CHAPTER 2

# Introducing Ajax

A little more than a year ago, an article by Jesse James Garrett was published describing an advanced web development technique that, even though individual components of it have existed for years, few web developers had ever stumbled across. I can guess the reason for this lack of knowledge; basically, in the last few years, the need to produce measurable results has gotten in the way of the need to practice our craft. Or, as a former manager of mine would say, it's "that mad scientist stuff," except, as I recall, he used another word in place of *stuff*. Unfortunately, nine times out of ten, the need to produce measurable results gets in the way of "that mad scientist stuff."

However, it's the tenth time that's important. The article didn't stop at just describing the technique; it went on to say that Google used the very same technique. Invoking that single name, Google, was enough to change a point of view. Quicker than you could say, "Igor, the kites!" the phrase "that mad scientist stuff" morphed into "Why aren't we doing it this way?" The reason for this change of perception is that the name Google made this a technique that could produce measurable results. All it took was that single name, Google, to make using the `XMLHttpRequest` object so that the browser could communicate with the server without the page ever unloading and reloading into an acceptable practice.

This chapter introduces you to that practice, the practice of updating web pages with information from the server. Beyond the `XMLHttpRequest` object, which has been around for several years as a solution looking for a problem, there is nothing weird needed. Basically, it is how the individual pieces are put together. When they're put together in one way, it is nothing more than a pile of parts; however, when put together in another way, the monster essentially rises from its slab.

## 2.1 Not a Mockup

A few years ago, I demonstrated an application that did what I just described. The demo ran for more than 2 hours with the same questions repeated over and over.

"It's a mockup, right?"

"No, it is the actual application."

"It can't be. The screen doesn't blink."

"That's because XML, HTTP, and SOAP are used to get the data directly from the server. JavaScript then updates only the parts of the page that have changed."

"It's a mockup, right?"

And so on. It took the client more than 2 hours to realize that the database was actually being updated without the page "blinking," as he referred to it.

## 2.2 A Technique Without a Name

Now, if I had been smart, I would have given the technology a name then and there, and thus ensured my place in Web history, shutting up the client as well. After all, a name is a thing of power, and the client, not wanting to sound stupid for not knowing what the acronym meant, would have saved more than 2 hours of my life that were spent re-enacting the scene of peasants with pitch forks from the 1931 version of *Frankenstein,* minus the tongs. Unfortunately, I drew an absolute blank and just called it as it was.

With apologies to the people who make the cleanser and the detergent, legend has it that the original Ajax was the second most powerful of the Greek warriors at Troy. Even though he had some issues (who in the *Illiad* didn't?), his strength and skill in battle were second to none (well, okay, second only to Achilles). In naming the technology Ajax, Jesse James Garrett gave the technology both Ajax's strengths and issues.

### 2.2.1 Names

An old idea dates back to the dawn of human civilization that to know someone's or something's true name is to have power over that person or thing. It is one of the basic concepts of what is commonly referred to as magic, and although magic isn't real, the idea that names can hold power isn't very far from the truth. Consider, if you will, a resumé. If ever a document held names of power, a resumé is it. Not very long ago, resumés invoking words such as *JavaScript, DHTML,* and *XML* were looked upon with envy, perhaps even

awe. After all, for a little while, it seemed as though web developers were rock stars that, thankfully, were never asked to sing. Unfortunately, those names are now considered passé or even a little old-fashioned.

In his essay describing this web development technique, Mr. Garrett did one final thing; he gave it a name, Ajax, and thus gave us power over it. The acronym refers to Asynchronous JavaScript And XML, and whether you love or hate the name, the technology now has a name. At the very least, this naming means that we can describe what we've been doing at work. Ajax is a lot easier to say than, "I've been using client-side JavaScript, SOAP, and XML to obtain data directly from the server using XMLHTTP instead of the standard unload/reload cycle."

## 2.3 What Is Ajax?

As stated previously, Ajax stands for Asynchronous JavaScript And XML, but what exactly does that mean? Is the developer limited to only those technologies named? Thankfully, no, the acronym merely serves as a guideline and not a rule. In some ways, Ajax is something of an art, as with cooking. Consider, for a moment, the dish called shrimp scampi; I've had it in restaurants up and down the East Coast of the United States, and it was different in every restaurant. Of course, there were some common elements, such as shrimp, butter, and garlic, but the plethora of little extras added made each dish unique.

