CHAPTER 1

An Introduction to Ajax, RPC, and Modern RIAs

If this is your first experience with Ajax, and even web development in general, this chapter will serve as a good introduction to get you up to speed for what is to come. If, however, you are a relatively experienced developer, and especially if Ajax is not new to you, feel free to skip most of this chapter, as it will likely be just a review for you (although I do suggest reading the section introducing DWR near the end, since that's presumably why you're here!). This chapter explores the wonderful world of Ajax by examining how applications in general, web applications in particular, have been developed over the past decade and a half or so. You'll discover an interesting cycle in terms of the basic structure of applications. We'll look at our first example of Ajax in action and talk about why Ajax is important and how it can fundamentally alter how you develop applications. We'll also briefly touch on some of the alternatives to Ajax, and some of the existing libraries and toolkits that make Ajax easier. Last but not least, I'll give you a first quick look at DWR, just to whet your appetite for what is to come later.

A Brief History of Web Development: The "Classic" Model

In the beginning, there was the Web. And it was good. All manner of catchy new words, phrases, and terms entered the lexicon, and we felt all the more cooler saying them. (Come on, admit it, you felt like Spock the first couple of times you used the word "hypertext" in conversation, didn't you?) *Webapps*, as our work came to be known, were born. These apps were in a sense a throwback to years gone by when applications were hosted on "big iron" and were accessed in a timeshare fashion. They were in no way, shape, or form as "flashy" as the Visual Basic, PowerBuilder, and C++ *fat clients* that followed them (which are still used today, of course, although less and less so with the advent of webapps).

In fact, if you really think about it, application development has followed a very up-and-down pattern, and if you walk down the timeline carefully and examine it, this pattern begins to emerge. Starting with what I term the "modern" era, that is, the era in which applications took a form that most of us would basically recognize, we first encounter simple terminal emulation applications (for the sake of this argument, we'll skip the actual terminal period!) used to access remotely running processes. Screens like the one shown in Figure 1-1 were typical of those types of applications.

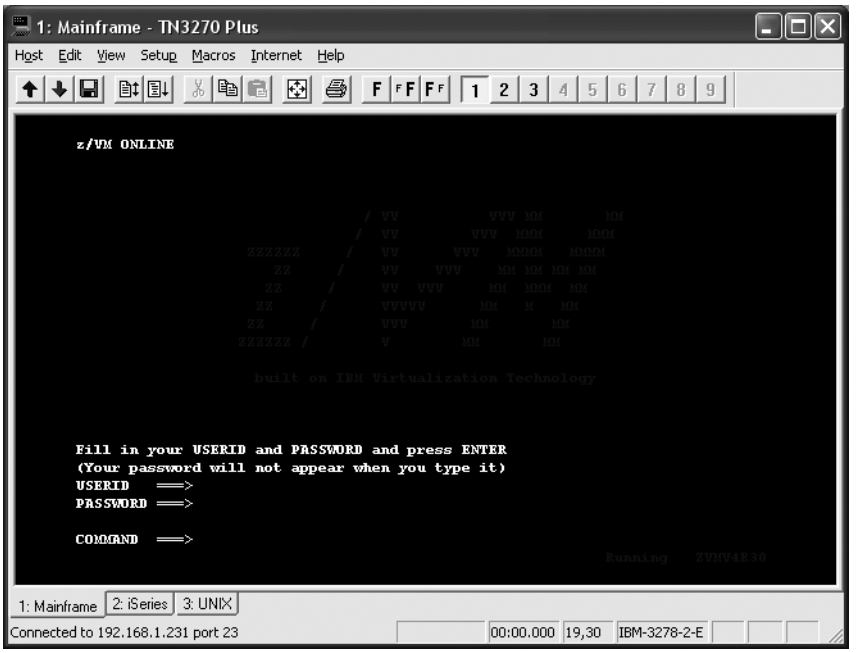


Figure 1-1. *3270 mainframe "green screen" terminal display*

3270 screens are, of course, completely relevant in the sense that they are still used quite a bit, especially in the business world, as anyone who has done any sort of mainframe work can attest to. There are two interesting things to note, for the sake of this discussion. First, notice the simple nature of the user interfaces (UIs) back then. They were text-only, usually limited to 80 columns by 25 lines of text, with extremely limited data entry capabilities, essentially just editable mapped regions. Things like drop-down menus, check boxes, and grids were completely unknown in this domain. If it was a well-written application, you would be fortunate and have a real menu like so:

C. Create record

D. Delete record

E. Edit record

If you were unlucky, you would just have something like this:

..... 01A7C0D9ABABAC00

..... 89A6B3E34D79E998

If you have never worked on a mainframe, let me briefly explain what that is. For editing files (called *data sets*) on a mainframe, you usually use a tool called *TSO/ISPF*. This is just a form of text editor. This editor can be flipped between textual display and hex display, and the preceding is the hex display version. The dots that start each line make up the command area.

For instance, to insert a line above the line that begins with 89, you would go to the first dot in that line and replace it with i, then press the Enter key. If you wanted to delete that line, plus the line that starts with 01, you would go to the line that starts with 01, type dd over the first two dots, then go to the line you just inserted and put dd there as well, then press Enter (dd is for deleting a block of lines; you can use a single d to delete a single line).

MORE ON 3270, FOR YOU HISTORY BUFFS OUT THERE

TN3270, or just plain 3270 as it is commonly called, was created by IBM in 1972. It is a form of *dumb terminal*, or *display device*, usually used to communicate with IBM mainframe systems. Because these displays typically only showed text in a single color, namely green, they have come to be informally referred to as *green screens*. 3270 was somewhat innovative in that it accepted large blocks of data known as *data-streams*, which served to minimized the number of I/O interrupts required, making it (at the time) a high-speed proprietary communication interface.

3270 terminals, the physical entities that is, have not been manufactured for a number of years; however, the 3270 protocol lives on to this day via emulation packages (Attachmate being a big one in the business world). Green screens can still be seen around the world though, most notably in call centers of many large businesses, where they are still the tool of choice . . . some would say tool of inertia. In any case, they are still used in many places because, ironically, these interfaces tend to be more productive and efficient than competing (read: web-based) interfaces. Sometimes, simplicity really is king!

Second, and more important here, is the question of what happens when the user performs an action that requires the application to do something. In many cases, what would happen is that the mainframe would redraw the entire screen, even the parts that would not change as a result of the operation. Every single operation occurred on the mainframe, and there was no local processing to speak of. Not even simple input validation was performed on the client; it was simply a view of a remote application's state, nothing more.

Dawn of a Whole New World: The PC Era

With the advent of the PC, when the amount of local processing power advanced orders of magnitude, a new trend emerged. At this point we began to see applications hosted locally instead of on central mainframes where at least some portion of the application actually executed locally. Many times, the entire application itself was running on the machine that the user was using. With the growth in popularity of Microsoft Windows more than anything else (I know, some will cringe at the thought of crediting Microsoft with **anything**, but there's no sense denying the reality of things), fat clients, as they came to be known, were suddenly the de facto standard in application development. The UI available in this paradigm was immensely more powerful and user-friendly, but the central hardware faded in importance for the most part (things like database servers notwithstanding). Screens like the one in Figure 1-2 became the norm.

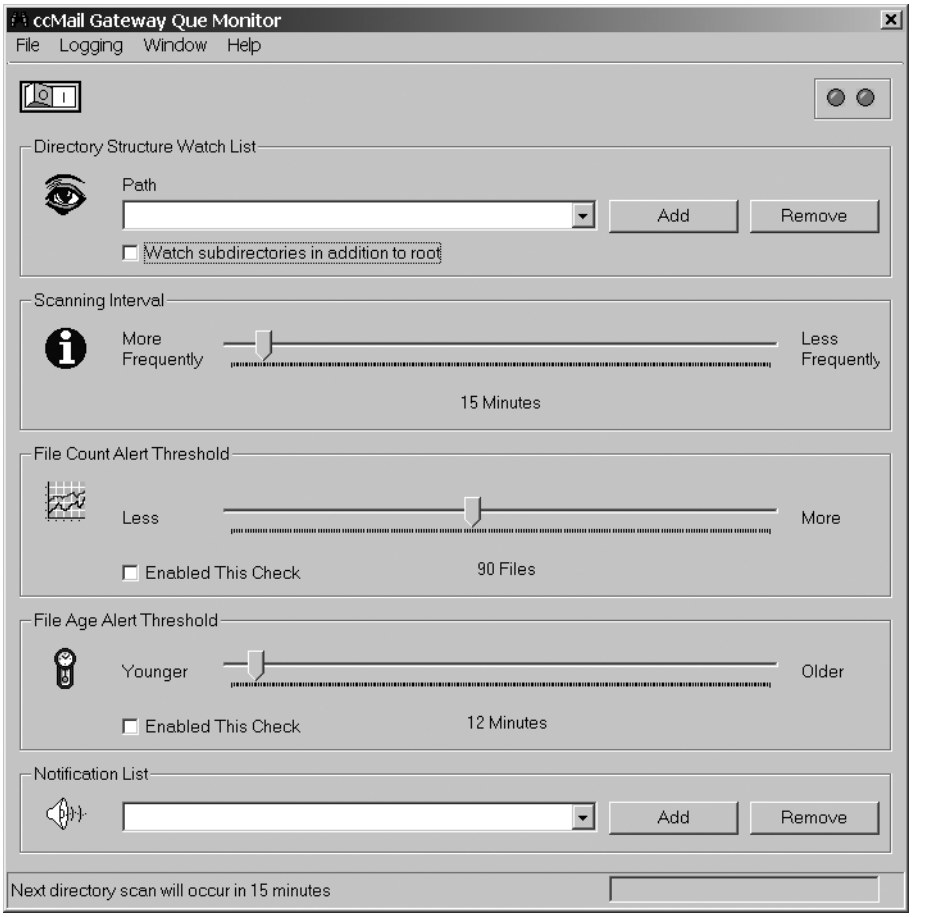


Figure 1-2. *A typical fat-client application*

More for my own ego than anything else, I'll tell you that this is a Visual Basic application I wrote pretty close to ten years ago. ccMail, which is a file-based e-mail system seldom used today but quite popular years ago, works by storing messages on a file system in queues. For a while, the IT guys at my company had an issue with messages getting "stuck" in queues. As you can imagine, that's not supposed to happen; they are supposed to hit a queue, and after some relatively brief period of time (a minute or two at most) get forwarded to their destination. Instead of finding the root cause of the problem, we got creative and wrote this application to check the queues for messages that were too old, or for an overabundance of messages in a queue, and send e-mail notifications alerting someone to the situation. Fortunately, we stopped using ccMail a short time after that, so it didn't matter that we never found the real problem.

Note how much richer the available UI metaphors are. It should come as no surprise to you and the rest of the user community out there that this is universally seen as "better" (you **do** see this as better, don't you?). Better is, of course, a relative term, and in some cases it is not better. You might think that people doing heads-down data entry all day might actually prefer those old green screens more because they lend themselves to more efficient keyboard-based data entry, and you would be right in many cases. No fussing with a mouse. No pointing and clicking. No need to take their eyes off the document they are keying off of to save a record. While all of that is true, it cannot be denied that, by and large, people will choose a fat-client version of a given application over a text-only version of it any day of the week. Go ahead, try it, I dare you! Take an existing mainframe application and put it side by side with a fat-client version of the same application and see how many users actually want the mainframe version. I'll give you a hint: it will be less than one, but not negative (although you may have to make the fat-client version "keying friendly" to make them happy, but that's another point).

Thinking about how the application actually functions, though, what happens here when the user clicks a button, for example, or slides a slider, or clicks a menu item? In most cases, only some region of the screen will be updated, and no external system is interacted with (usually). This is obviously more efficient and in all likelihood more user-friendly as well.

Yet Another Revolution: Enter the Web

But what happened next in our timeline? A bit of a monkey wrench got thrown in the works with the rise of the Internet, and more specifically, the World Wide Web component, or just the Web for short (remember, the Web is not, in and of itself, the Internet!). With the emergence of the Web, screens like the one in Figure 1-3 became commonplace.

Wait a second, what happened? Where did all our fancy radio buttons, 3D buttons, list boxes, and all that go? The first iteration of the Web looked a heck of a lot visually like the old mainframe world. More important, though, is what was happening under the hood: we went back to the old way of doing things in terms of centralized machines actually running the applications, and entire screens at a time being redrawn for virtually every user interaction.

