3 CHAPTER

XMLHttpRequest Object

The techniques discussed in the previous chapter use common JavaScript and XHTML features, often in ways and for purposes other than those for which they were intended. As such, these communication approaches generally lack necessary features for building a robust Ajax-style application. Specifically, to build a proper Ajax-style application you will need fine control over communication, including the ability to get and set HTTP headers, read response codes, and deal with different kinds of server produced content. JavaScript's `XMLHttpRequest` (XHR) object can be used to address nearly all of these problems and thus is at the heart of most Ajax applications. However, there are limitations to XHRs that should be acknowledged, so in this chapter, the aim is to present not only a complete overview of the object's syntax and its use, but an honest discussion of its limitations as well.

Overview of XHRs

At the heart of Ajax is the `XMLHttpRequest` object. A bit misnamed, this object provides generalized HTTP or HTTPS access for client-side scripting and is not limited to just making requests or using XML, as its name would suggest. The facility was first implemented in Internet Explorer 5 for Windows to support the development of Microsoft Outlook Web Access for Exchange 2000, and this object has come to be widely supported in all major desktop browsers. Native implementations can be found in Safari 1.2+, Mozilla 1+, Netscape 7+, Opera 8+, and Internet Explorer 7+. ActiveX-based implementations are found in Internet Explorer 5, 5.5, and 6. Browser support for XHRs is summarized in Table 3-1.

Given the ubiquity of the object, the W3C aims to standardize its syntax (http://www.w3.org/TR/XMLHttpRequest/), though browser variations do exist, as you will see in a moment. Table 3-2 summarizes the common properties and methods for the XHR object.

NOTE *While XML prefixes the name of this object, its only major tie-in with XML is that responses may be parsed as XML via the responseXML property. XML data interchange certainly is not required by XHRs as will be demonstrated in numerous examples.*

Like anything in a Web browser, specific features can be found in XHR objects, as shown in Table 3-3. Why so much "innovation" occurs in Web developers is a matter of debate, with some citing conspiracy and others simple acknowledging that we Web developers are never satisfied with the status quo.

Browser	Native	ActiveX
Mozilla 1+	Yes	No
Netscape 7+	Yes	No
Internet Explorer 5	No	Yes
Internet Explorer 5.5	No	Yes
Internet Explorer 6	No	Yes
Internet Explorer 7	Yes	Yes
Opera 8+	Yes	No
Safari 1.2+	Yes	No

Table 3-1 `XMLHttpRequest` Object Support by Browser

Property	Description
`readyState`	Integer indicating the state of the request, either: 0 (uninitialized) 1 (loading) 2 (response headers received) 3 (some response body received) 4 (request complete)
`onreadystatechange`	Function to call whenever the `readyState` changes
`status`	HTTP status code returned by the server (e.g., "200, 404, etc.")
`statusText`	Full status HTTP status line returned by the server (e.g., "OK, No Content, etc.")
`responseText`	Full response from the server as a string
`responseXML`	A `Document` object representing the server's response parsed as an XML document
`abort()`	Cancels an asynchronous HTTP request
`getAllResponseHeaders()`	Returns a string containing all the HTTP headers the server sent in its response. Each header is a name/value pair separated by a colon and header lines are separated by a carriage return / linefeed pair.
`getResponseHeader(`*headerName*`)`	Returns a string corresponding to the value of the *headerName* header returned by the server (e.g., `request.getResponseHeader("Set-cookie")`)

Table 3-2 Common Properties and Methods of the `XMLHttpRequest` Object

Property	Description
`open`(*method*, *url* `[`, *asynchronous* `[`, *user*, *password*`]]`)	Initializes the request in preparation for sending to the server. The *method* parameter is the HTTP method to use, for example "GET" or "POST". The value of method is not case sensitive. The *url* is the relative or absolute URL the request will be sent to. The optional *asynchronous* parameter indicates whether *send()* returns immediately or after the request is complete (default is **true**, meaning it returns immediately). The optional *user* and *password* arguments are to be used if the URL requires HTTP authentication. If none are specified and the URL requires authentication, the user will be prompted to enter it.
`setRequestHeader`(*name*, *value*)	Adds the HTTP header given by the *name* (without the colon) and *value* parameters.
`send`(*body*)	Initiates the request to the server. The *body* parameter should contain the body of the request, i.e., a string containing *fieldname=value&fieldname2=value2…* for POSTs or a null value for GET request.

TABLE 3-2 Common Properties and Methods of the **`XMLHttpRequest`** Object (*continued*)

Property or Method	Description	Browser Support
`onload`	Event triggered when whole document has finished loading, similar to looking at `onreadystatechange` when the `readyState` value is 4.	Firefox 1.5+
`onprogress`	Event triggered as partial data becomes available. The event will fire continuously as data is made available.	Firefox 1.5+
`onerror`	Event triggered when a network error occurs.	Firefox 1.5+ (still buggy as of Firefox 2)
`overrideMimeType`('*mime-type*')	Method takes a string for a MIME type value (for example, text/xml) and overrides whatever MIME type is indicated in the response packet.	Firefox 1.5+, Opera (buggy)

TABLE 3-3 Browser-specific XHR Properties and Methods

With a basic syntax overview complete, let's continue our discussion with concrete examples of XHRs in use.

Instantiation and Cross Browser Concerns

From the previous section, it is clear that there are inconsistencies in browser support for XHRs. Many browsers support the `XMLHttpRequest` object natively, which makes it quite simple to instantiate.

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

This code is all that is required to create an XHR in browsers such as Firefox 1+, Opera 8+, Safari 1.2+, and Internet Explorer 7+, but what about older Internet Explorer browsers, particularly IE6?

ActiveX XHR Anxiety

In the case of older Internet Explorer browsers (5, 5.5, and 6), the XHR object is instantiated a bit differently via the `ActiveXObject` constructor and passing in a string indicating the particular Microsoft XML (MSXML) parser installed. For example:

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

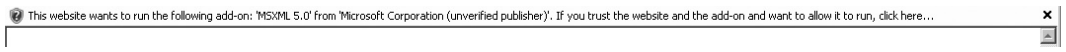
would attempt to instantiate the oldest form of the MSXML parser. As Internet Explorer matured and other software needed XML support, various other editions of MSXML were made available. Table 3-4 shows the standard relationships between IE and the XML ActiveX version supported.

Based upon this data, most Ajax libraries thus also use the program ID strings "Msxml2.XMLHTTP.3" and "Msxml2.XMLHTTP" to instantiate an ActiveX-based XHR object. Yet it is possible that other versions of MSXML outside those listed in Table 3-4 may also be available because of operating system or applications installed on a client system, and you might opt to use them. However, proceed with caution. For example, MSXML 4 is buggy, and MSXML 5

Internet Explorer Version	MSXML Version (file version)
5.0a	2.0a (5.0.2314.1000)
5.0b	2.0b (5.0.2614.3500)
5.01	2.5a (5.0.2919.6303)
5.01 SP1	2.5 SP1 (8.0.5226)
5.5	2.5 SP1 (8.0.5226)
5.5 SP2	2.5 Post-SP2 (8.00.6611.0)
6.0	3.0 SP2 (8.20.8730.1)
6.0 SP1	3.0 SP3 (8.30.9926.0)

Table 3-4 Internet Explorer—MSXML Relationship

should be avoided as it is focused on the scripting needs of MS Office Applications and will trigger an ActiveX security dialog when used in Internet Explorer.

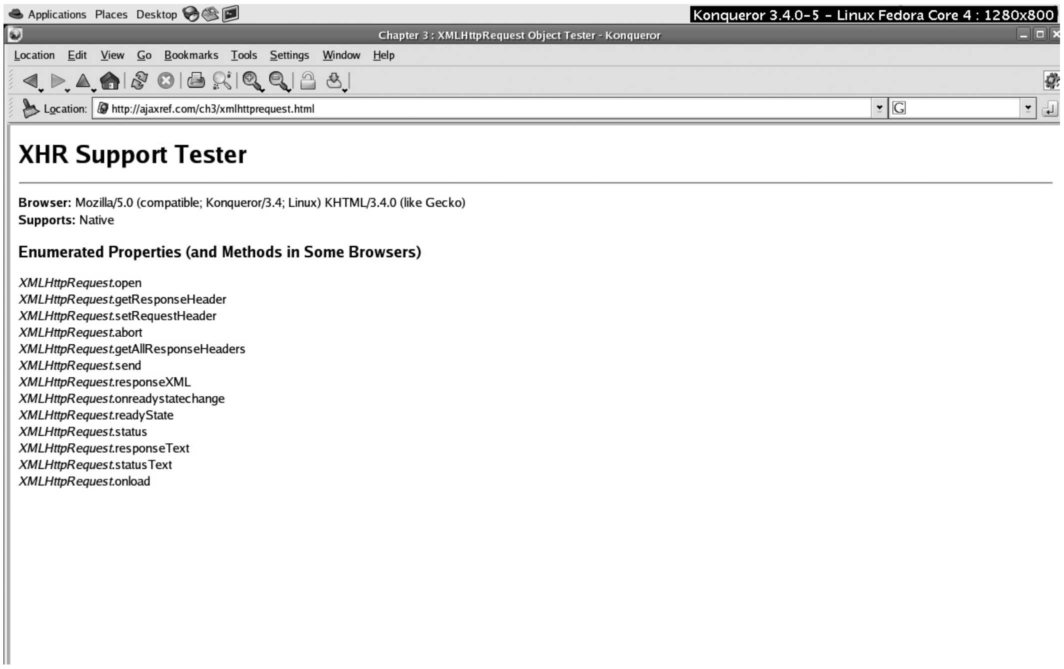

This website wants to run the following add-on: 'MSXML 5.0' from 'Microsoft Corporation (unverified publisher)'. If you trust the website and the add-on and want to allow it to run, click here...

At the time of this edition's writing, MSXML 6, which is provided with Vista, is the most up to date and standards-compliant XML parser released by Microsoft. However, if you are running Vista or have installed IE7 you won't need to know this for basic Ajax duties as the browser can use the native XHR. Given the room for confusion as to what ActiveX XHR possibilities are available, a simple testing program is provided for you to see what is supported by your browser. The script is quite straight forward and simply tries a variety of ways to instantiate the XHR object, as well as enumerate its properties and methods. A few captures of this script in action are shown in Figure 3-1.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Chapter 3 : XMLHttpRequest Object Tester</title>
<script type="text/javascript">

function XHRTester()
{
   var nativeXHR = false;
   var activeX = "";
   var commObject = null;
   try
   {
      commObject = new XMLHttpRequest();
      nativeXHR = true;
   }
   catch(e) {}

   /*
    * Testing purposes only. See createXHR wrapper for adopted pattern
    * If you use "MSXML2.XMLHTTP.5.0" you will be prompted by IE so it is
    * omitted here
    */
   var activeXStrings = ["Msxml2.XMLHTTP.6.0", "Msxml2.XMLHTTP.4.0",
                         "Msxml2.XMLHTTP.3.0", "Msxml2.XMLHTTP",
                         "Microsoft.XMLHTTP"];

   for (var i=0; i < activeXStrings.length; i++)
      {
       try {
              commObject = new ActiveXObject(activeXStrings[i]);
              activeX += activeXStrings[i] + ", ";
            }
         catch (e) { }
        }
   var userAgent = navigator.userAgent;
   var result = "";
   if (activeX === "" && !nativeXHR)
        result += "<em>None</em>";

   if (nativeXHR)
       result += "Native" ;