The same can be said of Ajax. Starting with a few simple ingredients, such as HTML and JavaScript, it is possible to cook up a web application with the feel of a Windows or, if you prefer, a Linux application. You might have noticed earlier that my ingredients list omitted XML; the reason for that omission is that XML is one of those optional ingredients. This might sound strange because the *x* in *Ajax* stands for XML, but it is also useful in those instances when a particular client does not support XML or doesn't support some of the more "mad scientist" methods of communicating with the server.

### 2.3.1 The Ajax Philosophy

How the client—in this case, a web browser—communicates with the server is one of the cornerstones of Ajax. Designed with the philosophy of not using bandwidth just because it's there, a web page coded using these techniques won't go through the unload/reload cycle, or "blink," as some refer to it, unless absolutely necessary. Why send 100,000 bytes back and forth to the server when 300 bytes will suffice?

Of course, this means that, to the casual observer, the browser is behaving strangely because sometimes only selected parts of a web page are updated.

This means that the page won't "blink," as the peasant—er, client—so elegantly put it. Instead, in a wink of an eye, parts of the page will update quicker than they believed possible. The speed difference can be compared to the difference between accessing a file on a floppy disk and accessing a file on the hard disk. Personally, my reaction was along the lines of "I am never going back!" But individual results can vary, so consult your doctor.

Another concept that Ajax uses is, why not make the client work for a living? Have the client's web browser handle parts of the processing rather than just parrot preprocessed information on the screen. The initial page load would consist of data and JavaScript, instructions on what to do with the data. To expand upon the earlier mad scientist analogy, imagine a do-it-yourself "mad scientist" kit consisting of a pile of parts and a minion that answers to Igor, and you'll get the idea.

With an Ajax application, the browser is expected to actually process the data supplied by the server. This means not only the little things that DHTML did, such as rollovers and hierarchical drop-down navigation menus, but real things, such as posting to the server and handling the response, whether it is handling it either synchronously or asynchronously. In addition, Ajax applications need to be able to not only find objects on the HTML page but also, if necessary, update them.

This leads to the question of how, short of the whole kites and Igor methodology, does one accomplish this unholy task? The answer is that it depends on just how and how far one wants to pursue this course. There are three ways to bring life to an Ajax application, and each has its own advantages and disadvantages. It all depends on just which parts of the Ajax toolset the developers are comfortable with. It also depends on how comfortable you are with excluding certain members of the planet from the application. Yes, I'm talking about those people who are still running Internet Explorer version 2.0. Fortunately, it isn't my job to issue decrees concerning browser compatibility; however, it is my job to cover how to implement an Ajax application.

### 2.3.2 Meddling with Unnatural Forces

Earlier I explained how I, and probably quite a few others, stumbled upon the then nameless technique that was to become Ajax. However, that was not my first brush with what my supervisor called "mad scientist stuff." Several years earlier, as a consultant for the group insurance division of a large insurance company, I had the good fortune to get the assignment to automate a paper-based request system.

Armed with a file layout, salespeople would try to sell group insurance to companies and, theoretically, would explain that enrollee information needed to conform to the file layout. However, possibly in an effort to make the sale and thereby get the commission, they would accept it in any conceivable electronic format. XML, Excel, or flat files—it was all the same to them because they would fill out a multipage form and the minions in systems would take care of it. Needless to say, quite a few of these pieces of paper got lost, got coffee spilled on them, or simply got filed under "it's real work and I don't want to do it" by the folks in systems.

Arriving onsite, I quickly got to work researching the various forms and how they were handled, which led to documenting how the process should work. Because I was the sole designer and developer for this new system, there was, shall I say, some freedom as to the technologies at my disposal. The back end was classic ASP and SQL Server, both of which are beyond the scope of this book. The front end, however, was a combination of HTML, JavaScript, and DOM, with a little CSS thrown in for good measure.

Here's how it worked: The user would enter multiple pages of information concerning the request. This information would be cached on the client side until the user reached the end of the chain of pages and clicked the final submit button. The caching was accomplished through the use of HTML frames; the first frame, as the user input frame, filled the entire browser's window. However, the second frame, the data frame, was the interesting one because it wasn't visible even though it was always there.

This trick, for lack of a better word, with hidden frames was that they had the advantage of speeding up the application. The speeding up was due to reduced interaction with both the web server and the database server. Another benefit was that, in addition to the performance improvements, the application seemed to flow better because the input was broken into convenient chunks instead of the usual approach of entering between 80 and 200 items at one time.