In a very real sense, we took a big step backward. The screen was redrawn by the server and returned to the user with each operation (amazingly, a paradigm that is still very much with us today, although if you're reading this book, you likely see the logic in changing that, and we'll do our part to make that happen!). Each and every user interaction (ignoring client-side scripting for the moment because it was not immediately available to the first web developers) required a call to a server to do the heavy lifting. See, we went back to the mainframe way of doing things, more or less! We didn't all just lose our minds overnight, of course; there were some good reasons for doing this. Avoidance of *DLL Hell*, the phenomenon in the Windows world where library versions conflict and cause all sorts of headaches, was certainly one of them, because that is even today a common complaint about fat clients on the Windows platform (newer versions of Windows alleviate it somewhat, but not entirely).

Another reason was the need to distribute applications. When an application runs on a centralized server, it can be accessed from any PC with a web browser without having to first install it. Another good reason was the relative ease of application development. At least in the beginning when webapps were fairly simple things done with little more than HTML and simple back-end CGI programs, almost anyone could quickly and easily pick it up. The learning curve was not all that high, even for those who had not done much application development before.

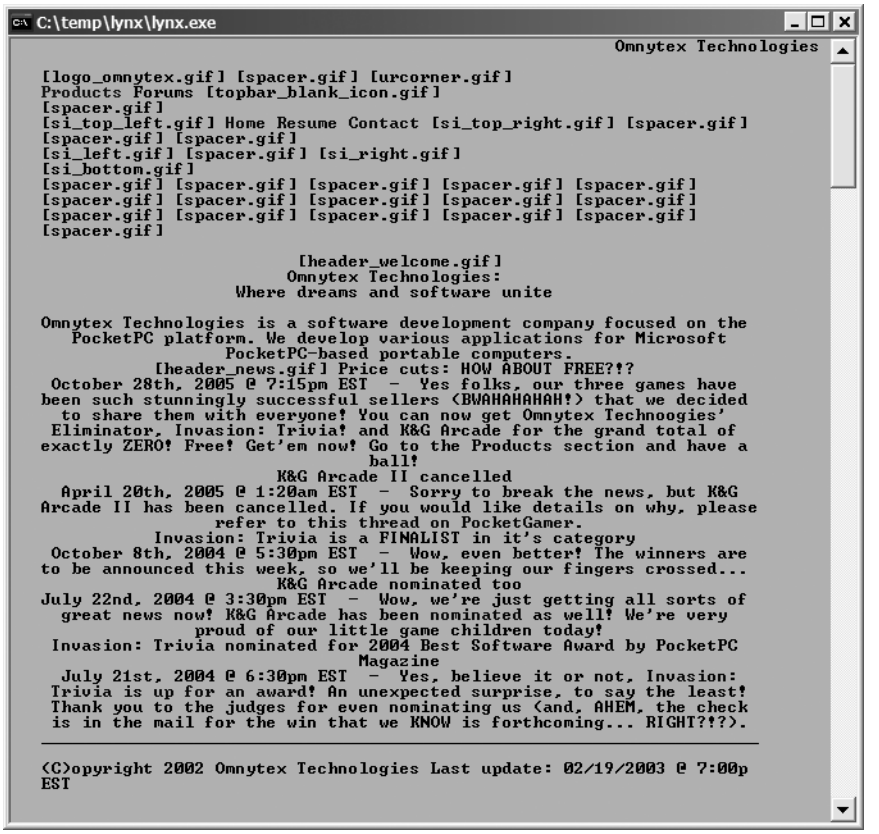


Figure 1-3. *The Omnytex Technologies site (*`www.omnytex.com`*) as seen in Lynx, a text-based browser*

MORE ON CGI

CGI, which stands for Common Gateway Interface, is a protocol used to interface an external software application with some information server, typically a web server. This mechanism allows for passing of requests from the client, a web browser in the case of a web server, to the server. The server then returns the output of the CGI application execution to the client.

CGI began its life based on discussions on the mailing list www-talk involving many individuals including Rob McCool. Rob was working at NCSA (National Center for Supercomputing Applications), and he wrote the initial CGI specification based on those discussions and also provided the first "reference" implementation for the NCSA HTTPd web server (which, after morphing a lot and being renamed, became the ubiquitous Apache web server we all know and love today). This implementation, and in fact most implementations ever written (although not all), used system environment variables to store the parameters passed in from the client and spawned a new thread of execution for executing the CGI application in, which as you can guess is highly inefficient. Later incarnations got around this problem in various creative ways.

CGI is still seen today, although far less often than before.

Of course, the Web grew up mighty quick! In what seemed like the blink of an eye, we moved from Figure 1-3 to Figure 1-4.

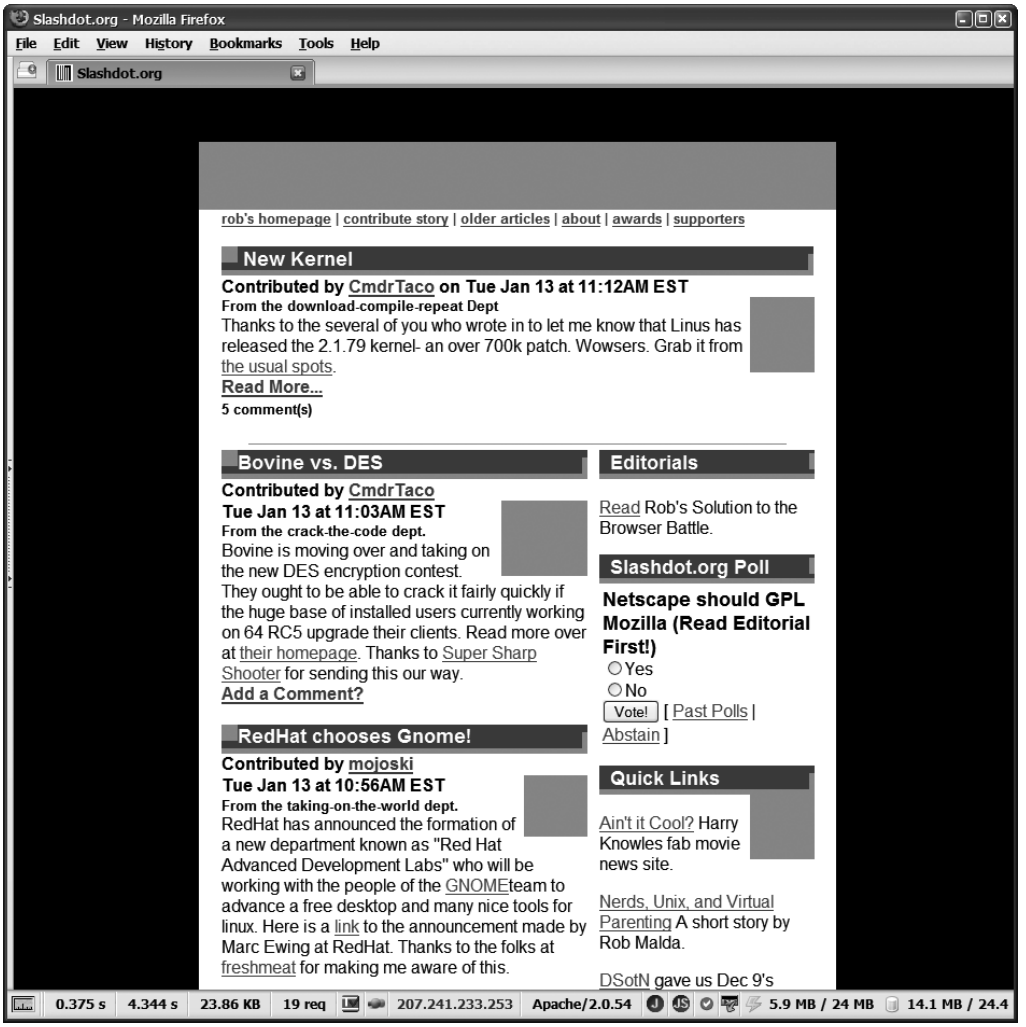


Figure 1-4. *Slashdot, circa 1998*

Now, that certainly looks a bit better, from a visual standpoint for sure. In addition to just the visual quality, we had a more robust palette of UI widgets available like drop-down menus, radio buttons, check boxes, and so forth. In many ways it was even better than the fat clients that preceded the rise of the Web because *multimedia* presentation was now becoming the norm. Graphics started to become a big part of what we were doing, as did even sound and video eventually, so visually things were looking a lot better.

What about those pesky user interactions, though? Yes, you guessed it: we were still redrawing the entire screen each time at this point. The beginnings of client-side scripting emerged though, and this allowed at least some functionality to occur without the server, but

by and large it was still two-tier architecture: a view tier and, well, the rest! For a while we had frames too, which alleviated that problem to some degree, but that was really more a minor divergence on the path than a contributor to the underlying pattern.

Before you could say "Internet time," though, we found ourselves in the "modern" era of the Web, or what we affectionately call "today" (see Figure 1-5).

Figure 1-5. *The "modern" Web, the Omnytex Technologies site as the example*

So, at least visually, as compared to the fat clients we were largely using throughout the late '80s and early '90s, we are now at about the same point, and maybe even a bit beyond that arguably. What about the UI elements available to us? Well, they are somewhere in the middle.

We have radio buttons and check boxes and drop-down menus and all that, but they are not quite as powerful as their fat-client counterparts. Still, it is clearly better than the text regions we had before the Web.

The term *Internet time* originated during the bubble years of the late '90s. It was (and still is, although like "Where's the beef?", it's not used much any more) a term meant to espouse the belief that everything moved faster on the Internet because getting information out with the Internet was so much faster than before.

But the underlying problem that we have been talking about all along remains: we are still asking the server to redraw entire screens for virtually every little user event and still asking the server to do the vast majority of the work of our application. We have evolved slightly from the earlier incarnations of the Web to the extent that client-side scripting is now available. So in fact, in contrast to how it was just a few years ago, not every user event requires a server call. Simple things like trivial data entry validations and such are now commonly performed on the client machine, independent of the server. Still, the fact remains that most events do require intervention by the server, whether or not that is ideal to do so. Also, where a user has scripting disabled, good design dictates that the site should "degrade" gracefully, which means the server again takes on most of the work anyway.

At this point I would like to introduce a term I am fond of that I may or may not have invented (I had never heard anyone use it before me, but I cannot imagine I was the first). The term is the *classic* Web. The classic Web to me means the paradigm where the server, for nearly every user event, redraws the entire screen. This is how webapps have been built for about 15 years now, since the Web first began to be known in a broad sense.

The webapps that have been built for many years now—indeed are still built today on a daily basis—have one big problem: they are by and large redrawing the entire screen each time some event occurs. They are intrinsically server-centric to a large degree. When the user does something, beyond some trivial things that can be processed client side such as mouse-over effects and such, the server must necessarily be involved. It has to do some processing, and then redraw what the user sees to incorporate the applicable updated data elements. This is, as I'm sure you have guessed, highly inefficient.

You may be asking yourself, "If we've been doing the classic Web thing for so long, and even still do it today, what's wrong with it?" We'll be discussing that very question next, but before we go further with that thought, let's go back briefly to our timeline. Do you see the pattern now? We started with centralized applications and complete redrawing of the screen in response to every single user action. Then we went to fat clients that largely ran locally and only updated the relevant portion of the screen. Then we went back to centralized applications, and also back to the central machine redrawing the entire screen.

So, what comes next? Well, simply put, the pendulum is in full swing right back the other direction, and with a ton of momentum to boot! Let's not jump to that quite yet though—let's first discuss why that shift is necessary in the first place.

What's So Wrong with the Classic Web?

In many ways, absolutely nothing! In fact, there is still a great deal of value to that way of designing webapps. The classic Web is great for largely linear application flows, and is also a wonderful medium for delivering information in an accessible way. It is easy for most people

to publish information and to even create rudimentary applications with basic user interactions. The classic Web is efficient, simple, ubiquitous, and accessible to most people. It is not, however, an ideal environment for developing complex applications. The fact that people have been able to do so to this point is a testament to the ingenuity of engineers rather than an endorsement of the Web as an application distribution medium!

It makes sense to differentiate now between a webapp and a web site, as summarized in Table 1-1.