   if (activeX !== "")
     {
       activeX = activeX.substring(0,activeX.length-2);
       result += " ActiveX [ " + activeX +" ]";
      }
   var message = "<strong>Browser:</strong> " + userAgent +
      "<br /><strong>Supports:</strong> " + result;
   return message;
}
</script>
</head>
<body>
<h1>XHR Support Tester</h1>
<hr />
<script type="text/javascript">
  document.write(XHRTester());
  if (window.XMLHttpRequest)
   {
     document.write("<h3>Enumerated Properties (and Methods in Some
Browsers)</h3>")
     var XHR = new window.XMLHttpRequest();
     for (var aprop in XHR)
        document.write("<em>XMLHttpRequest</em>."+aprop + "<br />");
    }
</script>
</body>
</html>
```

NOTE *There is some skepticism in the Web development community about the purity of the native implementation of XHRs in IE7. You'll note, as shown by the previous example, that things like object prototypes do not work on XHRs in IE7. In the prerelease versions, even adding instance properties (expandos) seemed to be problematic, though no longer in the final release.*

Because Internet Explorer 7 still supports the legacy ActiveX implementation of XMLHTTP as well as the native object, you need to be a bit careful. While the benefit of this side-by-side installation of XML implementations is that older legacy applications using only ActiveX will not have to be rewritten, scripts may incur unneeded performance hits in newer versions of IE unless you are careful. When creating an XHR, make sure to always try native first before invoking ActiveX as it is more efficient, particularly if you are going to be creating many objects for individual requests. Furthermore, if you play with various settings in your Internet Explorer 7 browser, you will see that ignoring the legacy ActiveX

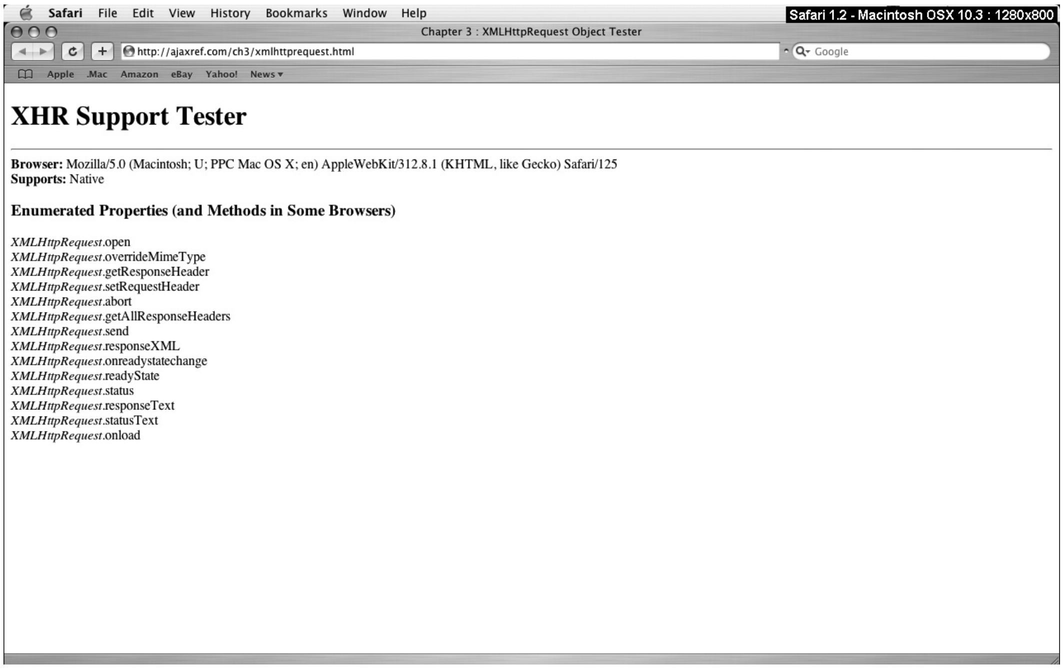

Applications Places Desktop
Konqueror 3.4.0-5 - Linux Fedora Core 4 : 1280x800
Chapter 3 : XMLHttpRequest Object Tester - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Location: http://ajaxref.com/ch3/xmlhttprequest.html

XHR Support Tester

Browser: Mozilla/5.0 (compatible; Konqueror/3.4; Linux) KHTML/3.4.0 (like Gecko)
Supports: Native

Enumerated Properties (and Methods in Some Browsers)

XMLHttpRequest.open
XMLHttpRequest.getResponseHeader
XMLHttpRequest.setRequestHeader
XMLHttpRequest.abort
XMLHttpRequest.getAllResponseHeaders
XMLHttpRequest.send
XMLHttpRequest.responseXML
XMLHttpRequest.onreadystatechange
XMLHttpRequest.readyState
XMLHttpRequest.status
XMLHttpRequest.responseText
XMLHttpRequest.statusText
XMLHttpRequest.onload


Safari File Edit View History Bookmarks Window Help
Safari 1.2 - Macintosh OSX 10.3 : 1280x800
Chapter 3 : XMLHttpRequest Object Tester
http://ajaxref.com/ch3/xmlhttprequest.html
Google
Apple .Mac Amazon eBay Yahoo! News

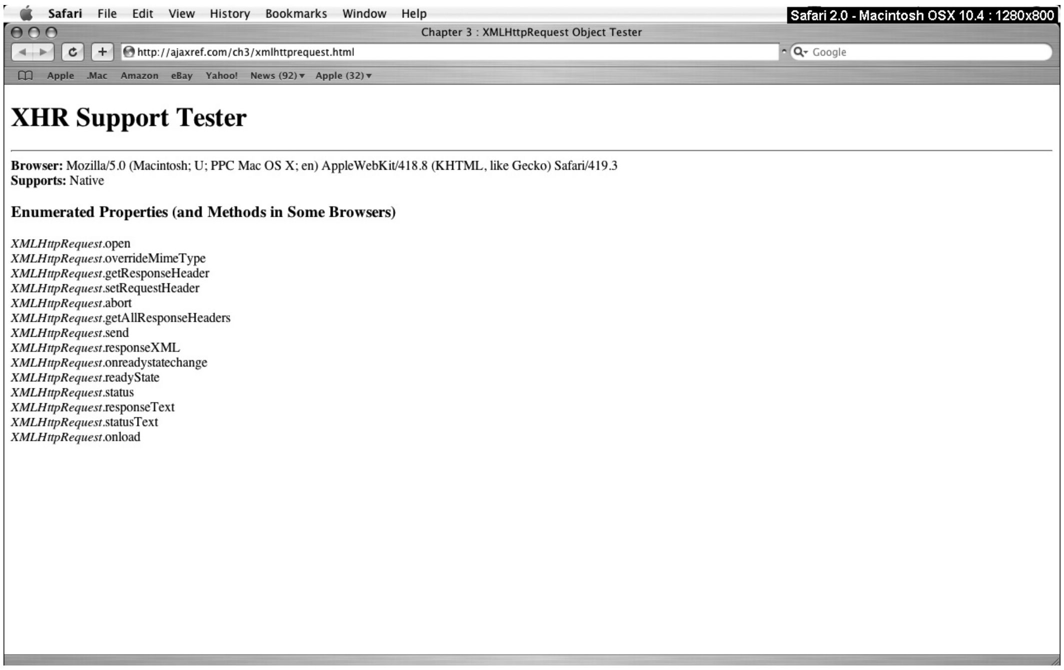
XHR Support Tester

Browser: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/312.8.1 (KHTML, like Gecko) Safari/125
Supports: Native

Enumerated Properties (and Methods in Some Browsers)

XMLHttpRequest.open
XMLHttpRequest.overrideMimeType
XMLHttpRequest.getResponseHeader
XMLHttpRequest.setRequestHeader
XMLHttpRequest.abort
XMLHttpRequest.getAllResponseHeaders
XMLHttpRequest.send
XMLHttpRequest.responseXML
XMLHttpRequest.onreadystatechange
XMLHttpRequest.readyState
XMLHttpRequest.status
XMLHttpRequest.responseText
XMLHttpRequest.statusText
XMLHttpRequest.onload

FIGURE 3-1 Various browsers reporting XHR support

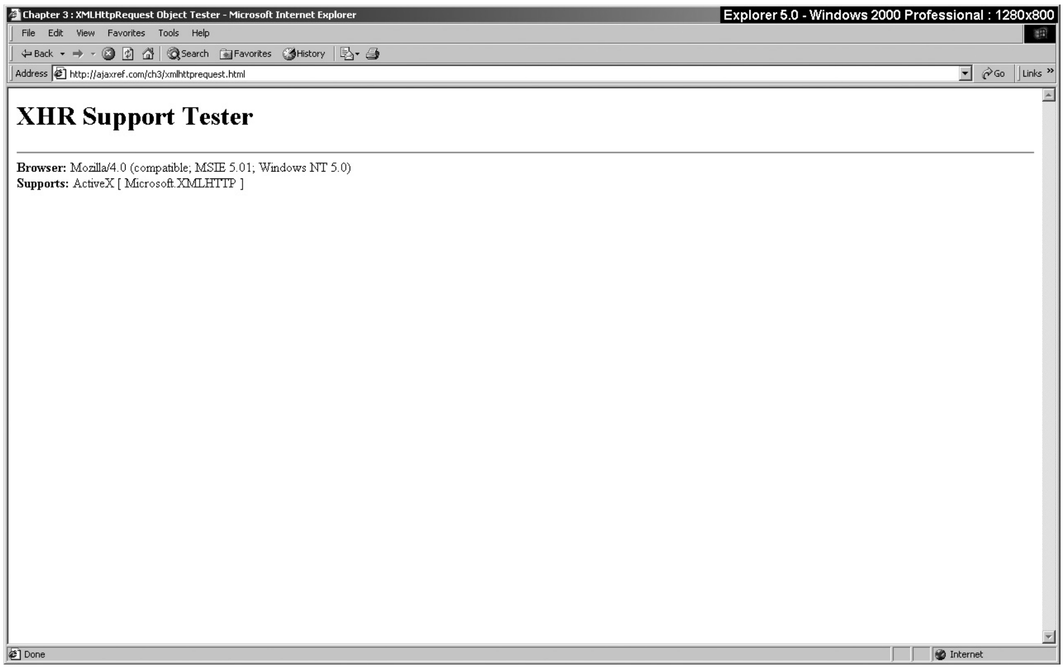

Safari File Edit View History Bookmarks Window Help
Safari 2.0 - Macintosh OSX 10.4 : 1280x800
Chapter 3 : XMLHttpRequest Object Tester
http://ajaxref.com/ch3/xmlhttprequest.html
Google
Apple .Mac Amazon eBay Yahoo! News (92) Apple (32)

XHR Support Tester

Browser: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/418.8 (KHTML, like Gecko) Safari/419.3
Supports: Native

Enumerated Properties (and Methods in Some Browsers)

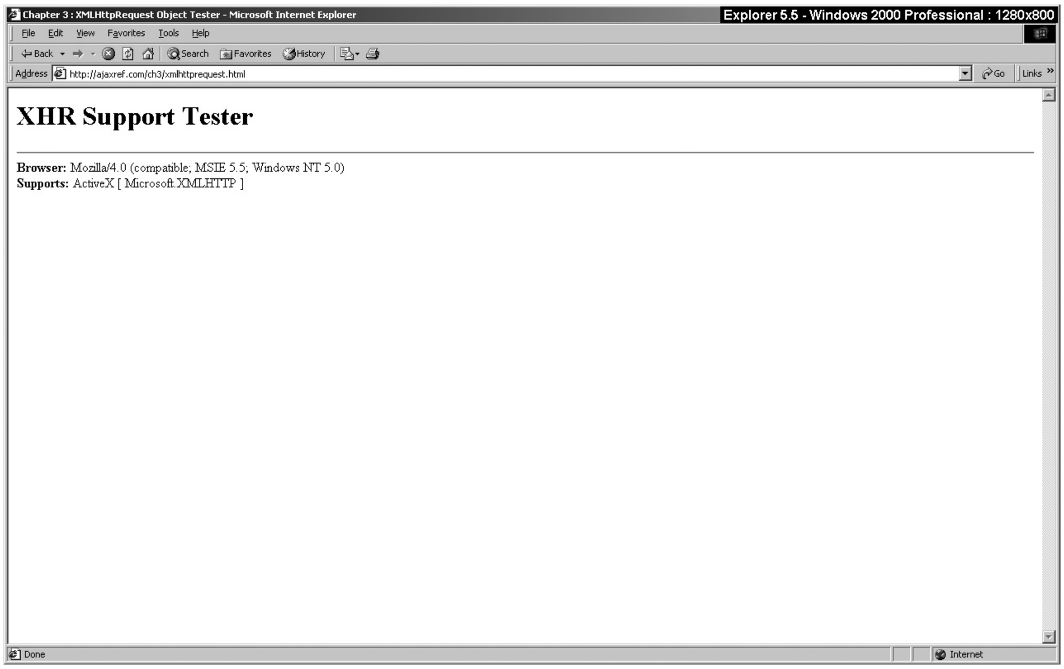
XMLHttpRequest.open
XMLHttpRequest.overrideMimeType
XMLHttpRequest.getResponseHeader
XMLHttpRequest.setRequestHeader
XMLHttpRequest.abort
XMLHttpRequest.getAllResponseHeaders
XMLHttpRequest.send
XMLHttpRequest.responseXML
XMLHttpRequest.onreadystatechange
XMLHttpRequest.readyState
XMLHttpRequest.status
XMLHttpRequest.responseText
XMLHttpRequest.statusText
XMLHttpRequest.onload


Chapter 3 : XMLHttpRequest Object Tester - Microsoft Internet Explorer
Explorer 5.0 - Windows 2000 Professional : 1280x800
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://ajaxref.com/ch3/xmlhttprequest.html
Go Links
Done
Internet

XHR Support Tester

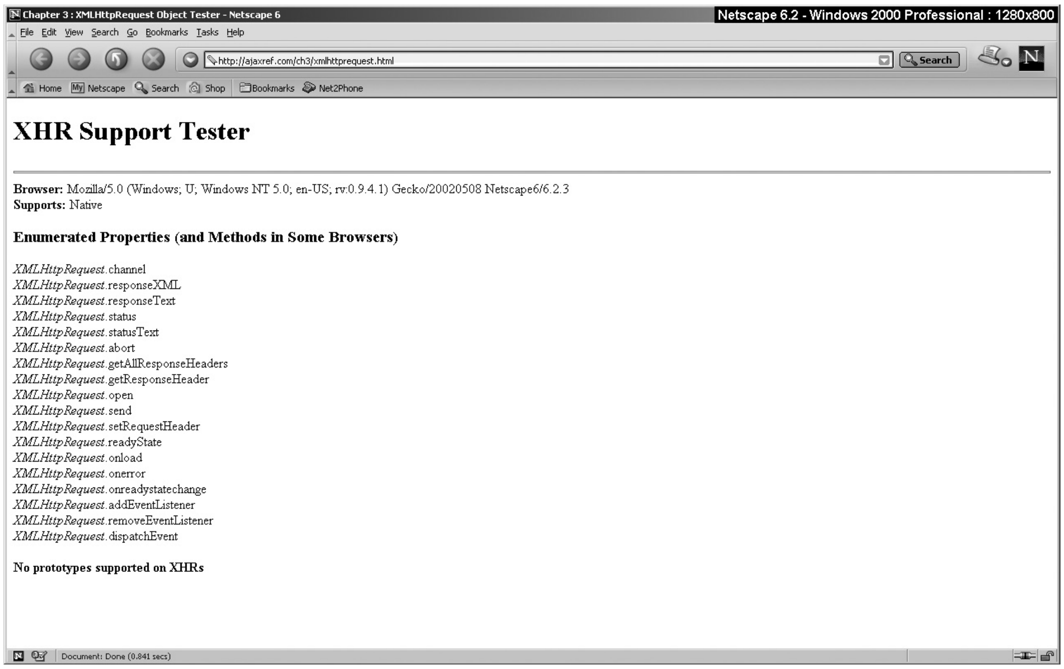
Browser: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)
Supports: ActiveX [Microsoft.XMLHTTP]

Figure 3-1 Various browsers reporting XHR support (*continued*)


Chapter 3 : XMLHttpRequest Object Tester - Microsoft Internet Explorer
Explorer 5.5 - Windows 2000 Professional : 1280x800
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://ajaxref.com/ch3/xmlhttprequest.html
Go Links
Done
Internet

XHR Support Tester

Browser: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Supports: ActiveX [Microsoft.XMLHTTP]


Chapter 3 : XMLHttpRequest Object Tester - Netscape 6
Netscape 6.2 - Windows 2000 Professional : 1280x800
File Edit View Search Go Bookmarks Tasks Help
http://ajaxref.com/ch3/xmlhttprequest.html
Search
Home My Netscape Search Shop Bookmarks Net2Phone
Document: Done (0.841 secs)

XHR Support Tester

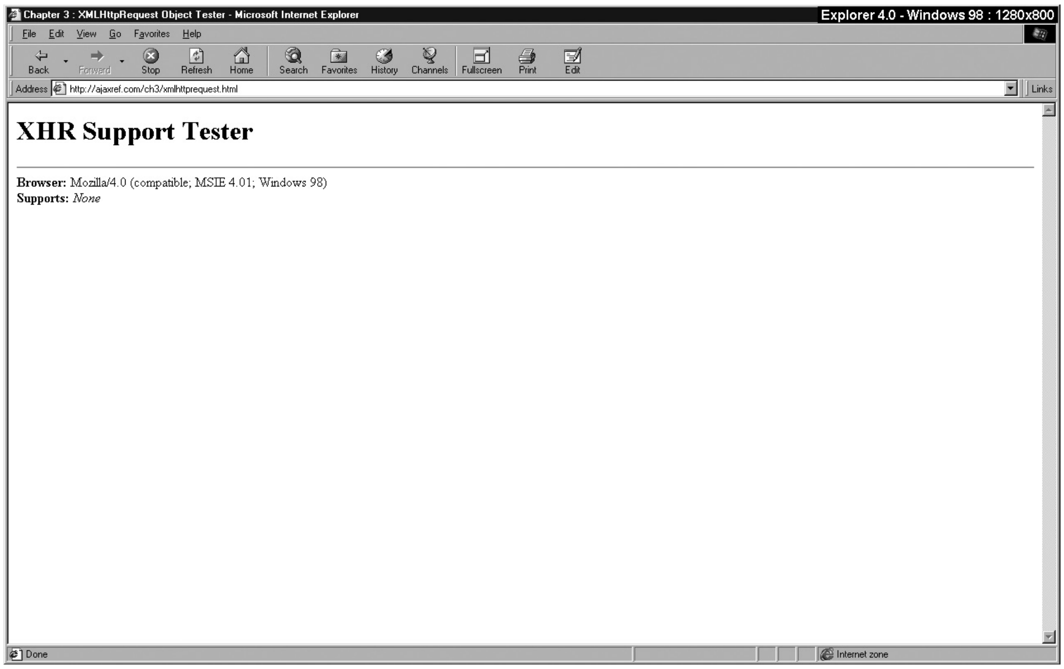
Browser: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:0.9.4.1) Gecko/20020508 Netscape6/6.2.3
Supports: Native

Enumerated Properties (and Methods in Some Browsers)

XMLHttpRequest.channel
XMLHttpRequest.responseXML
XMLHttpRequest.responseText
XMLHttpRequest.status
XMLHttpRequest.statusText
XMLHttpRequest.abort
XMLHttpRequest.getAllResponseHeaders
XMLHttpRequest.getResponseHeader
XMLHttpRequest.open
XMLHttpRequest.send
XMLHttpRequest.setRequestHeader
XMLHttpRequest.readyState
XMLHttpRequest.onload
XMLHttpRequest.onerror
XMLHttpRequest.onreadystatechange
XMLHttpRequest.addEventListener
XMLHttpRequest.removeEventListener
XMLHttpRequest.dispatchEvent

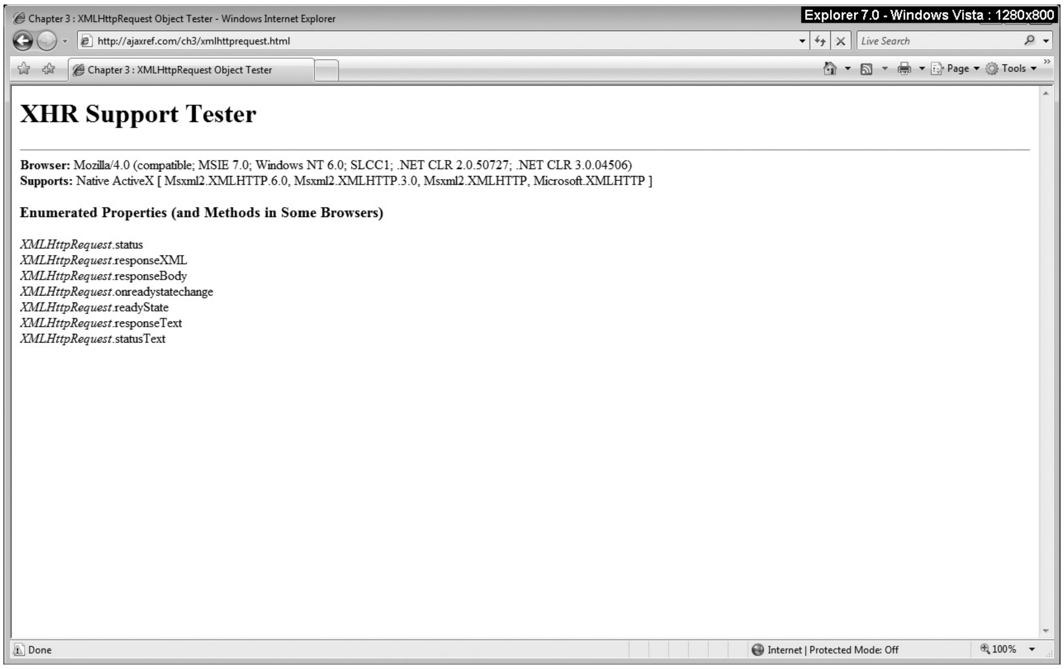
No prototypes supported on XHRs

Figure 3-1 Various browsers reporting XHR support (*continued*)


Chapter 3 : XMLHttpRequest Object Tester - Microsoft Internet Explorer
Explorer 4.0 - Windows 98 : 1280x800
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Print Edit
Address http://ajaxref.com/ch3/xmlhttprequest.html
Links
Done
Internet zone

XHR Support Tester

Browser: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Supports: *None*


Chapter 3 : XMLHttpRequest Object Tester - Windows Internet Explorer
Explorer 7.0 - Windows Vista : 1280x800
http://ajaxref.com/ch3/xmlhttprequest.html
Live Search
Chapter 3 : XMLHttpRequest Object Tester
Page Tools
Done
Internet | Protected Mode: Off
100%

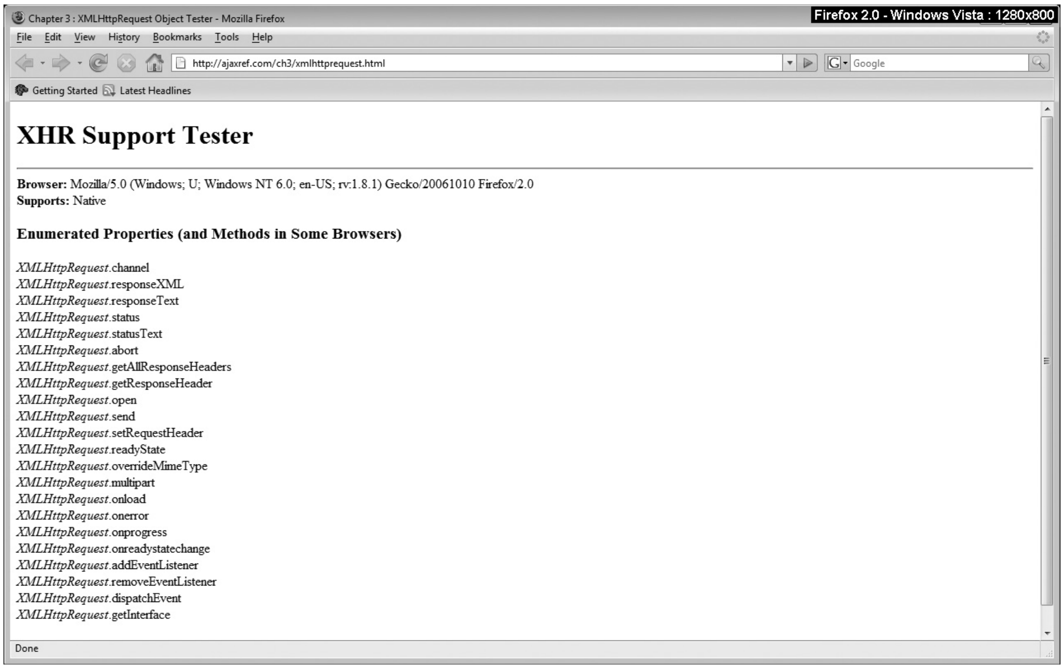
XHR Support Tester

Browser: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR 2.0.50727; .NET CLR 3.0.04506)
Supports: Native ActiveX [Msxml2.XMLHTTP.6.0, Msxml2.XMLHTTP.3.0, Msxml2.XMLHTTP, Microsoft.XMLHTTP]

Enumerated Properties (and Methods in Some Browsers)

XMLHttpRequest.status
XMLHttpRequest.responseXML
XMLHttpRequest.responseBody
XMLHttpRequest.onreadystatechange
XMLHttpRequest.readyState
XMLHttpRequest.responseText
XMLHttpRequest.statusText

Figure 3-1 Various browsers reporting XHR support (*continued*)

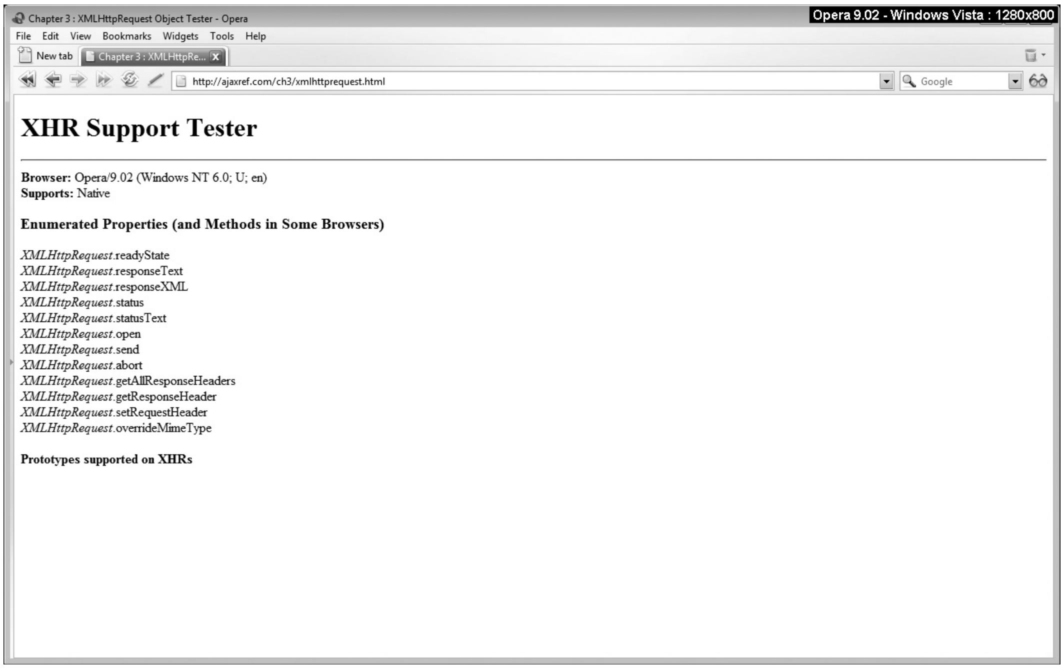

Chapter 3 : XMLHttpRequest Object Tester - Mozilla Firefox
Firefox 2.0 - Windows Vista : 1280x800
File Edit View History Bookmarks Tools Help
http://ajaxref.com/ch3/xmlhttprequest.html
Google
Getting Started Latest Headlines
Done

XHR Support Tester

Browser: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.8.1) Gecko/20061010 Firefox/2.0
Supports: Native

Enumerated Properties (and Methods in Some Browsers)

XMLHttpRequest.channel
XMLHttpRequest.responseXML
XMLHttpRequest.responseText
XMLHttpRequest.status
XMLHttpRequest.statusText
XMLHttpRequest.abort
XMLHttpRequest.getAllResponseHeaders
XMLHttpRequest.getResponseHeader
XMLHttpRequest.open
XMLHttpRequest.send
XMLHttpRequest.setRequestHeader
XMLHttpRequest.readyState
XMLHttpRequest.overrideMimeType
XMLHttpRequest.multipart
XMLHttpRequest.onload
XMLHttpRequest.onerror
XMLHttpRequest.onprogress
XMLHttpRequest.onreadystatechange
XMLHttpRequest.addEventListener
XMLHttpRequest.removeEventListener
XMLHttpRequest.dispatchEvent
XMLHttpRequest.getInterface


Chapter 3 : XMLHttpRequest Object Tester - Opera
Opera 9.02 - Windows Vista : 1280x800
File Edit View Bookmarks Widgets Tools Help
New tab
Chapter 3 : XMLHttpRe...
http://ajaxref.com/ch3/xmlhttprequest.html
Google

XHR Support Tester

Browser: Opera/9.02 (Windows NT 6.0; U; en)
Supports: Native

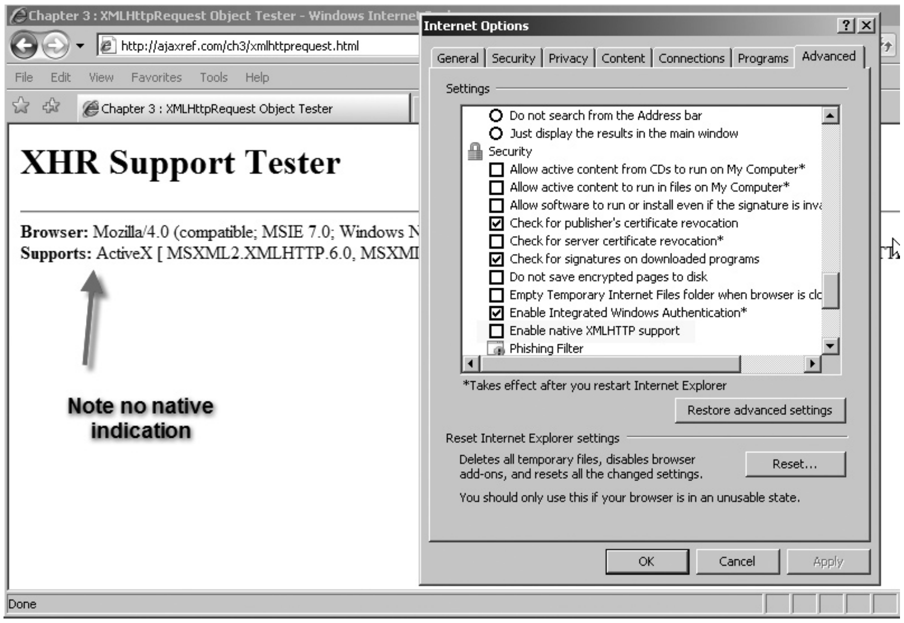
Enumerated Properties (and Methods in Some Browsers)

XMLHttpRequest.readyState
XMLHttpRequest.responseText
XMLHttpRequest.responseXML
XMLHttpRequest.status
XMLHttpRequest.statusText
XMLHttpRequest.open
XMLHttpRequest.send
XMLHttpRequest.abort
XMLHttpRequest.getAllResponseHeaders
XMLHttpRequest.getResponseHeader
XMLHttpRequest.setRequestHeader
XMLHttpRequest.overrideMimeType

Prototypes supported on XHRs

Figure 3-1 Various browsers reporting XHR support (*continued*)

approach may not be the best course of action. Consider that it is possible for the user to turn off native `XMLHttpRequest` under the Advanced tab of Internet Options, which will then only allow for an ActiveX XHR.


Chapter 3 : XMLHttpRequest Object Tester - Windows Internet

http://ajaxref.com/ch3/xmlhttprequest.html

XHR Support Tester

Browser: Mozilla/4.0 (compatible; MSIE 7.0; Windows N
Supports: ActiveX [MSXML2.XMLHTTP.6.0, MSXML

Note no native indication

Internet Options — Advanced

Settings
- ○ Do not search from the Address bar
- ○ Just display the results in the main window
- Security
  - ☐ Allow active content from CDs to run on My Computer*
  - ☐ Allow active content to run in files on My Computer*
  - ☐ Allow software to run or install even if the signature is inva
  - ☑ Check for publisher's certificate revocation
  - ☐ Check for server certificate revocation*
  - ☑ Check for signatures on downloaded programs
  - ☐ Do not save encrypted pages to disk
  - ☐ Empty Temporary Internet Files folder when browser is clo
  - ☑ Enable Integrated Windows Authentication*
  - ☐ Enable native XMLHTTP support
  - Phishing Filter

*Takes effect after you restart Internet Explorer

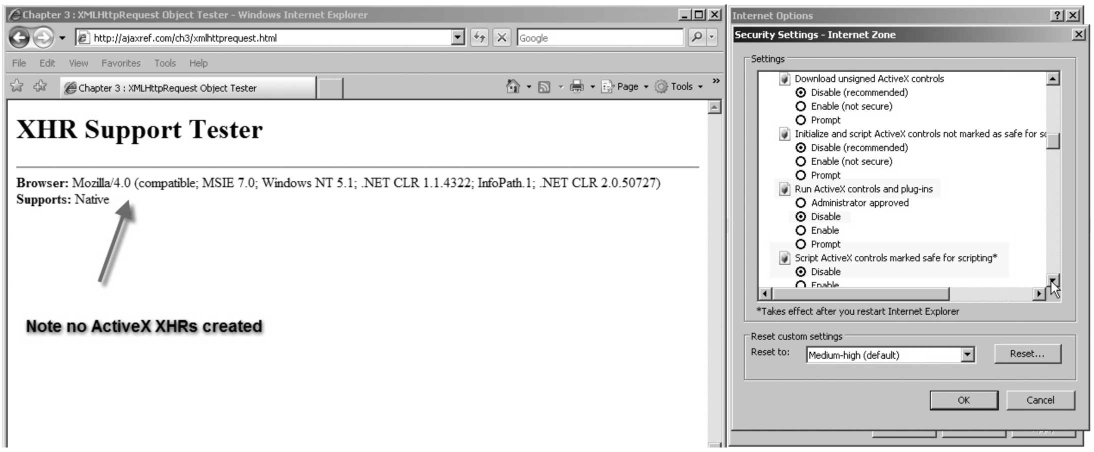
Reset Internet Explorer settings

Deletes all temporary files, disables browser add-ons, and resets all the changed settings.

You should only use this if your browser is in an unusable state.

General | Security | Privacy | Content | Connections | Programs | Advanced
File Edit View Favorites Tools Help
Restore advanced settings
Reset...
OK
Cancel
Apply
Done

More likely, it is possible that the user has turned off ActiveX support in Internet Explorer by adjusting their security settings, as shown next.


Chapter 3 : XMLHttpRequest Object Tester - Windows Internet Explorer

http://ajaxref.com/ch3/xmlhttprequest.html

XHR Support Tester

Browser: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; InfoPath.1; .NET CLR 2.0.50727)
Supports: Native

Note no ActiveX XHRs created

Security Settings - Internet Zone

Settings
- Download unsigned ActiveX controls
  - ● Disable (recommended)
  - ○ Enable (not secure)
  - ○ Prompt
- Initialize and script ActiveX controls not marked as safe for sc
  - ● Disable (recommended)
  - ○ Enable (not secure)
  - ○ Prompt
- Run ActiveX controls and plug-ins
  - ○ Administrator approved
  - ● Disable
  - ○ Enable
  - ○ Prompt
- Script ActiveX controls marked safe for scripting*
  - ● Disable
  - ○ Enable

*Takes effect after you restart Internet Explorer

Reset custom settings

Reset to: Medium-high (default)

Of course, it might be possible that the user disables both features but somehow keeps JavaScript on. In this case, it is necessary to degrade to an alternate JavaScript communication mechanism from the previous chapter, degrade to a standard post-and-wait style form of communication, or provide some error message and potentially block the user from the site or application. Architecturally, this can introduce some complexity to the design of your application. We will take up this expansive topic in Chapter 9.

Given that you can disable XHRs in Internet Explorer, you might wonder if it is possible to do the same in other browsers. Opera and Safari do not appear to support a way to disable XHRs without disabling all JavaScript. In Firefox, you can modify the browser's capabilities in a very fine grain manner. For example, to disable XHRs you could disable the `open()` method for the object. To accomplish this, first type **about:config** in Firefox's address bar. Next, right-click and create a new string. Name the property `capability.policy.default.XMLHttpRequest.open` and set the value to be `noAccess`. You should now find that XHRs are denied. Likely someone will modify Firefox to make it easy to do this by the time you read this, but regardless, you can see it is possible to slice out just the feature of JavaScript you need to.

NOTE *It is also possible to disable XHRs by modifying your browser's user.js file (or creating a new one) and adding the line*