## 2.4 An Ajax Encounter of the First Kind

Now that I've gushed about the *why* of this technique, let me offer a little insight on the *how* of this technique. Let's start with the three HTML documents shown in Listing 2-1, Listing 2-2, and Listing 2-3. Some readers might not consider this a true example of Ajax, but it does share many of the same qualities of Ajax, in much the same way that a *Star Trek* fan and a *Star Wars* fan share many of the same qualities.

**Listing 2-1** HTMLfs.htm

```
<html>
  <head>
    <title>HTMLfs</title>
  </head>
  <frameset rows="100%,*">
    <frame name="visible_frame" src="visible.htm">
    <frame name="hidden_frame" src="hidden.htm">
    <noframes>Frames are required to use this Web site.</noframes>
  </frameset>
</html>
```

**Listing 2-2** visible.htm

```
<html>
  <head>
    <title>visible</title>
    <script language="javascript">
/*
    Perform page initialization.
*/
function initialize() { }

/*
    Handle form visible form onchange events.  Values from the visible
    form are copied to the hidden form.
*/
function changeEvent(obj)
{
  parent.frames[1].document.getElementById(obj.id).value = obj.value;
}

/*
    Submits the form in the hidden frame then reloads the hidden frame.
*/
function submitForm() {
  parent.frames[1].document.getElementById('hidden_form').submit();
  parent.frames[1].document.location = "hidden.htm";
}
    </script>
  </head>
    <body onload="initialize()">
      <form name="visible_form" id="visible_form"></form>
    </body>
</html>
```

**Listing 2-3** hidden.htm

```
<html>
  <head>
    <title>hidden</title>
    <script language="javascript">
var reBrowser = new RegExp('internet explorer','gi');

/*
    Perform page initialization, waits for the visible frame to load and
clones the hidden form to the visible form.
*/
function initialize()
{
  var hiddenForm = document.getElementById('hidden_form');

  if(reBrowser.test(navigator.appName))
  {
    while(parent.document.frames.item(0).document.readyState !=
'complete') { }

    parent.frames[0].document.getElementById('visible_form').innerHTML =
hiddenForm.innerHTML;
  }
  else
  {
    var complete = false;

    while(!complete)
    {
      try
      {

parent.frames[0].document.getElementById('visible_form').appendChild
(hiddenForm.cloneNode(true));

        complete = true;
      }
      catch(e) { }
    }
  }

}
    </script>
  </head>
  <body onload="initialize()">
    <form name="hidden_form" id="hidden_form" action="post.aspx">
      <h1>Address Information</h1>
      <table border="0" width="100%">
        <tr>
          <th width="30%" align="right">Name: </th>
          <td align="left">
```

*continues*

**Listing 2-3** continued

```
            <input type="text" name="name" id="name" value=""
onchange="changeEvent(this)">
          </td>
        </tr>
        <tr>
          <th align="right">Address Line 1: </th>
          <td align="left">
            <input type="text" name="address1" id="address1" value=""
onchange="changeEvent(this)">
          </td>
        </tr>
        <tr>
          <th align="right">Address Line 2: </th>
          <td align="left">
            <input type="text" name="address2" id="address2" value=""
onchange="changeEvent(this)">
          </td>
        </tr>
        <tr>
          <th align="right">City: </th>
          <td align="left">
            <input type="text" name="city" id="city" value=""
onchange="changeEvent(this)">
          </td>
        </tr>
        <tr>
          <th align="right">State: </th>
          <td align="left">
            <input type="text" name="state" id="state" value=""
onchange="changeEvent(this)">
          </td>
        </tr>
        <tr>
          <th align="right">Zip Code: </th>
          <td align="left">
            <input type="text" name="zip" id="zip" value=""
onchange="changeEvent(this)">
          </td>
        </tr>
      </table>
      <br>
      <input type="button" value="Submit" onclick="submitForm()">
    </form>
  </body>
</html>
```

### 2.4.1 A World Unseen

Any developer familiar with the use of frames and framesets will find Listing 2-1 pretty normal looking. However, one item isn't plain vanilla:
the `rows="100%,*"` attribute on the frameset element, which states that the first frame gets 100 percent of available rows. The asterisk (`*`) states that anything left over goes to the second frame. In this example, there is nothing left over, so it is the equivalent of coding zero. This results in the first frame being visible and the second frame being hidden. In essence, this is a sneaky way to hide what's going on from prying eyes—namely, the user. The next two listings are the visible frame, Listing 2-2, and the hidden frame, Listing 2-3. Listing 2-3 is where the real mad science happens.

