1

Introducing Ajax

The path of history is littered with splits, branches, and what-if's. The pace of development of technology is relentless and often merciless. Past battles have seen VHS triumph over Betamax, PCs over microcomputers, Internet Explorer (IE) over Netscape Navigator, and plenty more similar conflicts are just waiting to happen in DVD formats. It doesn't mean that one technology was necessarily better than the other; it's just that one format or technology had the features and functionality required at that time to make it more popular. You'll still find enthusiasts now waxing lyrical about the benefits of Betamax tape, claiming that it was smaller, had better quality and such. It doesn't mean they were wrong. Perhaps they were being a little sad and obsessive, but beneath it all, they had a point.

The evolution of the Internet has had its own such forks. One that continues to rumble is the so-called "fat-client" versus "thin-client" debate. Briefly put, this is the choice between getting your browser to do most of the work, as opposed to getting a server at the other end to do the processing. Initially, in the mid-1990s, it looked as if the "fat-client" ideology was going to win out. The introduction of IE 4 and Netscape Navigator 4 brought with them the advent of Dynamic HTML, which used scripting languages to alter pages so that you could drag and drop items or make menus appear and disappear without requiring a page refresh. Within a year, though, there was a rush toward the "thin-client," with the introduction of server-side technologies such as Active Server Pages and PHP. The client-side techniques still exist, but the model of current Internet and web page usage is broadly based on the server-side method of "enter your data, send the page to the server, and wait for a response."

When one format predominates in the stampede to adoption, you can often forget what was good about the other format. For example, some aspects of page validation can be performed equally as well on the browser. If you were to type "fake e-mail" into an e-mail textbox, you wouldn't need to go to the server to check this. JavaScript can perform a check for you equally as efficiently, and also much more quickly. While plenty of people sensibly do validation on both client and server, many pages attempt to perform the processing only on the server. If there has been one continual bugbear about the Web, it is that it is slow. Timeouts, page-not-found errors, unresponsive buttons and links haven't gone away, despite the fact that bandwidth has increased tenfold. So, other ways of addressing this sluggishness are becoming more common.

Companies have begun to reevaluate the way they are doing things to see if they can improve the user experience on several levels — making pages faster and more responsive, but also offering a

more seamless and richer experience. This often involved going back to old techniques. The first and best example of creating web applications in this "new" way was Google's Gmail. Google also used these techniques in the applications Google Suggest and Google Maps, although neither application showcases them quite in such an effective way or enjoys quite the same notoriety. Windows Live Mail (formerly named Kahuna), Amazon's search engine `A9.com`, Yahoo's `flickr.com` for organizing photos online all were lacking a common way of describing their features, until an online article in 2005 changed all that by christening these techniques *Ajax*.

What Is Ajax?

Ajax is the catchy term coined by Jesse James Garrett in his 2005 article for Adaptive Path called "Ajax: A New Approach to Web Applications," which can still be found at `http://adaptivepath.com/publications/essays/archives/000385.php`. You should read this article if you haven't already, although not before you finish this chapter, because it can be slightly misleading as to exactly what Ajax is! Ajax is also an acronym, but for the same reasons, let's defer explaining just what it stands for right now. Ajax didn't exist before this article, but the features the article described certainly did.

In short, Ajax is a set of programming techniques or a particular approach to web programming. These programming techniques involve being able to seamlessly update a web page or a section of a web application with input from the server, but without the need for an immediate page refresh. This doesn't mean that the browser doesn't make a connection to the web server. Indeed, the original article paints a slightly incomplete picture in that it fails to mention that server-side technologies are often still needed. It is very likely that your page, or data from which the page is drawn, must still be updated at some point by a rendezvous with the server. What differs in the Ajax model is that the position at which the page is updated is moved. We'll look at the two models in more detail shortly.

Garrett's article envisaged a world where web applications could be mirrored Windows applications in their functionality. "Richness," "responsiveness," and "simplicity" were the key words involved. He envisioned a new breed of applications, one that would close the gap between the worlds of Windows and web applications. He cited Gmail, Google Suggest, and Google Maps as key exponents of this new approach.

The article — and even the term "Ajax" — polarized people. While plenty of people loved it and took up its creed, many developers criticized aspects from the name "Ajax," calling it banal, to the techniques described, which weren't (by any stretch of the imagination) new. There was definitely a hint of the modern art hater's typical criticism about abstract art — "Hey, I could do that and so could my 10-year-old" — about the complaints. Just because people could have been using these techniques to create their web pages and applications didn't mean they had been. Unfortunately, jealousy and backbiting reigned.

What emerged, though, was a consensus that the techniques and ideas that Jesse James Garrett described really struck a chord (such as "*If we were designing the Web from scratch for applications, we wouldn't make users wait around*" and "*The challenges are for the designers of these applications: to forget what we think we know about the limitations of the Web and begin to imagine a wider, richer range of possibilities*"). It was a call to arms to use existing mature and stable methods to create web applications rather than the latest flaky beta. It invited developers to leverage the existing knowledge of JavaScript, style sheets, and the Document Object Model (DOM), instead of sweating blood to get up to speed on the latest tag-based page-building language. It was liberating, and overnight job ads were reworded — "Wanted: developers with five years `JavaScript` Ajax experience."

This doesn't really give you a feel for what Ajax does, and as always, the best way is to walk though some Ajax techniques currently being used on the Web.

Ajax in Action

Undoubtedly, the Ajax "killer" application is Gmail, which allows users to edit and update their e-mails and inbox without hundreds of page refreshes. Overnight, it convinced people who would use applications such as Outlook, Outlook Express, or Thunderbird on their own machines to use a web-based e-mail system instead. Unfortunately, Gmail can't easily be demonstrated without signing up, and currently signups are available only to a limited amount of countries. So, let's take a look at some other examples.

flickr

Yahoo's `flickr.com` is a photo-organizing site that lets you "sort, store, search and share photos online." Previously, flickr had used Flash as the main tool behind its photo display interface, but in May 2005 it announced that it was moving over to using Dynamic HTML and Ajax (`http://blog.flickr.com/flickrblog/2005/05/from_flash_to_a.html`). You have to sign up for an account to be able to see the tool in action, but because it's free and photo manipulation tools on web applications are a great way of demonstrating Ajax techniques, you should look into it.

Once you've logged in, you can access the tool via the Organize menu by selecting the Organize All Your Photos option (Figure 1-1). You can drag and drop photos into a single batch, and then you can rotate them and amend their tags.

Figure 1-1: Organize your photos option.

This is all done seamlessly using Ajax techniques. In addition, the "Send to Group," "Add to Set," and "Blog this" buttons also perform their functionality right there on the page. flickr even talks about the fact that the pages use an "old technology" to achieve this. The Dynamic HTML techniques replaced Flash because Flash requires you to have the latest Macromedia plug-in installed and the wrapper for Flash can take a long time to load. Users complained about this. Flash is a great application, but here Ajax techniques prove more efficient.

Basecamp

Basecamp is a web-based tool for managing and tracking projects. You can find it at `www.basecamphq.com`. Once again, you have to sign up to use it, but there is a free sign-up option with a reasonable range of capabilities included.

Basecamp employs Ajax when adding people to a company, adding companies to a project, adding/editing/deleting/reordering/completing to-do items, adding/editing/deleting/reordering to-do lists, adding/editing/deleting time items, and renaming attachments to messages (Figure 1-2).