Table 1-1. *Summary Comparison of Webapps vs. Web Sites*

Webapps	Web Sites
Designed with much greater user interaction in mind.	Very little user interaction aside from navigation.
Main purpose is to perform some function or functions, usually in real time, based on user inputs.	Main purpose is to deliver information, period.
Uses techniques that require a lot more of the clients accessing them in terms of client capabilities.	Tends to be created for the lowest common denominator.
Accessibility tends to take a back seat to functionality out of necessity and the simple fact that it's hard to do complex and yet accessible webapps.	Accessibility is usually considered and implemented to allow for the widest possible audience.
Tends to be more event-based and nonlinear.	Tends to be somewhat linear with a path the user is generally expected to follow with only minor deviations.

There are really two different purposes served by the Web at large. One is to deliver information. In this scenario, it is very important that the information be delivered in a manner that is readily accessible to the widest possible audience. This means not only people with disabilities who are using screen readers and such devices, but also those using more limited capability devices like cell phones, PocketPCs, and kiosk terminals. In such situations, there tends to be no user interactions aside from jumping from static document to static document, or at most very little interaction via simple fill-out forms. This mode of operation for the Web, if you will, can be classified as web sites.

The category of webapps on the other hand has a wholly different focus. Webapps are not concerned with simply presenting information, but in performing some function based on what the user does and what data the user provides. The user can be another automated system in many cases, but usually we are talking about real flesh-and-blood human beings. Webapps tend to be more complex and much more demanding of the clients that access them. In this case, "clients" refer to web browsers.

This does not have to be true. There are indeed some very complex webapps out there that do not require any more capability of clients than a web site does. While it clearly is not impossible to build complex applications in the "web site" mode, it is limiting and more difficult to do well in terms of user-friendliness, and it tends to require sacrifices in terms of capabilities or robustness of the capabilities provided.

This is the problem with the classic model: you generally have to design to the lowest common denominator, which severely limits what you can do.

Let's think a moment about what the lowest common denominator means in this context. Consider what you could and could not use to reach the absolute widest possible audience out there today. Here is a list of what comes to mind:

- *Client-side scripting*: Nope, you could not use this because many mobile devices do not yet have scripting support or are severely limited. This does not even consider those people on full-blown PCs who simply choose to disable scripting for security or other reasons.
- *Cascading Style Sheets (CSS)*: You could use it, but you would have to be very careful to use an older specification to ensure most browsers would render it properly—none of the fancier CSS 2.0 capabilities, for instance.
- *Frames*: No, frames are not universally supported, especially on many portable devices. Even when they are supported, you need to be careful because a frame is essentially like having another browser instance in terms of memory (and in some cases it very literally is another browser instance), and this can be a major factor in mobile devices.
- *Graphics*: Graphics can be tricky in terms of accessibility because they tend to convey more information than an `alt` attribute can. So, some of the meaning of the graphic can easily be lost for those with disabilities, no matter how vigilant you are to help them.
- *Newer HTML specs*: There are still many people out there using older browsers that may not even support HTML 4.01, so to be safe you will probably want to code to HTML 3.0. You will lose some capabilities obviously in doing so.

Probably the most important element here is the lack of client-side scripting. Without client-side scripting, so many possibilities are not available to you as a developer. Most important in the context of this book is the fact that you have virtually no choice but to have the server deal with every single user interaction. You may be able to get away with some meta-refreshes in frames in some cases, or perhaps other tricks of the trade, but frames too are on the list, so you might not even have that option!

You may be wondering, "What is the problem with the server rendering entire pages?" Certainly there are benefits, and the inherent security of being in complete control of the runtime state of the application (i.e., the user can't hack the code, at least not as easily) is a big one. Not having to incur the initial startup delay of downloading the code to the client is another. However, there are indeed some problems that in many cases overshadow the benefits. Perhaps the most obvious is the load on the server. Asking a server to do all this work on behalf of the client many times over across a number of simultaneous requests means that the server must be more robust and capable than it might otherwise need to be. This all translates to dollars and cents in the long run because you will have to purchase more server power to handle the load. Now, many people have the "just throw more hardware at it" mentality, and we are indeed in an age where that works most of the time. But that is much like saying that because we can throw bigger and bigger engines in cars to make them go faster, then that's exactly what we should always do when we need or want more speed. In fact, we can make cars go faster by making a smaller engine more efficient in design and execution, which in many ways is much more desirable, that is, if you like clean, fresh air to breathe!

Perhaps an even better metaphor would be to say it is like taking a midsized car and continually adding seats tied to it around the outside to allow for more people to ride "in" the car rather than trying to find a more efficient way for them to get where they are going. While this

duct-tape solution might work for a while, eventually someone is going to fall off and get crushed by the 18-wheeler driving behind us!

Another problem with the server-does-it-all approach is that of network traffic. Network technology continues to grow in leaps and bounds at a fantastic rate. Many of us now have broadband connections in our homes that we could not fully saturate if we tried (and I for one have tried!). However, that does not mean we should have applications that are sending far more information per request than they need to. We should still strive for thriftiness, should we not?

The other big problem is simply how the user perceives the application. When the server has to redraw the entire screen, it generally results in a longer wait time to see the results, not to mention the visual redrawing that many times occurs in webapps, flickering, and things of that nature. These are things users universally dislike in a big way. They also do not like losing everything they entered when something goes wrong, which is another common failing of the classic model.

At the end of the day, the classic model still works well on a small scale and for delivering mostly static information, but it doesn't scale very well, and it doesn't deal with the dynamic nature of the Web today nearly as well. In this context, "scale" refers to added functionality in the application, not simultaneous request-handling capability (although it is quite possible that is in play, too). If things do not work as smoothly, or if breakages result in too much lost, or if perceived speed is diminished, then the approach didn't scale well.

The classic model will continue to serve us well for some time to come in the realm of web sites, but in the realm of webapps, the realm you are likely interested in if you are reading this book, its demise is at hand, and its slayer is the hero of our tale: Ajax!

Enter Ajax

Ajax (see Figure 1-6 . . . now you'll always know what code and architectures would look like personified as a plucky super hero!) came to life, so to speak, at the hands of one Jesse James Garrett of Adaptive Path (`www.adaptivepath.com`). I am fighting my natural urge to make the obvious outlaw jokes here! Mr. Garrett wrote an essay in February 2005 (you can see it here: `www.adaptivepath.com/publications/essays/archives/000385.php`) in which he coined the term Ajax.

Figure 1-6. *Ajax to the rescue!*

In Figures 1-7 through 1-9, you can see some examples of Ajax-enabled applications. There are, of course, tons of examples from which to choose at this point, but these show diversity in what can be accomplished with Ajax.

Figure 1-7. *The PhotoShare application, one of the Ajax applications from my first book,* Practical Ajax Projects with Java Technology

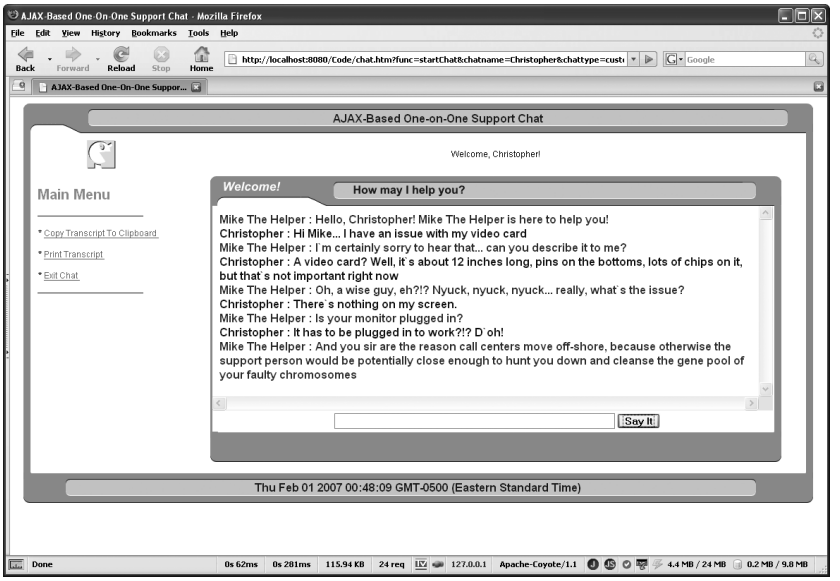


Figure 1-8. *An Ajax-based chat application, this one from my second book,* Practical JavaScript, DOM Scripting, and Ajax Projects *(that's the good thing about writing subsequent books: you have source material for new ones!)*

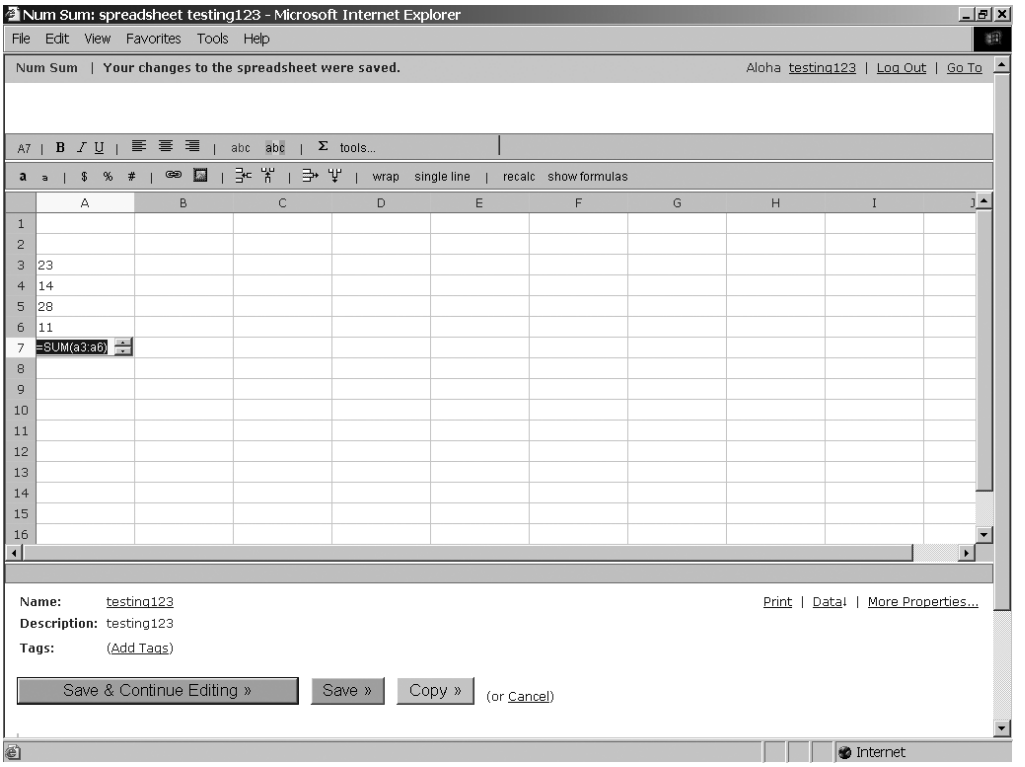


Figure 1-9. *Num Sum is an excellent Ajax-based spreadsheet application.*

Ajax, as I'd be willing to bet my dog you know already (well, not really, my wife and kids will kill me if I gave away the family dog, although my wallet would thank me), stands for Asynchronous JavaScript and XML. The interesting thing about Ajax, though, is that it doesn't have to be asynchronous (but virtually always is), doesn't have to involve JavaScript (but virtually always does), and doesn't need to use XML at all (and more and more frequently doesn't). In fact, one of the most famous Ajax examples, Google Suggest, doesn't pass back XML at all! The fact is that it doesn't even pass back data per se; it passes back JavaScript that contains data! (The data is essentially "wrapped" in JavaScript, which is then interpreted and executed upon return to the browser. It then writes out the list of drop-down results you see as you type.)

Ajax is, at its core, an exceedingly simple, and by no stretch of the imagination original, concept: it is not necessary to refresh the entire contents of a web page for each user interaction, or each *event*, if you will. When the user clicks a button, it is no longer necessary to ask the server to render an entirely new page, as is the case with the classic Web. Instead, you can define regions on the page to be updated, and have much more fine-grained control over user events as well. No longer are you limited to simply submitting a form or navigating to a new page when a link is clicked. You can now do something in direct response to a non-submit button being clicked, a key being pressed in a text box—in fact, to any event happening! The server is no longer completely responsible for rendering what the user sees; some of this logic is now performed in the user's browser. In fact, in a great many cases it is considerably better

to simply return a set of data and not a bunch of markup for the browser to display. As we traced along our admittedly rough history of application development, we saw that the classic model of web development is in a sense an aberration to the extent that we actually had it right before then!