### 2.4.2 Enter JavaScript

Listing 2-2 is short and sweet, basically two short JavaScript functions that don't appear to do anything. The first of these functions, `changeEvent`, is just what it says it is, a handler for an `on change` event. When fired, it copies the value associated with the current object on the current frame to one with the same ID on the hidden frame. The second function, `submitForm`, submits a form; however, like the previous function, it works with the hidden frame by locating and submitting the form there.

This leaves just one question: Where does the HTML for the visible form come from? The answer lies in Listing 2-3, the one for the hidden frame. Like the visible frame, it has JavaScript functions and a form. There is, however, a major difference in the form. Unlike its visible counterpart, it has all of the HTML necessary to make a nice little form. The trick is getting it from the hidden frame to the visible frame.

This magic is accomplished in the pages' `on load` event handler, `initialize`. This function waits for the other frame to load and then copies this form's inner HTML to the other frame. When this is done, the result is the normal-looking web page shown in Figure 2-1. The way it behaves, however, is almost application-like, with parts of the visible page being updated each time the hidden frame does an unload/reload cycle.

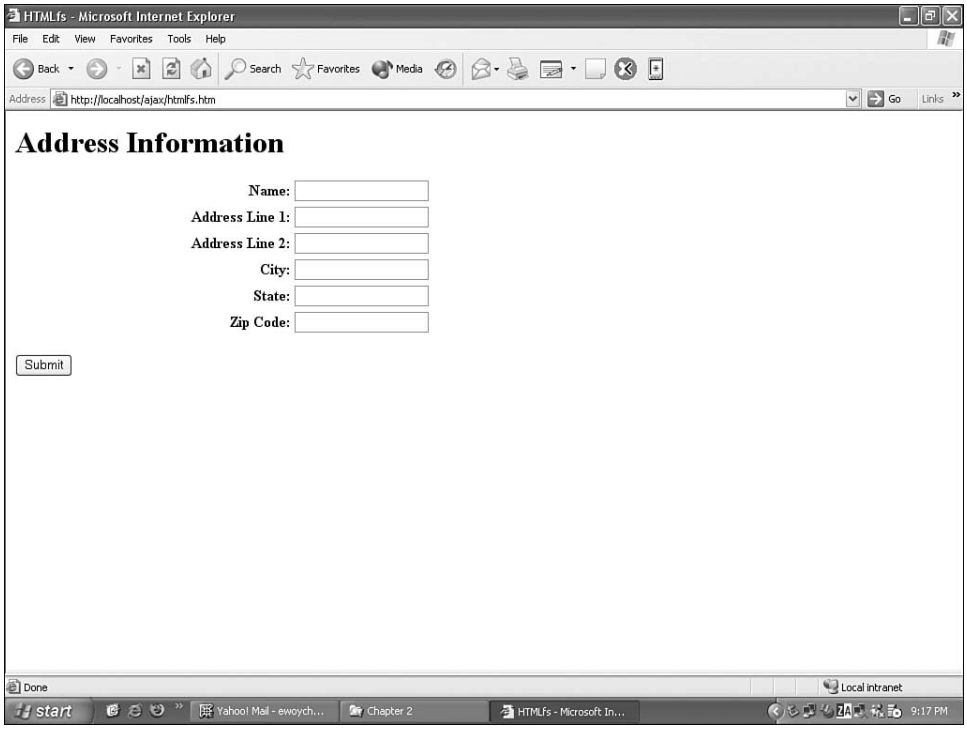


**Figure 2-1** A normal-looking web page that functions almost like a desktop application

## 2.5 An Ajax Encounter of the Second Kind

As flexible and cross-browser capable as the "hidden frames" method of implementing Ajax is, all that has been accomplished is the "AJ" part of Ajax. Which is sort of like the sound of one hand clapping, and that usually means that Igor has been slacking off again. Thankfully, there's another part—eh, make that technology—available: XML. The problem with XML is that it has developed a reputation of being difficult; however, it doesn't have to be. Just keep in mind that, in those situations, code has a tendency to follow you around, like Igor.