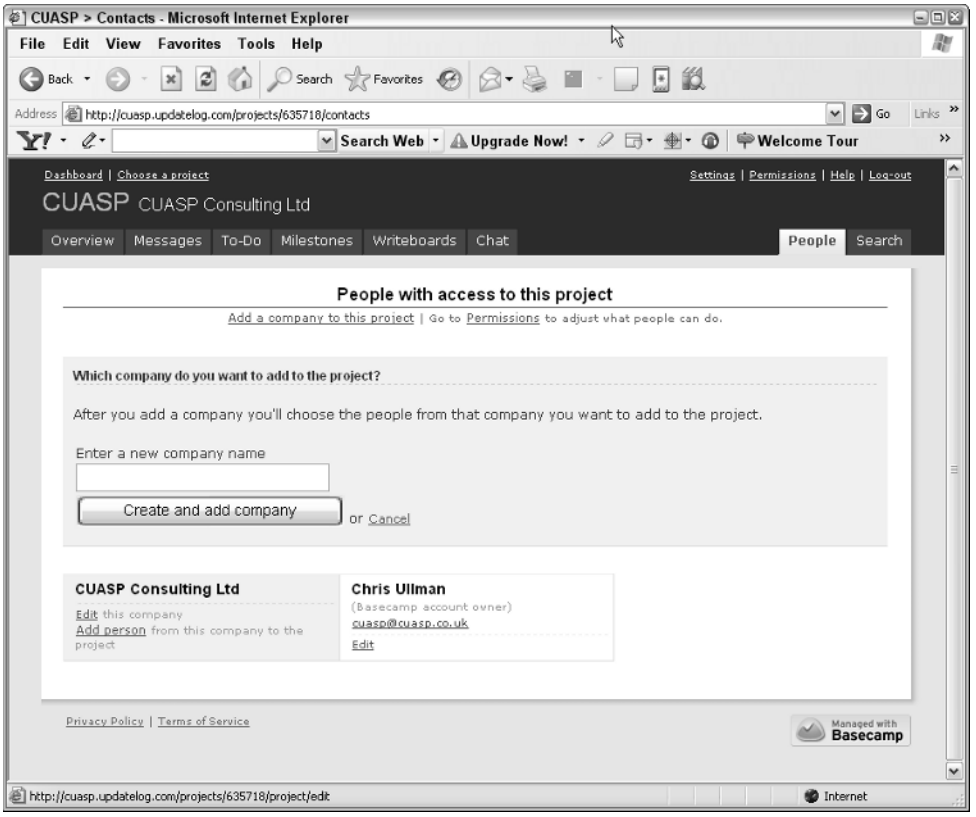

Figure 1-2: Adding people to a company.

When you click to add a person to the company, the dialog drops down smoothly without a screen refresh. Also, the hints and tips panel disappears when you select the button "Don't show this again."

Not all of Basecamp employs Ajax, but what makes it compelling is the mixture of server-side interaction and Ajax techniques that aren't overused, but are put into practice when they can be of benefit to the end-user experience.

Amazon (A9.com)

Amazon is well known for its store site, but less well known for its search engine, `www.A9.com`, which combines search results from a whole host of sources. Search engines have remained fairly static in terms of their user interface since their inception. You type a word and click on Search. What has changed is the complexity of the searches behind the engine and the different sources you can now interrogate. Amazon's A9 allows you to search on a single term across media such as movies, books, Wiki pages, and blogs. Figure 1-3 shows what a typical search on "Tour De France" would yield.

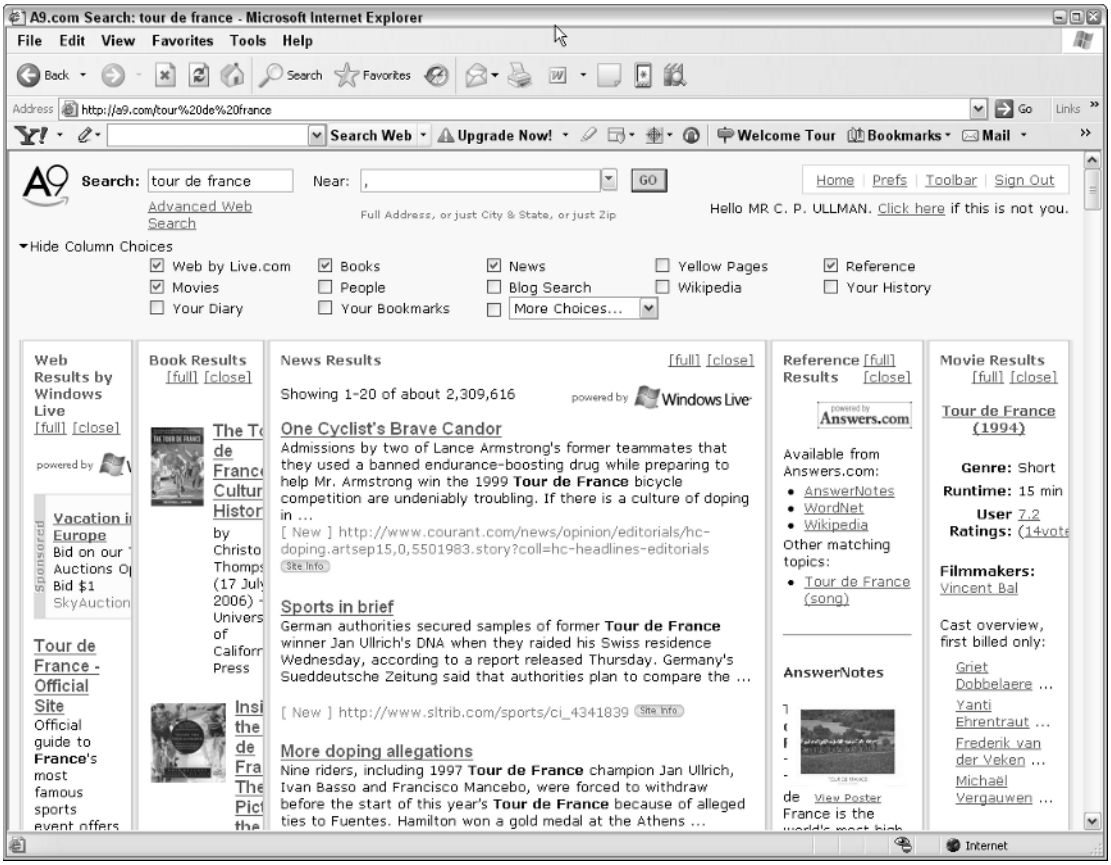

Figure 1-3: Results of a typical search on "Tour De France."

Ajax comes into play where you click on the check boxes to add or remove searches. You can click on the People link, and a box will appear without a refresh, even though you didn't include it in the original search. You can uncheck the References and Movies boxes, and immediately these links disappear. It's instantaneous and very much in the spirit of a rich and responsive user interface.

Google Suggest and Google Maps

Google Suggest and Google Maps were both mentioned in the Adaptive Path article as good examples. These are mentioned last, though, because their use of Ajax techniques is less pervasive than in the other web applications mentioned and because they are probably overfamiliar as examples. Google Suggest is a version of a search engine that attempts to offer suggestions as you type other similar searches. As shown in Figure 1-4, it can be found at `www.google.com/webhp?complete=1&hl=en`.

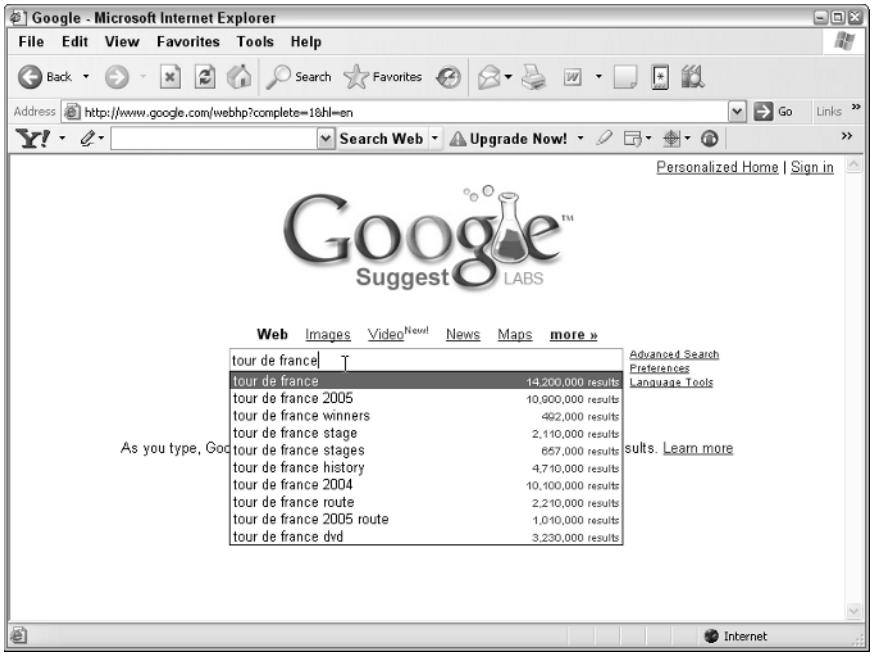

Figure 1-4: Google Suggest.

One thing that makes me slightly reluctant to recommend this site for Ajax techniques is that auto-suggest boxes have a bad press with some users. They can be seen as intrusive or confusing or both. Users of Short Message Service (SMS) text services on mobile phones will be only too familiar with the sense of frustration when words such as "no" are helpfully changed to "on." This has much to do with the code that does the "predicting" of what you're going to type. In Google Suggest, this is a good application of the technique, but when using something similar to perform form validation, you should be careful not to intrude on the usability of the form. A simple error message will often suffice.