```
user_pref("capability.policy.default.XMLHttpRequest.open", "noAccess").
```

A Cross-Browser XHR Wrapper

Given the previous discussion, if you wanted to do a quick and dirty abstraction for XHRs and didn't care so much about making sure to address the very latest ActiveX based XHR facility, you might just use a `?` operator, like so:

```
var xhr = (window.XMLHttpRequest) ?
new XMLHttpRequest() : new ActiveXObject("MSXML2.XMLHTTP.3.0");
```

or you could attempt to make older IEs look like they support native XHRs with code like this:

```
// Emulate the native XMLHttpRequest object of standards compliant browsers
if  (!window.XMLHttpRequest)
  window.XMLHttpRequest = function () {
     return new ActiveXObject("MSXML2.XMLHTTP.3.0"); }
```

If there was some concern about this code in non-IE browsers, you could employ the conditional comment system supported in Jscript to hide this override.

```
/*@cc_on @if (@_win32 && @_jscript_version >= 5)

if  (!window.XMLHttpRequest)
   window.XMLHttpRequest = function() { return new
ActiveXObject("MSXML2.XMLHTTP.3.0"); }
@end @*/
```

We opt instead to write a simple wrapper function `createXHR()` ,so that other techniques can easily be added if ever required. In this implementation, first the native instantiation is attempted followed by the most supported ActiveX solutions eventually returning null if nothing can be created.

```
function createXHR()
{
   try { return new XMLHttpRequest(); } catch(e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP.6.0"); } catch (e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP.3.0"); } catch (e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
   try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
   return null;
}
```

To create a cross-browser XHR object, all you need to do is call the wrapper function and make sure it returns something.

```
var xhr = createXHR();
if (xhr)
 {
   // Engage the XHR!
 }
```

Now with XHR in hand it is time to use it to make a request.

NOTE *There is a Java-based browser called IceBrowser that supports an alternate form of XHR creation,* `window.createRequest()`*, which you could have added to your wrapper. Other esoteric browsers may also use alternative XHR syntax, but we avoid promoting such esoteric oddities except to make you aware of their possible existence.*

XHR Request Basics

Once the XHR object is created, most of the cross-browser concerns subside—for the moment, at least. To invoke an XHR request, all browsers use the same syntax:

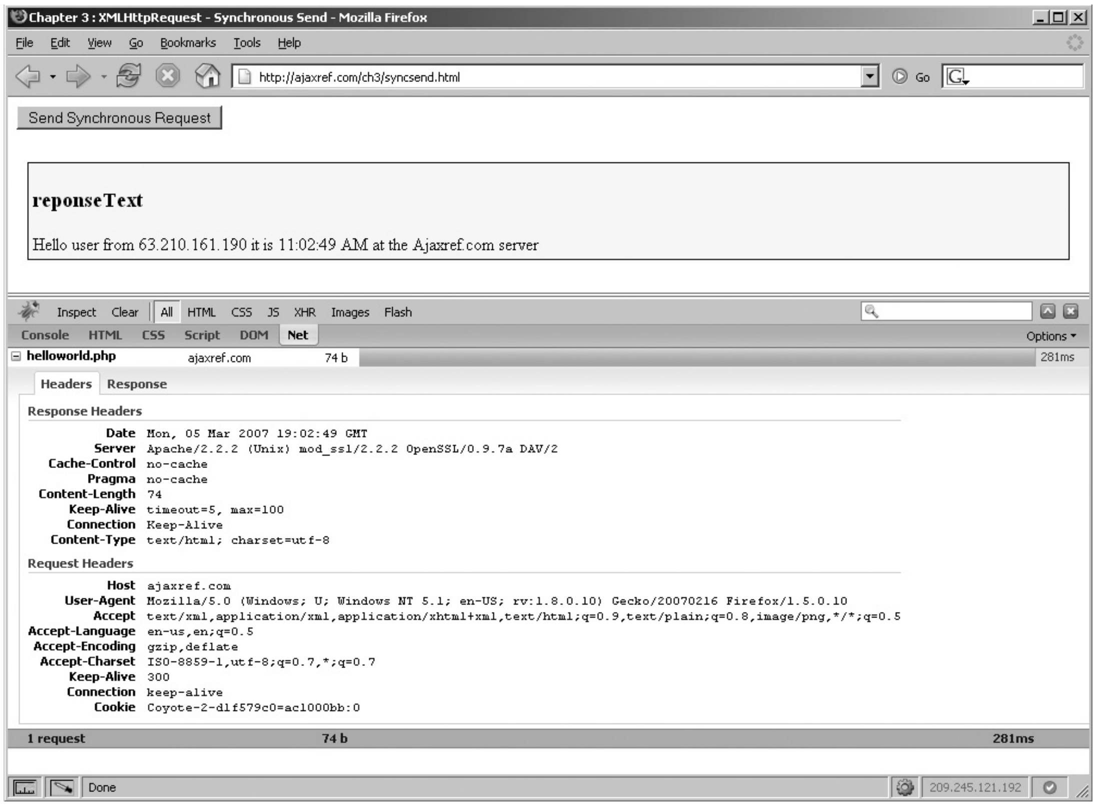
```
xhr.open(method, url, async [ ,username, password ])
```

where *method* is an HTTP method like GET, POST, HEAD. While these values are not case-sensitive, they should be in uppercase as per the HTTP specification. The parameter *url* is the particular URL to call and may be either relative or absolute. The *async* parameter is set to true if the request is to be made asynchronously or false if it should be made synchronously. If not specified, the request will be made asynchronously. The optional parameters *username* and *password* are used when attempting to access a resource that is protected with HTTP Basic authentication. We will explore that later in the chapter, but these parameters won't be very useful given the way browsers implement this feature.

Synchronous Requests

We start the discussion of XHR-based communication with the simplest example: performing a synchronous request. In this particular case, first, the wrapper function is used to create an XHR. Next a connection is opened using the syntax presented in the previous section. In this case, the URL is set to a very basic PHP program that will echo back the IP address of the user accessing it and the local server time. Finally, the request is sent on its way by invoking the XHR's `send()` method. It should be noted at this point that the URL requested must be within the same domain, using the same port and the same protocol from which a page is served. Browsers will deny other requests as breaking the same-origin policy. More details can be found on this and other security concerns in Chapter 7. Also note that a null value is sent in this particular example because there is no data to submit in the message body. When using POST to send data later in this chapter, that will not be the case. To keep things simple, the raw response is used and accessed via the XHR's `responseText` property and then added to the page using standard DOM methods. To be precise, `innerHTML` isn't actually isn't as of yet a W3C specified DOM property, but it is so ubiquitously supported, it is often assumed to be. The complete example is shown here with a communication trace in Figure 3-2.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Chapter 3 : XMLHttpRequest - Synchronous Send</title>
<link rel="stylesheet" href="http://ajaxref.com/ch3/global.css"
type="text/css" media="screen" />
```


Chapter 3 : XMLHttpRequest - Synchronous Send - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://ajaxref.com/ch3/syncsend.html
Go
Send Synchronous Request
Inspect Clear All HTML CSS JS XHR Images Flash
Console HTML CSS Script DOM Net
Options
Headers Response
Done
209.245.121.192

reponseText

Hello user from 63.210.161.190 it is 11:02:49 AM at the Ajaxref.com server

```
helloworld.php     ajaxref.com     74 b     281ms

Response Headers
           Date Mon, 05 Mar 2007 19:02:49 GMT
         Server Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.7a DAV/2
  Cache-Control no-cache
         Pragma no-cache
 Content-Length 74
     Keep-Alive timeout=5, max=100
     Connection Keep-Alive
   Content-Type text/html; charset=utf-8
Request Headers
           Host ajaxref.com
     User-Agent Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.10) Gecko/20070216 Firefox/1.5.0.10
         Accept text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language en-us,en;q=0.5
Accept-Encoding gzip,deflate
 Accept-Charset ISO-8859-1,utf-8;q=0.7,*;q=0.7
     Keep-Alive 300
     Connection keep-alive
         Cookie Coyote-2-d1f579c0=ac1000bb:0

1 request     74 b     281ms
```

FIGURE 3-2 Simple synchronous request

```
<script type="text/javascript" >
function createXHR()
{
   try { return new XMLHttpRequest(); } catch(e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP.6.0"); } catch (e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP.3.0"); } catch (e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
   try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
   return null;
}

function sendRequest()
{
    var responseOutput = document.getElementById("responseOutput");
    responseOutput.style.display = "";
    var xhr = createXHR();
    if (xhr)
    {
     xhr.open("GET", "http://ajaxref.com/ch3/helloworld.php", false);
     xhr.send(null);
     responseOutput.innerHTML = "<h3>reponseText</h3>" + xhr.responseText;
    }
}

window.onload = function ()
{
 document.requestForm.requestButton.onclick = function () { sendRequest(); };
};
</script>
</head>
<body>
<form action="#" name="requestForm">
  <input type="button" name="requestButton" value="Send Synchronous Request" />
</form>
<br />
<div id="responseOutput" class="results" style="display:none;">&nbsp;</div>
</body>
</html>
```

The PHP code that responds to this request is quite simple and the only details have to do with the cache control issues that will be discussed shortly.

```
<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");

$ip = GetHostByName($_SERVER['REMOTE_ADDR']);
echo "Hello user from $ip it is " .  date("h:i:s A") . " at the Ajaxref.com server";
?>
```

Of course, this previous example isn't really Ajax if you are a stickler for the precise meaning of the acronym as it used synchronous communication and no XML; it was *Sjat* (Synchronous JavaScript and Text), if you want to be precise. All jesting aside, it is important to note the implications of the synchronous communication. The browser, in effect, blocks on the line `xhr.send(null)` until the communication returns. Given the possibility for network delays and problems, this probably isn't the way to go except for important transactions. You can demonstrate this for yourself by running the example at http://ajaxref.com/ch3/syncsendslow.html. This example will block on the server for five seconds, giving plenty of time to note that your browser won't let you do anything else. While the asynchronous requests discussed in the next section do not exhibit such problems, they do introduce extra complexity to address.

Asynchronous Requests

To make the previous example perform its request asynchronously, the first change is to set the appropriate flag in the `open()` method.

```
xhr.open("GET", "http://ajaxref.com/ch3/helloworld.php", true);
```

However, where to put the code to handle the returned data is not immediately obvious. To address the response, a callback function must be defined that will be awoken as the response is received. To do this, associate a function with the XHR's `onreadystate` property. For example, given a function called `handleResponse`, set the `readystatechange` property like so:

```
xhr.onreadystatechange = handleResponse;
```

Unfortunately, when set like this is it not possible to pass any parameters to the callback function directly and thus it tends to lead to the use of global variables. Instead, use an inner function called a closure to wrap the function call and any values it might use or be passed, like so:

```
xhr.onreadystatechange = function(){handleResponse(xhr);};
```

Now the `handleResponse` function is going to get called a number of times as the request is processed. As the function is called, it is possible to observe the progress of the request by looking at the XHR's `readyState` property. However, at this point in the discussion the focus is simply on knowing when the request is done as indicated by a `readyState` value of 4. Also, it is important that the HTTP request must be successful as indicated by a `status` property value of 200 corresponding to the HTTP response line "`200 OK`". The `handleResponse` function shown next shows all these ideas in action.

```
function handleResponse(xhr)
{
  if (xhr.readyState == 4  && xhr.status == 200)
    {
     var responseOutput = document.getElementById("responseOutput");
     responseOutput.innerHTML = "<h3>reponseText</h3>" + xhr.responseText;
     responseOutput.style.display = "";
    }
}
```

The complete example is now shown. It also can be found online at http://ajaxref.com/ch3/asyncsend.html.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Chapter 3 : XMLHttpRequest - Asynchronous Send</title>
<link rel="stylesheet" href="http://ajaxref.com/ch3/global.css"
type="text/css" media="screen" />
<script type="text/javascript">
function createXHR()
{
   try { return new XMLHttpRequest(); } catch(e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP.6.0"); } catch (e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP.3.0"); } catch (e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
   try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
   return null;
}

function sendRequest()
{
    var xhr = createXHR();
    if (xhr)
    {
     xhr.open("GET", "http://ajaxref.com/ch3/helloworld.php", true);
     xhr.onreadystatechange = function(){handleResponse(xhr);};
     xhr.send(null);
    }
}
function handleResponse(xhr)
{
  if (xhr.readyState == 4  && xhr.status == 200)
    {
     var responseOutput = document.getElementById("responseOutput");
     responseOutput.innerHTML = "<h3>reponseText</h3>" + xhr.responseText;
     responseOutput.style.display = "";
    }
}
window.onload = function ()
{
 document.requestForm.requestButton.onclick = function () { sendRequest();
};
};
</script>
</head>
<body>

<form action="#" name="requestForm">
  <input type="button" name="requestButton"
         value="Send an Asynchronous Request" />
```

```
</form>
<br />
<div id="responseOutput" class="results" style="display:none;">&nbsp;</div>

</body>
</html>
```

Obviously, given the "browser lock-up" limitation presented in the previous section, you might want to try http://ajaxref.com/ch3/asyncsendslow.html to prove to yourself the value of using asynchronous communication. However, do note that with this power comes a price as now you must keep track of the connections made and make sure that they return in a timely fashion and without errors. You will also find that, if the ordering of requests and responses matter, asynchronous communication introduces much more complexity than may be expected. The richer network provides Ajax tremendous power and flexibility, but should not be trifled with. We'll begin to present some of these issues in more detail when we discuss `readyState` and `status` more in-depth later in this chapter and much more detail will be provided in Chapter 6 which discusses network concerns. For now, let's expand the XHR examples by transmitting some data to the server.

Sending Data via GET

As mentioned in the previous chapter, data can be sent via any HTTP GET request by adding the data to send to a query string in the URL to send to. Of course, the same is also true in the case of XHR-based communication, just create the XHR object and set it to request the desired URL with a query string appended, like so:

```
var xhr = createXHR();
if (xhr)
  {
    xhr.open("GET","http://ajaxref.com/ch3/setrating.php?rating=5",true);
    xhr.onreadystatechange = function(){handleResponse(xhr);};
    xhr.send(null);
  }
```

As you can see, it is quite easy to make a request but it is still necessary to respect the encoding concerns and make the payload URL safe, as well as acknowledge that there are limits to the amount of data that can be passed this way. As previously mentioned in Chapter 2, when passing more than a few hundred characters, you should start to worry about the appropriateness of the data transfer method. We revisit the rating example of the previous chapter done with an XHR communication mechanism for your inspection.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Chapter 3 : XMLHttpRequest - Sending Data with GET Query Strings </title>
<script type="text/javascript">
function encodeValue(val)
{
 var encodedVal;
 if (!encodeURIComponent)
 {
   encodedVal = escape(val);
   /* fix the omissions */
   encodedVal = encodedVal.replace(/@/g,"%40");
   encodedVal = encodedVal.replace(/\//g,"%2F");
   encodedVal = encodedVal.replace(/\+/g,"%2B");
 }
 else
 {
   encodedVal = encodeURIComponent(val);
   /* fix the omissions */
   encodedVal = encodedVal.replace(/~/g,"%7E");
   encodedVal = encodedVal.replace(/!/g,"%21");
   encodedVal = encodedVal.replace(/\(/g,"%28");
   encodedVal = encodedVal.replace(/\)/g,"%29");
   encodedVal = encodedVal.replace(/'/g,"%27");
 }
 /* clean up the spaces and return */
 return encodedVal.replace(/\%20/g,"+");
}
function createXHR()
{
   try { return new XMLHttpRequest(); } catch(e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP.6.0"); } catch (e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP.3.0"); } catch (e) {}
   try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
   try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
   return null;
}
function sendRequest(url, payload)
{
    var xhr = createXHR();

    if (xhr)
     {
       xhr.open("GET",url + "?" + payload,true);
       xhr.onreadystatechange = function(){handleResponse(xhr);};
       xhr.send(null);
     }
}

function handleResponse(xhr)
{
  if (xhr.readyState == 4  && xhr.status == 200)
    {
     var responseOutput = document.getElementById("responseOutput");
     responseOutput.innerHTML = xhr.responseText;
    }
}
function rate(rating)
```

```
{
    var url = "http://ajaxref.com/ch3/setrating.php";
    var payload = "rating=" + encodeValue(rating);


         sendRequest(url, payload);
     }
     window.onload = function ()
     {
      var radios = document.getElementsByName("rating");
      for (var i = 0; i < radios.length; i++)
       {
        radios[i].onclick = function (){rate(this.value);};
       }
     };

     </script>
     </head>
     <body>
     <h3>How do you feel about Ajax?</h3>
     <form action="#" method="get">
     <em>Hate It - </em> [
     <input type="radio" name="rating" value="1" /> 1
     <input type="radio" name="rating" value="2" /> 2
     <input type="radio" name="rating" value="3" /> 3
     <input type="radio" name="rating" value="4" /> 4
     <input type="radio" name="rating" value="5" /> 5
     ] <em> - Love It</em>
     </form>
     <br />
     <div id="responseOutput">&nbsp;</div>
     </body>
     </html>
```

Sending Data via Post

Sending data via an HTTP POST request is not much more difficult than the GET example—a welcome change to the iframe examples of the previous chapter. First, change the call to `open()` to use the POST method.

```
xhr.open("POST",url,true);
```

Next, if sending any data to the server, make sure to set a header indicating the type of encoding to be used. In most cases, this will be the standard `x-www-form-urlencoded` format used by Web browsers doing form posts.

```
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

A common mistake is to omit this header, so be careful to always add it with the appropriate encoding value when transmitting data via POST.

Then, like the previous asynchronous example, a callback function must be registered, but this time when initiating the request using the `send()` method, pass the payload data as a parameter to the method.

```
xhr.send("rating=5");
```

The previous example's `sendRequest` function is now easily modified use the POST method:

```
function sendRequest(url, payload)
{
    var xhr = createXHR();
    if (xhr)
     {
       xhr.open("POST",url,true);
       xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
       xhr.onreadystatechange = function(){handleResponse(xhr);};
       xhr.send(payload);
     }
}
```

An example of XHR based POST requests in action can be foundat http://ajaxref.com/ch3/post.html.

NOTE *While most likely all POST requests will be set to use* `application/x-www-form-urlencoded` *content encoding, it is possible to set just about any desired encoding method. Chapter 4 will present an in-depth discussion of many possible request and response data formats and their use with XHRs.*

Request Headers

One thing that was sorely missing from the traditional JavaScript communication methods was the ability to control requests, particularly setting any needed headers. As seen in the previous POST example, XHRs provide a method `setRequestHeader()` to do just that. The basic syntax is like so:

```
xhr.setRequestHeader("header-name", "header-value");
```

where *header-name* is a string for the header to transmit and *header-value* a string for the corresponding value. Both standard and custom headers can be set with this method. Following HTTP conventions, when setting custom headers, the header would typically be prefixed by an "X-". For example, here a header that indicates the JavaScript transport scheme used is set to show an XHR was employed.

```
xhr.setRequestHeader("X-JS-Transport", "XHR");
```

The `setRequestHeader()` method can be used multiple times and, when behaving properly, it should append values.

```
xhr.setRequestHeader("X-Client-Capabilities", "Flash");
xhr.setRequestHeader("X-Client-Capabilities", "24bit-color");

// Header should be X-Client-Capabilities: Flash, 24bit-color
```

As shown in the previous section, the most likely known HTTP headers, particularly the `Content-Type` header, will be needed when posting data.

```
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

This method is also useful with GET requests to set headers to influence cache control in browsers that inappropriately (or appropriately) cache XHR requests. This directive can be performed on the client side by setting `If-Modified-Since` HTTP request header to some date in the past, like so:

```
xhr.setRequestHeader("If-Modified-Since", "Wed, 15 Nov 1995 04:58:08 GMT");
```

This is just another common example of the `setRequestHeader()` method. We will explore cache control quite a bit in Chapter 6.

Given the previous discussion of custom headers, you might wonder what would happen if you try to add to or even change headers that maybe you shouldn't. For example, can the `Referer` header be changed to look like the request is coming from another location?

```
xhr.setRequestHeader("Referer", "http://buzzoff.ajaxref.com");
```

How about the `User-Agent` header? Or how about actions that might be useful, like adding other `Accept` header values? Unfortunately, you'll see in the next section that the belief that XHR support is the same in browsers is not quite all it is cracked up to be.

Request Header Headaches

According to the emerging `XMLHttpRequest` specification from the W3C, for security reasons, browsers are supposed to ignore the use of `setRequestHeader()` for the headers shown in Table 3-5.

Also, when setting the headers in Table 3-6, the values specified by `setRequestHeader()` should replace any existing values.

Finally, all other headers set via this method are supposed to add to the current value being sent, if defined, or create a new value if not defined. For example, given:

```
xhr.setRequestHeader("User-Agent", "Ajax Browser ");
```

data should be added to the existing `User-Agent` header, not replace it.

While the specification may indicate one thing, the actual support in browsers for setting headers seems to be, in a word, erratic. For example, the `Referer` header is sent in XHR requests by Internet Explorer, Safari, and Opera, but it is not settable as per the specification by these browsers. However, some versions of Firefox (1.5) do not send the header normally and allow you to set it. For other headers, the situation may be the opposite, with Firefox

Accept-Charset	Date	TE
Accept-Encoding	Host	Trailer
Content-Length	Keep-Alive	Transfer-Encoding
Expect	Referer	Upgrade

TABLE 3-5 `setRequestHeader` Values That Should Be Ignored

`Authorization`	`Delta-Base`	`If-Unmodified-Since`
`Content-Base`	`Depth`	`Max-Forwards`
`Content-Location`	`Destination`	`MIME-Version`
`Content-MD5`	`ETag`	`Overwrite`
`Content-Range`	`From`	`Proxy-Authorization`
`Content-Type`	`If-Modified-Since`	`SOAPAction`
`Content-Version`	`If-Range`	`Timeout`

Table 3-6 `setRequestHeader` Values That Should Replace Existing Values

conforming or coming close and the others not doing so. Figure 3-3 shows the results of testing the common browsers at this edition's writing; the complete results can be found at http://ajaxref.com/ch3/requestexplorerresults.php.

Very likely, this situation is going to change as browser vendors start shoring up the details and inconsistencies when developers start really exercising XHRs. Rather than rely on results at one point in time, run the script at http://ajaxref.com/ch3/requestexplorerscript.html yourself. By doing so, you help keep the chart automatically updated until these details are worked out by the browser vendors. You may also find it useful to use a browser HTTP debugging tool or run the Request Explorer at http://ajaxref.com/ch3/requestexplorer.php to experiment with header values.

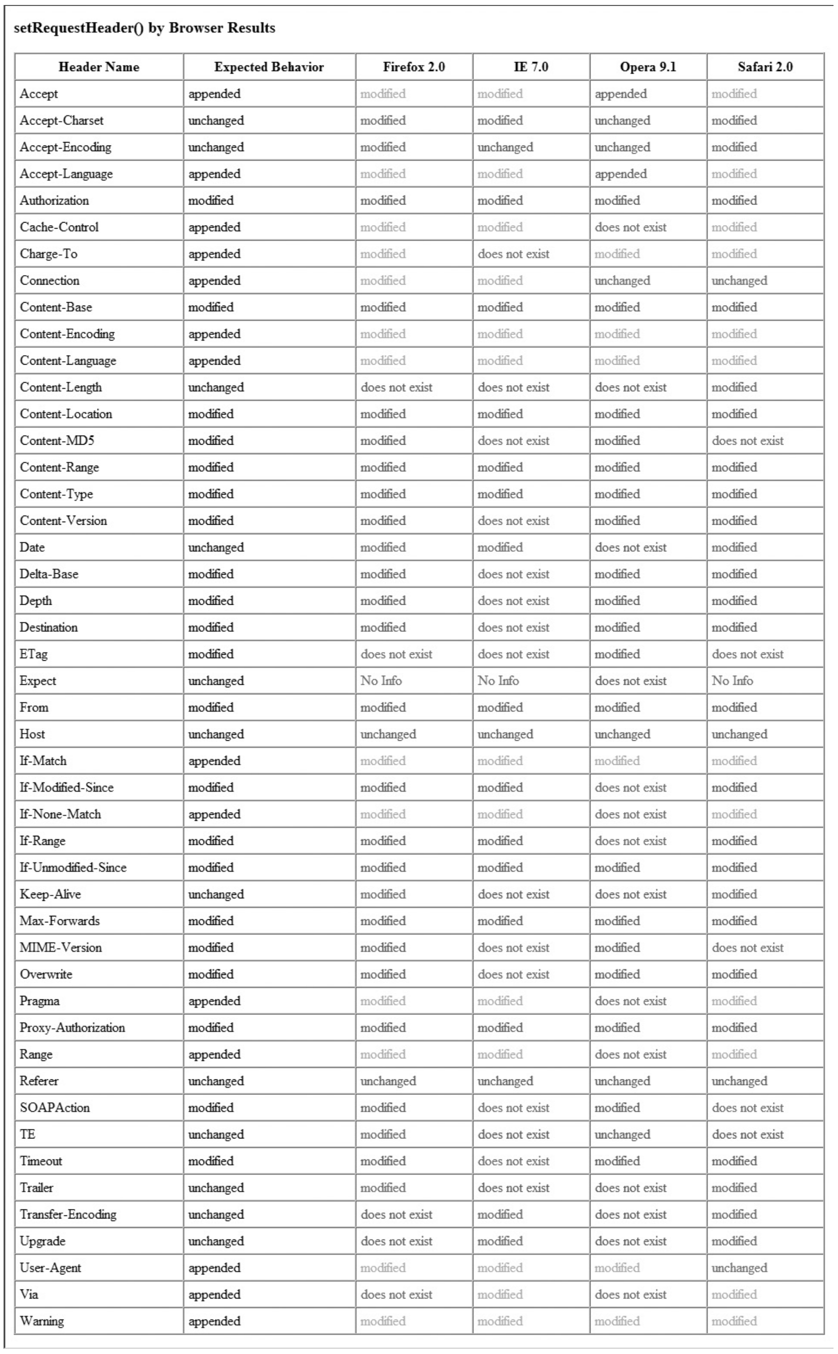
Other HTTP Requests

While most of time, GET and POST will be used in Ajax communication, there is a richer set of HTTP methods that can be used. For security reasons, many of these may be disabled on your server. You may also find that some methods are not supported in your browser, but the first request method, HEAD, should be available in just about any case.

Head Requests

The HTTP HEAD method is used to check resources. When making a HEAD request of an object, only the headers are returned. This may be useful to check for the existence for something, the size of something, or to see if it has been recently updated before committing to fetch or use the resource. Syntactically, there isn't much to do differently versus previous examples save setting the method differently, as shown here:

```
var url = "http://ajaxref.com/ch3/headrequest.html";
var xhr = createXHR();

if (xhr)
  {
    xhr.open("HEAD", url, true);
    xhr.onreadystatechange = function(){handleResponse(xhr);};
    xhr.send(null);
  }
```



setRequestHeader() by Browser Results

Header Name	Expected Behavior	Firefox 2.0	IE 7.0	Opera 9.1	Safari 2.0
Accept	appended	modified	modified	appended	modified
Accept-Charset	unchanged	modified	modified	unchanged	modified
Accept-Encoding	unchanged	modified	unchanged	unchanged	modified
Accept-Language	appended	modified	modified	appended	modified
Authorization	modified	modified	modified	modified	modified
Cache-Control	appended	modified	modified	does not exist	modified
Charge-To	appended	modified	does not exist	modified	modified
Connection	appended	modified	modified	unchanged	unchanged
Content-Base	modified	modified	modified	modified	modified
Content-Encoding	appended	modified	modified	modified	modified
Content-Language	appended	modified	modified	modified	modified
Content-Length	unchanged	does not exist	does not exist	does not exist	modified
Content-Location	modified	modified	modified	modified	modified
Content-MD5	modified	modified	does not exist	modified	does not exist
Content-Range	modified	modified	modified	modified	modified
Content-Type	modified	modified	modified	modified	modified
Content-Version	modified	modified	does not exist	modified	modified
Date	unchanged	modified	modified	does not exist	modified
Delta-Base	modified	modified	does not exist	modified	modified
Depth	modified	modified	does not exist	modified	modified
Destination	modified	modified	does not exist	modified	modified
ETag	modified	does not exist	does not exist	modified	does not exist
Expect	unchanged	No Info	No Info	does not exist	No Info
From	modified	modified	modified	modified	modified
Host	unchanged	unchanged	unchanged	unchanged	unchanged
If-Match	appended	modified	modified	modified	modified
If-Modified-Since	modified	modified	modified	does not exist	modified
If-None-Match	appended	modified	modified	does not exist	modified
If-Range	modified	modified	modified	does not exist	modified
If-Unmodified-Since	modified	modified	modified	modified	modified
Keep-Alive	unchanged	modified	does not exist	does not exist	modified
Max-Forwards	modified	modified	modified	modified	modified
MIME-Version	modified	modified	does not exist	modified	does not exist
Overwrite	modified	modified	does not exist	modified	modified
Pragma	appended	modified	modified	does not exist	modified
Proxy-Authorization	modified	modified	modified	modified	modified
Range	appended	modified	modified	does not exist	modified
Referer	unchanged	unchanged	unchanged	unchanged	unchanged
SOAPAction	modified	modified	does not exist	modified	does not exist
TE	unchanged	modified	does not exist	unchanged	does not exist
Timeout	modified	modified	does not exist	modified	modified
Trailer	unchanged	modified	does not exist	does not exist	modified
Transfer-Encoding	unchanged	does not exist	modified	does not exist	modified
Upgrade	unchanged	does not exist	modified	does not exist	modified
User-Agent	appended	modified	modified	modified	unchanged
Via	appended	does not exist	modified	does not exist	modified
Warning	appended	modified	modified	modified	modified

Figure 3-3 Browser `setRequest` Header Support Circa 2007

However, in the `handleResponse` function, it wouldn't be useful to look at the `responseText` or `responseXML` properties. Instead `getAllResponseHeaders()` or `getResponseHeader()` would be used to look at particular returned header values. These methods will be discussed shortly, but if you want to try a HEAD request, try http://ajaxref.com/ch3/head.html or use the Request Explorer (http://ajaxref.com/ch3/requestexplorer.php), which can reveal very interesting results.

Method Madness

The `XMLHttpRequest` specification indicates that user-agents supporting XHRs must support the following HTTP methods: GET, POST, HEAD, PUT, DELETE, and OPTIONS. However, it also states that they should support any allowable method. This includes the various WebDAV (www.webdav.org) methods such as MOVE, PROPFIND, PROPPATCH, MKCOL, COPY, LOCK, UNLOCK, POLL, and others. In theory, you might even have your own methods, though that wouldn't be safe on the Web at large as it would likely get filtered by caches or Web application firewalls encountered during transit. Even while avoiding anything too radical, testing methods beyond GET, POST, and HEAD with XHR in various browsers, the results were found to be a bit inconsistent.

Some browsers, like Opera and Safari, reject most extended methods, turning them into GETs if not understood or supported. This is very bad because it will potentially trigger server-side problems and produce totally unexpected behavior. In the case of Internet Explorer, it throws errors when trying to feed it methods it doesn't know. This is a more reasonable approach though, per the specification, it is still wrong. On the plus side, IE does support all the WebDAV methods, which are heavily used in Outlook Web Access. Firefox seems the closest to the emerging specification. It allows other methods, including WebDAV methods or even your own custom defined methods, though you'd obviously have to have a server with an ability to handle any custom created methods.

To see what your browser currently supports, we encourage readers to play with the Request Explorer at http://ajaxref.com/ch3/requestexplorer.php and shown in Figure 3-4. You can use it to set any type of method, header, and payload combination that may interest you.

Response Basics

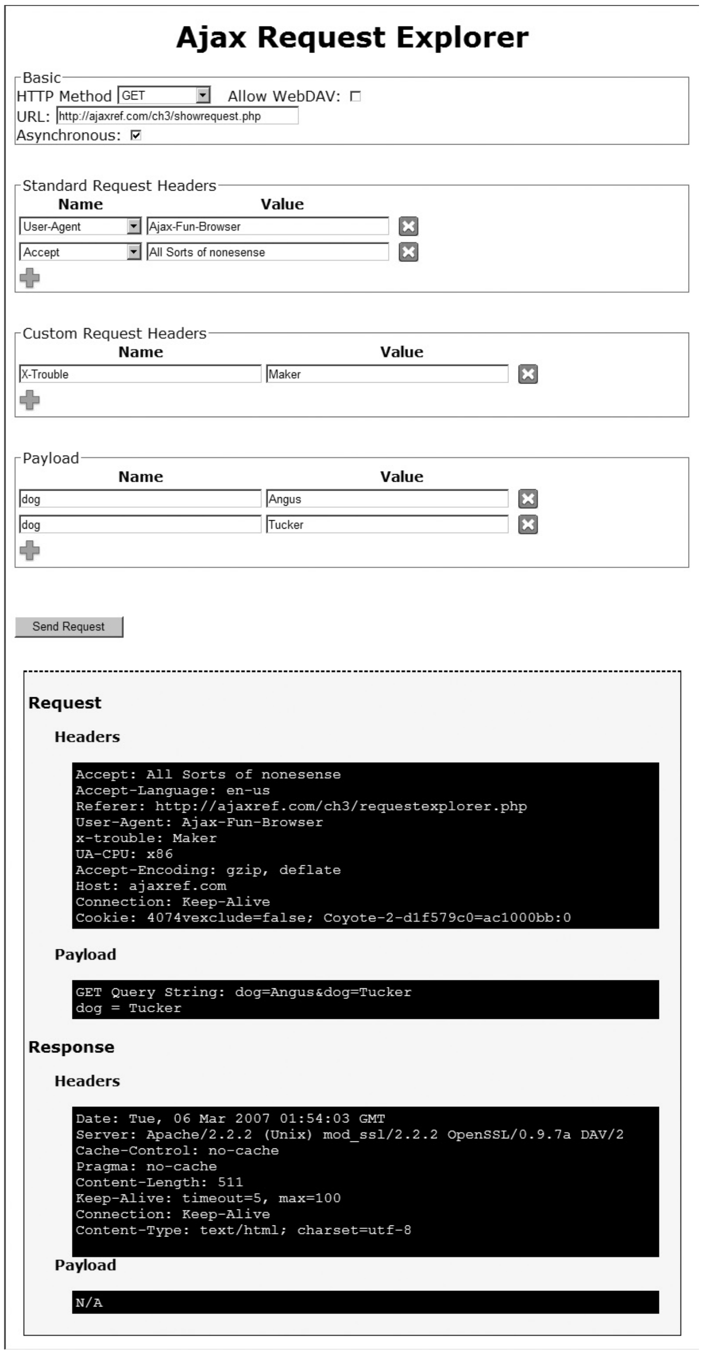
We have alluded to handling responses in order to demonstrate making requests with XHRs. However, this discussion has omitted a number of details, so we present those now.

readyState Revisited

As shown in the callback functions, the `readyState` property is consulted to see the state of an XHR request. The property holds an integer value ranging from 0–4 corresponding to the state of the communication, as summarized in Table 3-7.