### 2.5.1 XML

In its simplest form, XML is nothing more than a text file containing a single well-formed XML document. Come to think of it, the same is pretty much true

in its most complex form as well. Looking past all of the hype surrounding XML, it is easy to see that XML is merely the text representation of self-describing data in a tree data structure. When this is understood, all that is left are the nitty-gritty little details, like "What's a tree data structure?" and "How exactly does data describe itself?"

A tree data structure is built of nodes, with each node having only one node connected above it, called a *parent* node. The sole exception to this rule is the *root* node, which has no parent node. Nodes can also have other nodes connected below, and these are called *child* nodes. In addition, nodes on the same level that have the same parent node are called children. Figure 2-2 is a graphical representation of a tree data structure.

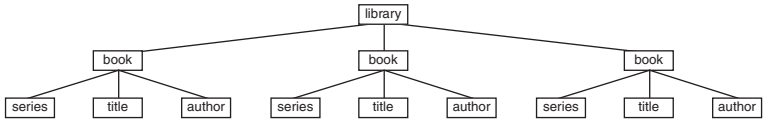


**Figure 2-2** Tree data structure

Figure 2-2 can also be represented as the XML document shown in Listing 2-4.

**Listing 2-4** XML Representation of the Same Information as in Figure 2-2

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<library>
  <book>
    <series>The Wonderland Gambit</series>
    <title>The Cybernetic Walrus</title>
    <author>Jack L. Chalker</author>
  </book>
  <book>
    <series>The Wonderland Gambit</series>
    <title>The March Hare Network</title>
    <author>Jack L. Chalker</author>
  </book>
  <book>
    <series>The Wonderland Gambit</series>
    <title>The Hot-Wired Dodo</title>
    <author>Jack L. Chalker</author>
  </book>
</library>
```

The nodes shown in Listing 2-4 are called *elements,* which closely resemble HTML tags. And like HTML tags, start tags begin with < while end tags

begin with </. However, unlike HTML tags, all XML tags either must have a closing tag or be self-closing or must be empty elements. Self-closing tags are recognizable by the ending />; if the forward slash was omitted, the document would not be a *well-formed* XML document. In addition, to all elements being either closed or self-closing, the tags must always match up in order. This means that the XML document in Listing 2-5 is well formed but the XML document in Listing 2-6 is not well formed. In a nutshell, "well formed" means that there is a right place for everything. Feet are a good example of this: Imagine if Igor used two left feet; the monster wouldn't be well formed and wouldn't be able to dance, either.

**Listing 2-5** A Well-Formed XML Document

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<one>
  <two>
    <three>
      <four/>
    </three>
  </two>
</one>
```

**Listing 2-6** An XML Document That Is Not Well Formed

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<one>
  <two>
    <three>
      <four/>
  </two>
    </three>
</one>
```

As neat and nifty as the hidden frames method of communicating with the server is, the addition of an XML document provides another option, XMLHTTP, or, as some refer to it the `XMLHttpRequest` object. Note all those capital letters, which are meant to indicate that it is important. The `XMLHttpRequest` object sends information to and retrieves information from the server. Although it doesn't have to be, this information is usually in the form of XML and, therefore, has the advantage of being more compact than the usual HTML that the server sends. Just in case you're interested, this was the means of communication for that page that I had handwritten and was using during the "it doesn't blink" fiasco.

### 2.5.2 The XMLHttpRequest Object

Unlike the hidden frames approach, in which the unload/reload cycle is still there but is tucked out of the way, using the `XMLHttpRequest` object means finally saying good-bye to the unload/reload cycle that we've all come to know and loathe. This means that, in theory, if not in practice, a single page could conceivably be an entire website. Basically, it's a load-and-go arrangement.

In theory, the original page loads and a user enters information into a form and clicks submit. A JavaScript event handler sends the user's information to the server via XMLHTTP and either waits penitently for a response (synchronous) or sets an event handler for the response (asynchronous). When the response is received, the JavaScript takes whatever action that it is programmed to, including updating parts of the page, hence the lack of an unload/reload cycle or "blink." This is great theory, but a theory is pretty useless if it cannot be put into practice; let's take a look in Listings 2-7 and 2-8 at how this can be implemented from a client-side perspective.

**Listing 2-7** Example Ajax Web Page

