a matter of money or economics. We simply can't get a new Web redesigned from the foundation up and implemented without disrupting or just slowing down service. The Web is now a fundamental commodity. We all need it. No serious disruptions are allowed.

What's AJAX, Exactly?

AJAX is not a technology. AJAX is not something you can install and run. AJAX doesn't require any plug-in modules and is not browser specific. Quite the opposite: the key to the success of AJAX is that virtually any browsers released in the past five years are great hosts for AJAX-based applications. So what's AJAX?

AJAX is a blanket term. As disappointing as it may sound, the term *AJAX* was coined primarily as a concise and cool way to sell a set of technologies and a new approach to Web development.

What initially was simply a clever approach to building pages, scaled to the size of an entire real-world Web front end, turned out to be the incarnation of a new paradigm for writing Web applications. The AJAX approach is probably destined to last for many years or until conditions exist for rebuilding the Web from scratch (whichever comes first).

A New Way to Do Web Programming

AJAX refers to using a set of specific browser technologies to build pages. It's amazing to note that all these technologies are nothing really new. We're talking about browser technologies that have been around for ten years now—*XMLHttpRequest*, DOM, and JavaScript.

It's simple to use these technologies to implement a given set of features in an individual page. It's much more complex to build an entire application according to the AJAX paradigm. Why?

Especially with the advent of ASP.NET, the world of Web programming has been simplified. Frameworks offer a thick layer of abstraction over basic HTML and HTTP interaction, and the ASP.NET development environment makes it easy with automated code generation and remote debugging. And all of it works on the assumption that the browser sends an HTML form to get back an HTML page, one of the foundational pillars of the Web.

It's relatively easy to change the paradigm for a single feature in a single page. It can be quite difficult, however, to extend the new paradigm to the whole application. Why? Because the world of AJAX programming has not been similarly simplified—most AJAX implementations (at least efficient and properly designed implementations) are still built by hand. But this will change.

The *XMLHttpRequest* Object

As I mentioned, AJAX stands for *Asynchronous JavaScript and XML*. Five years after its introduction, and from a more technological point of view, we can say that the first part of the acronym is acceptable but the second part is arguable.

The AJAX development model revolves around one common software element—the *XMLHttpRequest* object. The availability of this object in most browsers' object model is the key to the current ubiquity and success of AJAX applications.

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.

The *XMLHttpRequest* object allows script code to send HTTP requests and handle their responses. 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. The object is scripted from client JavaScript code and, with regard to the browser, it operates asynchronously. (With respect to your code, on the other hand, the call can be either synchronous or asynchronous.)

When a connection to a Web server is led by the browser, the current page displayed to the user is lost. The page becomes inactive and frozen as soon as the user clicks to submit the content to some remote server.

With *XMLHttpRequest*, conversely, developers directly control the placement and outcome of the request. The actual mechanics of the request/response don't make any difference to the user. However, the possibility of using *XMLHttpRequest* enables Web developers to build features that ultimately deliver a much better user experience.

The Document Object Model

In addition to *XMLHttpRequest*, a second technology contributes to making AJAX so effective and attractive—the availability of an object model that exposes the current content of the page in an updatable manner.

Microsoft pioneered updatable Web pages in the late 1990s. With Internet Explorer 4.0 (released back in 1997), Microsoft introduced Dynamic HTML (DHTML), which is a powerful combination of HTML, style sheets, and scripts that allows programmatic changes to any displayed page. Several companies since then have worked out their own DHTML object model—often referred to as the Browser Object Model (BOM). The W3C committee worked hard to get vendors to agree on an interoperable and language-neutral solution for exposing Web pages through an updatable programming interface. The result is the Document Object Model (DOM) as opposed to a browser-specific BOM.

The DOM is a platform-independent and language-neutral representation of the contents of a Web page that scripts can access and use to modify the content, structure, and style of the document.

Note I'd even dare say that without an updatable DOM the whole AJAX approach wouldn't be possible at all. Using *XMLHttpRequest*, a developer can asynchronously connect to a URL and grab some fresh data. However, how could she integrate such fresh data into the current page without an updatable representation of the page? That's why the DOM is required and critical.