Ajax is a return to that thinking. Notice I said "thinking." That should be a very big clue to you about what Ajax really is. It is not a specific technology, and it is not the myriad toolkits available for doing Ajax, and it is not the `XMLHttpRequest` object (don't worry if you've never seen that before, we'll get to it soon enough). It is a way of thinking, an approach to application development, a mindset.

The interesting thing about Ajax is that it is in no way, shape, or form new; only the term used to describe it is. I was reminded of this fact a while ago at the Philadelphia Java Users Group. A speaker by the name of Steve Banfield was talking about Ajax, and he said (paraphrasing from memory), "You can always tell someone who has actually done Ajax because they are pissed that it is all of a sudden popular." This could not be truer! I was one of those people doing Ajax years and years ago; I just never thought what I was doing was anything special and hence did not give it a "proper" name. Mr. Garrett holds that distinction.

I mentioned that I personally have been doing Ajax for a number of years, and that is true. What I did not say, however, is that I have been using XML or that I have been using the `XMLHttpRequest` object, or any of the Ajax toolkits out there. I've written a number of applications in the past that pulled tricks with hidden frames and returning data to them, then using that data to populate existing portions of the screen. This data was sometimes in the form of XML, other times not. The important point here is that the approach that is at the heart of Ajax is nothing new as it does not, contrary to its very own name, require any specific technologies (aside from client-side scripting, which is, with few exceptions, required of an Ajax or Ajax-like solution).

When you get into the Ajax frame of mind, which is what we are really talking about, you are no longer bound by the rules of the classic Web. You can now take back at least some of the power the fat clients offered, while still keeping the benefits of the Web in place. Those benefits begin, most importantly perhaps, with the ubiquity of the web browser.

SHAMELESS SELF-PROMOTION

If you're looking for a book on Ajax that touches on a number of the libraries out there available to help you get the job done, might I suggest my own *Practical Ajax Projects with Java Technology* (Apress, 2006)? It not only covers DWR, but also Dojo, the AjaxParts Taglib (APT) in Java Web Parts (JWP), and more. It does so in the same way as this book, that is, as a collection of full, working applications.

If you're looking for something that covers Ajax, but not as the focal point, then you might be interested in my book *Practical JavaScript, DOM Scripting, and Ajax Projects* (Apress, 2007). Whereas the first book is very much Java-centric, the second virtually never touches on the server side of things, squarely focusing on the client side and covering even more libraries: Dojo, Prototype, MooTools, and YUI, just to name a few.

Both of these books should be on your bookshelf because, aside from the fact that my son is addicted to video games and needs to feed his habit and because my daughter is a true princess by any definition and needs all the clothes and trimmings to go along with it, they're good references and learning materials besides (and hopefully there's some entertainment value there as well!).

Have you ever been at work and had to give a demo of some new fat-client app, for example, a Visual Basic app, that you ran on a machine you have never touched before? Ever have to do it in the boardroom in front of top company executives? Ever had that demo fail miserably because of some DLL conflict you couldn't possibly anticipate (see Figure 1-10)? You are a developer, so the answer to all of those questions is likely yes (unless you work in the public sector, and then it probably was not corporate executives, which I suppose means you may have run the risk of being lined up against a wall and shot for your "crimes," but either way, you get the point). If you have never done Windows development, you may not have had these experiences (yeah, right . . . if you believe it only happens on Windows, then I've got a big hunk of cheese to sell you . . . it's on display every evening, just look up in the sky and check it out). You will have to take my word for it when I say that such situations were, for a long time, much more common than any of us would have liked. With a web-based application, this is generally not a concern. Ensure the PC has the correct browser and version, and off you go 98 percent of the time.

Figure 1-10. *We've all been there: live demos and engineers do not mix!*

The other major benefit of a webapp is distribution. No longer do you need a 3-month shakedown period to ensure your new application does not conflict with the existing suite of corporate applications. An app running in a web browser, security issues aside, will not affect, or be affected by, any other application on the PC (and I am sure we all have war stories about exceptions to that, but they are just that: exceptions!).

Of course, you probably knew those benefits already, or you probably wouldn't be interested in web development in the first place, so we won't spend any more time on this.

Why Is Ajax a Paradigm Shift? On the Road to RIAs

Ajax does in fact represent a paradigm shift for some people (even most people, given what most webapps are today) because it can fundamentally change the way you develop a webapp. More important perhaps is that it represents a paradigm shift for the *user*, and in fact it is the user who will drive the adoption of Ajax. Believe me; you can no longer ignore Ajax as a tool in your toolbox.

Put a non-Ajax webapp in front of users, and then put that same app using Ajax techniques in front of them, and guess which one they are going to want to use all day nine times out of ten? The Ajax version! They will immediately see the increased responsiveness of the application and will notice that they no longer need to wait for a response from the server while they stare at a spinning browser logo wondering if anything is actually happening. They will see that the application alerts them on the fly of error conditions they would have to wait for the server to tell them about in the non-Ajax webapp. They will see functionality like type-ahead suggestions and instantly sortable tables and master-detail displays that update in real time—things that they would *not* see in a non-Ajax webapp. They will see maps that they can drag around like they can in the full-blown mapping applications they spent $80 on before. All of these things will be obvious advantages to the user. Users have become accustomed to the classic webapp model, but when confronted with something that harkens back to those fat-client days in terms of user-friendliness and responsiveness, there is almost an instantaneous realization that the Web as they knew it is dead, or at least should be!

If you think about many of the big technologies to come down the pike in recent years, it should occur to you that we technology folks rather than the users were driving many of them. Do you think a user ever asked for an Enterprise JavaBean (EJB)–based application? No, we just all thought it was a good idea (how wrong we were there!). What about web services? Remember when they were going to fundamentally change the way the world of application construction worked? Sure, we are using them today, but are they, by and large, much more than an interface between cooperating systems? Not usually. Whatever happened to Universal Description, Discovery, and Integration (UDDI) directories and giving an application the ability to find, dynamically link to, and use a registered service on the fly? How good did that sound? To us geeks it was the next coming, but it didn't even register with users.

Ajax is different, though. Users can see the benefits. They are very real and very tangible to them. In fact, we as technology people, especially those of us doing Java web development, may even recoil at Ajax at first because more is being done on the client, which is contrary to what we have been drilling into our brains all these years. After all, we all believe scriptlets in JavaServer Pages (JSPs) are bad, eschewing them in favor of custom tags. Users do not care about elegant architectures and separation of concerns and abstractions allowing for code reuse. Users just want to be able to drag the map around in Google Maps (see Figure 1-11) and have it happen in real time without waiting for the whole page to refresh like they do (or did anyway) when using Yahoo's mapping solution.

The difference is clear. They want it, and they want it now (stop snickering in your head, we're all adults here!).

Ajax is not the only new term floating around these days that essentially refers to the same thing. You may have also heard of Web 2.0 and RIAs. RIA is a term I particularly like, and I will discuss it in a bit.

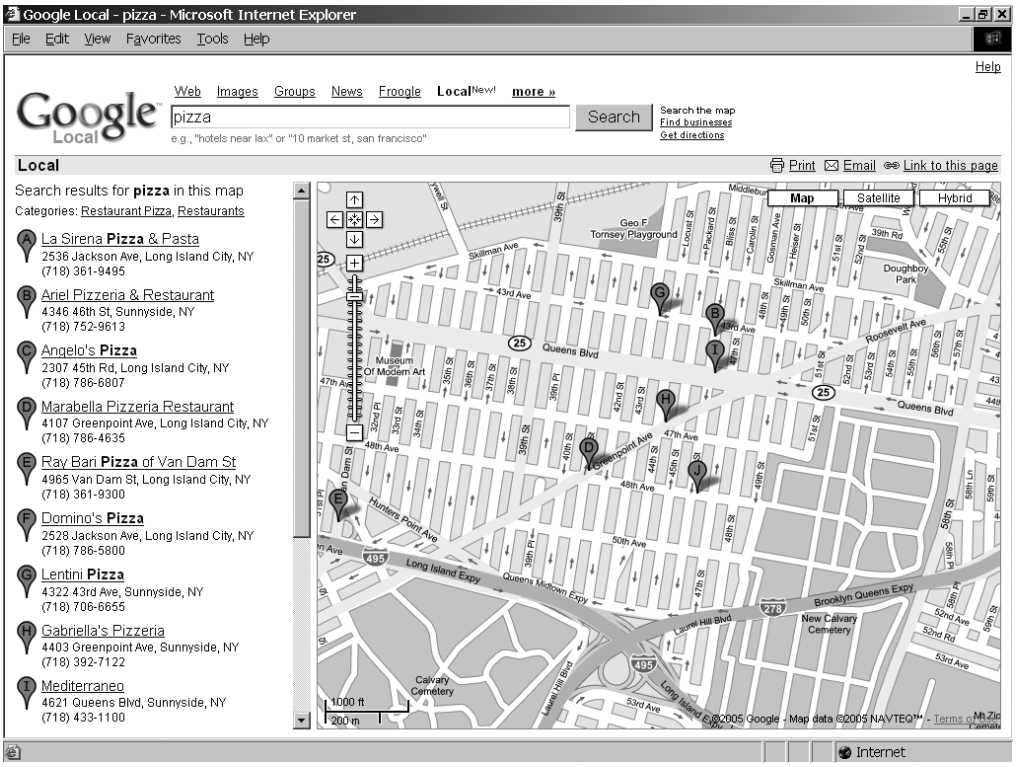


Figure 1-11. *Google Maps, a fantastic example of the power of Ajax*

WITHER WEB 2.0?

Web 2.0 is a term I'm not especially a fan of, partially because its meaning isn't really fixed precisely enough. For instance, if you take a cruise over to Wikipedia, you'll find this definition (assuming no one has changed it since I wrote this):

> *Web 2.0, a phrase coined by O'Reilly Media in 2003 and popularized by the first Web 2.0 conference in 2004, refers to a perceived second-generation of web-based communities and hosted services such as social networking sites, wikis, and folksonomies that facilitate collaboration and sharing between users. O'Reilly Media titled a series of conferences around the phrase, and it has since become widely adopted.*
>
> *Though the term suggests a new version of the Web, it does not refer to an update to World Wide Web technical specifications, but to changes in the ways systems developers have used the web platform. According to Tim O'Reilly, "Web 2.0 is the business revolution in the computer industry caused by the move to the Internet as platform, and an attempt to understand the rules for success on that new platform."*

> *Some technology experts, notably Tim Berners-Lee, have questioned whether one can use the term in a meaningful way, since many of the technology components of "Web 2.0" have existed since the beginnings of the World Wide Web.*
>
> Isn't it ironic that the man most directly responsible for the Internet in the first place, Tim Berners-Lee, basically says the same thing about the term? Sounds to me like someone just needed a term around which to base conferences, but that's more cynical a view than even **I** am comfortable with!
>
> The problem is that if you go somewhere else and look up the definition, you may well find something completely different. To some, for instance, Web 2.0 means the Web with all the Ajax goodness thrown in. To others, it means all sorts of transition effects, animations, and that sort of thing. To yet others, it means the ability of one system to connect with another via some form of web service. For some, it's a return to simplistic designs without tables using only CSS.
>
> In truth, all those things probably factor into the equation, but whichever meaning you like, Web 2.0 is a term that you'll hear a lot, and my suggestion is to try and glean its meaning not from any formal definition but rather from the context of the discussion you're involved in because it's likely the only context in which that particular definition of Web 2.0 will have any meaning!
>
> Oh yeah, and you should **definitely** put it on your resume somewhere. HR drones just **love** seeing it thrown around as if **you** are the only one with **THE** definition of it!

RIA stands for Rich Internet Application. Although there is no formal definition with which I am familiar, most people get the gist of its meaning without having to Google for it.

In short, the goal of an RIA is to create an application that is web based—that is, it runs in a web browser but looks, feels, and functions more like a typical fat-client application than a "typical" web site. Things like partial-page updates are taken for granted, and hence Ajax is always involved in RIAs (although what form of Ajax is involved can vary; indeed you may not find the `XMLHttpRequest` object, the prototypical Ajax solution, lurking about at all!). These types of applications are always more user-friendly and better received by the user community they service. In fact, your goal in building RIAs should be for users to say, "I didn't even know it was a webapp!"

Gmail (see Figure 1-12) is a good example of an RIA, although even it isn't perfect because while it has definite advantages over a typical web site, it still looks and feels very much like a web page, albeit one with lots of niceties mixed in. Microsoft's Hotmail is another good example (note that Hotmail is being supplanted by Microsoft Live Hotmail, which is the newer, more Web 2.0–ish version of Hotmail).