```
<html>
  <head>
    <title>AJAX Internet Explorer Flavor</title>
    <script language="javascript">
var dom = new ActiveXObject('MSXML2.FreeThreadedDOMDocument.3.0');
var objXMLHTTP = new ActiveXObject('Microsoft.XMLHTTP');

/*
    Obtain the XML document from the web server.
*/
function initialize()
{
  var strURL = 'msas.asmx/getTime';

  objXMLHTTP.open('POST',strURL,true);
  objXMLHTTP.onreadystatechange = stateChangeHandler;

  try
  {
    objXMLHTTP.send();
  }
  catch(e)
  {
  alert(e.description);
  }
}

/*
    Handle server response to XMLHTTP requests.
*/
function stateChangeHandler()
```

*continues*

**Listing 2-7** continued

```
{
  if(objXMLHTTP.readyState == 4)
    try
    {
      dom.loadXML(objXMLHTTP.responseText);
      document.getElementById('time').innerText =
dom.selectSingleNode('time').text;
    }
  catch(e) { }
}
    </script>
  </head>
  <body onload="initialize()">
    <div id="time"></div>
  </body>
</html>
```

**Listing 2-8** XML Document

```
<?xml version="1.0" encoding="utf-8" ?>
<time>3:30 PM</time>
```

If this were *CSI, Columbo* or *The Thin Man,* now is the time when the hero explains how the deed was done. It goes something like this: The HTML page loads, which causes the `onload` event handler, `initialize`, to fire. In this function, the `XMLHttpRequest` object's `open` method is invoked, which only sets the method (`POST`), gives the relative URL of a web service, and states that the request will be asynchronous (`true`). Next, the `onreadystatechage` event handler is set; this is the function that handles what to do when the web service responds. Finally, the `send` method of the `XMLHttpRequest` object is invoked, sending our request on its merry way.

When a response is received from the web service, the `stateChangeHandler` is fired. You've probably noticed the test of the `readyState` property. The reason for this is that there are more than one possible `readyState` values, and we're interested in only four, complete. When the response is complete, the result is loaded into an XML document, the appropriate node is selected, and the HTML is updated.

Listings 2-7 and 2-8 could be considered by some a pure example of Ajax. Unfortunately, the way it is currently coded, browsers other than Microsoft Internet Explorer would have real issues with it. What sort of issues? The code simply won't work because of differences in how XML and the `XMLHttpRequest` object work in various browsers. This doesn't mean that this form of Ajax is an

IE-only technology; it simply means that careful planning is required to ensure cross-browser compatibility.

On the subject of compatibility, I don't want to scare you off, but let me point out that the more advanced the client-side coding is, the more likely it is that there will be issues. The majority of these issues are merely little annoyances, similar to flies buzzing around. These "flies" aren't fatal, but it is a good idea to keep these things in mind.

## 2.6 An Ajax Encounter of the Third Kind

The fifth part of Ajax, an optional part, isn't for the faint of heart. It transcends the "mad scientist stuff" into the realm of the magical, and it is called eXtensible Stylesheet Language for Transformations, or XSLT. In other words, if Ajax really was mad science and it was taught in school, this would be a 400-level course. Why? The reason is that the technology is both relatively new and very, very browser dependent. However, when it works, this method provides an incredible experience for the user.

### 2.6.1 XSLT

XSLT is an XML-based language that is used to transform XML into other forms. XSLT applies a style sheet (XSLT) as input for an XML document and produces output—in most cases, XHTML or some other form of XML. This XHTML is then displayed on the browser, literally in the "wink of an eye."

One of the interesting things about XSLT is that, other than the XML being well formed, it really doesn't make any difference where the XML came from. This leads to some interesting possible sources of XML. For example, as you are probably aware, a database query can return XML. But did you know that an Excel spreadsheet can be saved as XML? XSLT can be used to transform any XML-derived language, regardless of the source.

Listing 2-9 shows a simple Internet Explorer–only web page along the same lines as the earlier examples. By using XSLT and the `XMLHttpRequest` object to retrieve both the XML and XSLT shown in Listing 2-10, it is extremely flexible. This is because after the initial page is loaded, any conceivable page can be generated simply by changing the XML and/or the XSLT. Sounds pretty powerful, doesn't it?

**Listing 2-9** A Simple IE-Only Web Page

```
<html>
  <head>
    <title>AJAX Internet Explorer Flavor</title>
```

*continues*

**Listing 2-9** continued