Google Maps uses Ajax. When a location on the map is pinpointed it will load the relevant section of the map. If you then scroll along, rather than keeping the entire map in memory, it will load the map in blocks as and when you need them. You can see this in action in Figure 1-5; the loading is very fast indeed.

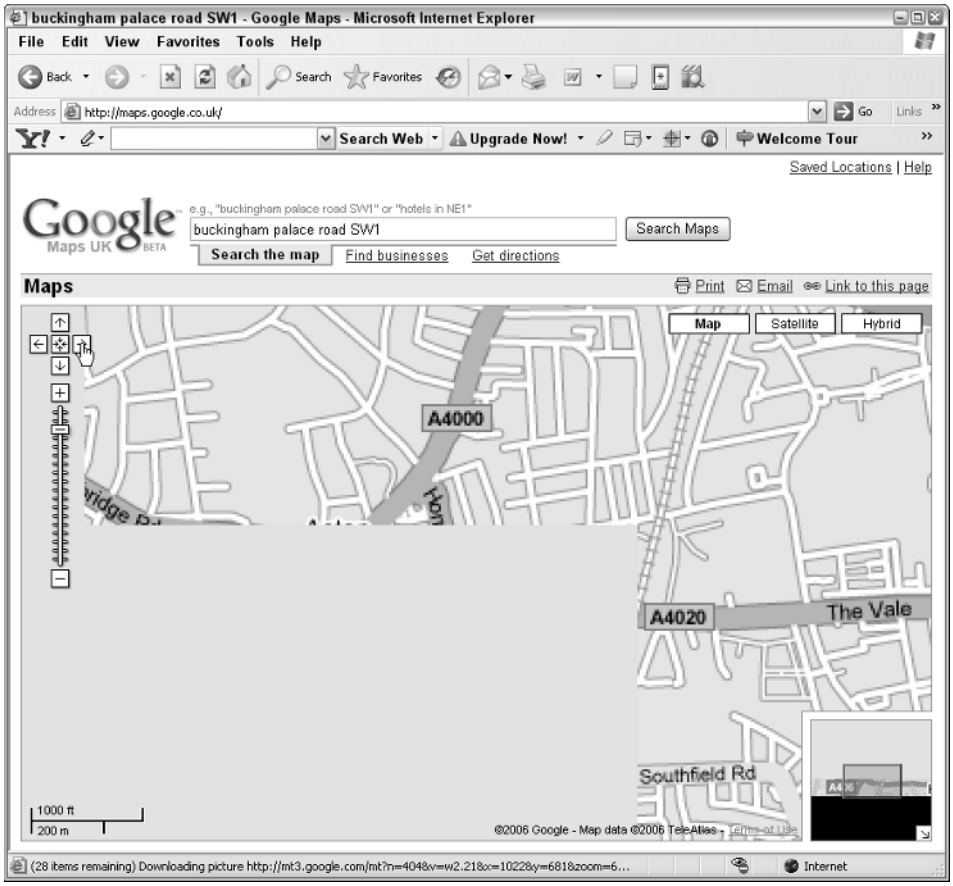

Figure 1-5: Google Maps.

This is another example of using just one technique to enhance the usability and end-user experience of a site.

Other Sites

Plenty of other sites employ Ajax techniques, and, of course, it's not possible to preview them all. A good place to start looking is A Venture Forth's list of the top 10 Ajax applications as created by start-ups. This can be found at: `www.aventureforth.com/2005/09/06/top-10-ajax-applications/`. Its list contains an online calendar, a word processor, an RSS reader, a project management tool and a version of Google Suggest for Amazon Suggest. This set gives a nice overview of the kinds of things you can put to use. You're limited only by your imagination.

Last, there is a good list of the kind of tasks you might use Ajax techniques to perform at the site `http://swik.net/Ajax/Places+To+Use+Ajax`. It's a Wiki, so it's interactive, and people are constantly adding suggestions. Here are some of the better suggestions:

- **Dynamic menus** — Web sites are constantly changing beasts, and having a rigid structure imposed on them only means more work when the structure changes. Menus are more responsive when handled by client-side code, and you can pull the data to fill them using Ajax.
- **AutoSave** — A good behind-the-scenes procedure is to save the contents of a textbox without a user's prompting.
- **AutoComplete** — Predictive text phrases, like Google Suggest, if done well, can speed up the process of typing.
- **Paginating or organizing large numbers of results** — When large amounts of data are returned by a query such as a search, then you could use an Ajax application to sort, organize, and display the data in manageable chunks.
- **User-to-user communication** — You would probably think twice about using MSN Messenger if you had to refresh the screen every time you sent or received a message. With online forums or chat areas on web applications, however, this can often be the case. Having communication come up instantly with another user in a web application is a good place where Ajax could be used.

You can see that these kinds of tasks are already achieved using server-side technologies. These tasks, though, could be better accomplished by reverting to the client to perform some of the processing, to achieve a faster and more responsive site.

Bad Examples

Of course, not all examples of Ajax in use on the Web are good ones. As already mentioned, not all areas are suitable for use with Ajax. In Alex Bosworth's list of Ajax mistakes at `http://alexbosworth.backpackit.com/pub/67688`, a new note has been appended to the list "Using Ajax for the sake of Ajax." Perhaps what irritates a lot of Ajax's detractors is that developers will take a cutting-edge technology or methodology and apply it, regardless of its suitability for a particular task.

So, what exactly makes for a bad Ajax example? Providing some URLs might be instructive and a little cheeky, but probably would encourage a lot of flames, not to mention lawsuits. It's also quite subjective. Let's define "bad" examples as sites that use Ajax and that are slower in what they achieve than they might be with a simple submit to the server.

For example, you may have seen a search engine paginator that, while it doesn't update the page, takes longer to return the results than it might otherwise do if it just sent the query off to the server. The problem is that by shifting the focus back to the client, you are also partially dependent on the user's machine resources and the browser involved. If you give them large amounts of data to process, then a Pentium 2 isn't going to process it as fast as Pentium 4.

Another bad example of Ajax would be in form validation if you actually interrupt what the users are typing before they've finished typing it. You don't want to interrupt the normal pattern of how a user types or enters data. One of the key aims of Ajax must be to improve the user's overall experience. In some cases, this might mean that the use of Ajax is very subtle or barely noticeable, but when it comes to user interfaces, that is a good thing.

Now it's time to talk about what Ajax stands for and why what it stands for isn't necessarily what Ajax is all about right now.

Ajax: The Acronym

If you read the Adaptive Path article, then you'll already know that Ajax the acronym stands for *Asynchronous JavaScript and XML*. Here's a curveball: Ajax doesn't have to use XML, and neither does it have to be asynchronous. Ajax applications can use XML, and they can be updated asynchronously. These are quite common tricks and techniques used to update the page, but they are not tied to these technologies.

To reiterate an earlier point, Ajax is "a set of programming techniques," "a particular approach to web programming." It isn't rigid; it isn't like a members-only club, if you don't use one technique then it isn't Ajax; it's an overall guiding philosophy. How you achieve these objectives on the client is up to you. The objectives, though, prove a good starting point. Jesse James Garrett mentioned in the article "several technologies... coming together in powerful new ways." Here are the technologies he specifically mentioned:

- XHTML and CSS
- The Document Object Model (DOM)
- JavaScript
- XML and XSLT
- The `XMLHttpRequest` object

In reality, to create an application using Ajax techniques you need only three of these: XHTML, the DOM, and JavaScript. If you do any amount of development with Ajax techniques, though, you will almost certainly need to use all of the technologies at some point.

You'll also probably need a server-side language to handle any interaction with the server. This is most typically one of the following three:

- PHP
- ASP.NET (Visual Basic.Net/C#)
- Java

When building a web page, you'll probably have encountered many or most of these technologies, but perhaps not all, so it's worth having a quick reminder of what each one is and does, its role in web development, and how it pertains to Ajax.

XHTML and CSS

You will be familiar with HyperText Markup Language (HTML), the lingua franca of the Web, but perhaps not so familiar with its successor, eXtensible HyperText Markup Language (XHTML). XHTML is the more exacting version of HTML. In fact, it is the HTML standard specified as an XML document. The main difference with this is that whereas HTML has been fairly easygoing and the browser will make a reasonable attempt to display anything you place in tags, XHTML now follows XML's rules. For example, XML documents must be well formed (tags are correctly opened and closed, and nested), and so must XHTML pages. For example, the following is correct nesting:

```
<div>
<h1>
```

```
            This is a correctly nested H1 tag
    </h1>
    </div>
```

The following is incorrect nesting:

```
    <div>
    <h1>
            This is an incorrectly nested H1 tag
    </div>
    </h1>
```

Although it might seem to go against the grain of HTML's easygoing and easy-to-code nature, if a page isn't correctly constructed, then you won't be able to perform the kind of Ajax techniques discussed in this book. To use the DOM, the page has to be correctly formed. Otherwise, you won't be able to access the different parts of the page.