It is very easy to test the `readyState` value moving through its stages as the callback function will be invoked every time the `readyState` value changes. In the following code, the value of the `readyState` property is displayed in alert dialog as the request goes along.

```
var url = "http://ajaxref.com/ch3/helloworld.php";
var xhr = createXHR();
if (xhr)
```


Ajax Request Explorer

Basic
HTTP Method: GET Allow WebDAV: ☐
URL: http://ajaxref.com/ch3/showrequest.php
Asynchronous: ☑

Standard Request Headers

Name	Value
User-Agent	Ajax-Fun-Browser
Accept	All Sorts of nonesense

Custom Request Headers

Name	Value
X-Trouble	Maker

Payload

Name	Value
dog	Angus
dog	Tucker

Send Request

Request

Headers

```
Accept: All Sorts of nonesense
Accept-Language: en-us
Referer: http://ajaxref.com/ch3/requestexplorer.php
User-Agent: Ajax-Fun-Browser
x-trouble: Maker
UA-CPU: x86
Accept-Encoding: gzip, deflate
Host: ajaxref.com
Connection: Keep-Alive
Cookie: 4074vexclude=false; Coyote-2-d1f579c0=ac1000bb:0
```

Payload

```
GET Query String: dog=Angus&dog=Tucker
dog = Tucker
```

Response

Headers

```
Date: Tue, 06 Mar 2007 01:54:03 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.7a DAV/2
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 511
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

Payload

```
N/A
```

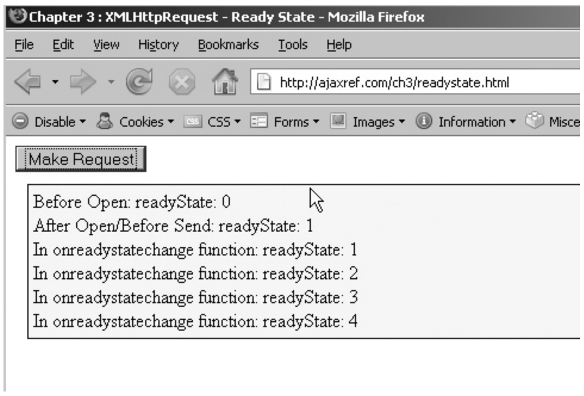
Figure 3-4 Exploring method support in XHR implementations

`readyState` Value	Meaning	Description
0	Uninitialized	The XHR has been instantiated, but the `open()` method has not been called yet.
1	Open	The XHR has been instantiated and the `open()` method called, but `send()` has not been invoked.
2	Sent	The `send()` method has been called, but no headers or data have been received yet.
3	Receiving	Some data has been received. Looking at headers or content. This phase of loading may cause an error in some browsers and not in others.
4	Loaded	All the data has been received and can be looked at. Note that the XHR may enter this state in abort and error conditions.

Table 3-7 `readyState` Values

```
  {
    alert("Before open method: readyState: " + xhr.readyState );
    xhr.open("GET",url,true);
    xhr.onreadystatechange = function(){alert("In onreadystatechange
function: readyState: " + xhr.readyState);};
    xhr.send(null);
  }
```

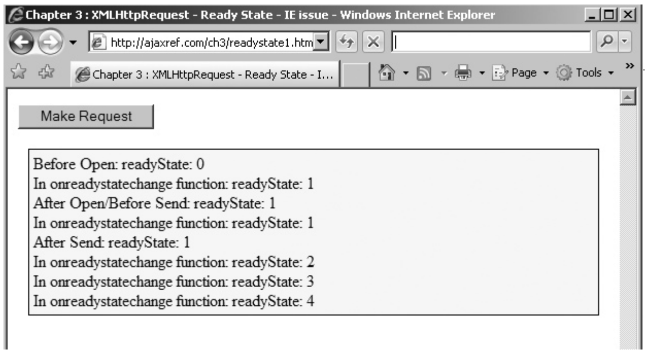
The alert is useful as it blocks the progress of the request so you can watch the process closely. However, if you want to see the progress of a request in a more real-time style, try the example at http://ajaxref.com/ch3/readystate.html, which is displayed here:



```
Before Open: readyState: 0
After Open/Before Send: readyState: 1
In onreadystatechange function: readyState: 1
In onreadystatechange function: readyState: 2
In onreadystatechange function: readyState: 3
In onreadystatechange function: readyState: 4
```

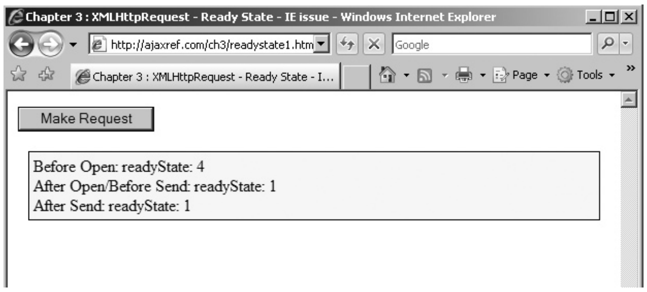
Like many of the details of XHRs `readyState`, values can be a bit quirky depending on the code and browser. For example, mysteriously, a `readyState` value of 2 may not be seen in Opera browsers, at least in version 9 or before. Moving the position of the `onreadystatechange` assignment, very different results will be experienced. Most of these are relatively harmless, save one in Internet Explorer that will break the object's functionality. To demonstrate this, first use a global XHR and set the `onreadystatechange` in the wrong place, in this case before the `open()` method. The `readyState` for the XHR object will not work properly on the second and subsequent uses, as demonstrated here:

Run: 1


Chapter 3 : XMLHttpRequest - Ready State - IE issue - Windows Internet Explorer
http://ajaxref.com/ch3/readystate1.htm
Make Request

```
Before Open: readyState: 0
In onreadystatechange function: readyState: 1
After Open/Before Send: readyState: 1
In onreadystatechange function: readyState: 1
After Send: readyState: 1
In onreadystatechange function: readyState: 2
In onreadystatechange function: readyState: 3
In onreadystatechange function: readyState: 4
```

Run: 2


Chapter 3 : XMLHttpRequest - Ready State - IE issue - Windows Internet Explorer
http://ajaxref.com/ch3/readystate1.htm
Make Request

```
Before Open: readyState: 4
After Open/Before Send: readyState: 1
After Send: readyState: 1
```

Interestingly, the various quirks of the `readyState` value are rarely felt in practice since most folks are looking solely for the final 4 value. However, you'll see later in the chapter that it is actually possible in some browsers and situations to look at data as it is loaded as opposed to waiting for the final `readyState` value to be reached.

`readyState` Needs Time to Change

One particularly important aspect of asynchronous communication related to the `onreadystatechange` functionality is the simple fact that in browser-based JavaScript, callback functions cannot be invoked until the script interpreter has a free moment to do so. There is no suspend and resume aspect to JavaScript execution in the typical single-threaded style implemented in Web browsers, so if you make a request and then enter into heavy calculations or other blocking activity, control will not be handed back long enough for the interpreter to invoke the `readyState` value change and the callback function will not be invoked. You can prove this to yourself by trying to run the example at http://ajaxref.com/ch3/longprocess.html, but given the difficulty in forcing the issue you might rather watch a movie that shows this situation in action at http://ajaxref.com/ch3/longprocessmovie.html.

With the increased interest in JavaScript from Ajax, we may see the eventual introduction of thread control or features like continuations that may allow for a more suspend-interrupt-continue style of coding. For the moment, however, you should be mindful that you may have to wait to get your data until your browser has a moment to deal with it.

status and statusText

After the `readyState` value has indicated that some headers have been received, the next step is to look at the success or failure of the response by looking at the XHR's status and `statusText` properties. The `status` property will contain the numeric HTTP status value such as 200, 404, 500, and so on, while the `statusText` property will contain the corresponding message or reason text like "OK", "Not Found", "Unavailable", "No Data", and so on.

Very often, the use of these values in Ajax applications is a bit rudimentary, usually looking to make sure that the XHR's response status value is 200 (as in 200 OK) and in all other cases failing, like so:

```
function handleResponse(xhr)
{
  if (xhr.readyState == 4  && xhr.status == 200)
    {
     // consume the response
    }
}
```

However, you might also opt to add more intelligence to your Ajax application based upon the status value. For example, given some errors like a 503 "Service Unavailable" returned when a server is busy, you might decide to automatically retry the request for the user after some time period. You also may find that some status values suggest letting the user know what exactly is happening rather than just raising an exception with a vague message about "Request Failed" as seen in some examples online. To restructure the callback function, you might first check for `readyState` and then carefully look at status values, like so:

```
function handleResponse(xhr)
{
  if (xhr.readyState == 4)
    {
      try {
        switch (xhr.status)
         {
          case 200: // consume response
                    break;
          case 403:
          case 404: // error
                    break;
          case 503: // error but retry
                    break;
          default: // error
        }
     }
     catch (e) { /* error */ }
  }
}
```

Yet, as you'll see next, even if you are very aware of typical HTTP status codes, it may not be enough under some conditions.

Unusual Status Values

As we are always reminded, any number of network problems can arise on the Internet. Some implementations of XHRs provide odd status values that can be useful under such extreme conditions. Internet Explorer implements the status values shown in Table 3-8, which are useful to detect error conditions.

You should note that these status codes do not relate to any standard HTTP status codes and are actually used to indicate TCP level problems as reported back to WinInet, which is used by Internet Explorer to drive XHRs. These values give the application level programmer some insight into what is going on with the connection so they can decide to handle things gracefully. You can see http://msdn2.microsoft.com/en-us/library/aa385465.aspx for a complete list of these codes, but Table 3-8 presents what you will likely encounter in practice.

Standard or not, IE's unusual codes seem useful, but what about other browsers—how do they react to network problems? Consider, for example, what happens if a connection doesn't go through due to the server being down or some problem with the network route. Firefox will eventually call `onreadystatechange` and even set the state to 4, but checking the value of status will raise an exception. Opera will also set `readyState` to 4, but the status will have a value of 0. IE will set `readyState` to 4 as well, but will inform us with the status code of 12029 what happened. You could add a try-catch block to deal with this problem.

```
if (xhr.readyState == 4)
  {
   try {
    if (xhr.status == 200)
       {
         // consume response
       }
   }
  catch (e) {alert("Network error");}
  }
```

Those are detected easily enough; use try and catch to handle the exception.

IE `status` Property Value	Corresponding `statusText` Value
12002	ERROR_INTERNET_TIMEOUT
12007	ERROR_INTERNET_NAME_NOT_RESOLVED
12029	ERROR_INTERNET_CANNOT_CONNECT
12030	ERROR_INTERNET_CONNECTION_ABORTED
12031	ERROR_INTERNET_CONNECTION_RESET
12152	ERROR_HTTP_INVALID_SERVER_RESPONSE

TABLE 3-8 Internet Explorer Special **`status`** Values

Other situations might not be so easy. What happens if there is a network problem or server crash midrequest? Internet Explorer will inform you of such problems with a status value of 12152 or potentially 12031, but browsers like Firefox may report things incorrectly, particularly if some headers have come back already. There may even be a 200 code sitting in the `status` property and a `readyState` of 4 with no reasonable data to work with!

If the server disconnects and other errors can result in 200 status codes, it would seem quite difficult to handle things under edge cases. How do you really know an Ajax request is successful if such cases are possible? You could try to look to see if there is content in `responseText` and inspect it very carefully with appropriate `try-catch` blocks.

204 Status Quirks and Beyond

The use of 204 No Data responses can be quite useful in applications that just "ping" a server and don't necessarily need a response with data. While the use of this type of response is common in traditional JavaScript communication patterns with XHRs, there are some troubling quirks. For example, in Opera you will have a 0 status and may not invoke `onreadystatechange` properly, while in Internet Explorer you will receive the odd status value of 1223. Like much of what you have seen in this chapter, when it comes to details you shouldn't take much for granted. To explore how your browser reacts to various status codes, use the Request Explorer on the book support site and enable "Force Status." You won't get any data back, but you will be able to evaluate the headers and `readyState` values.

responseText

The `responseText` property holds the raw text of a response body, not including any headers. Despite the name suggesting differently, XHRs are actually neutral to data format. Just about anything can be passed back and held in this property from plain text to XHTML fragments to comma separated values, JavaScript or, as some have demonstrated, even encoded binary style data. The example http://ajaxref.com/ch3/responsetextmore.html, shown in Figure 3-5, proves this point as it provides a way to receive the same response in a variety of formats.

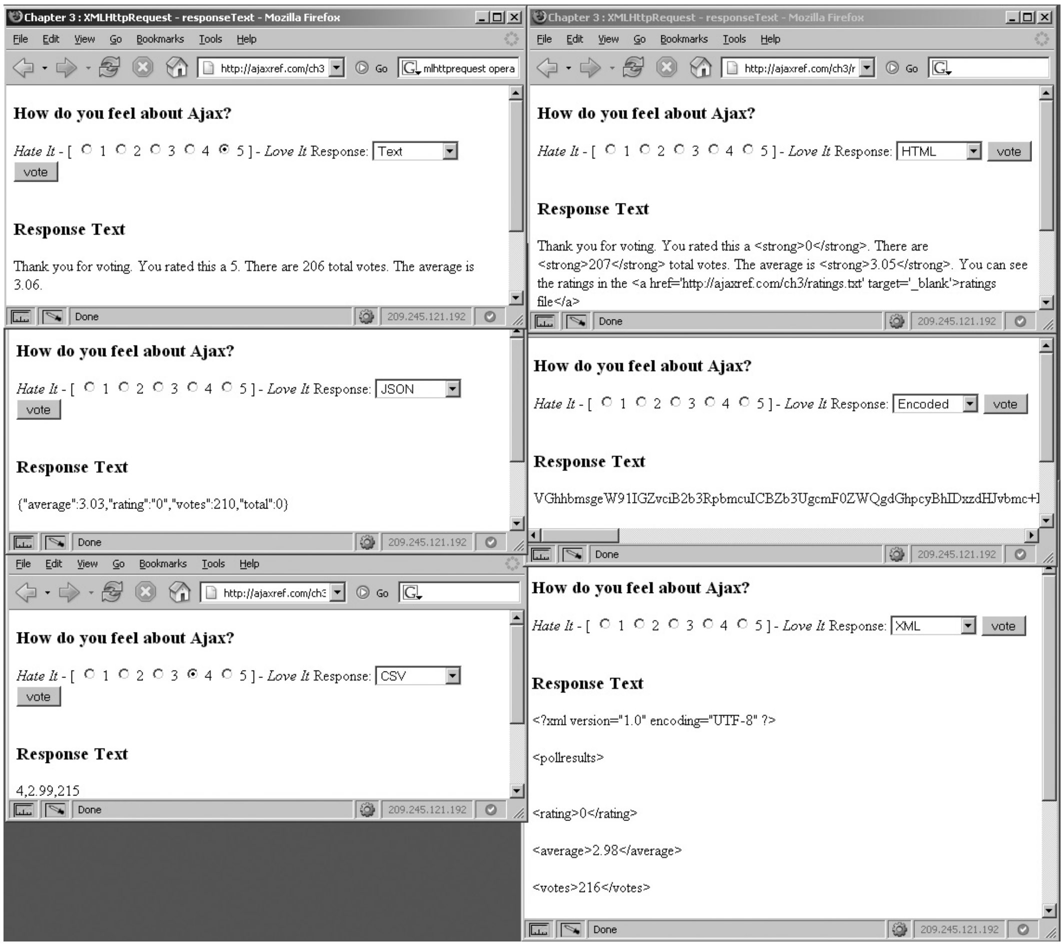
We'll look at the particulars of data formats used in Ajax in the next chapter, but for now the main point to take away is that `responseText` holds the raw unprocessed response from the server, which could be just about any text format you can dream up.

Another interesting aspect to the `responseText` property is that it can be polled continuously as data is received and that data can be utilized before it is complete in some browsers. Since this is not supported everywhere, this is discussed in the section entitled "onProgress and Partial Responses" later in the chapter when we discuss the proprietary, emerging, and inconsistently supported features of XHRs.

NOTE *While Ajax is somewhat neutral on data type, it is not on character set. UTF-8 is the default character encoding in most XHR implementations.*

responseXML

While `responseText` is a very flexible property, there is a special place for XML in the heart of `XMLHttpRequest` objects: the `responseXML` property. The idea with this property is that when a request is stamped with a MIME type of text/xml, the browser will go ahead and parse the content as XML and create a `Document` object in the object that is the parse tree of


Chapter 3 : XMLHttpRequest - responseText - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://ajaxref.com/ch3
Go
Done
209.245.121.192

How do you feel about Ajax?

Hate It - [○ 1 ○ 2 ○ 3 ○ 4 ● 5] - *Love It* Response: Text vote

Response Text

Thank you for voting. You rated this a 5. There are 206 total votes. The average is 3.06.

How do you feel about Ajax?

Hate It - [○ 1 ○ 2 ○ 3 ○ 4 ○ 5] - *Love It* Response: HTML vote

Response Text

Thank you for voting. You rated this a <strong>0</strong>. There are <strong>207</strong> total votes. The average is <strong>3.05</strong>. You can see the ratings in the <a href='http://ajaxref.com/ch3/ratings.txt' target='_blank'>ratings file</a>

How do you feel about Ajax?

Hate It - [○ 1 ○ 2 ○ 3 ○ 4 ○ 5] - *Love It* Response: JSON vote

Response Text

{"average":3.03,"rating":"0","votes":210,"total":0}

How do you feel about Ajax?

Hate It - [○ 1 ○ 2 ○ 3 ○ 4 ○ 5] - *Love It* Response: Encoded vote

Response Text

VGhhbmsgeW91IGZvciB2b3RpbmcuICBZb3UgcmF0ZWQgdGhpcyBhIDxzdHJvbmc+I

How do you feel about Ajax?

Hate It - [○ 1 ○ 2 ○ 3 ● 4 ○ 5] - *Love It* Response: CSV vote

Response Text

4,2.99,215

How do you feel about Ajax?

Hate It - [○ 1 ○ 2 ○ 3 ○ 4 ○ 5] - *Love It* Response: XML vote

Response Text

<?xml version="1.0" encoding="UTF-8" ?>

<pollresults>

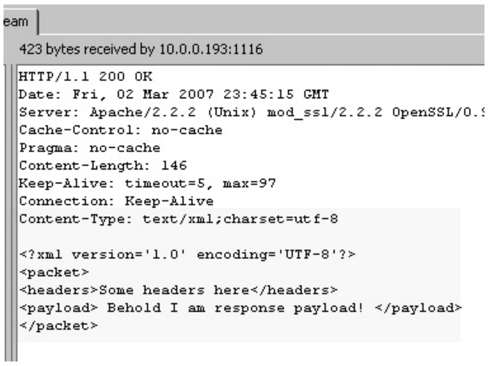
<rating>0</rating>

<average>2.98</average>

<votes>216</votes>

Figure 3-5 XHR's `responseText` property allows for a multitude of data formats

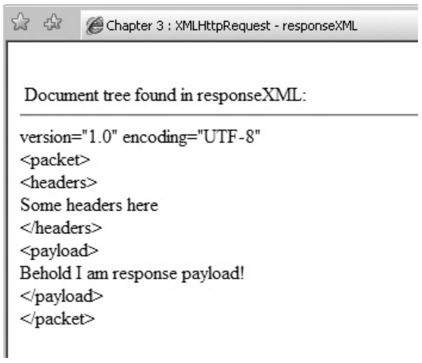
the returned markup. With most analysis tools, it is easy enough to see the raw XML text, or you can peak at the whole body by looking at `responseText`.