The Paradigm Shift

We're all witnessing (and as users, we're also contributing) to an interesting and fairly unique phenomenon—the Web is undergoing an epochal change right before our eyes as a result of our actions.

Only ten 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.

As drastic as it might sound, the Web revolutionized the concept of an application. Now AJAX is revolutionizing the concept of a Web application.

The Web will always remain separate from the desktop, but Web applications are going to enter a new age.

The Pages-for-Forms Model

Today, communication between the browser and the Web server occurs through *forms*. A form is a collection of values stored in a group of HTML input fields.

From a user's perspective, the transition occurs through *pages*. A page is a piece of HTML markup returned by the Web server. Each user action that originates a new request for the server results in a new page (or a revamped version of the current page) being downloaded and displayed.

The browser-to-server communication employs the classic HTTP protocol. As is widely known, the HTTP protocol is stateless, which means that each request is not related to the next and no state is automatically maintained, neither on the client nor on the server.

The state objects developers know and use in, say, ASP.NET are nothing more than an abstraction provided by the server programming environment. The state objects developers know and use on the client (for example, cookies) are nothing more than an abstraction provided by the client browser.

The Pages-for-Forms model was just fine in the beginning of the Web age when pages contained little more than formatted text, hyperlinks, and maybe some images. The success of the Web has prompted users to ask for increasingly more powerful features, and it has led developers and designers to create more sophisticated services and graphics. As a result, today's pages are heavy and cumbersome.

Given the current architecture of Web applications, each user action requires a complete redraw of the page. Subsequently, 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 perfect for causing great frustration to the poor end user.

Because nobody is willing to come back to the scanty, "Times New Roman" pages of the mid-1990s, a new Web model is possible only via a smarter form of interaction between the client and the Web server.

The Data-for-Data Model

For too many years, the old Web model survived because of compatibility and reach. To accommodate businesses, Web sites had to be as easy as possible to reach for any potential customer. From a technology perspective, the AJAX revolution was ready to start back in 1999 when *XMLHttpRequest* and an updatable DOM were designed and implemented.

It took a few more years instead.

This happened because for quite some time only high-end browsers (also known as rich, up-level browsers) provided support for both *XMLHttpRequest* and an updatable DOM. For a long time, only companies that could exercise strict control over the capabilities of the client browsers were able to choose the AJAX model for their sites. In short, for too long a *rich* browser also has meant a browser with too limited *reach*. For too long, using such a browser definitely has been a bad choice for most businesses.

Around 2004, many people realized at the same time that, perhaps because of a rare astral convergence, 90 percent of the browsers available in the marketplace were supporting the same set of features—in particular, both *XMLHttpRequest* and an updatable DOM.

This made it possible for Web architects and developers to set up the Data-for-Data interaction model. According to this model, a Web page puts plain data in the body of a HTTP packet instead of inserting the content of an HTML form. And the Web server just returns plain data—not a whole new HTML page—as its response. Figure 1-2 offers a graphical view of the model.

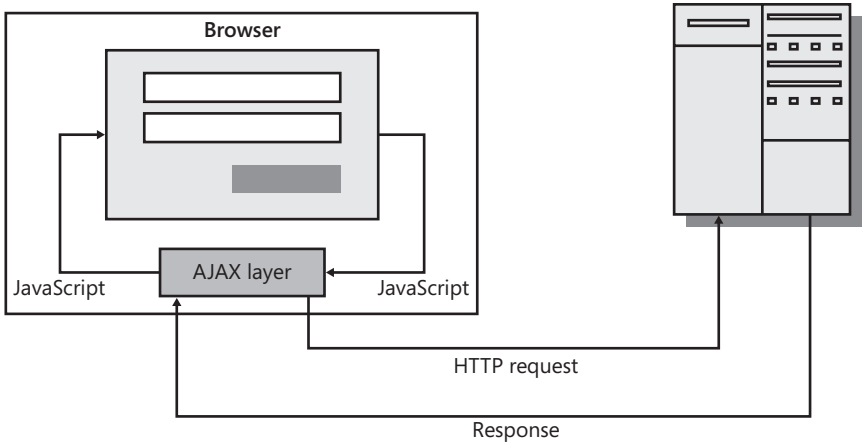


FIGURE 1-2 The AJAX Web application model

Some JavaScript code embedded in the client page triggers an HTTP request to the Web server using *XMLHttpRequest*. When the response comes back, another piece of client code parses it to some JavaScript object and, using DOM, integrates the new content in the current page.