```
    <script language="javascript">
var dom = new ActiveXObject('MSXML2.FreeThreadedDOMDocument.3.0');
var xslt = new ActiveXObject('MSXML2.FreeThreadedDOMDocument.3.0');
var objXMLHTTP;

/*
    Obtain the initial XML document from the web server.
*/
function initialize()
{
  doPOST(true);
}

/*
    Use the XMLHttpRequest to communicate with a web service.
*/
function doPOST(blnState) {
  var strURL = 'http://localhost/AJAX/msas.asmx';

  objXMLHTTP = new ActiveXObject('Microsoft.XMLHTTP');

  objXMLHTTP.open('POST',strURL,true);

  if(blnState)
    objXMLHTTP.setRequestHeader('SOAPAction','http://
tempuri.org/getState');
  else

  objXMLHTTP.setRequestHeader('SOAPAction','http://tempuri.org/getXML');

  objXMLHTTP.setRequestHeader('Content-Type','text/xml');

  objXMLHTTP.onreadystatechange = stateChangeHandler;

  try
  {
    objXMLHTTP.send(buildSOAP(blnState));
  }
  catch(e)
  {
  alert(e.description);
  }
}

/*
    Construct a SOAP envelope.
*/
function buildSOAP(blnState) {
  var strSOAP = '<?xml version="1.0" encoding="UTF-8"?>';
```

```
  strSOAP += '<soap:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">';
  strSOAP += '<soap:Body>';

  if(blnState)
  {
    strSOAP += '<getState xmlns="http://tempuri.org/">';
    strSOAP += '<state_abbreviation/>';
    strSOAP += '</getState>';
  }
  else
  {
    strSOAP += '<getXML xmlns="http://tempuri.org/">';
    strSOAP += '<name>xsl/state.xsl</name>';
    strSOAP += '</getXML>';
  }

  strSOAP += '</soap:Body>';
  strSOAP += '</soap:Envelope>';

  return(strSOAP);
}

/*
    Handle server response to XMLHTTP requests.
*/
function stateChangeHandler()
{
  if(objXMLHTTP.readyState == 4)
    try
    {
      var work = new ActiveXObject('MSXML2.FreeThreadedDOMDocument.3.0');

      work.loadXML(objXMLHTTP.responseText);

      switch(true) {
        case(work.selectNodes('//getStateResponse').length != 0):
          dom.loadXML(objXMLHTTP.responseText);
          doPOST(false);

          break;
        case(work.selectNodes('//getXMLResponse').length != 0):
          var objXSLTemplate = new
ActiveXObject('MSXML2.XSLTemplate.3.0');


xslt.loadXML(work.selectSingleNode('//getXMLResult').firstChild.xml);

          objXSLTemplate.stylesheet = xslt;

          var objXSLTProcessor = objXSLTemplate.createProcessor;

          objXSLTProcessor.input = dom;
```

*continues*

**Listing 2-9** continued

```
          objXSLTProcessor.transform();

          document.getElementById('select').innerHTML =
objXSLTProcessor.output;

          break;
        default:
          alert('error');

          break;
      }
    }
    catch(e) { }
}
    </script>
  </head>
  <body onload="initialize()">
  <div id="select"></div>
</html>
```

**Listing 2-10** The XML and XSLT Part

```
<?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" version="1.0" encoding="UTF-8" indent="yes"/>

  <xsl:template match="/">
    <xsl:element name="select">
      <xsl:attribute name="id">state</xsl:attribute>
      <xsl:attribute name="name">selState</xsl:attribute>
      <xsl:apply-templates select="//Table[country_id = 1]"/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="Table">
    <xsl:element name="option">
      <xsl:attribute name="value"><xsl:value-of
select="state_abbreviation"/></xsl:attribute>
      <xsl:value-of select="state_name"/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

### 2.6.2 Variations on a Theme

At first glance, the JavaScript in the previous example appears to be very similar to that shown in Listing 2-7; however, nothing could be further from the

truth. The first of these differences is due to two calls being made to a web service and the use of XSLT to generate the HTML to be displayed in the browser. Let's look at this in a little more detail.

First, the only thing that the `initialize` function does is call another function, `doPOST`, passing a `true`. Examining `doPOST` reveals that the purpose of the `true` is to indicate what the `SOAPAction` in the request header is, `http://tempuri.org/getState` to get information pertaining to states and provinces from the web service, or `http://tempuri.org/getXML` to get XML/XSLT from the web service. The first time through, however, we're getting the XML.