```
423 bytes received by 10.0.0.193:1116
HTTP/1.1 200 OK
Date: Fri, 02 Mar 2007 23:45:15 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 146
Keep-Alive: timeout=5, max=97
Connection: Keep-Alive
Content-Type: text/xml;charset=utf-8

<?xml version='1.0' encoding='UTF-8'?>
<packet>
<headers>Some headers here</headers>
<payload> Behold I am response payload! </payload>
</packet>
```

However, it is not so easy to see the parse tree so we show a simple example here of a walked `responseXML` parse tree output to the document.

You can access this example at http://ajaxref.com/ch3/responsexmlwalk.html.



Chapter 3 : XMLHttpRequest - responseXML

Document tree found in responseXML:

```
version="1.0" encoding="UTF-8"
<packet>
<headers>
Some headers here
</headers>
<payload>
Behold I am response payload!
</payload>
</packet>
```

Assuming there is a correctly MIME-stamped and well-formed XML packet, its DOM tree should be in the `responseXML` property, begging the question: how do you consume the response data? Very often, people will use DOM methods to extract bits and pieces of the content returned. The `document.getElementsByTagName()` method might be used to find a particular tag and extract its contents. For example, given a packet that looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<pollresults>
  <rating>4</rating>
  <average>2.98</average>
  <votes>228</votes>
</pollresults>
```

as the response payload, it is possible to extract the data items with the following code:

```
var xmlDoc = xhr.responseXML;
var average = xmlDoc.getElementsByTagName("average")[0].firstChild.nodeValue;
var total = xmlDoc.getElementsByTagName("votes")[0].firstChild.nodeValue;
var rating = xmlDoc.getElementsByTagName("rating")[0].firstChild.nodeValue;
```

Doing a straight walk of the `Document` tree is also an option if you understand its structure. In order to look for the average node in the previous example, you might walk directly to it with:

```
var average = xmlDoc.documentElement.childNodes[1].firstChild.nodeValue;
```

Of course, this type of direct walk is highly dangerous, especially if you consider that the DOM tree may be different in browsers, particularly Firefox, as it includes whitespace nodes in its DOM tree (http://developer.mozilla.org/en/docs/Whitespace_in_the_DOM). Normalizing responses to account for such a problem is a possibility, but frankly both of these approaches seem quite messy. JavaScript programmers familiar with the DOM should certainly wonder why we are not using the ever-present `document.getElementById()` method or some shorthand `$()` function, as provided by popular JavaScript libraries. The simple answer is, as it stands right now, you can't with an XML packet passed back to an XHR. The `id` attribute value is not supported automatically in an XML fragment. This attribute must be defined in a DTD or schema with the name `id` and type `ID`. Unless an `id` attribute of the appropriate type is known, a call to `document.getElementById()` method will return `null`. The sad situation is that, as of the time of this book's writing, browsers are not (at least by default) directly schema- or DTD-aware for XML data passed back from an XHR. To rectify this, it would be necessary to pass any XHR received XML data to a DOM parser and then perform selections using `document.getElementById`. Unfortunately, this cannot be done effectively in a cross-browser fashion, as will be demonstrated in Chapter 4. It is possible, however, to perform a hard walk of a tree looking for the attribute of interest, which certainly isn't elegant but will work. If you are looking for ease of node selection in XML, you might turn to related technologies like XPath to access returned data and XSLT to transform. This topic will be covered more in the next chapter, but for now note simply that there is more than a bit of work involved in handling XML data in many cases, thus the increased developed interest in text, HTML fragments, and JSON formatted data.

XML Challenges: Bad MIME Types

One important question that should come to mind when working with `responseXML` is what happens if the MIME type of the data returned is not `text/xml`? Does the browser populate the `responseXML` and, if so, can you safely look at it? Using a simple example that changes the MIME type on the returned packet, you can see that this is yet another example of browser variation. Most of the browsers will not parse the response unless it is stamped with `text/xml` or `application/xml`, though interestingly Opera will seem to attempt to parse just about anything you received, even something with a completely bogus MIME type.

If you attempt to look at `responseXML` after the data has loaded from a non-XML MIME type, what happens will vary by browser. Placing a simple `if` statement that looks for existence on the `responseXML` property will indicate a problem in Firefox, but not in the other browsers. A far better way to do things is to first look at the response header to make sure the `Content-type:` returned is appropriate. You may be tempted to do something as simple as:

```
if (xhr.getResponseHeader("Content-Type")  == "text/xml")
  {
    // use XML response
   }
```

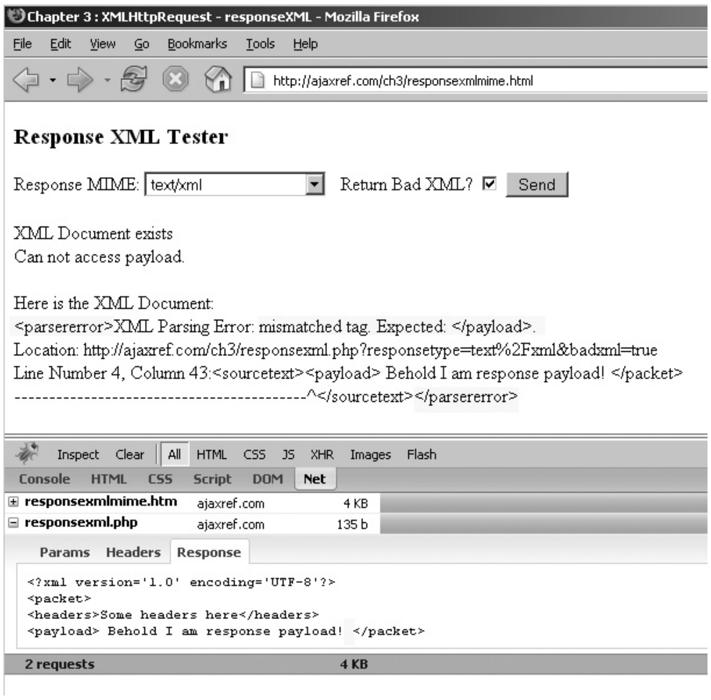
However, note that the returned MIME type may be more than `text/xml` and contain information about the character encoding used like so: `text/xml;charset=utf-8`. In this case you would probably need a statement more like this:

```
if (xhr.readyState == 4 && xhr.status == 200)
  {
    if (xhr.getResponseHeader("Content-Type").indexOf("text/xml") >= 0)
      {
        var xmlDoc = xhr.responseXML;
        // use XML response
      }
   }
```

If you also want to address `application/xml`, you will need to add further code. Yet even if the response is stamped correctly it says very little about if the content is well formed or valid.

XML Challenges: Well Formedness and Validity

If an XML packet is not well formed, meaning it doesn't follow XML's syntax rules such as not crossing elements, quoting attribute values, matching an element's name case, properly closing elements including empty elements, and addressing special characters, the `responseXML` value will not be populated with a DOM tree. However, looking in Firefox, you will find DOM nodes inside the `responseXML` property even in such a case because the


Chapter 3 : XMLHttpRequest - responseXML - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://ajaxref.com/ch3/responsexmlmime.html
Inspect Clear All HTML CSS JS XHR Images Flash
Console HTML CSS Script DOM Net
Params Headers Response

Response XML Tester

Response MIME: text/xml Return Bad XML? ☑ Send

XML Document exists
Can not access payload.

Here is the XML Document:
<parsererror>XML Parsing Error: mismatched tag. Expected: </payload>.
Location: http://ajaxref.com/ch3/responsexml.php?responsetype=text%2Fxml&badxml=true
Line Number 4, Column 43:<sourcetext><payload> Behold I am response payload! </packet>
--^</sourcetext></parsererror>

⊞ responsexmlmime.htm	ajaxref.com	4 KB	
⊟ responsexml.php	ajaxref.com	135 b	

```
<?xml version='1.0' encoding='UTF-8'?>
<packet>
<headers>Some headers here</headers>
<payload> Behold I am response payload! </packet>
```

2 requests 4 KB

FIGURE 3-6 responseXML parse tree in Firefox on error

parser returns an XML tree with a root node of `<parsererror>` that contains information about the parse error encountered, as shown in Figure 3-6.

Assuming no syntactical errors are made in the XML, you might desire to dive in and start using the data, but that begs yet another question: is the actual returned data valid? What this means is it is important to not only look to make sure that the various tags found in the response are syntactically well formed, but also whether they are used properly according to some defined Document Type Definition (DTD) or schema. Unfortunately, by default XHR objects do not validate the contents of the responses. This can be addressed by invoking a DOM parser locally in some browsers like Internet Explorer, but in others it isn't possible to validate at all, which eliminates some of the major value of using XML as a data transmission format. We will pursue this issue in greater detail when data formats are covered in the next chapter, but to explore all the variations of MIME types, well formedness, and validity now, you can use the example at http://ajaxref.com/ch3/xmlrequestexplorer.html.

XML Challenges and Benefits

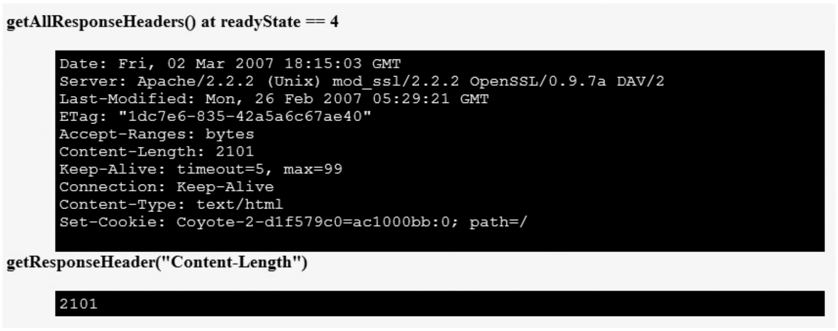
There are a number of other challenges facing those who wish to use XML as a response format even beyond what has been mentioned here. XML may be a bulkier format and need compression and for large data sets may have local parsing time consideration. Partial responses are pretty much out of the question when using XML, but obviously there aren't only downsides to the format. The various tools such as Xpath and XSLT to consume the received content are quite powerful. Further, the ability to generally validate the syntactical and semantic integrity of a received data packet is certainly quite appealing. Yet our goal in this chapter is primarily to focus on the XHR object itself, so let's return to that discussion directly.

Response Headers

XHRs have two methods to read response headers: `getResponseHeader(`*`headername`*`)` and `getAllResponseHeaders()`. As soon as the XHR has reached `readyState` 3, it should be possible to look at the response headers that have been returned by the server. Here are two simple examples:

```
xhr.getResponseHeader("Content-Length"); // fetches a single header
xhr.getAllResponseHeaders(); // fetches all the headers
```

Some possible values are shown next:



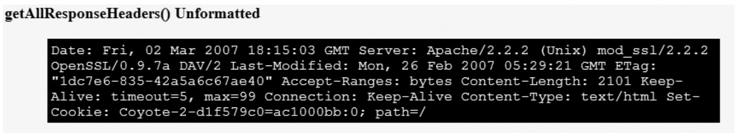
getAllResponseHeaders() at readyState == 4

```
Date: Fri, 02 Mar 2007 18:15:03 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.7a DAV/2
Last-Modified: Mon, 26 Feb 2007 05:29:21 GMT
ETag: "1dc7e6-835-42a5a6c67ae40"
Accept-Ranges: bytes
Content-Length: 2101
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html
Set-Cookie: Coyote-2-d1f579c0=ac1000bb:0; path=/
```

getResponseHeader("Content-Length")

```
2101
```

Both methods return strings, but note that in the case of multiple headers, the results will contain `\n` for newlines.



getAllResponseHeaders() Unformatted

```
Date: Fri, 02 Mar 2007 18:15:03 GMT Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2
OpenSSL/0.9.7a DAV/2 Last-Modified: Mon, 26 Feb 2007 05:29:21 GMT ETag:
"1dc7e6-835-42a5a6c67ae40" Accept-Ranges: bytes Content-Length: 2101 Keep-
Alive: timeout=5, max=99 Connection: Keep-Alive Content-Type: text/html Set-
Cookie: Coyote-2-d1f579c0=ac1000bb:0; path=/
```

If you plan on placing the headers in an XHTML page, you will have to convert the \n to break tags or use some other preformatting mechanism to output them nicely to the screen.

```
var allHeaders = xhr.getAllResponseHeaders();
allHeaders = allHeaders.replace(/\n/g, "<br/>");
```

Looking at edge cases, there are only minor variations in browsers. For example, attempting to fetch a header that does not exist with `getResponseHeader()`, may result in a slight difference in what is returned. Firefox returns null, while IE returns nothing. Given the loose typing system of JavaScript, this difference likely won't be noted. Both browsers agree what to do when you attempt to invoke these methods before headers are available: throw a JavaScript error.

Controlling Requests

The `XMLHttpRequest` object has fairly limited ability to control requests once they're sent outside the `abort()` method. This method provides the basic functionality of the stop button in the browser and will very likely be used in your Ajax applications to address network timeouts. For example, you might imagine that you can write a `cancelRequest()` function that will set a timer to be invoked after a particular period of time of nonresponsiveness from the server.

```
function sendRequest(url,payload)
{
 var xhr = createXHR();
 if (xhr)
   {
       xhr.open("GET",url + "?" + payload,true);
       xhr.onreadystatechange = function(){handleResponse(xhr);};
       xhr.send(null);
    }
// set timeout for 3 seconds
timeoutID = window.setTimeout( function() {cancelRequest(xhr);}, 3000);
}

function cancelRequest(xhr)
{
 xhr.abort();
 alert("Sorry, your request timed out.  Please try again later.");
}
```

Unfortunately, this won't work quite correctly because once the request is aborted, the `readyState` value will be set to 4 and the on`ReadyStateChange` handler will have to be called. There might be a partial response, or even an incorrect status message as mentioned in the previous sections, and then the on`ReadyStateChange` handler might inadvertently use it. To address this potential problem, there will likely need to be a flag to indicate if a request has been aborted. For example, as a simple demo, a global variable, `g_abort`, is created to indicate the abort status. After creating the XHR, it is set to false.

```
g_abort = false;
```

Within the request cancellation function, the abort flag is set to true for later use.

```
function cancelRequest(xhr)
{
  g_abort = true;
  /* we have to use this variable because after it aborts,
     the readyState will change to 4 */
  xhr.abort();
  alert("Sorry, your request timed out.  Please try again later.");
}
```

Now when `handleResponse` gets invoked because the `readyState` has changed, nothing is done based upon the `true` value of the abort flag.

```
function handleResponse(xhr)
{
  if (!g_abort)
   {
    if (xhr.readyState == 4)
     {
       clearTimeout(timeoutID);  // don't want to timeout accidentally
       switch (xhr.status)
```

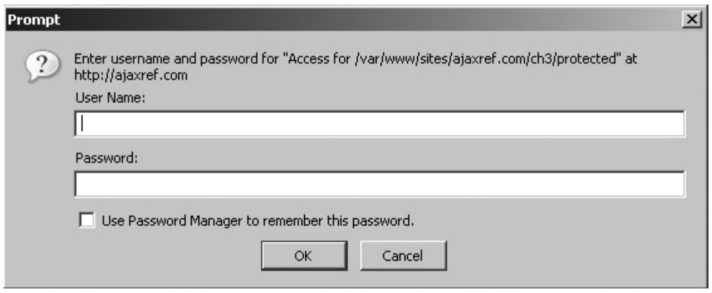
```
         {
          // handle response
         }
      }
    }
}
```

To see this idea in action, visit the example at http://ajaxref.com/ch3/abort.html. We will cover techniques for handling timeouts and other network problems in much more depth in Chapter 6.

NOTE *It is not really correct to manually force a connection to go away by nulling out various aspects of the request object or callback function or overwriting an existing XHR with another request. It may have a similar feeling to the end user in some cases because a callback won't happen, but any changes may still happen on the server as the request, once sent, still happened.*

Authentication with XHRs

In the course of building applications, you often want to restrict access to certain resources, such as particular directories or files. A simple form of authentication called HTTP Basic Authentication may be implemented on the Web server resulting in a browser challenging a user like so:



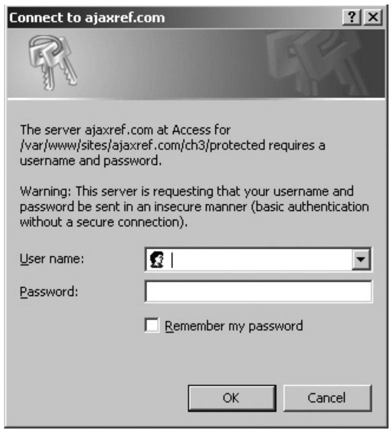
Prompt

Enter username and password for "Access for /var/www/sites/ajaxref.com/ch3/protected" at http://ajaxref.com

User Name:

Password:

☐ Use Password Manager to remember this password.



Connect to ajaxref.com

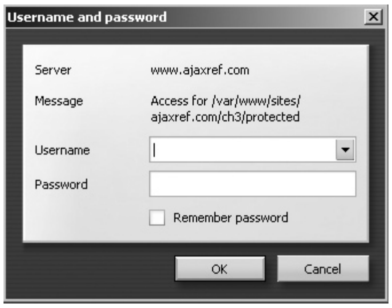
The server ajaxref.com at Access for /var/www/sites/ajaxref.com/ch3/protected requires a username and password.

Warning: This server is requesting that your username and password be sent in an insecure manner (basic authentication without a secure connection).

User name:

Password:

☐ Remember my password



Username and password

Server	www.ajaxref.com
Message	Access for /var/www/sites/ajaxref.com/ch3/protected
Username	
Password	

☐ Remember password

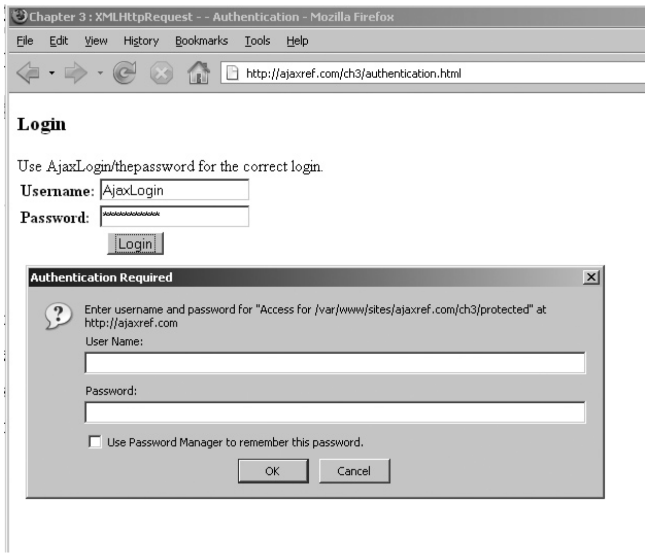
The `XMLHttpRequest` object supports HTTP authentication in that it allows specification of a username and password in the parameters passed to the `open()` method.

```
xhr.open("GET", "bankaccount.php", true, "drevil", "onemillion$");
```

Of course, you will need to make sure that such a request runs over SSL if you are worried about password sniffing during the transmission. Furthermore, you wouldn't likely hardcode such values in a request, but rather collect this data from a user via a Web form.

Interestingly, while the `open()` method accepts credentials passed via parameter, those credentials are not automatically sent to the server upon first request in all browsers. Opera sends it this way. Internet Explorer does not and waits until the server challenges the client for credentials with a `401 - Access Denied` response code. You can see that in the communication trace presented in Figure 3-7. Otherwise, Internet Explorer 7 acts just as you would expect and does not throw any user prompts regardless of correctness or incorrectness of authentication attempt. Other browsers like Opera and Firefox may not act so graceful when authentication fails; they may present the browser's normal challenge dialogs to the user despite the authentication being handled by an XHR. However, in all cases, once the authentication is verified in whatever manner, the `onreadystatechange` function gets called with `readyState` equal to 4 as expected.

There may also be a variety of problems in browsers even with successful authentication tries. Numerous older versions of Opera and Firefox and, in some cases, newer versions did throw user challenges up even on successful tries, which defeats the whole purpose of using this method. Yet in other installations and operating system combinations, they did not exhibit such problems.


Chapter 3 : XMLHttpRequest - - Authentication - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://ajaxref.com/ch3/authentication.html

Login

Use AjaxLogin/thepassword for the correct login.