From the user's perspective, the operation takes place asynchronously, and the user can keep on reading and scrolling the page without interruption.

Important Is the Data-for-Data interaction model—the essence of AJAX—really a faster model? Certainly, the Data-for-Data model moves around much less information than the classic HTTP page response. However, the big issue is network latency, which is more significant a factor than the transmitted quantity of data. And network latency affects the Data-for-Data interaction model because requested data is delayed. Moreover, the more roundtrips you make, the more network latency affects your application (that is, the effects are additive). So what's the point?

Performance, though, is not only made of raw numbers. Where a user and a user interface are concerned, the concept of performance morphs into the concept of *perceived* performance. A user who can keep on working with a page will feel much better than one who cannot. Therefore, data requests are made in the background and performed asynchronously. The user never knows the data was requested, and the user interface never "freezes" while waiting for new data. Most commonly, smaller portions of the page are independently updated, further providing the feeling of (increasing) perceived performance.

Is that all? As my old friend understood quite quickly, unfortunately *XMLHttpRequest* and an updatable DOM are only the starting point of a much longer revolution that necessarily will need to touch on the architecture of pages and applications.

Refactoring to AJAX: Features, Pages, and Applications

Gaining the ability to place asynchronous calls to the Web server while bypassing the browser's standard procedure is only the first, and largely preliminary, step to building an AJAX site.

When the benefits of the AJAX model are being discussed, often the following example is given. Suppose you want to know the balance of your bank account or any other simple and small piece of information. With the standard Web model, you submit a request to a server URL and wait for a new page to be (downloaded and) served. Intertwined with advertising, banners, graphics, menus, and disclaimers is the number you were looking for. With AJAX, on the other hand, the page remains up and running (with all of its banners, menus, and disclaimers) and only the number is downloaded.

Unfortunately, the example addresses only the *feature* level. It says nothing about the rest of the *page* and the rest of the *application*.

AJAX is a paradigm shift. And a paradigm shift always has a dramatic impact because it requires that people change their habits and embrace new and largely unknown practices.

Refactoring is a key word in AJAX.

What should you refactor in your application? The whole application? Or only a bunch of individual pages? Or should you simply consider optimizing just one critical feature or two?

AJAX and New Web Projects

After a decade of increasingly powerful tools and technologies designed for effective and quick development of Web sites and applications (such as ASP, Microsoft Visual InterDev, Dreamweaver, Java Server Pages, ASP.NET, and Microsoft Visual Studio), we've been pushed into the AJAX age where no such tools exist.

The world of AJAX development is not yet embraced by the tools you use. Everything you need can be manually created; however, very few tools exist. This is the issue that project leads (such as my old friend) and IT managers face when they get past the initial enthusiasm.

As I see things, there are three ways to approach AJAX. One is to just add AJAX capabilities to an existing solution or to a new solution designed in the traditional, non-AJAX way. Another is sticking to the Web paradigm (HTML and HTTP) but rethinking the architecture of the application and its implementation. This means learning new patterns, facing new issues, solving new problems, and using new tools. The third approach is to take the route of a Rich Internet Application (RIA)—a desktop-like application hosted in the Web browser via a plug-in.

I'm going to give a quick strategic overview of these three approaches in the rest of the chapter. The remainder of this book goes into more detail about a particular approach. Part I, "The (Much Needed) Facelift for the Old Web," covers the first approach in more technical depth. The second approach and RIAs are covered in Part II, "Power to the Client."

Adding AJAX Capabilities

Most Web sites today might be significantly improved in terms of usability and user experience with a touch of AJAX. As mentioned, the core of the AJAX model is an internal browser object and the DOM. The interface of both is defined according to standards—still a *de facto* standard for *XMLHttpRequest* and an official W3C standard for the DOM.

This means that adding AJAX capabilities requires only a bit of script code. You can add AJAX capabilities to any page regardless of the underlying programming platform—be it classic ASP, ASP.NET, Java Server Pages, PHP, or plain HTML.

Selective Updates

Adding AJAX capabilities entails working at the page level, when not directly at the feature level. The scaffolding of the application doesn't change, and so it is for the inspiring principles and overall architecture.