Yet another good example, shown in Figure 1-13, also comes courtesy of Google and is an online RSS feed reader.

You may have noticed that these examples of Ajax RIAs are from Google. That is not a coincidence. Google has done more to bring Ajax to the forefront of people's minds than anyone else. They were not the first to do it, or even the best necessarily, but they certainly have been some of the most visible examples and have really shown people what possibilities Ajax opens up.

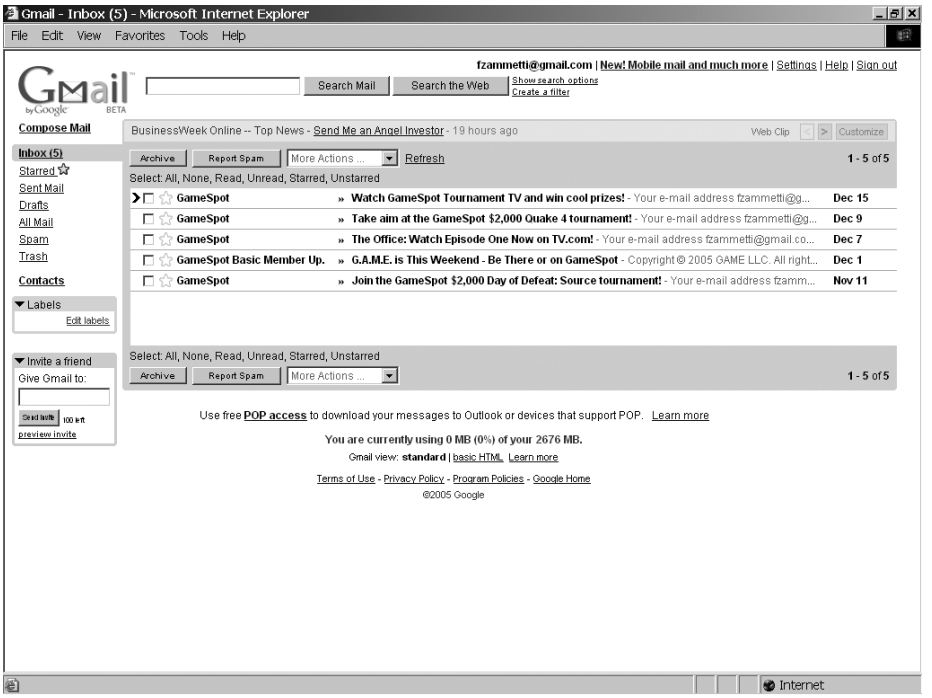


Figure 1-12. *Gmail, an Ajax webmail application from Google*

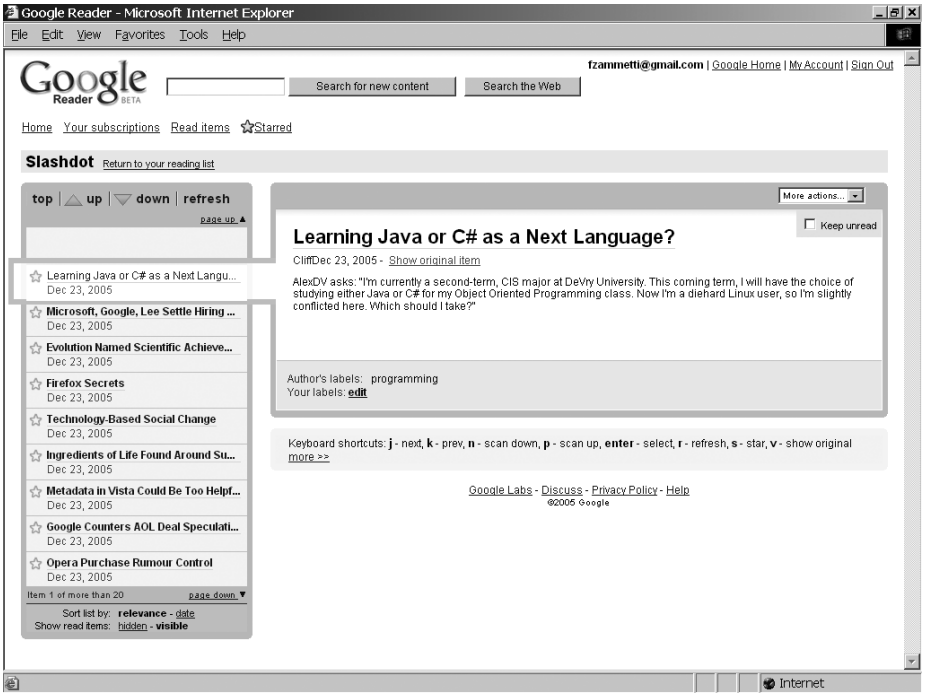


Figure 1-13. *Google Reader is a rather good Ajax-enabled RSS reader*

The Flip Side of the Coin

Ajax sounds pretty darned good so far, huh? It is not all roses in Ajax land, however, and Ajax is not without its problems. Some of them are arguably only perceived problems, but others are concrete.

First and foremost, in my mind at least, is accessibility. You will lose at least some accessibility in your work by using Ajax because devices like screen readers are designed to read an entire page, and since you will no longer be sending back entire pages, screen readers will have trouble. My understanding is that some screen readers can deal with Ajax to some degree, largely depending on how Ajax is used. If the content is literally inserted into the Document Object Model (the DOM), using DOM manipulation methods, vs. just inserting content into a `<div>` using `innerHTML`, for example, that makes a big difference with regard to whether a screen reader will be able to handle it or not and alert the user of the change; using DOM methods is a little bit more supported at this point, although it's still not perfect. In any case, extreme caution should be used if you know people with disabilities are a target audience for your application, and you will seriously want to consider (and test!) whether Ajax will work in your situation. I am certain this problem will be addressed better as time goes on, but for now it is definitely a concern. Even still, there are some things you can do to improve accessibility:

- Put a note at the top of the page that says the page will be updated dynamically. This will give users the knowledge that they may need to periodically request a reread of the page from the screen reader to hear the dynamic updates.
- Depending on the nature of the Ajax you are using on a page, use `alert()` pop-ups when possible as these are read by a screen reader. This is a reasonable enough suggestion for things like Ajax-based form submission that will not be happening too frequently, but obviously if you have a timed, repeating Ajax event, this suggestion would not be a good one.
- Accessibility generally boils down to two main concerns: helping the vision-impaired and helping those with motor dysfunctions. Those with hearing problems tend to have fewer issues with web applications, although with more multimedia-rich applications coming online each day, this may be increasingly less true. Those with motor disorders will be concerned with things like keyboard shortcuts, since they tend to be easier to work with than mouse movements (and are generally easier than mouse movements for specialized devices to implement).
- Often overlooked is another kind of vision impairment: color blindness. Web developers usually do a good job of helping the blind, but they typically don't give as much attention to those who are color-blind. It is important to understand that color-blind people do not usually see the world in only black and white. Color blindness, or rather color deficiencies, is a failing of one of the three pigments that work in conjunction with the cone cells in your eyes. Each of the three pigments, as well as the cones, is sensitive to one of the three wavelengths of light: red, green, or blue. Normal eyesight, and therefore normal functioning of these pigments and cone cells, allows people to see very subtle differences in shades of the colors that can be made by mixing red, green, and blue. Someone with color blindness cannot distinguish these subtle shading differences as well as someone with normal color vision can, and sometimes cannot distinguish such differences at all. To someone with color blindness, a field of blue dots

with subtle red ones mixed in will appear as a field of dots all the same color, just as one example. A page that demonstrates the effects of color deficiencies to someone with normal vision can be found at `http://colorvisiontesting.com/what%20colorblind%20people%20see.htm`.

- Remember that it is not only those who have disabilities that have accessibility concerns, it can be folks with no impairments at all. For instance, you should try to use visual cues whenever possible. For instance, briefly highlighting items that have changed can be a big help. Some people call this the *Yellow Fade Effect*, whereby you highlight the changed item in yellow and then slowly fade it back to the nonhighlighted state. Of course, it does not have to be yellow, and it does not have to fade, but the underlying concept is the same: highlight changed information to provide a visual cue that something has happened. Remember that changes caused by Ajax can sometimes be very subtle, so anything you can do to help people notice them will be appreciated.

The term "Yellow Fade Effect" seems to have originated with a company called 37signals, as seen in this article: `www.37signals.com/svn/archives/000558.php`.

Another disadvantage of Ajax, many people feel, is added complexity. Many shops do not have in-house the client-side coding expertise Ajax requires (the use of toolkits that make it easier notwithstanding). The fact is, errors that originate client side are still, by and large, harder to track down than server-side problems, and Ajax does not make this any simpler. For example, View Source does not reflect changes made to the DOM (there are some tools available for Firefox that actually do allow this). Another issue is that Ajax applications will many times do away with some time-honored web concepts, most specifically back and forward buttons and bookmarking. Since there are no longer entire pages, but instead fragments of pages being returned, the browser cannot bookmark things in many cases. Moreover, the back and forward buttons cease to have the same meanings because they still refer to the last URL that was requested, and Ajax requests almost never are included (requests made through the `XMLHttpRequest` are not added to history, for example, because the URL generally does not change, especially when the method used is `POST`).

All of these disadvantages, except for perhaps accessibility to a somewhat lesser extent, have solutions, and we will see some of them later in the example apps. They do, however, represent differences in how webapps are developed for most developers, and they cause angst for many people. So they are things you should absolutely be aware of as you move forward with your Ajax work.

I'd say that's enough of theory, definitions, history, and philosophy behind Ajax and RIAs and all that. Let's go get our hands dirty with some actual code!

Let's Get to the Good Stuff: Our First Ajax Code, the Manual Way

This book aims to be different from most Ajax books in that it is based around the concept of giving you concrete examples to learn from, explaining them, explaining the decisions behind them (even the debatable ones), and letting you get your hands dirty with code. We are not going to spend a whole lot of time looking at UML diagrams, sequence diagrams, use case diagrams, and the like (although there will be some because, hey, we are code monkeys after all). You are more than welcome to pick up any of the fine UML books out there for that.

CODE MONKEY? DID YOU JUST CALL ME A CODE MONKEY?!?

Some people consider the term *code monkey* to be derogatory, but I never have. To me, a code monkey is someone who programs for a living and who enjoys writing code. I suppose by that definition you wouldn't even have to earn a living from programming, so long as you love hacking bits. Either way, it's all about the code!

By the way, just because someone is a code monkey doesn't mean they can't do architecture, and vice versa. If you like all of the facets of building software, if you like being up 'til all hours of the night trying to figure out why your custom-built double-linked list is corrupting elements when you modify them, if you like playing with that new open source library for no other reason than you're curious, if you like the feeling you get from seeing a working application come spewing out the back end of a development cycle (even if it's some otherwise dull business application), then you're a code monkey, plain and simple, and you should never take offense to being called that name.

Of course, some people **do** in fact mean it in a derogatory way, and you'll know who they are, in which case you should be a good little code monkey and throw feces at them (Frank Zammetti and Apress cannot be held liable if you actually follow this advice!).

Oh yes, and no discussion of the term code monkey would be complete with referencing the fantastic parody song "Code Monkey" by one Jonathan Coulton. His web site is here: `www.jonathancoulton.com`. Sadly, at the time I wrote this, it appeared to be having problems. Hopefully they are just temporary, but in any case, if you like Weird Al–style funny songs and want to hear a good one about us code monkeys, of which I am proudly one, try that site, and failing that, spend a few minutes with Google trying to find it (I don't even think the RIAA, which is the Recording Industry Association of America, the American organization famous for suing grandmothers, children, and even dead people for illegally downloading pirated music, will have a problem with it, but you never can tell with those people, so I'm **not** suggesting any file-sharing networks you might try!).

With that in mind, I am not going to waste any more time telling you what Ajax is, why it is the greatest thing since sliced bread, where the name came from, or any of that. Instead, we are going to jump right into some code!

This first example is somewhat unique in that it doesn't require Java. In fact, it does not require a server at all. Rest assured that all the other examples in this book do, just not this one. But we want to cover a simple Ajax app without server interaction first, just to get some of the basics covered, so here goes (see Listing 1-1).