Username: AjaxLogin

Password: ***********

Login
Authentication Required

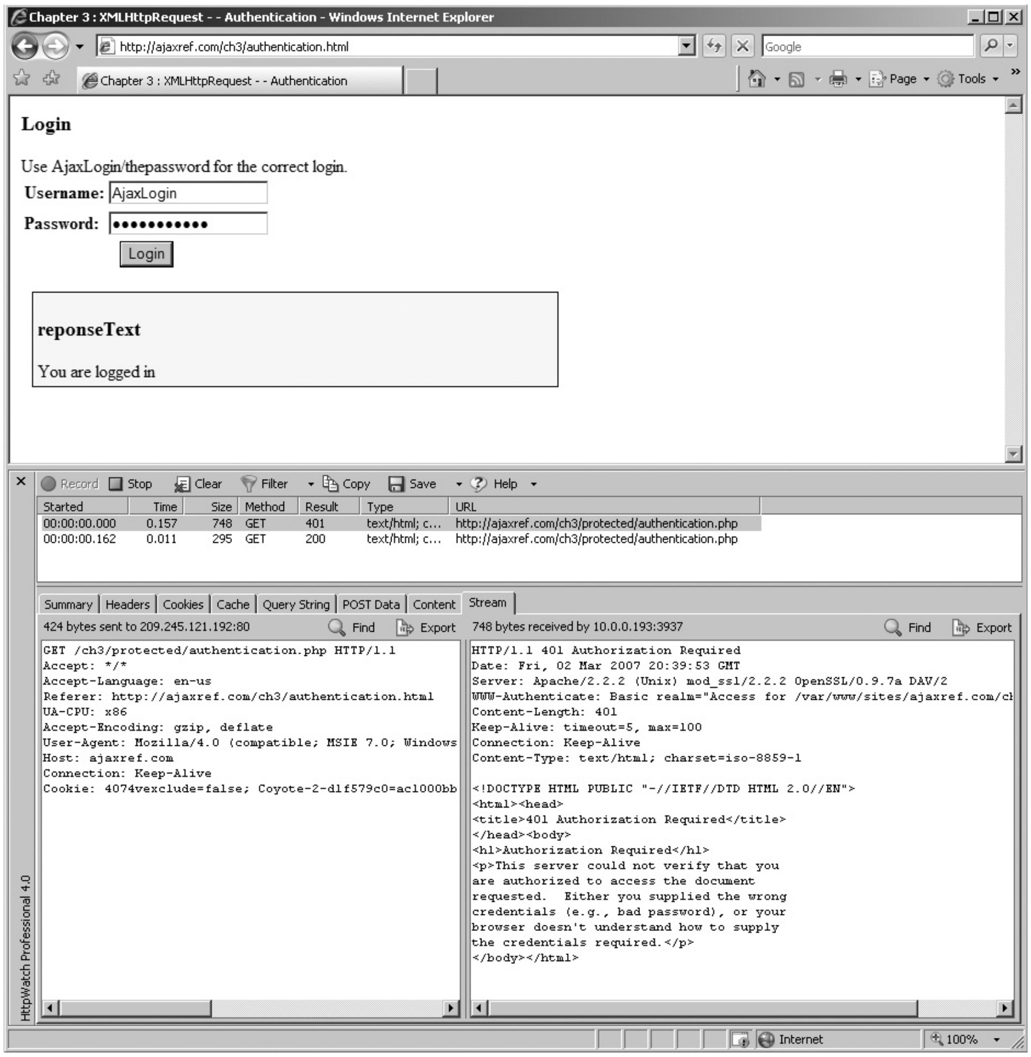
Enter username and password for "Access for /var/www/sites/ajaxref.com/ch3/protected" at http://ajaxref.com

User Name:

Password:

Use Password Manager to remember this password.
OK
Cancel

Given the inconsistency of how HTTP authentication is handled in XHRs, you are advised to avoid it and use your own form of user credential checking. However, if for some reason you must use it, you should thoroughly test the state of authentication support in browsers yourself by running the code at http://ajaxref.com/ch3/authentication.html.



Login

Use AjaxLogin/thepassword for the correct login.

Username: AjaxLogin

Password: ●●●●●●●●●●●

reponseText

You are logged in

Started	Time	Size	Method	Result	Type	URL
00:00:00.000	0.157	748	GET	401	text/html; c...	http://ajaxref.com/ch3/protected/authentication.php
00:00:00.162	0.011	295	GET	200	text/html; c...	http://ajaxref.com/ch3/protected/authentication.php

424 bytes sent to 209.245.121.192:80

```
GET /ch3/protected/authentication.php HTTP/1.1
Accept: */*
Accept-Language: en-us
Referer: http://ajaxref.com/ch3/authentication.html
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows
Host: ajaxref.com
Connection: Keep-Alive
Cookie: 4074vexclude=false; Coyote-2-d1f579c0=ac1000bb
```

748 bytes received by 10.0.0.193:3937

```
HTTP/1.1 401 Authorization Required
Date: Fri, 02 Mar 2007 20:39:53 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.7a DAV/2
WWW-Authenticate: Basic realm="Access for /var/www/sites/ajaxref.com/ch
Content-Length: 401
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>401 Authorization Required</title>
</head><body>
<h1>Authorization Required</h1>
<p>This server could not verify that you
are authorized to access the document
requested.  Either you supplied the wrong
credentials (e.g., bad password), or your
browser doesn't understand how to supply
the credentials required.</p>
</body></html>
```

Chapter 3 : XMLHttpRequest - - Authentication - Windows Internet Explorer
http://ajaxref.com/ch3/authentication.html
Google
Chapter 3 : XMLHttpRequest - - Authentication
Page
Tools
Login
Record
Stop
Clear
Filter
Copy
Save
Help
Summary
Headers
Cookies
Cache
Query String
POST Data
Content
Stream
Find
Export
HttpWatch Professional 4.0
Internet
100%

FIGURE 3-7 Internet Explorer XHR authentication communication trace

Propriety and Emerging XHR Features

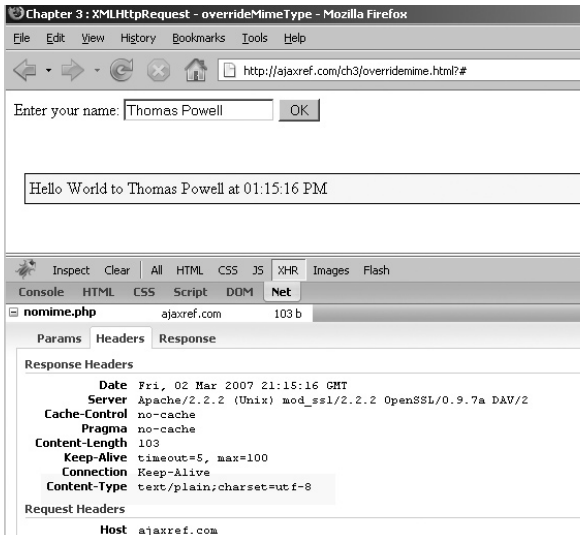
Given the intense interest in Ajax, the `XMLHttpRequest` object is starting to be exercised a great deal more than it has been in the past. Admittedly, the object is missing useful features and lacks some capabilities to deal with common problems with the network or received content. Without a strong specification, the browser vendors are adding various innovations to the object at a furious pace. It is pretty likely that this section will not cover all the features that may have been added by the time you read this, but we cover those that are currently implemented in shipping or prerelease browsers, and later in the chapter point out what is likely to come.

Managing MIME Types

It is very important for Ajax applications that any called server-side code correctly set the MIME type of the returned data. You must always remember if the XHR object receives a data stream with a `Content-type:` header not set to `text/xml` it shouldn't try to parse and populate the `responseXML` property. If that happens and you go ahead and try to access that property anyway and perform DOM manipulations, you will raise a JavaScript exception. If content is being retrieved that is truly a particular MIME type (like `text/xml)` and for some reason can't be set properly server-side, it is possible to rectify this in Firefox and Opera by using the `overrideMimeType()` method. Usage is fairly simple; set this method to indicate the desired MIME type before sending the request, and it will always treat the response as the MIME type specified, regardless of what it is. This is demonstrated here:

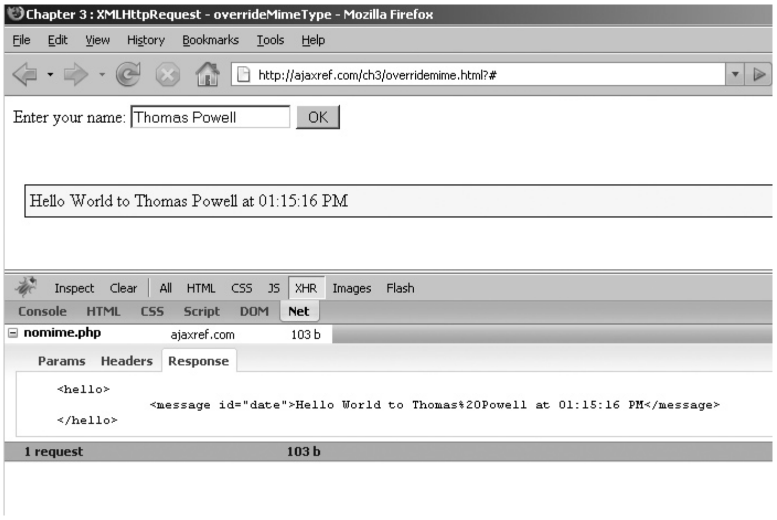
```
var xhr = createXHR();
if (xhr)
  {
    xhr.open("GET", url, true);
    xhr.overrideMimeType("text/xml");
    xhr.onreadystatechange = function(){handleResponse(xhr);};
    xhr.send(null);
   }
```

The communications trace here shows that the browser is passed content with format `text/plain` that is then overriden to `text/xml` so that it is parsed.


Chapter 3 : XMLHttpRequest - overrideMimeType - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://ajaxref.com/ch3/overridemime.html?#
Enter your name: Thomas Powell OK
Inspect Clear All HTML CSS JS XHR Images Flash
Console HTML CSS Script DOM Net
Params Headers Response

Hello World to Thomas Powell at 01:15:16 PM

```
nomime.php          ajaxref.com      103 b

Response Headers
          Date  Fri, 02 Mar 2007 21:15:16 GMT
        Server  Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.7a DAV/2
 Cache-Control  no-cache
        Pragma  no-cache
Content-Length  103
    Keep-Alive  timeout=5, max=100
    Connection  Keep-Alive
  Content-Type  text/plain;charset=utf-8
Request Headers
          Host  ajaxref.com
```


Chapter 3 : XMLHttpRequest - overrideMimeType - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://ajaxref.com/ch3/overridemime.html?#
Enter your name:
Thomas Powell
OK
Inspect Clear All HTML CSS JS XHR Images Flash
Console HTML CSS Script DOM Net
nomime.php ajaxref.com 103 b
Params Headers Response
1 request 103 b

Hello World to Thomas Powell at 01:15:16 PM

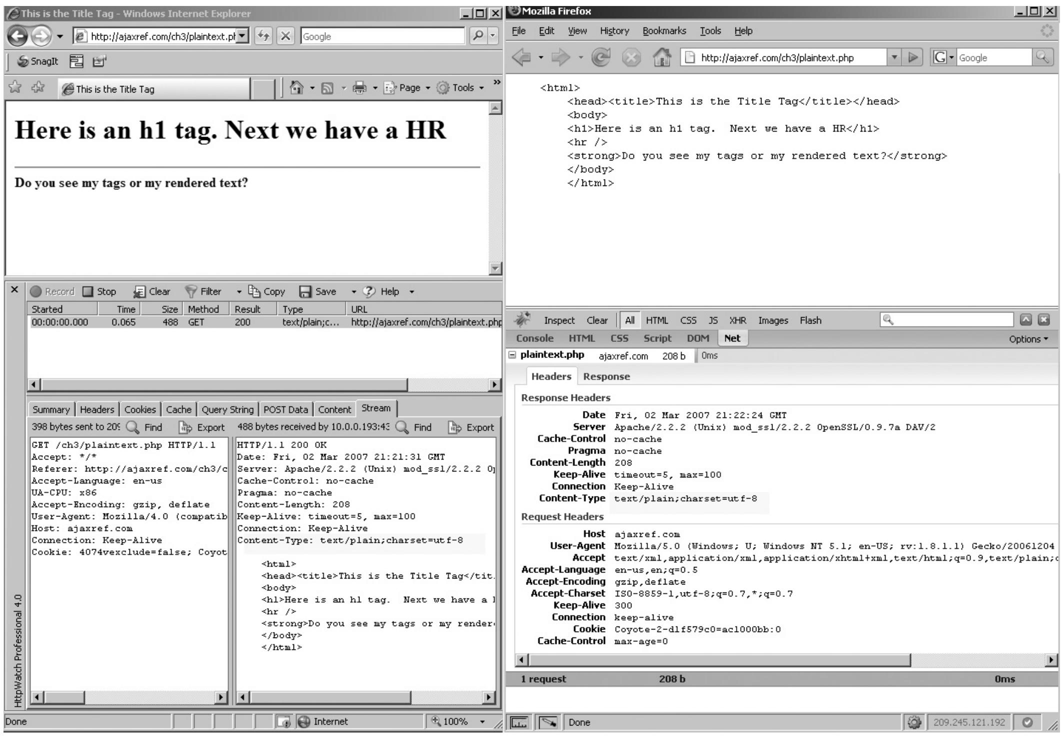
```
<hello>
            <message id="date">Hello World to Thomas%20Powell at 01:15:16 PM</message>
</hello>
```

You might wonder about the value of such a method given that typically you will be responsible for forming the data packet to be consumed by the client-side JavaScript. Sorry to say, proper MIME type usage is not something that many server-side developers have paid enough attention to. The main reason for this is that browsers, particularly Internet Explorer, are a bit too permissive in what they do with incorrect MIME types, so developers often are not forced to get the details right. Internet Explorer will often flat out ignore MIME types, instead peeking inside the response packet to decide what it is and favoring that over any `Content-type` header value encountered. As an example, you can serve a file as `text/plain`, but if you have some HTML tags in the first few lines of the file, Internet Explorer will happily render it as HTML, while more conformant browsers will not and display the file properly as text. You can see this in Figure 3-8.

Setting MIME types incorrectly on a Web server or in programs has lead to numerous "works in browser X but not in browser Y" errors that the author has observed, including something as common as Flash content being handled differently in various browsers. Readers are encouraged to get this particular detail right in the server side of their Ajax application to avoid headaches and the need for methods like `overrideMimeType()`. An `overrideMimeType()` example can be found at http://ajaxref.com/ch3/overridemime.html.

Multipart Responses

Some browsers like Firefox support an interesting property called `multipart` that allows you to handle responses that come in multiple pieces. Traditionally this format was used in an ancient Web idea called *server push*, where data was continuously streamed from the Web server and the page was updated. In the early days of the Web, this type of feature was used to display changing images, simple style video, and other forms of ever-changing data. Today you still see the concept employed in Webcam pages where images refresh continuously.


This is the Title Tag - Windows Internet Explorer
http://ajaxref.com/ch3/plaintext.php
Mozilla Firefox
File Edit View History Bookmarks Tools Help
Record Stop Clear Filter Copy Save Help
Summary Headers Cookies Cache Query String POST Data Content Stream
HttpWatch Professional 4.0
Inspect Clear All HTML CSS JS XHR Images Flash
Console HTML CSS Script DOM Net
Headers Response
Done
Internet
100%
209.245.121.192

Here is an h1 tag. Next we have a HR

Do you see my tags or my rendered text?

```
Started        Time    Size  Method  Result  Type           URL
00:00:00.000   0.065   488   GET     200     text/plain;c...  http://ajaxref.com/ch3/plaintext.php
```

398 bytes sent to 209 Find Export 488 bytes received by 10.0.0.193:43 Find Export

```
GET /ch3/plaintext.php HTTP/1.1
Accept: */*
Referer: http://ajaxref.com/ch3/c
Accept-Language: en-us
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatib
Host: ajaxref.com
Connection: Keep-Alive
Cookie: 4074vexclude=false; Coyot
```

```
HTTP/1.1 200 OK
Date: Fri, 02 Mar 2007 21:21:31 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 O
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 208
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/plain;charset=utf-8

    <html>
    <head><title>This is the Title Tag</tit
    <body>
    <hl>Here is an hl tag.  Next we have a 
    <hr />
    <strong>Do you see my tags or my render
    </body>
    </html>
```

```
<html>
    <head><title>This is the Title Tag</title></head>
    <body>
    <h1>Here is an h1 tag.  Next we have a HR</h1>
    <hr />
    <strong>Do you see my tags or my rendered text?</strong>
    </body>
    </html>
```

plaintext.php ajaxref.com 208 b 0ms

Response Headers

```
          Date  Fri, 02 Mar 2007 21:22:24 GMT
        Server  Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.7a DAV/2
 Cache-Control  no-cache
        Pragma  no-cache
Content-Length  208
    Keep-Alive  timeout=5, max=100
    Connection  Keep-Alive
  Content-Type  text/plain;charset=utf-8
```

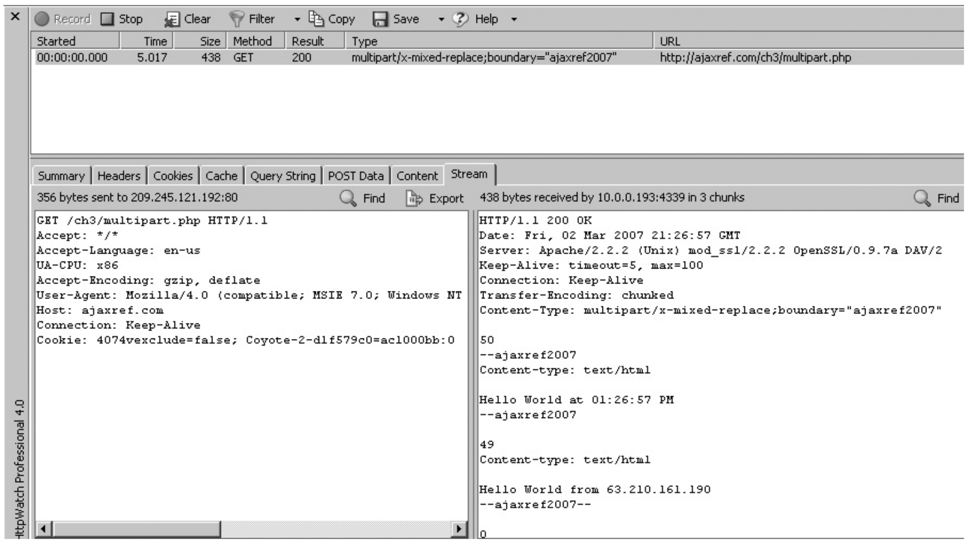
Request Headers

```
           Host  ajaxref.com
     User-Agent  Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.1) Gecko/20061204
         Accept  text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;
Accept-Language  en-us,en;q=0.5
Accept-Encoding  gzip,deflate
 Accept-Charset  ISO-8859-1,utf-8;q=0.7,*;q=0.7
     Keep-Alive  300
     Connection  keep-alive
         Cookie  Coyote-2-dlf579c0=acl000bb:0
  Cache-Control  max-age=0
```

1 request 208 b 0ms

Figure 3-8 Internet Explorer and Firefox deal with MIME types differently

Looking at a communication trace of a multipart response, you can see chunks of individual data with size and boundary indicators, as shown here:


Record Stop Clear Filter Copy Save Help
Summary Headers Cookies Cache Query String POST Data Content Stream
Find Export
HttpWatch Professional 4.0

```
Started        Time    Size  Method  Result  Type                                              URL
00:00:00.000   5.017   438   GET     200     multipart/x-mixed-replace;boundary="ajaxref2007"  http://ajaxref.com/ch3/multipart.php
```

356 bytes sent to 209.245.121.192:80 Find Export 438 bytes received by 10.0.0.193:4339 in 3 chunks

```
GET /ch3/multipart.php HTTP/1.1
Accept: */*
Accept-Language: en-us
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT
Host: ajaxref.com
Connection: Keep-Alive
Cookie: 4074vexclude=false; Coyote-2-dlf579c0=acl000bb:0
```

```
HTTP/1.1 200 OK
Date: Fri, 02 Mar 2007 21:26:57 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.7a DAV/2
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: multipart/x-mixed-replace;boundary="ajaxref2007"

50
--ajaxref2007
Content-type: text/html

Hello World at 01:26:57 PM
--ajaxref2007

49
Content-type: text/html

Hello World from 63.210.161.190
--ajaxref2007--

0
```

With Firefox, it is possible to set the `multipart` property of an XHR instance to true to enable support for this format. Since this is a proprietary feature, you will likely set the `onload` event handler, which fires when data is loaded (`readyState` = 4), but you should also be able to set `onreadystate` change approach for your callback as well, if you like.

```
var url = "http://ajaxref.com/ch3/multipart.php";
var xhr = createXHR();
if (xhr)
  {
   xhr.multipart = true;
   xhr.open("GET", url, true);
   xhr.onload = handleLoad;
   xhr.send(null);
  }
```

When the data is received, just look at it as a normal XHR, though given the format, you will likely be only using `responseText`.

```
function handleLoad(event)
{
  document.getElementById("responseOutput").style.display = "";
  document.getElementById("responseOutput").innerHTML +=
"<h3>xhr.responseText</h3>" + event.target.responseText;
}
```

To see this example working under supporting browsers, visit http://ajaxref.com/ch3/multipart.html.

onProgress and Partial Responses

Firefox already implements a few useful event handlers for the `XMLHttpRequest` object. The most interesting is the `onprogress` handler, which is similar to `readyState` with a value of 3 but is different in that it is called every so often and provides useful information on the progress of any transmission. This can be consulted to not only look at the `responseText` as it is received, but also to get a sense of the current amount of content downloaded versus the total size. The following code snippet sets up an XHR to make a call to get a large file and associates a callback for the `onprogress` handler:

```
var url = "http://ajaxref.com/ch3/largefile.php";
var xhr = createXHR();
if (xhr)
  {
    xhr.onprogress = handleProgress;
    xhr.open("GET", url, true);
    xhr.onload = handleLoad;

    xhr.send(null);
   }
```

The `handleProgress` function receives an event object that can be examined to determine the progress made versus the total size, as well as to access the received content in `responseText`.

```
function handleProgress(e)
{
  var percentComplete = (e.position / e.totalSize)*100;

  document.getElementById("responseOutput").style.display = "";
  document.getElementById("responseOutput").innerHTML += "<h3>reponseText -
" + Math.round(percentComplete) + "%</h3>" + e.target.responseText;
}
```

This Firefox-specific example can be run at http://ajaxref.com/ch3/partialprogress.html and should be quite encouraging because it suggests that there will be a time in the near future when we will be able to very quickly get an accurate sense of request progress beyond a spinning circle animated GIF.