The second difference, also in `doPOST`, is the addition of a call to `buildSOAP` right smack in the middle of the `XMLHttpRequest` object's `send`. This is how arguments are passed to a web service, in the form of text—a SOAP request, in this instance. Checking out `buildSOAP`, you'll notice that `Boolean` from `doPOST` is passed to indicate what the body of the SOAP request should be. Basically, this is what information is needed from the web service, states or XSLT.

You'll remember the `stateChangeHandler` from the earlier set of examples, and although it is similar, there are a few differences. The first thing that jumps out is the addition of a "work" XML document that is loaded and then used to test for specific nodes; `getStateResponse` and `getXMLResponse`. The first indicates that the SOAP response is from a request made to the web service's `getState` method, and the second indicates a response from the `getXML` method. Also notice the `doPOST` with an argument of `false` in the part of the function that handles `getState` responses; its purpose is to get the XSLT for the XSL transformation.

Speaking of a transformation, that is the purpose of the code that you might not recognize in the `getXML` portion of the `stateChangeHandler` function. Allow me to point out the `selectSingleNode` method used, the purpose of which is to remove the SOAP from the XSLT. The reason for this is that the XSLT simply won't work when wrapped in a SOAP response. The final lines of JavaScript perform the transformation and insert the result into the page's HTML.

The use of XSLT to generate the HTML "on the fly" offers some interesting possibilities that the other two methods of implementing Ajax do not. For instance, where in the earlier example the look of the page was dictated by the hard-coded HTML, this doesn't have to be the case when using XSLT. Consider for a moment the possibility of a page using multiple style sheets to change the look and feel of a page. Also, with the speed of XSLT, this change would occur at Windows application speeds instead of the usual crawl that web applications proceed at.

## 2.7 The Shape of Things to Come

The sole purpose of this chapter is to offer a glimpse of the shape of things to come, both in this book and in the industry. All joking aside, this glimpse wasn't the result of mad science or any other dark art. It is the result of several years of beating various web browsers into submission, consistently pushing a little further to create rich application interfaces with consistent behavior.

The wide range of technologies that comprise Ajax can be a double-edged sword. On one hand, there is extreme flexibility in the tools available to the developer. On the other hand, currently Ajax applications are often sewn together in much the same way that DHTML pages were in the late 1990s. Unfortunately, although the hand-crafted approach works for furniture and monsters, it relies heavily on the skill level of Igor—eh, the developer.

In future chapters, it is my intention to elaborate on the various techniques that were briefly touched upon in this chapter. Also, even though Ajax is currently considered a technique that takes longer to develop than the "traditional" methods of web development, I'll show some ideas on how to reduce this time. After all, what self-respecting mad scientist cobbles together each and every monster by hand? It's all about tools to make tools—eh, I mean monsters.

## 2.8 Summary

This chapter started with a brief introduction to Ajax that included some of the origins and problems associated with using "mad scientist stuff," such as the accusations of attempting to pass off a mock-up as an actual application and the inability to describe just how something works. Of course, some people still will think Corinthian helmets and hoplites at the very mention of Ajax, but you can't please everyone.

Next there was a brief outline of the philosophy behind Ajax, which centers on the idea of not bothering the server any more than is necessary. The goal is that of reducing, if not eliminating, the unload/reload cycle—or "blink," as some call it. The Ajax philosophy also includes the idea of making the client's computer work for a living. After all, personal computers have been around in some form for close to 30 years; they should do some work—take out the trash, mow the lawn, or something.

Finally, I presented the three simple examples of how Ajax can be implemented. The first example, although not quite Ajax, does much to show something of the first attempts to implement a web application with the feel of a Windows application. Although it's primitive by today's standard, it is still better than 99 percent of the web pages out there today.

Using the `XMLHttpRequest` object, the second example is dead on as to what is expected from an Ajax application. Broken are the bonds that limit updates to the unload/reload cycle that has been confronting us on the Web since Day 1. In addition, XML plays well with the concept of reducing traffic.

The third and final example pushes Ajax to the current limits with the addition of XSLT to the mix. XSLT allows XML to be twisted and stretched into any conceivable shape that we can imagine. No longer are our creations limited to the parts that we can dig up here and there; we can make our own parts on demand.