With this approach, you apply selective updates to the parts of a page that need a facelift. You do so by employing smart tricks to work around the classic behavior of the page.

Vendors provide some tools to make this process quick and effective. Effectiveness here is a critical parameter. A solution that applies AJAX updates to a Web page can be based only on JavaScript and must work well in a cross-browser manner.

The perfect example of what "adding AJAX capabilities" means is ASP.NET partial rendering, which I'll cover in Chapter 2, "The Easy Way to AJAX." Other possibilities exist, too. For example, vendors of UI suites such as Telerik, Infragistics, ComponentArt, and Gaiaware offer their own products that, in the ASP.NET world, allow you to reuse your skills entirely while getting a fully AJAX-enabled presentation layer.

Costs and Benefits

By simply adding AJAX capabilities, you don't turn your architecture upside down and you save significant time and costs. It's by far the cheapest option, and it still gets you a Web site that is perceived to be much faster than the old one.

For developers, the impact is limited, as all they have to learn is how to use a small set of new controls and features. Adding AJAX capabilities is the most conservative choice; take what you have and make it better.

In my opinion, this approach is ideal for existing Web sites when it's ascertained they need some updating. If you have a complex site and are concerned about the architecture, this option is probably as good (or as bad) as others. Selecting a different option certainly gives rise to additional issues, such as possible shortage of skills, higher learning curves, and longer development times. Like everything else in AJAX, there's a tradeoff to be considered.

Note Currently, the world of the Web is evolving and it's hard to see which products and approaches will emerge from the process. For what it's worth, this strategy has no significant future.

It certainly can be used, and it still makes your site work for you; however, the underlying approach is a dead end. It's likely that in a few years new tools will be created to make building AJAX solutions a walk in the park in much the same way it is today with classic ASP.NET.

Architecture Is the Concern

If ASP.NET fully embraces the old model of the Web, which is centered around JavaScript and HTML, should we conclude that ASP.NET is dead? And if so, what does the future have in store for us?

The ASP.NET application model based on postbacks and view states is, technologically speaking, probably a thing of the past. However, this doesn't mean that thousands of pages

will be wiped out tomorrow and that hundreds of applications must be rewritten. More simply, a superior model is coming out that is more powerful both technologically and architecturally.

To take full advantage of the AJAX model, a different architecture is necessary and new patterns must be taken into account.

Some Common Architectural Concerns

If adding AJAX capabilities to an existing site doesn't have a huge impact on any of the parts involved, why on earth should we ever consider a different approach?

In ASP.NET, the classic Web model is implemented through the Web Forms API. The Web Forms API is based on the concept of the postback. The current page contains just one HTML form and one or more submit buttons. When the user clicks, the content is uploaded and the new page is downloaded. The new page is created based on the content that page controls have stored in the view state and based on the outcomes of the postback event.

The Web Forms model was created to make Windows and Web development nearly the same in the .NET platform. ASP.NET also has the merit of bringing a new family of developers to the arena of building Web applications. For years, Web development has required a radically different set of skills (such as HTML, JavaScript, DOM, and CSS) than smart C++ developers possessed. With ASP.NET, building Web applications has become a matter of doing plain old programming with a first-class language such as C#.

The Web Forms model sacrificed, almost entirely, JavaScript and client-side interaction. With AJAX, instead, we are moving back to the original characteristics of the Web. And the Web Forms model is less adequate every day.

The Web Forms model can still work if you plan to add only a few new features. It stops working if you want to design a more interactive application from scratch.

Two Tiers and a Data Format

The original enthusiasm for AJAX tends to wane when project leads figure out what it takes to build a true AJAX application from the ground up. They can see the benefits (interactivity, responsiveness, user experience, performance, and scalability) of an AJAX application, but they find it difficult to plan for it. Why? Most often it's the lack of tools and a clear vision of the final architecture.

Don't be too surprised to see different people talk about AJAX with different, often opposite, feelings—one saying it is the next big cool thing, and the other replying that its rate of adoption is slowing down.

AJAX is a plus and a necessity. But it requires a new architecture, new patterns, and a new ad hoc platform from vendors, including Microsoft. This is coming, but slowly.