Listing 1-1. *Our First Real Ajax Application! (This Is the File* index.htm*.)*

```
<html>

  <head>

    <title>Simple Non-Server AJAX Example</title>

    <script>

      // This is a reference to an XMLHttpRequest object.
      xhr = null;

      // This function is called any time a selection is made in the first
      // <select> element.
      function updateCharacters() {

        var selectedShow = document.getElementById("selShow").value;
        if (selectedShow == "") {
          document.getElementById("divCharacters").innerHTML = "";
          return;
        }

        // Instantiate an XMLHttpRequest object.
        if (window.XMLHttpRequest) {
          // Non-IE.
          xhr = new XMLHttpRequest();
        } else {
          // IE.
          xhr = new ActiveXObject("Microsoft.XMLHTTP");
        }
        xhr.onreadystatechange = callbackHandler;
        url = selectedShow + ".htm";
        xhr.open("post", url, true);
        xhr.send(null);

      }

      // This is the function that will repeatedly be called by our
      // XMLHttpRequest object during the life cycle of the request.
      function callbackHandler() {

        if (xhr.readyState == 4) {
          document.getElementById("divCharacters").innerHTML = xhr.responseText;
        }

      }
```

```
    </script>

  </head>

  <body>

    Our first simple AJAX example
    <br><br>

    Make a selection here:
    <br>
    <select onChange="updateCharacters();" id="selShow">
      <option value=""></option>
      <option value="b5">Babylon 5</option>
      <option value="bsg">Battlestar Galactica</option>
      <option value="sg1">Stargate SG-1</option>
      <option value="sttng">Star Trek The Next Generation</option>
    </select>

    <br><br>

    In response, a list of characters will appear here:
    <br>
    <div id="divCharacters">
      <select>
      </select>
    </div>

  </body>

</html>
```

Figure 1-14 shows what it looks like on the screen (don't expect much here, folks!).

As you can see, there is no content in the second drop-down menu initially. This will be dynamically populated once a selection is made in the first, as shown in Figure 1-15.

Figure 1-15 shows that when a selection is made in the first drop-down menu, the contents of the second are dynamically updated. In this case we see characters from the greatest television show ever, *Babylon 5* (don't bother arguing, you know I'm right! And besides, you'll get your chance to put in your favorites later!). Now let's see how this "magic" is accomplished.

Listing 1-1 shows the first page of our simple Ajax example, which performs a fairly typical Ajax-type function: populate one `<select>` box based on the selection made in another. This comes up all the time in web development, and the "classic" way of doing it is to submit a form, whether by virtue of a button the user has to click or by a JavaScript event handler, to the server and let it render the page anew with the updated contents for the second `<select>`. With Ajax, none of that is necessary.

Figure 1-14. *Note that there is no content in the second drop-down menu because nothing has yet been selected in the first.*

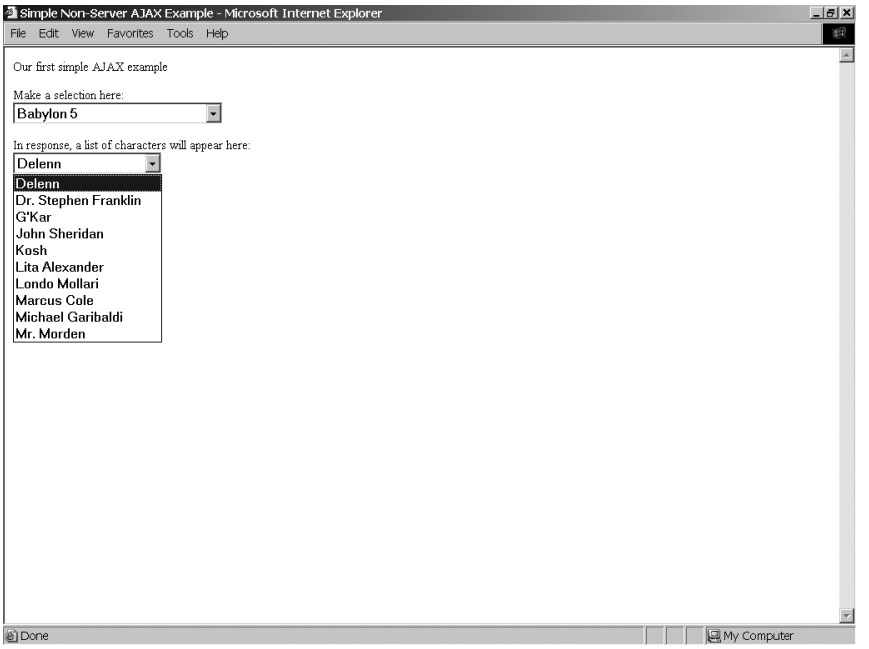


Figure 1-15. *A selection has been made in the first drop-down menu, and the contents of the second have been dynamically created from what was returned by the "server."*

A Quick Postmortem

Let's walk through the code and see what is going on. Note that this is not meant to be a robust, production-quality piece of code. It is meant to give you an understanding of basic Ajax techniques, nothing more. There is no need to write in with all the flaws you find!

First things first: the markup itself. In our `<body>` we have little more than some text and two `<select>` elements. Notice that they are not part of a `<form>`. You will find that forms tend to have less meaning in the world of Ajax. You will many times begin to treat all your form UI elements as top-level objects along with all the other elements on your page (in the `<body>` anyway).

Let's look at the first `<select>` element. This `<select>` element is given the ID `selShow`. This becomes a node in the DOM of the page. DOM, if you are unfamiliar with the term, is nothing more than a tree structure where each of the elements on your page can be found. In this case, we have a branch on our tree that is our `<select>` element, and we are naming it `selShow` so we can easily get at it later.

```
<select onChange="updateCharacters();" id="selShow">
  <option value=""></option>
  <option value="b5">Babylon 5</option>
  <option value="bsg">Battlestar Galactica</option>
  <option value="sg1">Stargate SG-1</option>
  <option value="sttng">Star Trek The Next Generation</option>
</select>
```

You will notice the JavaScript event handler attached to this element. Any time the value of the `<select>` changes, we will be calling the JavaScript function named `updateCharacters()`. This is where all the "magic" will happen. The rest of the element is nothing unusual. I have simply created an `<option>` for some of my favorite shows. After that we find another `<select>` element . . . sort of:

```
<div id="divCharacters">
  <select>
  </select>
</div>
```

It is indeed an empty `<select>` element, but wrapped in a `<div>`. You will find that probably the most commonly performed Ajax function is to replace the contents of some `<div>`. That is exactly what we will be doing here. In this case, what will be returned by the "server" (more on that in a minute) is the markup for our `<select>` element, complete with `<option>`'s listing characters from the selected show. So, when you make a show selection, the list of characters will be appropriately populated, and in true Ajax form, the whole page will not be redrawn, but only the portion that has changed—the second `<select>` element in this case (or more precisely, the `<div>` that wraps it) will be.

Let's quickly look at our mock server. Each of the shows in the first `<select>` has its own HTML file that in essence represents a server process. You have to take a leap of faith here and pretend a server was rendering the response that is those HTML pages. They all look virtually the same, so I will only show one as an example (see Listing 1-2).

Listing 1-2. *Sample Response Listing Characters from the Greatest Show Ever,* Babylon 5*!*

```
<select>
  <option>Delenn</option>
  <option>Dr. Stephen Franklin</option>
  <option>G'Kar</option>
  <option>John Sheridan</option>
  <option>Kosh</option>
  <option>Lita Alexander</option>
  <option>Londo Mollari</option>
  <option>Marcus Cole</option>
  <option>Michael Garibaldi</option>
  <option>Mr. Morden</option>
</select>
```

As expected, it really is nothing but the markup for our second `<select>` element.

Hey, I Thought This Was Ajax?!?

So, now we come to the part that does all the work here, our JavaScript function(s). First is the `updateCharacters()` function, shown here:

```
// This function is called any time a selection is made in the first
// <select> element.
function updateCharacters() {

        var selectedShow = document.getElementById("selShow").value;
        if (selectedShow == "") {
          document.getElementById("divCharacters").innerHTML = "";
          return;
        }

  // Instantiate an XMLHttpRequest object.
  if (window.XMLHttpRequest) {
    // Non-IE.
    xhr = new XMLHttpRequest();
  } else {
    // IE.
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
  }
  xhr.onreadystatechange = callbackHandler;
  url = selectedShow + ".htm";
  xhr.open("post", url, true);
  xhr.send(null);

}
```

If you were to do manual Ajax on a regular basis, this basic code would very soon be imprinted on the insides of your eyelids (fortunately for you, this book is about DWR, so you

won't have to deal with these details manually, but it's still good to get an idea what's going on under the covers, and that's exactly what this is). This is the basic prototypical Ajax function, and you'd find similar code in an Ajax application, or in any library providing Ajax functionality. Let's tear it apart, shall we?

First, the function deals with the case where the user selects the blank option from the drop-down menu. In that case, we want to remove any second drop-down that may be present, so it's a simple matter of writing a blank string to the `innerHTML` property of the target `<div>`.

The first thing we need to add to our Ajax call, as one would expect, is an `XMLHttpRequest` object. This object, a creation of Microsoft (believe it or not!), is nothing more than a proxy to a socket. It has a few (very few) methods and properties, but that is one of the benefits: it really is a very simple beast.

Notice the branching logic here. It turns out that getting an instance of the `XMLHttpRequest` object is different in Internet Explorer than in any other browser (although it's worth noting that in IE7, an `XMLHttpRequest` object **is** now available, which means the same instantiation code can be used across browsers! However, to support the widest possible audience, if you have to code manual Ajax, you will want to stick with this type of branched code, which still works in IE7 as well). Now, before you get your knickers in a knot and get your anti-Microsoft ire up, note that Microsoft invented this object, and it was the rest of the world that followed. So, while it would be nice if Microsoft updated its API to match everyone else's, it isn't Microsoft's fault we need this branching logic! The others could just as easily have duplicated what Microsoft did exactly too, so let's not throw stones here—we're all in glass houses on this one! (Well, that's not **technically** true because Microsoft's implementation uses ActiveX, which means only Windows-based browsers could really implement it the same way. On the other hand, we could envision a very simple emulation of ActiveX to the extent that the browser could recognize this particular instantiation method and spawn the native object, as it would do normally in the absence of such emulation, so it still really does hold true.)

This is probably a good time to point out that `XMLHttpRequest` is pretty much a de facto standard at this point. It is also being made a true W3C standard, but for now it is not. It is safe to assume that any "modern" browser—that is, a desktop web browser that is no more than a few versions old—will have this object available. More limited devices, such as PocketPCs, cell phones, and the like, will many times not have it, but by and large it is a pretty ubiquitous little piece of code.

Continuing on in our code review . . . once we have an `XMLHttpRequest` object instance, we assign the reference to it to the variable `xhr` in the global page scope. Think about this for just a minute; what happens if more than one `onChange` event fires at close to the same time? Essentially, the first will be lost because a new `XMLHttpRequest` object is spawned, and `xhr` will point to it. Worse still, because of the asynchronous nature of `XMLHttpRequest`, a situation can arise where the callback function for the first request is executing when the reference is `null`, which means that callback would throw errors due to trying to reference a `null` object. If that was not bad enough, this will be the case only in some browsers, but not all (although my research indicates most would throw errors), so it might not even be a consistent problem.

Remember, I said this was not robust, production-quality code! This is a good example of why. That being said, it is actually many times perfectly acceptable to simply instantiate a new instance and start a new request. Think about a fat client that you use frequently. Can you spot instances where you can kick off an event that in essence cancels a previous event that was in the process of executing? For example, in your web browser, can you click the Home button

while a page is loading, thereby causing the page load to be prematurely ended and the new page to begin loading? Yes you can, and that is what in essence happens by starting a new Ajax request using the same reference variable. It is not an unusual way for an application to work, and sometimes is downright desirable. That being said, though, you do need to keep it in mind and be sure it is how you want and need things to work.

The next step we need to accomplish is telling the `XMLHttpRequest` instance what callback handler function to use. An Ajax request has a well-defined and specific life cycle, just like any HTTP request (and keep in mind, that's all an Ajax request is at the end of the day: a plain ole HTTP request!). This cycle is defined as the transitions between ready states (hence the property name, `onreadystatechange`). At specific intervals in this life cycle, the JavaScript function you name as the callback handler will be called. For instance, when the request begins, your function will be called. As the request is chunked back to the browser, in most browsers at least (IE being the unfortunate exception), you will get a call for each chunk returned (think about those cool status bars you can finally do with no complex queuing and callback code on the server!). Most important for us in this case, the function will be called when the request completes. We will see this function in just a moment.