NOTE *A limitation of using XML responses is that you cannot look at partial responses. The reason for this is that an entire XML packet is required for parsing the tree properly.*

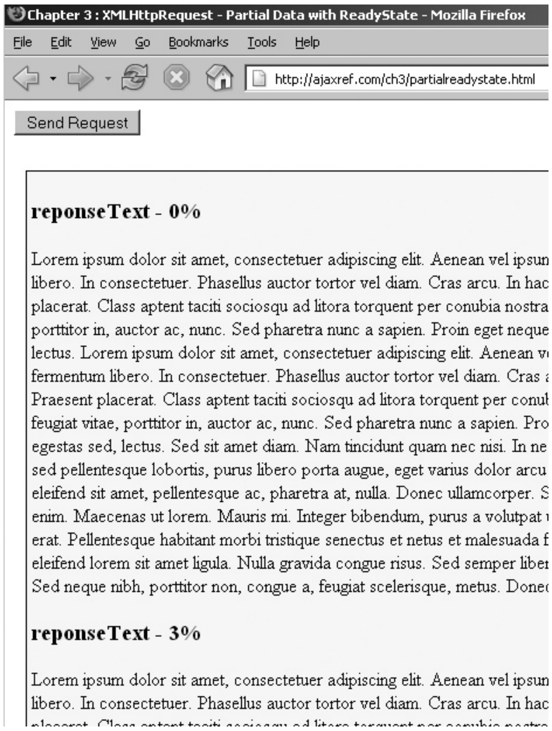
Partial Responses with `readyState`

It is possible to perform the same partial data example using a timer to wake up every so often and look at `responseText`. In this case, the callbacks are set to wake up every 50ms using either `setTimeout()` or `setInterval()`. The callbacks then handle the partial data.

```
var url = "http://ajaxref.com/ch3/largefile.php";
var xhr = createXHR();
if (xhr)
  {
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function(){handleResponse(xhr);};
    xhr.send(null);
    window.setTimeout( function() {handlePartialResponse(xhr);}, 50);
}
```

In `handlePartialResponse` we look at the `responseText` field to grab whatever data has been provided. We can also look at the `Content-Length` response header, assuming it is provided to calculate the percentage progress.

```
function handlePartialResponse(xhr)
{
    if (xhr.readyState == 3)
      {
        document.getElementById("responseOutput").style.display = "";

        var length = xhr.getResponseHeader("Content-Length");
        var percentComplete = (xhr.responseText.length / length)*100;

        document.getElementById("responseOutput").innerHTML += "<h3>reponseText -
" + Math.round(percentComplete) + "%</h3>" + xhr.responseText;
     }

    /* wake up again in 50ms to handle more data if not done now */
    if (xhr.readyState != 4)
       window.setTimeout( function() {handlePartialResponse(xhr);}, 50);
}
```


Chapter 3 : XMLHttpRequest - Partial Data with ReadyState - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://ajaxref.com/ch3/partialreadystate.html
Send Request

reponseText - 0%

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean vel ipsun
libero. In consectetuer. Phasellus auctor tortor vel diam. Cras arcu. In hac
placerat. Class aptent taciti sociosqu ad litora torquent per conubia nostra
porttitor in, auctor ac, nunc. Sed pharetra nunc a sapien. Proin eget neque
lectus. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean v
fermentum libero. In consectetuer. Phasellus auctor tortor vel diam. Cras
Praesent placerat. Class aptent taciti sociosqu ad litora torquent per conul
feugiat vitae, porttitor in, auctor ac, nunc. Sed pharetra nunc a sapien. Pro
egestas sed, lectus. Sed sit amet diam. Nam tincidunt quam nec nisi. In ne
sed pellentesque lobortis, purus libero porta augue, eget varius dolor arcu
eleifend sit amet, pellentesque ac, pharetra at, nulla. Donec ullamcorper. S
enim. Maecenas ut lorem. Mauris mi. Integer bibendum, purus a volutpat
erat. Pellentesque habitant morbi tristique senectus et netus et malesuada f
eleifend lorem sit amet ligula. Nulla gravida congue risus. Sed semper liber
Sed neque nibh, porttitor non, congue a, feugiat scelerisque, metus. Donec

reponseText - 3%

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean vel ipsun
libero. In consectetuer. Phasellus auctor tortor vel diam. Cras arcu. In hac

Figure 3-9 Partial data consumption is possible in some browsers.

The results of this example are in Figure 3-9, which looks pretty much the same as the previous example. To see if this technique works in your browser try the example found at http://ajaxref.com/ch3/partialreadystate.html.

Note *Internet Explorer 7 and before cannot use the readyState to access partial data as it disallows looking at* `responseText` *when you are in* `readyState` *3 or before.*

Other Firefox Event Handlers

Firefox also implements the `onload` and `onerror` event handlers for XHRs. The `onload` handler is a convenience feature and corresponds to `onreadystatechange` reaching a `readyState` value of 4. Given that most developers just use this `readyState` value, this is an obvious change and certainly is a bit less cryptic than the integer codes. It also is beneficial because you do not have to use closures to access the XHR object, which certainly makes coding life more pleasant.

The `onerror` seems a promising feature as well and would be invoked when a network error occurs. Unfortunately, this handler doesn't seem to work properly yet and is poorly documented. Until it is fixed, you will likely need to trap network errors using status codes, creating timeouts, and using `try-catch` blocks as we have already alluded to in this chapter; these will be presented in more depth in Chapter 6.

XHR Arcana and Future Changes

If you dig around enough in browser documentation or write code to reflect the innards of the XHR object, you might find things you don't expect. For example, Internet Explorer supports `responseBody` and `responseStream` properties to get access to raw encoded response data. While this sounds quite interesting, there is no way to use JavaScript to utilize these features. Firefox has similar things lurking around, such as the `channel` property, which represents the underlying channel communication mechanism used in Mozilla to make the request. If you inspect it with Firebug, you will see it contains a variety of interesting values about the network request and appears to have a variety of methods to control it. However, you will not be able to access these items in a typical JavaScript application as they require elevated privileges. You'll also find scant documentation on exactly what everything does and what the various numeric values mean, so if you like to hunt for arcane knowledge this will certainly keep you busy.

While we don't know for sure what the future holds for XHRs, it isn't too hard to guess that, given the excitement around Ajax, there is likely to be great innovation with the `XMLHttpRequest` object, for better or worse. Looking at the emerging specification discussion, listening to various browser vendors, and simply thinking about what is missing, you see a few likely areas for change, including:

- More request header related methods like `getRequestHeader()` and `removeRequestHeader()`
- Some way to deal with byte streams
- A method to invoke cross-domain `XMLHttpRequests` that can break the same origin restriction without using a service proxy
- New event handlers like `onabort`, `ontimeout`, on-particular types of errors
- Features to support offline content availability
- Features to support client-side session management

While the previous list is just speculation, until a browser vendor commits to it, don't be surprised if you see a few of these things implemented either natively or in some Ajax extension library that you may encounter.

As we wind down the chapter, we have a few more things that should be covered. First we need to see a few common problems people run into with XHRs and then finally present a full example to put everything into context.

XHR Implications and Challenges

Besides dealing with all the cross-browser syntax concerns that have been presented, there are still numerous coding-specific challenges facing an aspiring Ajax developer.

- **Handling Network Problems** The network is really what makes Ajax interesting and is where you will face many of the most difficult challenges. You saw already that dealing with network errors and timeouts is not yet intrinsically part of XHRs, though it really should be. We have but scratched the surface of the edge case possibilities that may range from incomplete and malformed responses, the need for timeouts, retries, and more meaningful indication of network conditions and download progress. We present this in great detail in Chapter 6.

- **Managing Requests** Handling many simultaneous requests can be a bit tricky if you use a global XHR object. With this style of coding, any request in progress will be overwritten if a new request is opened with an existing object. However, beyond such a basic problem, you will certainly encounter difficulties when handling many simultaneous requests. First, may be limitations in browsers as to the number of network requests that can be made to a particular domain at once. In all standard configured browsers it is a mere two. Second, if the requests are dependent on each other, you may be forced to implement some form of queuing and locking mechanism to make sure that requests are handled in the right order. Now we have reached a difficult aspect of coding known as concurrent programming.
- **User Interface Improvements** The improved data availability and page changing possibilities with JavaScript and Ajax have a large impact on user interface design. Once you employ XHRs and build a more responsive Web application, be prepared to adopt new interface conventions to fully take advantage of the newfound power. Usually Ajax is coupled with much richer user interface conventions such as drag-and-drop, type-ahead, click-to-edit, and many others. We will briefly touch on some of these in examples throughout the book, especially in Chapter 8.
- **Degrading Gracefully** A big question is whether we should allow older browsers and even search bots that don't support XHRs to access our Ajax-driven site or application. Yet even if these user-agents are rejected, what happens if XHR support is disabled in modern browsers by a user out of security paranoia? How are you going to degrade gracefully or at least inform users of limitations they may face without XHR support? You saw in the previous chapter that it is possible to perform Ajax-style communication without XHRs, so maybe you should employ these techniques in such conditions? You'll see over the next few chapters that building very resilient web application architecture is possible, but it takes more than a bit of planning. We'll wrap up that discussion in Chapter 9.
- **A Need for JavaScript and Ajax Libraries** You may wonder why, with so many Ajax libraries available, you bothered studying the underlying properties and methods of XHRs? Why not just adopt a popular library and let it hide all the details from you? Frankly, there are so many of them it is tough to choose, and you don't want to learn examples for a library that isn't supported in the future. At the time of this edition there are literally 200+ Ajax-related libraries and toolkits to choose from! Be prepared to be shocked if you evaluate some of these offerings to find that a number of the ideas presented in this chapter are not handled, and quite a number from the following chapters are certainly not. So don't be fooled by nice UI widget demos during your evaluations until you are certain they aren't layered upon an XHR facility that isn't browser quirk network edge case aware enough. To help you understand such considerations, we'll develop a sample library of our own starting in Chapter 5, but don't take this as a definitive suggestion to only roll your own or use ours; we certainly believe that well-supported libraries will ultimately be the way to go.

We certainly didn't fully cover each of these issues since most require large sections or complete chapters for an adequate discussion and are more appropriately found in later chapters. However, we will finish the section with a complete discussion of one Ajax-related issue that is quite misunderstood—closures and memory leaks.

Ajax and Memory Leaks

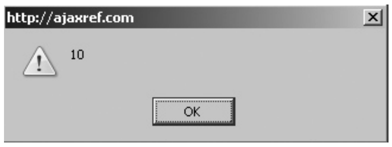
Ajax doesn't cause memory leaks. We need to get that out in the open right away. Misuse of JavaScript coupled with a bad garbage collection implementation in one browser is where the cause of this misconception truly lies. The skeptical reader may wonder why then have you only heard of such problems since Ajax became popular? The answer is that generally JavaScript wasn't used enough and you didn't stay on a page long enough to encounter the problem. Consider that you likely may have had memory leaks in your pre-Ajax applications but since you were posting and repainting pages fairly often, you might not have been executing code long enough to leak too much memory. Now with Ajax you see the problem more often, but folks who used JavaScript for games and building Web editors have been quite aware of JavaScript memory challenges for a number of years. So you probably wonder: where do these memory leaks come from? There is no simple answer. There are bugs as well as user-caused memory leaks, but in the case of Ajax you are likely facing trouble in Internet Explorer 5, 5.5, and 6 because of event handlers and closures.

Exploring Closures

A *closure* is an inner function in JavaScript that becomes available outside of the enclosing function and thus must retain variable state to act in a meaningful way. For example, consider the following function:

```
function outer()
{
 var x = 10;
 function inner() { alert(x); }
 setTimeout(inner, 1000);
}
outer();
```

When you run this code fragment, the function `outer` is invoked and has a locally scoped function `inner` that will print out the variable. This `inner` function will be called in one second, but you will have left the `outer` function by the time the `inner` function is called, so what would the value of `x` be? Because of the way JavaScript binds the values of the needed variables to the function, it will actually have a value in `x` of 10 at the time the inner is invoked.

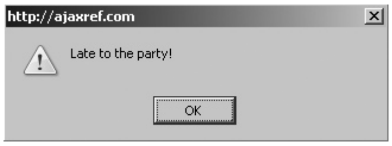


```
10
```

This gets quite interesting if you note when these bindings actually happen. Consider the following code, which resets the value of `x`.

```
function outer()
{
 var x = 10;
 function inner() { alert(x); }
 setTimeout(inner, 1000);
 x = "Late to the party!";

}
outer();
```

It might be surprising to you, since the timeout and the reassignment happens after the function is defined, that the value of `x` is the string value "Late to the party!" as shown here. This shows that the inner function is not just copying the value of the variable `x` but also holds a reference to that variable.


http://ajaxref.com

Late to the party!

Do not assume that closures are related solely to timeouts and other asynchronous activity such as what Ajax thrives on. The following little example shows you could just as easily use them when doing high-order JavaScript programming when you return functions as values for later use:

```
function outer()
{
 var x = 10;
 var inner = function() { alert(x); }
 x = "Late to the party!";
 return inner;
}
var alertfunction = outer();
alertfunction();
```

You have seen closures throughout this chapter. Every time we make an XHR, we assign a function to be called back upon a `readyState` value change and we want to capture the local variable associated with the created XHR for reference.

```
function sendRequest(url,payload)
{
 var xhr = createXHR();
 if (xhr)
   {
       xhr.open("GET",url + "?" + payload,true);
       xhr.onreadystatechange = function(){handleResponse(xhr);};
       xhr.send(null);
    }
}
```

Note that the variable `xhr` is local to the function `sendRequest` but through the closure is made available to the `handleResponse` callback function.

Beyond such a rudimentary example of closures, you will also encounter such constructs all over typical Ajax applications because of the need of setting up various event handlers to address user activity in a richer user interface. This is where the trouble begins.

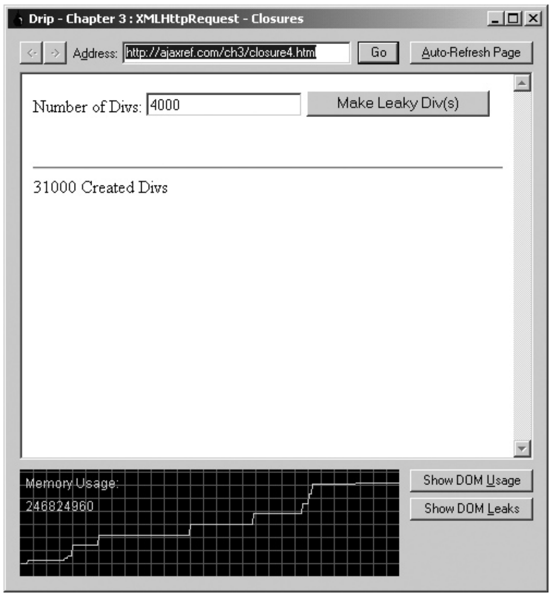
Closures and Memory Leaks

Internet Explorer has a problem freeing memory and closures when there are circular references. A circular reference is when one object has a pointer that points to another object and that object creates a reference back to the first. There are other ways to make such a cycle, of course, but the place that we most often see circular references is in event handlers where the event handling function references the bound node that the event was triggered upon. For example, a mouse click against a particular form element references a function that then references back to that particular form element that the event was captured upon. The example here creates the number of **<div>** tags you specify, with event handlers referencing each.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
<title>Chapter 3 : Memory Leak Tester</title>
<script type="text/javascript" language="javascript">
 function createDivs()
  {
    var countSpan = document.getElementById("countSpan");
    var numDivs = document.requestForm.numberDivs.value;
    var oldNumDivs = parseInt(countSpan.innerHTML,10);
    var newNumDivs = oldNumDivs + parseInt(numDivs,10);
    if (newNumDivs > 0)
      {
        for (var i=oldNumDivs; i<newNumDivs; i++)
           {
            createClosure(i);
           }
        countSpan.innerHTML = newNumDivs;
      }
   }

 function createClosure(i)
  {
    var div = document.createElement("div");
    div.id = "leakydiv" + i;
    div.onclick = function() { this.innerHTML = "Clicked"; };
    document.body.appendChild(div);
  }
 window.onload = function ()
  {
    document.requestForm.requestButton.onclick = createDivs;
   };
</script>
</head>
<body>
<form action="#" name="requestForm">
  Number of Divs: <input type="text" name="numberDivs" />
  <input type="button" name="requestButton" value="Make Leaky Div(s)" />
```

```
</form>
<br />
<hr />
<span id="countSpan">0</span> Created Divs
</body>
</html>
```

This simple example will begin to leak memory in versions of Internet Explorer 6 and before, which can't handle the circular references setup. You can see that in the capture shown here from a memory leak tool for Internet Explorer appropriately called Drip:

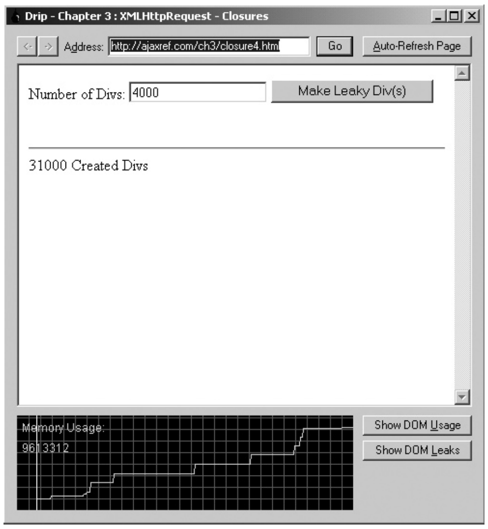


http://ajaxref.com/ch3/closure4.html

Number of Divs: 4000

31000 Created Divs

Memory Usage:
246824960



http://ajaxref.com/ch3/closure4.html

Number of Divs: 4000

31000 Created Divs

Memory Usage:
9613312

From this simple example, you can see a small amount of memory being leaked each time the button is clicked, but it does not seem too big of a deal because of the small amount of memory leaked. However, imagine this on a larger scale. In Ajax applications, there may be hundreds of event handlers. If each of these contains a circular reference, it can be enough to crash to the older versions of Internet Explorer. If you have Internet Explorer 6 or before around and want to try crashing your browsers, simply adjust the number of **<div>** tags to make to 200,000 or more and you will likely crash regardless of the gigabytes of RAM you may have.

Closures and memory leaks are actually the least of your worries. You'll see in the upcoming chapters that Ajax-style programming is going to introduce significant challenges from dealing with network, security, and interface concerns you may have been able to avoid before. So in the next chapter, let's finish one more core topic, data formats, before we vigorously tackle these challenges.

Summary

The `XMLHttpRequest` object is the heart of most Ajax applications. This useful object provides a general purpose facility for JavaScript to make HTTP requests and read the responses. It does not force the use of XML as payload, as we will discuss greatly in the next chapter, though it does provide some conveniences for dealing with XML in the form of the `responseXML` property, which will automatically parse any received XML data.

The syntax of the XHR object is somewhat of an industry de facto standard at the moment, with browser vendors implementing the core syntax introduced initially by Microsoft. For basic usage, the browser vendors are pretty consistent in their support; however, in the details, there is quite a bit of variation across browsers in areas such as XHR object instantiation, header management, ready states, status codes, extended HTTP methods, authentication, and error management. Browser vendors have also already begun to introduce proprietary features, some of which are quite useful, but sadly not widely supported.

Based upon the various quirks and inconsistencies presented at the point of this book's writing in 2007, readers are duly warned that they should not take the details of XHRs for granted and should be cautious with any Ajax communication library they may adopt. Fortunately, the W3C has begun the standardization process which should eventually bring some much needed stability and polish to the XHR syntax.