Cascading Style Sheets (CSS) are the templates behind HTML pages that describe the presentation and layout of the text and data contained within an HTML page. CSS is of particular interest to the developer because changes made to the style sheet are instantly reflected in the display of the page. The style sheets are linked into the document commonly with the HTML `<link>` tag, although it is possible (but not preferable) to specify style attributes for each individual HTML tag on a page. You can also access CSS properties via the DOM.

In the design of any web site or web application, you should make the division between the content/structure of the page and the presentation as clear as possible. Suppose you have 100 pages and you specify the font size on all 100 pages as a style attribute. When you're forced to change the font size you will have to change it on each individual page, instead of changing it just once in the style sheet.

Having a style sheet isn't 100 percent essential, but to keep good organization, style sheets are an indispensable aid.

The Document Object Model (DOM)

The DOM is a representation of the web page as a hierarchy or tree structure, where every part of the page (the graphics, the text boxes, the buttons, and the text itself) is modeled by the browser.

Before IE 4 and Netscape Navigator 4, not every part of the web page was accessible to code. In fact, changing text on a web page had to be done by using server-side technologies or not at all. The whole page was termed a *document*, and that document contained all the HTML tags and text that made up the page. The DOM is specified as a standard by the World Wide Web Consortium, also known as W3C (www.`w3.org`), and so it is a standard way for all browsers to represent the page. You can pretty much guarantee that when you use JavaScript to alter the background color of a page in IE, it will correctly do so in Mozilla, Safari, or Opera as well. There are exceptions to the rule, though. There are several non-standard methods in IE, and items such as ActiveX controls can't be used in the other browsers.

You can add items to the DOM or alter them using a scripting language (such as JavaScript or VBScript), and they will appear on the page immediately. They are typically addressed in the format that addresses the page in hierarchical format, such as the following code, which addresses a button called "button" on a form and changes the text of that button. Note that in this code fragment, the form element has the name attribute set to `form1`:

```
document.form1.button.value = "Click Me";
```

Or, you can use methods that can access the specific elements or subsets of elements on the page, such as the `document.getElementById` method, which will return a specific instance of an element that matches the criteria:

```
var myTextBox = document.getElementById("myTextbox");
```

You can then assign values to the variable you have created to alter the values. To make the text box invisible, you could call the following:

```
myTextBox.style.visibility = "visible";
```

Another related method is the `getElementsByTagName` method. The `getElementsByTagName` method will return an array of elements on the web page of type `NodeList`, all with a given tag name, even if there is only one occurrence of that element on the page. The following code will return all the image elements on the page:

```
var imageElements = document.getElementsByTagName("img");
```

It is also possible to assemble the page by adding new sections to the document known as *nodes*. These can be elements, attributes, or even plain text. For example, you could create a span tag that contains a short message and add it to the page as follows:

```
var newTag = document.createElement("span");
var newText = document.createTextNode("Here is some New Text. Ho Hum.");
newTag.appendChild(newText);
document.body.appendChild(newTag);
```

All of these DOM techniques are applicable to the client side, and as a result, the browser can update the page or sections of it instantly. Ajax leans on these capabilities very heavily to provide the rich user experience. Also, as mentioned, these techniques have been around since version 4 of IE. It's just that they have been underused. The DOM is an important topic, and it is discussed in much more detail in Chapter 2.

JavaScript

JavaScript is the scripting language of choice of most web developers. You must know JavaScript to be able to use this book. This book doesn't teach JavaScript, although a quick refresher of the most salient points is available in Chapter 2.

Ajax techniques aren't solely the preserve of JavaScript. VBScript also offers the same capabilities for dynamic updates as well — albeit tied to IE only. While JavaScript has a standard specified in the ECMAScript standard, JavaScript was initially created in Netscape Navigator before such standards existed. Microsoft created its own version of JavaScript (called *JScript*) in parallel, and as a result, each browser's version of JavaScript is slightly different. Although JavaScript remains a very powerful method for updating your web pages, some amount of dual-coding is necessary to make sure that the web pages and applications function in the correct way across browsers, When not possible, some error-handling code will also be necessary.

A fair amount of Ajax code will deal with handling cross-browser code and handling errors if and when they arise, unless you can guarantee that your target audience will only ever use one browser (such as on a local intranet). This is an unfortunate set of circumstances that even new versions of IE and Firefox are not able to rectify. Later chapters address both of these dilemmas.

XML, XSLT, and XPath

XML is another familiar cornerstone, the language that is used to describe and structure data, to any web developer.

With XML comes a whole plethora of supporting technologies, each allocated its own individual niche. An XML document contains no information about how it should be displayed or searched. Once the data is rendered as an XML document, you typically need other technologies to search and display information from it. IE and Firefox both contain XML engines that can parse XML documents.

XSLT is a language for *transforming* XML documents into other XML documents. It isn't restricted to transforming XML documents into XML. You could also specify HTML or pure text as the output format. When you transform a document, you start with a source XML document, such as the following fragment:

```
<HotelList>
  <Hotel>
    <Name>Hotel Shakespeare</Name>
    <Rooms>50</Rooms>
    <City>Birmingham</City>
  </Hotel>
  <Hotel>
    <Name>Hotel Washington</Name>
    <Rooms>500</Rooms>
    <City>Chicago</City>
  </Hotel>
</HotelList>
```

You apply a second document in XSLT to the first document. The XSLT document contains a set of rules for how the transformation should be conducted, as shown here:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
      <table>
     <tr>
          <td>
            Hotel Name
          </td>
          <td>
          Number of Rooms
          </td>
          <td>
          Location
          </td>
     </tr>
        <xsl:for-each select="//Hotel">
```

```
          <tr>
            <td>
                <xsl:value-of select="Name" />
            </td>
            <td>
                <xsl:value-of select="Rooms" />
            </td>
            <td>
                <xsl:value-of select="City" />
            </td>
           </tr>
          </xsl:for-each>
       </table>
    </xsl:template>

  </xsl:stylesheet>
```

And you get a resulting XML document (in this case, also an XHTML document) looking like this:

```
         <table>
        <tr>
             <td>
               Hotel Name
             </td>
             <td>
             Number of Rooms
             </td>
             <td>
             Location
             </td>
        </tr>
           <tr>
             <td>
                 Hotel Shakespeare
             </td>
             <td>
                 50
             </td>
             <td>
                 Birmingham
             </td>
            </tr>
           <tr>
             <td>
                 Hotel Washington
             </td>
             <td>
                 500
             </td>
             <td>
                 Chicago
             </td>
            </tr>
          </table>
```

You could then insert the XHTML document fragment into the HTML page dynamically to update the page.

XSLT is another standard maintained by W3C. Both IE and Mozilla have XSLT processors; however, they don't always treat their XSLT documents in the same way. XSLT uses another language, XPath, to query the XML document when applying its transformations. XPath queries are used to address elements within the original XML document, such as the following:

```
            <xsl:for-each select="//Hotel">
```

The `//Hotel` statement instructs the browser's XSLT processor to look for elements called `<Hotel>` that are descendants of the root element `<HotelList>`. XPath queries can be used to locate specific items or groups of items within an XML document, using a syntax that is superficially similar to the way browsers can locate web pages, with the following as an example:

```
//HotelList/Hotel/Rooms
```

These techniques can be used to retrieve sections of data and display it on the page using the browser. Again, they are able to offer instantaneous updates without the need for a trip back to the server. Both XSLT and XPath are discussed in more detail in Chapter 8.

The XMLHttpRequest Object

If there's one thing you haven't come across before, it's likely to be the XMLHttpRequestObject. Microsoft introduced quite an obscure little ActiveX control in Internet Explorer version 5 (IE 5) called the `XMLHttp` object. Of course, ActiveX controls are tied to IE, so shortly afterward, Mozilla engineers followed suit and created their own version for Mozilla 1 and Netscape 7, the corresponding `XMLHttpRequest` object. The status of these objects was elevated considerably because of their inclusion in the original Ajax article. They are now more commonly referred to singularly as the `XMLHttpRequest` object, and, in IE 7, there is a native `XMLHttpRequest` object in addition to the ActiveX Control. Versions of this object have been included in Safari 1.2 and Opera as well.

But what does it do? Well, it allows a developer to submit and receive XML documents in the background. Previously, you could use hidden frames or IFrames to perform this task for you, but the `XMLHttpRequest` is rather more sophisticated in the ways in which it allows you to send and pick up data.

Unfortunately, because it isn't yet standard, that means there are still two separate ways of creating it. In versions of IE prior to version 7, it is created as an `ActiveXObject` in JavaScript as follows:

```
var xHRObject = new ActiveXObject("Microsoft.XMLHTTP");
```

> **There are several versions of the MSXML library with respective `ProgIDs` to instantiate `XMLHttp` object. Specifically, you cannot instantiate any version of `XMLHttp` object higher than `'Msxml2.XMLHTTP.3.0'` with the given version independent `ProgID` `Microsoft.XMLHTTP`. For the examples in this book, though, this will be sufficient.**

In Mozilla Firefox, IE 7, and other browsers, it is created as follows:

```
var xHRObject = new XMLHttpRequest();
```

In turn, this means that you have to do a little bit of browser feature detection before you can determine in which way the object needs to be created. Because ActiveX controls and objects are unique to IE, you can test for them by attempting to call the `ActiveXObject` method of the window object. For IE 7, Mozilla Firefox, Safari, and Opera, there is an `XMLHttpRequest` method. These calls use implicit type conversion to return `true` or `false`, depending on which browser is being used to view the page. In other words, if there is an `ActiveXObject` on that browser, it will return `true`. If not, it will return `false`.

Typically your browser feature detection and object creation code would look something like this:

```
var xHRObject = false;
if (window.XMLHttpRequest)
{
xHRObject = new XMLHttpRequest();
}
else if (window.ActiveXObject)
{
xHRObject = new ActiveXObject("Microsoft.XMLHTTP");
}
else
{
     //Do something to handle non-Ajax supporting browsers.
}
```

There are plans to standardize this functionality in level 3 of the DOM specification, but until then, you will need to make allowances for this fact.

The role the `XMLHttpRequest` object plays is a major one. It is used heavily in Google's Gmail to ferry the data backward and forward behind the scenes. In fact, it can provide the asynchronous part of the Ajax application. It uses the `onreadystatechange` event to indicate when it has finished loading information. You can tell it to get an XML document as shown here:

```
       xHRObject.open("GET", "SuiteList.xml", true);
```

Then, at a later point, you can check the value of the `readyState` property to see if it has finished loading and, if it has, then extract the information that you need.

The `XMLHttpRequest` object is misleading in one sense, in that you don't have to transfer XML with it. You can quite happily use it to transfer HTML or just plain text back and forth as well. The `XMLHttp Request` is examined in more detail in Chapter 4. The difference between synchronous and asynchronous methods of data transfer is examined later in this chapter.

Server-Side Technologies

The last piece of this equation (and one not really touched on in Garrett's original article) is the server-side technologies that Ajax will need to use. This book uses PHP and ASP.NET (where appropriate) to service requests to the server. Any one method is not better than another; rather, you should use the one that you are most familiar with.

As with JavaScript, this book does not teach you how to use PHP and ASP.NET. It's expected that you will know either one or the other. Server-side technologies are examined more closely in Chapter 3, but the current discussion attempts to avoid going into them in too much detail because they provide the glue for Ajax, rather than the backbone.

With all the pieces of the Ajax puzzle in place, let's now look more closely at how Ajax changes the traditional method of web interaction.

The Ajax Application Model

At first, the Web intended to display only HTML documents. This means that the classic web application has an "enter your data, send the page to the server, and wait for a response" model, intended only for web pages. Second, there is the problem of synchronous communication. A good example of a real-world synchronous device is a public telephone booth. You have to call somebody, and that person has to be available to communicate with you. With many public telephone booths, you cannot get the receiver to call you back. You can only call the receiver and impart information. After the call you must leave the booth, as quite often there will be other people waiting to use it. If the receiver hasn't given you the information you need, then that is too bad. An example of asynchronous communication would be with your home phone where you phone someone, and they can't give you the information you need right now, so they agree to call you back when they do have the information, whenever that might be.

On the Web, synchronous means that the user requests an HTML page, and the browser sends an HTTP request to a web server on his or her behalf (Figure 1-6). The server performs the processing, then returns a response to the browser in the form of an HTML page. The browser displays the HTML page requested. The browser always initiates the requests, whereas the web server merely responds to such browser requests. The web server never initiates requests — the communication is always one way. The "request/response" cycle is synchronous, during which the user has to wait.

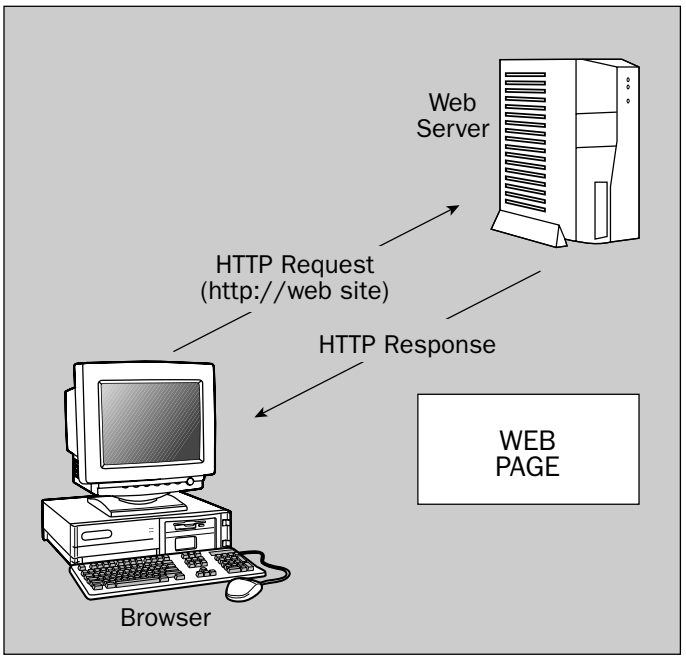

Figure 1-6: Synchronous model.

As mentioned, this model works for browsing web pages, but the development of more and more complex web applications means that this model is now breaking down. The first area in which it breaks down is that of performance. The "enter, send, and wait" approach means there is wasted time. Second, whenever you refresh a page, you are sending a new request back to the server. This takes up extra server processing, and it leads to time lost while waiting for a response and higher bandwidth consumption caused by redundant page refreshes. The underlying problem is that there is a complete lack of two-way, real-time communication capability, and the server has no way in which to initiate updates.

This scenario leads to slow, unreliable, low-productivity and inefficient web applications. You have two basic problems here: one of having to wait for a response for the server and one of the server not being able to initiate an update. The Ajax application model seeks to produce higher performance, thereby creating more efficient web applications by subtly altering the way in which this works.

Ajax introduces the idea of a "partial screen update" to the web application model. In an Ajax application, only the user interface elements that contain new information will be updated. The rest of the user interface should be unchanged. This means that you don't have to send as much information down the line, and you're not left waiting for a response because the previous page is already operational. This model enables continuous operation of a web page, and it also means that work done on the page doesn't have to follow a straight, predictable pattern.

Instead of a synchronous model, you can now have either an asynchronous model or a polling one. In an Ajax application, the server can leave a notification today when it's ready, and the client will pick it up when it wants to. Or, the client can poll the server at regular intervals to see if the server's ready, but it can continue with other operations in the meantime, as shown in Figure 1-7.

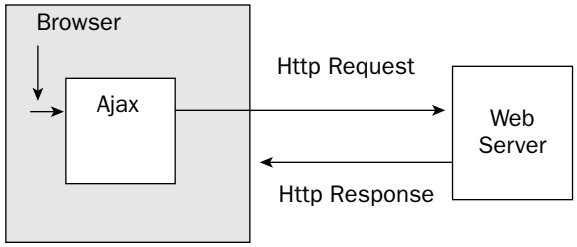

Figure 1-7: Model in Ajax application.

As a result, the user can continue to use the application while the client requests information from the server in the background. When the new data finally does turn up, only the related parts of the user interface need to be updated.

Why Should I Use Ajax?

We've looked at the model, but let's now spell out in real terms the advantages of using Ajax to create your applications. This is my equivalent of the second-hand car dealer's hard-sell.

Partial Page Updating

You don't have to update the data on an entire page. You can update only the portions of the page that require it. This should mean no full page refreshes, less data to be transferred, and an improved flow for the user experience. You don't have to stutter from page to page.

Invisible Data Retrieval

The longer you look at a web page, the greater its chance to go out of date. With an Ajax application, even though on the surface the web page might not be told to do anything, it could be updating itself behind the scenes.

Constant Updating

Because you're not waiting for a page refresh every time, and because Ajax can retrieve data under the covers, the web application can be constantly updated. A traditional software application such as Word or Outlook will alter the menus it displays or the views it shows, dependent on its configuration, or the data it holds or the situation or circumstances it finds itself in. It doesn't have to wait for a server or user to perform an action before it can download new mail or apply a new template. Ajax techniques enable web applications to behave more like Windows applications because of this.

Smooth Interfaces

An interface that doesn't have to be changed is almost inevitably a user interface that is easier to use. Ajax can cut both ways here in that you can use it to modify parts of the interface and simply confuse users by changing the ground beneath their feet. In theory, by making subtle alterations, you could aid the user's passage through an interface or wizard and speed up the process.

Simplicity and Rich Functionality

As shown in the previous examples, some of the most impressive Ajax applications are those where you had to look for the Ajax functionality, such as Basecamp. If Ajax can be used to make your applications simpler while improving the user's experience, then that must be an improvement.

Drag and Drop

Drag-and-drop functionality is one of the neatest features of most software applications, from Windows Explorer to Windows Desktop. It doesn't strictly qualify as Ajax functionality. It's something that's been possible for a great many years in web applications, even before the introduction of the `XMLHttpRequest` object. Most developers seem to opt for Flash or some similarly heavyweight solution rather than using the JavaScript and DOM solutions. In the reassessment of user-interface creation techniques that Ajax has introduced, drag-and-drop functionality can be used to manage front-end changes, and then these changes are submitted via Ajax to the server. For example, you drag several items on the screen into new positions, and then you log out. Later, when you come back, those items are located in the same positions.

With such a host of benefits, it might be difficult to imagine why everyone isn't switching over to using Ajax applications. There are definitely quite a few pitfalls, though, to consider.

When Not to Use Ajax

Once you've learned Ajax techniques, you shouldn't go back and tear up all your old web pages and start again. Rather, you should consider how you could improve the usability of your pages and whether Ajax is a solution that will help improve them. Without advances in usability, you might as well not bother at all.

A considerable number of articles explain why you shouldn't use Ajax techniques in certain situations, such as those found at the following URLS:

- **Ajax sucks** — `http://www.usabilityviews.com/ajaxsucks.html` (a spoof article on Jakob Nielsen's why frames suck)
- **Ajax promise or hype** — `http://www.quirksmode.org/blog/archives/2005/03/ajax_promise_or.html`
- **Ajax is not cool** — `http://www.lastcraft.com/blog/index.php?p=19`

Poor Responsiveness

Perhaps the most glaring problem noticed with Ajax is when it is used in theory to speed up the interaction, and it actually slows down the page.

If you continually use Ajax to submit data to the server, then you could find that the page has slowed down. Alternately, if the server returns large datasets to the client, you can find that your browser will struggle to cope with the data in a timely manner. In theory, a search engine might seem like a good place to use Ajax to break up the data so that you don't have to refresh the page when you move from pages 1–10 to pages 11–20. In reality, you'd still be much better off, though, using the server to slice the dataset up into small chunks, rather than getting the browser to attempt to deal with a 1,000-record-plus dataset. Intelligent and judicious applications of Ajax techniques are the key here.

Breaks the Back Button on Your Browser

Ajax applications can, if not judiciously applied, break the Back button on your browser.

One of the prime examples of a great Ajax application, Gmail, uses the Back and Forward buttons very effectively. Also, there are plenty of web sites out there that don't use Ajax techniques that disable the Back button anyway. In my list of crimes, this one doesn't rate particularly highly.

Breaking Bookmarks and Blocking Search Engine Indexes

If you bookmark a page in an Ajax application, you could, of course, end up with a different page in your list of favorites. JavaScript creates a particular view of the page, and this "view" has no dependence on the original URL. Also, conversely, with a dynamic menu system, you might think you have shut off part of your site that is no longer accessible via the menu and regard it as defunct when, in fact, others can still access it. Plenty of sites have thousands of hits for old versions of pages that just cause errors on the server.

It is possible to create web applications that handle bookmarks and Back buttons correctly. A good example of how to handle both effectively can be found at `http://onjava.com/lpt/a/6293`.

Strain on the Browser

If you push the entire burden for processing data back to the browser, you then become more reliant on the browser and the user's machine to do the processing for you. Waiting for the browser to process a page is no better than waiting for a web server to get back to you with a response. In fact, in some cases, it could be worse if it is IE you're waiting for and it decides to hog all of the processing space on your machine while it's doing the processing and make the whole machine respond at a crawl.

You should be getting a feel by now that each of these reasons is qualified — the use of Ajax can hinder usability, not aid it; however, if development is considered, if you take steps to address the problems with Back buttons and bookmarks, then there's no reason why Ajax techniques shouldn't be used. It does raise a question, though.

If Garrett's seminal article was so flawed, then why is it so popular? This might be because it articulated what a lot of people were thinking and doing already, but didn't have a name for. It didn't matter that there were some inaccuracies about the methods of execution. It completely struck a chord with developers. You could liken it to some of the early Beatles recordings that weren't the best played, but they're rightfully remembered affectionately because the tunes always shone through. Even developers who had issues with the article agreed that it described a sensible approach to creating web applications and that it revisited old techniques that had been either ignored or forgotten.

Who Can/Can't Use Ajax?

It might go without saying that not everyone will be able to use Ajax. You should always be aware of any portion of your web site that might exclude some users by its use of a particular technology, and you should consider how to address that portion of the target audience.

Here are some of the main considerations:

- Users of reasonably modern versions of the main browsers (IE 4+, Mozilla Firefox/Netscape 7+, Safari, and Opera 5+) will find they can use Ajax applications just fine. Users of older browsers won't be able to.
- People who have scripting languages disabled won't be able to use Ajax (or indeed JavaScript applications). Before you scoff, just remember that after some of the recent vulnerabilities in both IE and Microsoft Outlook, the initial advice from Microsoft was to disable active scripting until a patch was in place to fix the vulnerability.
- People browse applications offline. Again, it might almost seem a prehistoric way of using the Web, but it is, in fact, a very modern way, too. Mobile communications charge a lot more for bandwidth and offer a lot less throughput, so it's very common to log in, download, and log out. Sites such as AvantGo allow you to browse a series of web pages offline that you have subscribed to. If Ajax were included in these pages, it would break them.

Once again, be aware of who your target audience is and what their needs are.

Create Your Own Example

You shouldn't come away from this chapter feeling "so why bother using Ajax?" This hasn't been the point. There is a time and a place for using Ajax correctly. Consider it like a fast car. Sure, it's capable of doing 0–60 mph in 7 seconds and can reach top speeds of 150 mph. But in town or on the freeway is not the place to demonstrate those capabilities. You will incur the wrath of the rest of the populace, not to mention the police. The same goes for Ajax. If you use it at every available opportunity, you won't find quite as many visitors to your site as you perhaps envisioned.

Creating an Ajax application is about the design and thinking about how the user interface and experience will be improved by your application. We're going to create a quick example that shows how you can use the `XMLHttpRequest` object and XML files to create a dynamic menu system for a business scenario.

The business scenario is this. You have a travel company whose web site displays information about hotels and suites. The hotels wish to display booking information about only those suites that currently are available. This information is published as an XML file. It is quite common for companies to publish data as text or XML files to the same location at a given interval because it can usually be generated automatically. This example does not create the facility to publish the XML file, but rather checks and fetches new data from a given location every 10 seconds. This, in turn, will have the effect of updating the options on the menu system without the need for a page refresh.

Note that this example is quite complex. It demonstrates a meaningful way in which Ajax can be used. You should skim over the code. Later chapters discuss the `XMLHttpRequest` object, the use of the DOM, and scenarios in which you can use XML and XSL in your applications.

Try It Out First Ajax Example

1. Create a new folder in your web server's root directory called `BegAjax`. Inside that, create a folder called `Chapter1`.

2. You will start by creating the data file in XML. Save this as `SuiteList.xml` in the `Chapter1` folder. Note that all of these files can be created in Notepad or your preferred text or web page editor. If you don't wish to type the code in, it can be downloaded from `www.wrox.com` in the Downloads section.