The next step is probably pretty obvious: we have to tell the object what URL we want to call. We do this by calling the `open()` method of the object. This method takes three parameters: the HTTP method to perform, the URL to contact, and whether we want the call to be performed asynchronously (true) or not (false). Because this is a simple example, each television show gets its own HTML file pretending to be the server. The name of the HTML file is simply the value from the `<select>` element with `.htm` appended to the end. So, for each selection the user makes, a different URL is called. This is obviously not how a real solution would work—the real thing would likely call the same URL with some sort of parameter to specify the selected show—but some sacrifices were necessary to keep the example both simple and not needing anything on the server side of things.

The HTTP method can be any of the standard HTTP methods: `GET`, `POST`, `HEAD`, etc. Ninety-eight percent of the time you will likely be passing `GET` or `POST`. The URL is self-explanatory, except for one detail: if you are doing a `GET`, you must construct the query string yourself and append it to the URL. That is one of the drawbacks of `XMLHttpRequest`: you take full responsibility for marshaling and unmarshaling data sent and received. Remember, it is in essence just a very thin wrapper around a socket. This is where any of the numerous Ajax toolkits can come in quite handy, but we will talk about that later.

Once we have the callback registered with the object and we have told it what we're going to connect to and how, we simply call the `send()` method. In this case, we are not actually sending anything, so we pass `null`. One thing to be aware of is that, at least when I tested it, you can call `send()` with no arguments in IE and it will work, but in Firefox it will not. `null` works in both, though, so `null` it is.

Of course, if you actually had some content to send, you would do so here. You can pass a string of data into this method, and the data will be sent in the body of the HTTP request. Many times you will want to send actual parameters, and you do so by constructing essentially a query string in the typical form `var1=val1&var1=val1` and so forth, but without the leading question mark. Alternatively, you can pass in an XML DOM object, and it will be serialized to a string and sent. Lastly, you could send any arbitrary data you want. If a comma-separated list does the trick, you can send that. Anything other than a parameter string will require you to deal with it; the parameter string will result in request parameters as expected.

So far we've described how a request is sent. It is pretty trivial, right? Well, the next part is what can be even more trivial, or much more complex. In our example, it is the former. I am referring to the callback handler function, which is as follows:

```
// This is the function that will repeatedly be called by our
// XMLHttpRequest object during the life cycle of the request.
function callbackHandler() {
  if (xhr.readyState == 4) {
    document.getElementById("divCharacters").innerHTML = xhr.responseText;
  }
}
```

Our callback handler function this time around does very little. First, it checks the `readystate` of the `XMLHttpRequest` object. Remember I said this callback will be called multiple times during the life cycle of the request? Well, the `readystate` code you will see will vary with each life cycle event. The full list of codes (and there is only a handful) can be found with about five seconds of not-too-strenuous effort with Google, but for the purposes of this example we are only interested in code 4, which indicates the request has completed. Notice that I didn't say completed *successfully*! Regardless of the response from the server, the `readystate` will be 4. Since this is a simple example, we don't care what the server returns. If an HTTP 404 error (page not found) is received, we don't care in this case. If an HTTP 500 error (server processing error) is received, we still do not care. The function will do its thing in any of these cases. I repeat my refrain: this is not an industrial-strength example!

When the callback is called as a result of the request completing, we simply set the `innerHTML` property of the `<div>` on the page with the ID `divCharacters` to the text that was returned. In this case, the text returned is the markup for the populated `<select>`, and the end result is the second `<select>` is populated by characters from the selected show.

Now, that wasn't so bad, was it?

For a fun little exercise, and just to convince yourself of what is really going on, I suggest adding one or two of your own favorite shows in the first `<select>`, and creating the appropriately named HTML file to render the markup for the second `<select>`.

Cutting IN the Middle Man: Ajax Libraries to Ease Our Pain

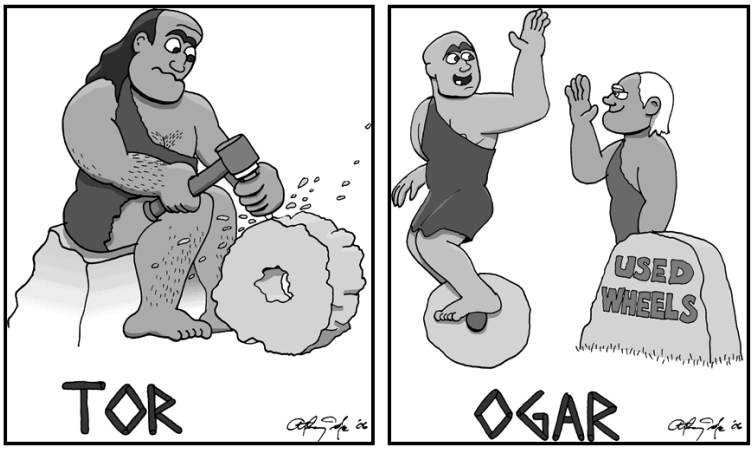
At this point you are probably thinking, "OK, Ajax is no big deal," and you would be right. At least in its more basic form, there is not much code to it, and it is pretty easy to follow.

However, if this is your first exposure to JavaScript and client-side development in general, it may seem like a lot of work. Or, if you intend to be doing more complex Ajax work, you may not want to be hand-coding handlers and functions all over the place. It is in these cases where a good Ajax library can come into play.

As you might imagine with all the hype surrounding Ajax recently, there are a ton of libraries and toolkits to choose from. Some are fairly blunt, general-purpose tools that make doing Ajax only a bit simpler, whereas others are rather robust libraries that seek to fill all your JavaScript and Ajax needs. Some provide powerful and fancy GUI widgets that use Ajax behind the scenes; others leave the widget building to you, but make it a lot easier.

Ajax libraries, JavaScript libraries in general really, have truly grown in leaps and bounds over the past two to three years. It used to be that you could spend a few hours scouring the web looking for a particular piece of code, and you indeed eventually found it. Often though, you might have to, ahem, appropriate it from some web site (this still happens of course). Many times, you could, and can, find what you need on one of a handful of *script sites* that are there expressly to supply developers with JavaScript snippets for their own use.

However, larger libraries that provide all sorts of bells and whistles, as exist in the big-brother world of Java, C++, PHP, and other languages, are a more recent development in the world of JavaScript. In many ways, we are now in a golden age in this regard, and you will find almost more options than you might want! So **use those libraries**, lest you wind up like Tor!



One thing that most of these modern-day libraries, at least the more well-known ones, have in common is that their quality is light-years beyond what used to be available. More importantly to you, each and every one of them will likely make your life (in terms of writing code anyway) considerably easier. There's usually no sense in reinventing the wheel after all. If you are doing Ajax, unless you need absolute control over every detail of things, I can't think of a good reason not to use a library for it. If you know your UI design requires some more advanced widgets that the browser doesn't natively provide, these libraries can absolutely be worth their weight in gold.

At the end of the day, though, keep in mind that "naked" Ajax, as seen in the example from this chapter, is also extremely common and has many of its own benefits, some of which are control over what is going on at the most fundamental level and more opportunity to tune for performance and overall robustness. It is also never a bad idea to know exactly what is going on in your own applications! If you are comfortable with JavaScript and client-side development in general, you may never need or want to touch any of these toolkits (I personally lean that way), but if you are just coming to the client-side party, these libraries can indeed make your life a lot easier.

JUST TO NAME A FEW . . .

There truly is a wealth of Ajax libraries out there, and of JavaScript libraries that have Ajax as one component. I saw a recent survey that listed something on the order of 236 choices at the moment, and that doesn't even count the ones that are specific to Java or .NET or PHP or what have you! As a service to you, dear reader, I'd like to list just a handful that I personally consider the cream of the crop. This isn't to shortchange any others because it's frankly likely I simply am not aware of some other great options, but these are ones I've personally used and think rather highly of (it's also true that most of these tend to top most peoples' lists).

- *Dojo* (`http://dojotoolkit.org`)

  Dojo is very much a jack-of-all-trades, covering a very broad range of topics, Ajax being just one of them. What Dojo is probably most famous for is its widget system and its widgets. It provides some very snazzy UI goodness, things like fisheye lists (think the dock bar on Macintosh systems) and more. Dojo is still a developing library, so expect some growing pains as you use it, but it gets a lot of attention and you'll quickly see why if you take a look.

- *MooTools* (`http://mootools.net`)

  MooTools is another of those "cover all the bases" libraries. One of its most interesting features is actually its web site! It allows you to build a custom version of it containing just the component you want. It deals with any dependencies that can arise so your build will always work. Aside from that, it has some great components including Ajax, of course, as well as effects and transitions and basic language enhancements.

- *Prototype* (`www.prototypejs.org`)

  Prototype serves as the foundation for many other famous libraries, and for good reason. It is small, relatively simple, and extremely useful. You can view it as an extension to JavaScript itself for the most part. You won't find some of the more fancy features of other libraries like widgets and effects, but it will enhance the language for you in ways you'll be very thankful for.

- *YUI* (`http://developer.yahoo.com/yui`)

  The Yahoo! User Interface library, while by and large about user interface widgets, also provides a good supply of utilities outside those widgets. YUI is one of the simpler and clearly written (and documented!) libraries out there. It won't wow people as much as something like Dojo will, but it makes up for it in spades in being very easy to use and very well supported.

- *script.aculo.us* (`http://script.aculo.us`)

  script.aculo.us, aside from being bizarrely named, is, to quote its tag line, "about the interface, baby!" Indeed it is. If you need visual effects, transitions, and so forth, this is the one library you'll want to check out. To see it in action, you can visit one of the more popular sites on the web: Digg (`www.digg.com`). Digg uses script.aculo.us to do all sorts of nifty little things along the way, like fading elements in and out when you post.

Alternatives to Ajax

No discussion of Ajax would be complete without pointing out that it is not the only game in town. There are other ways to accomplish the goals of building RIAs than using Ajax.

Have you ever heard of Flash? Of course you have! Who hasn't been annoyed by all those animations all over some of the most popular web sites around? Flash can be very annoying at times, but when used properly, it can provide what Ajax provides.

One good example is Mappr (`www.mappr.com`), which is a Flash-based application for exploring places based on pictures people take and submit for them (see Figure 1-16). This is an example of how Flash, used properly, can truly enhance the web users' experience.

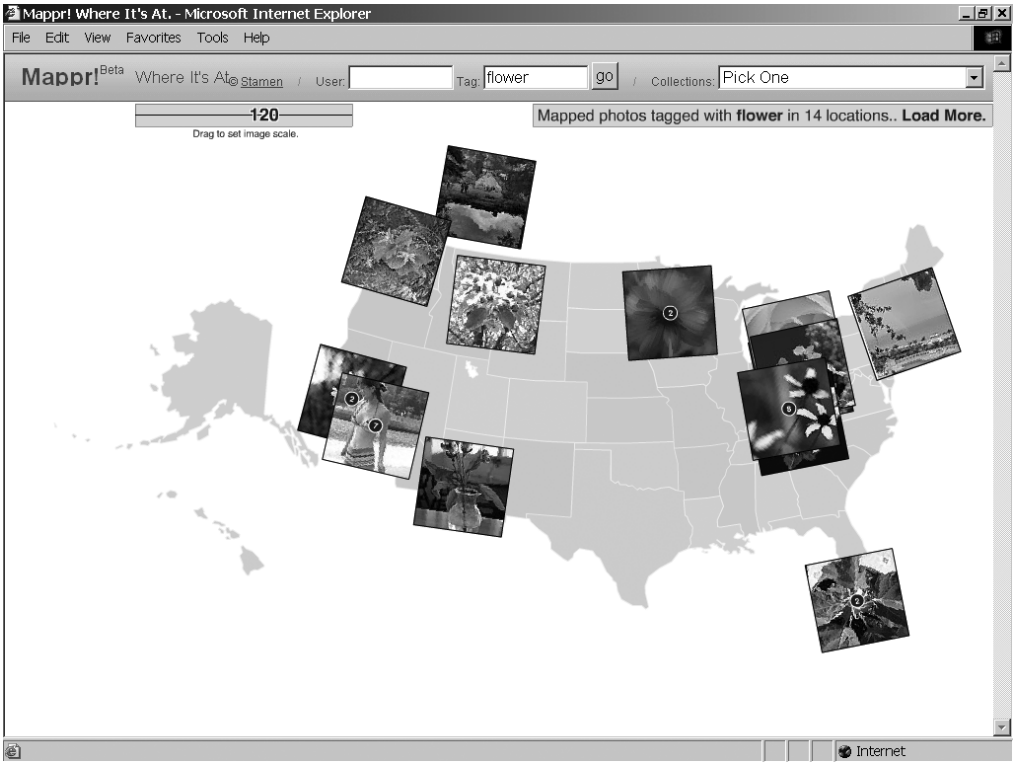


Figure 1-16. *Mappr, an excellent use of Flash*

Flash is a more "proper" development environment as well, complete with its own highly evolved integrated development environment (IDE), debugging facilities, and all that good stuff. It is also a tool that graphic designers can readily use to create some truly awe-inspiring multimedia presentations.