```
<?xml version="1.0" encoding="utf-8" ?>
<SuiteList>
  <Suite>
    <SuiteID>1001</SuiteID>
    <Name>Ponderosa</Name>
    <Size>2</Size>
    <Price>50</Price>
    <WeeksFree>10</WeeksFree>
  </Suite>
  <Suite>
    <ProductID>1002</ProductID>
    <Name>Colenso</Name>
    <Size>2</Size>
    <Price>30</Price>
    <WeeksFree>10</WeeksFree>
  </Suite>
  <Suite>
    <ProductID>1003</ProductID>
    <Name>Dunroamin
    </Name>
    <Size>2</Size>
    <Price>60.00</Price>
    <WeeksFree>10</WeeksFree>
  </Suite>
```

```
  <Suite>
    <ProductID>1003</ProductID>
    <Name>Family</Name>
    <Size>6</Size>
    <Price>90.00</Price>
    <WeeksFree>10</WeeksFree>
  </Suite>
</SuiteList>
```

3. Next, you must create a JavaScript script to handle the client-side processing. Save this as `ajax.js`, and, once again, in the `Chapter1` folder.

```
// Create the XMLHttpRequest
    var xHRObject = false;
    if (window.XMLHttpRequest)
    {
        xHRObject = new XMLHttpRequest();
    }
    else if (window.ActiveXObject)
    {
        xHRObject = new ActiveXObject("Microsoft.XMLHTTP");
    }
    function getData()
    {
        //Check to see if the XMlHttpRequest object is ready and whether it has
        //returned a legitimate response
        if (xHRObject.readyState == 4 && xHRObject.status == 200)
        {
            var xmlDoc = xHRObject.responseXML;

if (window.ActiveXObject)
            {
                //Load XSL
                var xsl = new ActiveXObject("Microsoft.XMLDOM");
                xsl.async = false;
                xsl.load("MenuDisplay.xsl");

                //Transform
                var transform = xmlDoc.transformNode(xsl);
                var spanb = document.getElementById("menuhere");

            }

            else
            {
              var xsltProcessor = new XSLTProcessor();

              //Load XSL
              XObject = new XMLHttpRequest();
              XObject.open("GET", "MenuDisplay.xsl", false);
              XObject.send(null);

              xslStylesheet = XObject.responseXML;
              xsltProcessor.importStylesheet(xslStylesheet);

              //Transform
```

```
                var fragment = xsltProcessor.transformToFragment(xmlDoc, document);

                document.getElementById("menuhere").innerHTML = "";
               document.getElementById("menuhere").appendChild(fragment);
              }
              if (spanb != null)
              {
                 spanb.innerHTML = transform;
              }

              //Clear the object and call the getDocument function in 10 seconds
              xHRObject.abort();
              setTimeout("getDocument()", 10000);
          }
      }

        function getDocument()
      {
       //Reset the function
       xHRObject.onreadystatechange = getData;

       //IE will cache the GET request; the only way around this is to append a
       //different querystring. We add a new date and append it as a querystring
       xHRObject.open("GET", "SuiteList.xml?id=" + Number(new Date), true);

       xHRObject.send(null);

      }
```

4. You will also need an XSL style sheet for this example. This handles the presentation of the data contained in the XML document. This will be the section that controls which items from the XML document are displayed and which items are not. Save the XSL style sheet as `MenuDisplay.xsl` to the `Chapter1` folder.

```
<?xml version="1.0" encoding="utf-8"?>

<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:template match="/">
    <div onmouseout=" var submenu =
document.getElementById('romesubmenu');submenu.style.visibility = 'hidden';return
true;" >
      <table>
        <tr>
        <td id="td1" class="menublock"
onMouseOver="td1.style.backgroundColor='#cccccc';"
onMouseOut="td1.style.backgroundColor='#eeeeff';" >
          Hotel Paris
        </td>
      </tr>
      <tr>
```

```
        <td id="td2" class="menublock"
onMouseOver="td2.style.backgroundColor='#cccccc';"
onMouseOut="td2.style.backgroundColor='#eeeeff';"  >
          Hotel London
        </td>
      </tr>
      <tr>
        <td  id="td3" class="menublock"
onmouseover="td3.style.backgroundColor='#cccccc';var submenu =
document.getElementById('romesubmenu');submenu.style.visibility = 'visible';return
true;" onMouseOut="td3.style.backgroundColor='#eeeeff';" >
          Hotel Rome
        </td>
      </tr>
      <tr>
        <td id="td4" class="menublock"
onMouseOver="td4.style.backgroundColor='#cccccc';"
onMouseOut="td4.style.backgroundColor='#eeeeff';" >
          Hotel New York
        </td>
      </tr>
      <tr>
        <td id="td5" class="menublock"
onMouseOver="td5.style.backgroundColor='#cccccc';"
onMouseOut="td5.style.backgroundColor='#eeeeff';" >
          Hotel Montreal
        </td>
        </tr>
      </table>
        <xsl:call-template name="DynamicSubMenu"></xsl:call-template>

    </div>
  </xsl:template>


    <xsl:template name="DynamicSubMenu">
    <div id="romesubmenu" class="submenublock" onmouseover="var submenu =
document.getElementById('romesubmenu');submenu.style.visibility = 'visible';return
true;">
        <xsl:for-each select="//Suite">

          <xsl:if test="WeeksFree&gt;0">
            <a href="{Name}.htm">
              <xsl:value-of select="Name"/>
            </a>
            <br/>
        </xsl:if>

      </xsl:for-each>

    </div>
  </xsl:template>

</xsl:stylesheet>
```

5. Next is the CSS, which controls the switching on and off of the dynamic submenu and the presentation of the menu blocks, such as the colors and font size. Save this as `SuiteListStyles.css` in the `Chapter1` folder.

```
td.menublock
{
     background-color:#eeeeff; width:120px;height:25px;border-style:ridge; border-
width:thin; border-color:gray; font-weight: bold; text-align:center;  font-family:
Arial; color:gray;
}
div.submenublock
{
     visibility:hidden; background-color:lightsteelblue; position:absolute; top:
70px; left: 133px; width:180px;border-style:ridge; border-width:thin; color:gray;
font-weight: bold; text-align:center; font-family: Arial;
}
a
{
     color: White; text-decoration: none;
}

a:hover
     {background-color: #28618E;
                font-weight: bold;

}
```

6. Last is the HTML page that contains very little detail. It simply has links to the script and the CSS and a single placeholder `<span>` tag into which you write the menu information. Save this as `default.htm` in the `Chapter1` folder.

```
<html>
<head>
    <title>Ajax First Example</title>
      <link rel="stylesheet" type="text/css" href="SuiteListStyles.css" />
       <script type="text/javascript" src="ajax.js"></script>
</head>
<body onload="GetDocument()">
    <span id="menuhere"></span>
</body>
</html>
```

Note that in the code download for Chapter 1, there are some extra pages for each of the suites. These are for presentation only and are not needed for the example.

7. Open your browser and view the `default.htm` page. Ensure that you do this via your web server. For example, the following typed into the browser Address bar would be correct:

```
http://localhost/BegAjax/Chapter1/default.htm
```

If you were to browse via Windows Explorer directly to the file and click on it, then you would see the following and this would not work:

```
C:\Inetpub\wwwroot\BegAjax\Chapter1\default.htm
```

Just as with server-side technologies, the `XMLHttpRequest` object requires the server to work correctly. If it works correctly, you will see a page with five menu options on the left-hand side. Four of these options are inactive. The active option is Hotel Rome, and if you hover the mouse over it, you will see a blue menu appear (Figure 1-8).

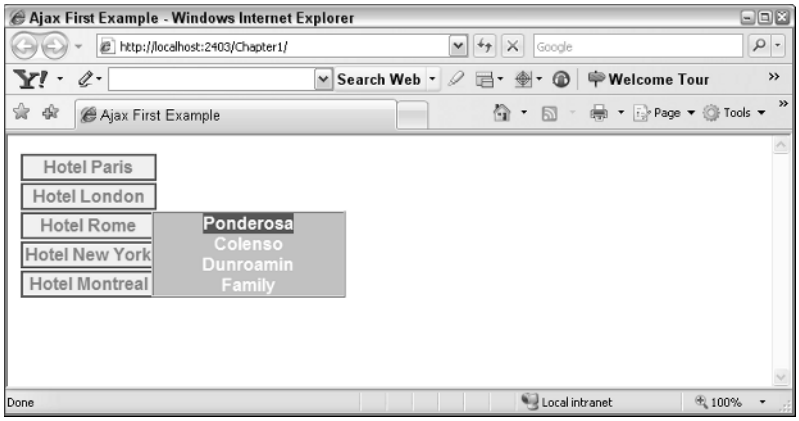

Figure 1-8: Hovering over the active option of Hotel Rome.

8. Now, this is the neat bit. Say that the XML document that you used has just been updated to reflect the fact that you have no availability on the Ponderosa suite. Go back to the `SuiteList.xml` file, and alter the weeks free to zero:

```
...
  <Suite>
    <SuiteID>1001</SuiteID>
    <Name>Ponderosa</Name>
    <Size>2</Size>
    <Price>50</Price>
    <WeeksFree>0</WeeksFree>
  </Suite>
  ...
```

9. Go back to the page, making sure you do not refresh it. Wait for about 10–15 seconds. Hover over Hotel Rome with the cursor once more, as shown in Figure 1-9.

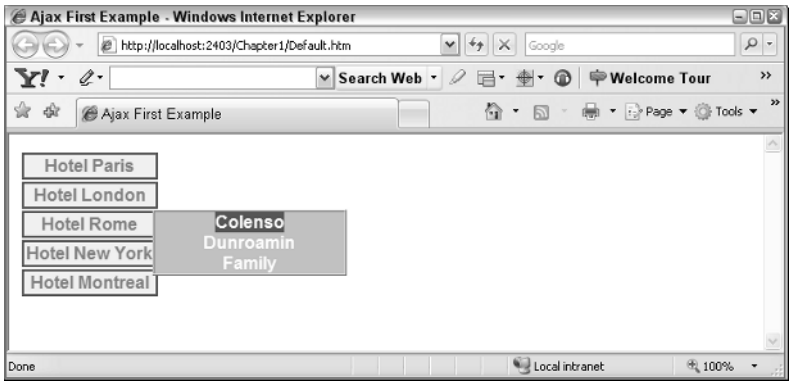

Figure 1-9: New results of hovering over Hotel Rome.

10. The top entry has disappeared.

How It Works

As mentioned previously, this discussion does not go into detail about this example because the techniques will be covered in later chapters. Instead, let's look at what the application is doing and how the different parts fit together. We've created a simple menu system in an XSL template. This consists of a table with five rows. The XSL template takes its data from the XML document `SuiteList.xml`. The middle row of the XSL template contains a JavaScript `mouseover` event, which turns on the visibility of the submenu. The XSL template contains a subtemplate, which is used to display the submenu:

```
  <xsl:template name="DynamicSubMenu">
    <div id="romesubmenu" class="submenublock" onmouseover="var submenu =
document.getElementById('romesubmenu');submenu.style.visibility = 'visible';return
true;">
      <xsl:for-each select="//Suite">
        <xsl:if test="WeeksFree&gt;0">
          <div>
            <a href="{Name}.htm">
              <xsl:value-of select="Name"/>
            </a>
          </div>
          <br/>
        </xsl:if>
      </xsl:for-each>
    </div>
  </xsl:template>
```

XSL statements are used in the template to iterate through the `SuiteList.xml` document. The `xsl:for-each` statement locates each element called `<Suite>` in the XML document. There are four elements in all:

```
...
    <Suite>
      <SuiteID>1001</SuiteID>
      <Name>Ponderosa</Name>
      <Size>2</Size>
      <Price>50</Price>
      <WeeksFree>0</WeeksFree>
    </Suite>
    ...
```

Then, the `<xsl:if>` statement is used to conditionally display the submenu item. You check the value of the `<WeeksFree>` element, and you say that, if this element is greater than zero, then you will display the `<span>` and `<a>` element's contained within. If not, you won't. This has the effect of showing only the suites on the submenu that have a `WeeksFree` value greater than zero, which corresponds with the requirement of the businesses that wish to display only the suites with available weeks.

What is happening behind the scenes is of most interest. Instead of waiting to go back to the server for a refresh, you are actually getting the code to reload the XML document every 10 seconds. This is all controlled in the JavaScript on the page. In four swift steps, you create an `XMLHttpRequest` object, you set the event to notify you when the XML document has been loaded, you get the XML document, and you send it. The creation of the `XMLHttpRequest` object is performed at the head of the code. It is browser independent, but you can add some code to make sure that this works on IE 6 as well. You create it at the head because you want the object to be accessible to all functions in your script.

```
// Create the XMLHttpRequest
var xHRObject = false;
if (window.XMLHttpRequest)
{
    xHRObject = new XMLHttpRequest();
}
else if (window.ActiveXObject)
{
    xHRObject = new ActiveXObject("Microsoft.XMLHTTP");
}
```

The other three steps are contained in the `getDocument()` function, which is loaded on startup, via the body element's onload event handler:

```
//Reset the function
     xHRObject.onreadystatechange = getData;

     //IE will cache the GET request; the only way around this is to append a
     //different querystring. We add a new date and append it as a querystring
     xHRObject.open("GET", "SuiteList.xml?id=" + Number(new Date), true);

     xHRObject.send(null);
```

When the `readystatechange` event is fired, it calls a function `getData`. `getData` checks to see whether the response (the XML) has been returned from the server and acts only if it has. It then performs three steps to load the XML document with the data, load the XSL style sheet (which controls which parts of the data are displayed on the menu), and then performs a transform on the XML document — meaning that the two are combined to produce an HTML document fragment.

```
if (window.ActiveXObject)
    {
        //Load XSL
        var xsl = new ActiveXObject("Microsoft.XMLDOM");
        xsl.async = false;
        xsl.load("MenuDisplay.xsl");

        //Transform
        var transform = xmlDoc.transformNode(xsl);
        var spanb = document.getElementById("menuhere");

    }

    else
    {
      var xsltProcessor = new XSLTProcessor();

      //Load XSL
      XObject = new XMLHttpRequest();
      XObject.open("GET", "MenuDisplay.xsl", false);
      XObject.send(null);

      xslStylesheet = XObject.responseXML;
      xsltProcessor.importStylesheet(xslStylesheet);

      //Transform
```

```
    var fragment = xsltProcessor.transformToFragment(xmlDoc, document);

    document.getElementById("menuhere").innerHTML = "";
   document.getElementById("menuhere").appendChild(fragment);
  }
```

XSLT is browser-specific, so you have one branch for IE and one branch for Firefox, but they both do the same thing. They store the XML response, they load an XSL style sheet, and they perform a transform on it. The fragment is then inserted into the page at the `<span>` element with the ID `"menuhere"`. When a change has been made to the document, you are no longer dependent on a page refresh. The XML document is being loaded and then placed into the DOM using the `innerHTML` property.

```
spanb.innerHTML = transform;
```

Last, you clear the `XmlHttpRequest` object with the `abort` method and then use the `setTimeout` to call another method `getDocument()`. `getDocument` performs the same processing on the `XmlHttpRequest` object, as you did at the beginning of the script, and it essentially sets up a cycle that loads the XML document every 10 seconds.

```
//Clear the object and call the getDocument function in 10 seconds
xHRObject.abort();
setTimeout("getDocument()", 10000);
```

Reloading a page every 10 seconds is definitely not the most efficient way of populating a dynamic menu system. You might wish to consider some alternatives. ASP.NET has a `Cache` object that allows you to create a dependency so that when the XML file is updated, the contents of the cache are automatically refreshed. You could shoehorn this into the application here, so that the menu updates every time the file is updated ; it is reloaded into the `XMLHttpRequest`. Currently, as it stands, if you were loading a large XML file every 10 seconds, you might end with the unresponsiveness you have been seeking to avoid. To demonstrate how this kind of updating works, though, it more than serves the purpose.

You have a flow — the data is contained in the XML document, and you use an XSL style sheet to interpret the document. You use the DOM and `XMLHttpRequest` to load the combined XML/XSL document into your HTML page every 10 seconds. You use the CSS to make the output look a bit more presentable.

If you feel this has been a bit fast, don't worry. Each individual aspect is examined in a later chapter. For now, you should see that you can have a model where you update a portion of the web page without ever having to refresh the page -- and without any delay. It's also worth noting that this is just one "view" of how Ajax might be used. Plenty of other techniques that use Ajax in radically different ways are explored throughout the book.

Summary

This chapter considered what Ajax is and some of the reasons you may have for using Ajax techniques. The chapter examined the original article by Jesse James Garrett and the different components that he described as going into an Ajax application. Because there has been a lot of contention over what is and what isn't an Ajax application, this chapter explained that not everything in the article has to be found in

an Ajax application. The discussion looked at why indiscriminate use of Ajax could have the reverse effect to the desired one. Examples showed some good Ajax applications in action, and the chapter discussed their techniques. Last, you created your own Ajax application that used a lot of the technologies to be discussed and used in this book.

Exercises

Suggested solutions to these questions can be found in Appendix A.

1. What are the defining points that will make a web application into an Ajax application?
2. Why might Ajax be considered a misleading acronym?