Flash is also fairly ubiquitous at this point. Virtually all browsers come with a Flash player bundled, or one can be added on with virtually no difficulty. Flash does, however, require a browser plug-in, whereas Ajax doesn't, and that is one advantage Ajax has. On the other hand, there are versions of Flash available for many mobile devices like PocketPCs and some cell phones, even those where `XMLHttpRequest` may not yet be available, so that's a plus for Flash.

Something of an extension to Flash is the relatively new product called Flex from Adobe. Flex is basically a markup-based UI toolkit built on top of Flash. What this means is you can code an application much like you do an HTML page, using special tags that represent UI elements such as tabbed dialogs, buttons, combo boxes, data grids and more, as well as script-based code written in Adobe's ActionScript language (which is fairly similar to JavaScript in most regards, certainly syntactically for the most part). The neat thing is that you compile a Flex page (which is a file with an extension of `.mxml`) into a Flash SWF file, which then runs in a Flash browser plug-in (or stand-alone Flash player), typically launched from an HTML page (just like any Flash movie embedded in a web page, but in this case the web page is literally nothing but a container and launcher for the Flash file). What this provides is a very rich UI experience, as well as providing more system-specific services such as the ability to write to the local file system, the ability to make HTTP requests to servers, the ability to make web service–like requests to a suitable server, and much more. Flex is starting to gain a lot of momentum among developers because it gives you the ability to present an extremely rich user interface in your applications while having most of the benefits of HTML and JavaScript-based applications, including cross-platform support and no deployment woes, not to mention saving you having to deal with the sometimes sticky details of Ajax, concurrent user events, etc.

In Figure 1-17, you can see an example of a Flex application. This is one of the sample applications that comes packaged with the Flex SDK, which is a free download. That's right; you can get started with Flex today if you want, at absolutely no cost! The SDK is free, as is of course the Flash runtime. There is, however, a commercial Flex IDE that Adobe sells as well, which you may want to look at if you decide to work with Flex on a regular basis. As you might imagine, this example has a much greater "wow" factor if you actually play with it, so I very much recommend having a look if you think Flex might be something you are interested in. It showcases quite a lot of, well, **Flash**, that is, various animations and effects, as you navigate the application.

All in all, Flex is, in my opinion (and in the growing opinion of many) a very viable alternative to Ajax-based applications, and may even represent the single best alternative available today.

What are the negatives about Flash, and by extension Flex? First, it has a bit of a bad reputation because it is so easy to abuse. Many people choose to not install it or block all Flash content from their browser. Second, in the case of Flash, it is not completely "typical" programming; it is a little more like creating a presentation in the sense that you have to deal with the concept of a "timeline," so there can be a bit of a learning curve for "traditional" developers. This is not true of Flex, which is very much like creating typical web applications. However, keep in mind that developers will still have a learning curve as they will need to get up to speed on the UI markup language, as well as ActionScript. Third, it can be somewhat heavyweight for users not on broadband connections, and even for those *with* broadband connections it can sometimes take longer than is ideal to start up. Lastly, Flash and Flex are owned and developed by Adobe, and the Flash runtime is not open sourced, although Flex recently was open sourced. In my opinion, it should still be viewed as something where vendor lock-in could become an issue, although it's certainly not any worse than many other technologies we all use, and better than some.

All in all, as alternatives go, Flash, and Flex more specifically, are good things to consider as you decide how to develop your RIAs. There are certainly pluses and minuses down each path.

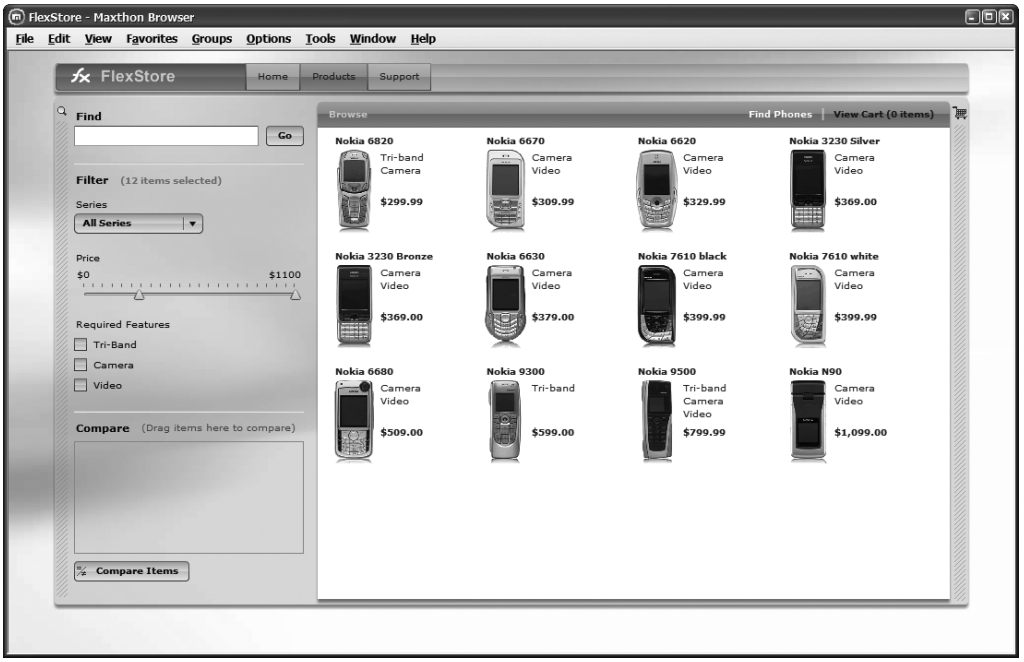


Figure 1-17. *The Flex store example application*

While not, strictly speaking, an "alternative" to Ajax, I feel that Google Web Toolkit, or GWT, should be discussed in this section as well. GWT is an interesting product because it allows you to build web applications in pure Java. That's right, no writing HTML or JavaScript, no dealing with various browser quirks and cross-browser concerns. Just write your code very much like you would a GUI application in Java and let GWT handle the rest. Just to prove it's not just a neat subject for a thesis paper, note that Google Maps and Gmail are both built using GWT.

Now, some among you, and frankly me included, may feel that this is a little too unique. Speaking for myself, I've never been a huge fan of the way GUIs are written in Java, and to write a webapp like that doesn't really fill me with warm fuzzies. Many people, however, feel very differently, and they have good reasons for feeling that way. Most importantly, simply not having to concern yourself with all the quirks of writing browser-based applications, of which there are certainly plenty, is a good one.

In short, if this concept of writing Java code to create a webapp has appeal for you, then check out GWT: `http://code.google.com/webtoolkit`. You most definitely will not be alone in your choice; many people sing the praises of GWT as loudly as they can, and there's certainly no arguing the success of many of the products Google has created with it.

Of course, since you're reading this book, you probably already have an idea which way you intend to go, so we should get to the real topic at hand, and that of course is DWR itself.

Hmm, Are We Forgetting Something? What Could It Be? Oh Yeah, DWR!

Since we're here to talk about DWR, we might as well not delay any further. Consider this a teaser of what is coming in the next chapter.

DWR (`http://getahead.ltd.uk/dwr`), which stands for Direct Web Remoting, is a free, open source product from a small IT consulting group called Getahead, whose members are Joe Walker and Mark Goodwin. DWR is an interesting approach to Ajax in that it allows you to treat Java classes running on the server as if they were local, that is, running within the web browser.

DWR is a Java-centric solution, unlike many of the popular Ajax libraries out there today. Some view this as a limitation, but I for one view it as a plus because it allows for capabilities that probably couldn't be provided in a cross-language way (not without being tremendously more difficult to use and most certainly to maintain for the developers).

If you are familiar with the concept of RPC, or Remote Procedure Call, then you will recognize quickly that DWR is, essentially, a form of RPC (and it had better be, it says so right there in the name: Direct Web **Remoting**!).

MORE ABOUT RPC

RPC is a mechanism by which code executing in one address space, typically one physical computing device, executes a subroutine or procedure on another or a shared network. This is accomplished without the programmer of the calling code explicitly coding for the details of the remote interaction. Stated another way, the programmer writes in essence the same code whether the function being called is on the local system or the remote system. Remote invocation or remote method invocation means the same thing as RPC, but is typically used when discussing object-oriented code.

The basic concepts behind RPC go back to at least 1976, as described in RFC 707. Xerox was one of the first businesses to use RPC, but they called it "Courier." This was in 1981. Sun, shortly thereafter, created what is now called ONC RPC, a form of RPC specifically for UNIX, which formed the basis of Sun's NFS. ONC RPC is still used widely on several platforms today.

Today, there are many analogies to RPC; most well known are probably Java's RMI and Microsoft .NET's Remoting, and there is also a web services implementation of RPC, just to name a few. All of these still share the same basic concepts developed back in 1976, and DWR follows the basic model as well.

In the DWR model, the calling "system" is JavaScript executing in the users' browser, and the system being called is the Java classes running on the server. Just like other forms of RPC, the code you write in JavaScript looks very similar to the code you'd write on the server to execute the same code. There are, of course, some differences as you'd expect given the inherent differences between JavaScript and Java, but by and large it's extremely similar.

To give you a brief preview, consider the following simple class definition:

```
package com.company.app;
public class MyClass {
  String sayHello(String name) {
    return "Hello, " + name;
  }
}
```

If you wanted to call this code via DWR from JavaScript running in the browser, assuming you'd already accomplished all the necessary setup, the following code is all it would take:

```
MyClass.sayHello("Frank", sayHelloHandler);
var sayHelloHandler = function(data) {
  alert(data);
}
```

As I mentioned, and as you can see, the call syntax is very much like you'd do from Java, with the addition of the second parameter, which as it turns out is a callback function that will be executed when the method invocation completes.

In the next chapter, we'll be getting into all the nitty-gritty details of how this works, why the callback is needed, how to configure DWR, etc. For now though, this is a good preview of what DWR is all about because, believe it or not, it doesn't get a whole lot more complex than that!

Summary

If this chapter has seemed like an attempt to brainwash you, that is because, in a sense, it was! Ajax can seem to some people like a really bad idea, but those people tend to only see the problems and completely ignore the benefits. Because of my belief that Ajax is more about philosophy and thought process than it is about specific technologies, it is important to sell you on the ideas underlying it. It is not enough to simply show you some code and hope you agree!

In this chapter, we have looked at the evolution of web development over the past decade and a half or so. We have discussed Ajax and what it means, and seen how it is accomplished. We have discovered the term RIA and talked about why all of this really is a significant development for many people.

We have also discussed how the most important thing about Ajax is not the technologies in use but the mindset and approach to application development that are its underpinnings.

We have seen and played with our first Ajax-based application and have discovered that the code involved is ultimately pretty trivial. We have discussed some alternatives to Ajax and have learned that Ajax has been done for years, but with different techniques and technologies.

We have also discussed how Ajax is more user-driven than developer-driven, which is a significant difference from many advances in the past (and also one good reason why Ajax isn't as likely to go away as easily).

We took a quick look at some alternatives to Ajax, and also some of the libraries out there besides DWR for doing Ajax.

Last but not least, we got our first quick glimpse of DWR itself, setting the stage for the next chapters in which we'll dive into it head first.

I will close with just a few take-away points that succinctly summarize this chapter:

- Ajax is more about a mindset, an approach to developing webapps, than it is about any specific technologies or programming techniques.
- Ajax does in fact represent a fundamental shift in how webapps are built, at least for many people.
- Ajax, at its core, is a pretty simple thing. What we build on top of it may get complex, but the basics are not a big deal.
- RIAs may or may not represent the future of application development in general, but almost certainly do in terms of web development.
- There are a number of alternatives to Ajax that you should not dismiss out-of-hand. Consider the problem you are trying to solve and choose the best solution. Ajax will be it in a great many (and a growing number) of cases, but may not always be. As you have bought this book, however, we'll assume from here on out you want to do Ajax!