I

PART

# Core Ideas

1
CHAPTER

# Introduction to Ajax

Ajax (Asynchronous JavaScript and XML) encompasses much more than the technologies that make up this catchy acronym. The general term *Ajax* describes the usage of various Web technologies to transform the sluggish batch submission of traditional Web applications into a highly responsive near desktop-software-like user experience. However, such a dramatic improvement does come with the price of a significant rise in programming complexity, increased network concerns, and new user experience design challenges. For now, let's avoid most of those details, as is appropriate in an introduction, and begin with an overview of the concepts behind Ajax illustrated by an example. Details of the example will then be presented hinting at future complexity. The chapter then concludes with a brief discussion of the historical rise and potential effects of Ajax upon Web development.

## Ajax Defined

Traditional Web applications tend to follow the pattern shown in Figure 1-1. First a page is loaded. Next the user performs some action such as filling out a form or clicking a link. The user activity is then submitted to a server-side program for processing while the user waits, until finally a result is sent, which reloads the entire page.

While simple to describe and implement, the down side with this model is that it can be slow, as it needs to retransmit data that makes up the complete presentation of the Web page over and over in order to repaint the application in its new state.

Ajax-style applications use a significantly different model where user actions trigger behind the scenes communication to the server fetching just the data needed to update the page in response to the submitted actions. This process generally happens asynchronously, thus allowing the user to perform other actions within the browser while data is returned. Only the relevant portion of the page is repainted, as illustrated in Figure 1-2.

Beyond this basic overview, the specifics of how an Ajax-style Web application is implemented can be somewhat variable. Typically JavaScript invokes communication to the server, often using the `XMLHttpRequest` (XHR) object. Alternatively, other techniques such as inline frames, `<script>` tag fetching remote .js files, image requests, or even the Flash player are used. After receiving a request, a server-side program may generate a response in XML, but very often you see alternate formats such as plain text, HTML fragments, or JavaScript Object Notation (JSON) being passed back to the browser. Consumption of the

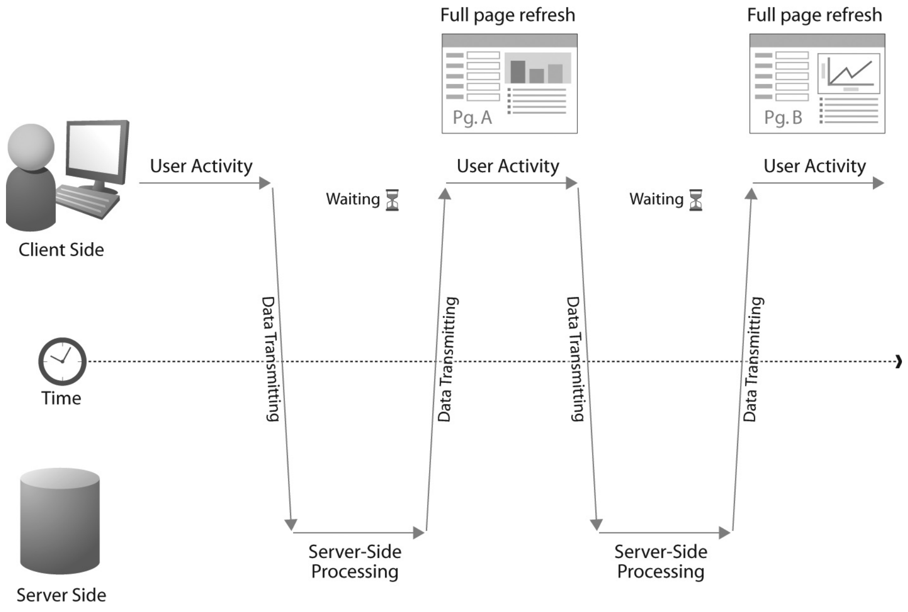


**FIGURE 1-1** Traditional Web application communication flow

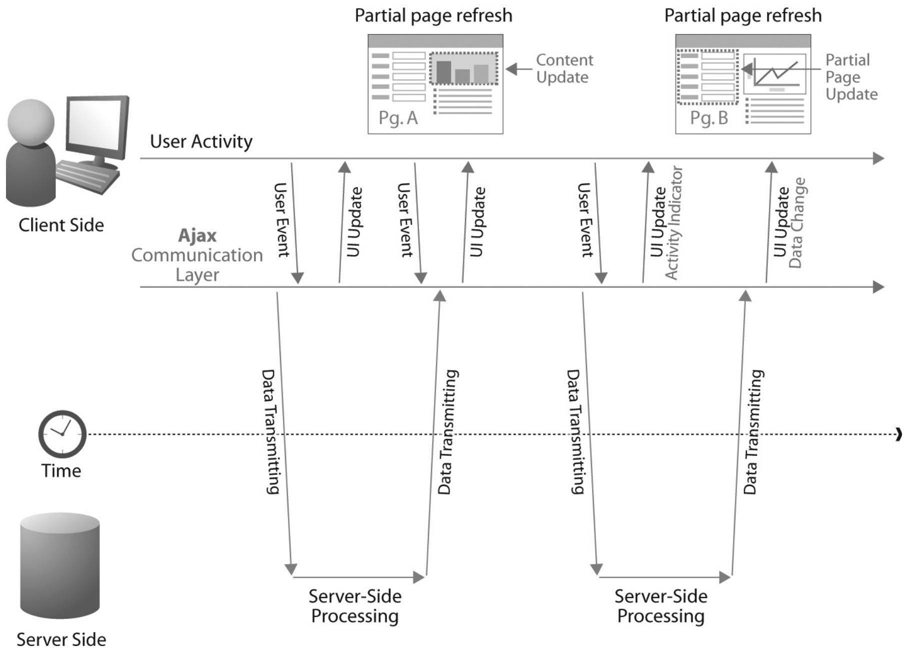


**FIGURE 1-2** Ajax-style communication flow

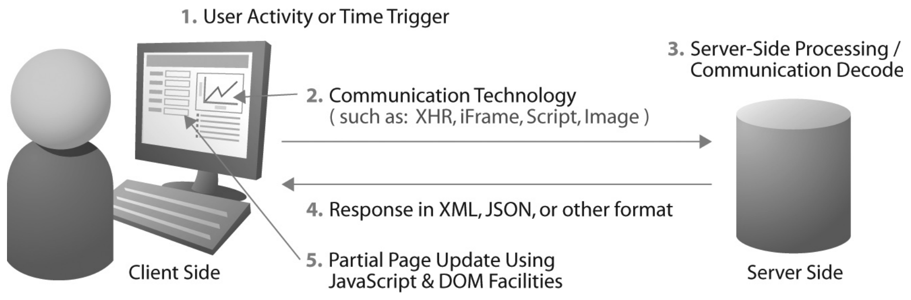

1. User Activity or Time Trigger
2. Communication Technology ( such as: XHR, iFrame, Script, Image )
3. Server-Side Processing / Communication Decode
4. Response in XML, JSON, or other format
5. Partial Page Update Using JavaScript & DOM Facilities
Client Side
Server Side

**FIGURE 1-3** Ajax applications may vary in implementation

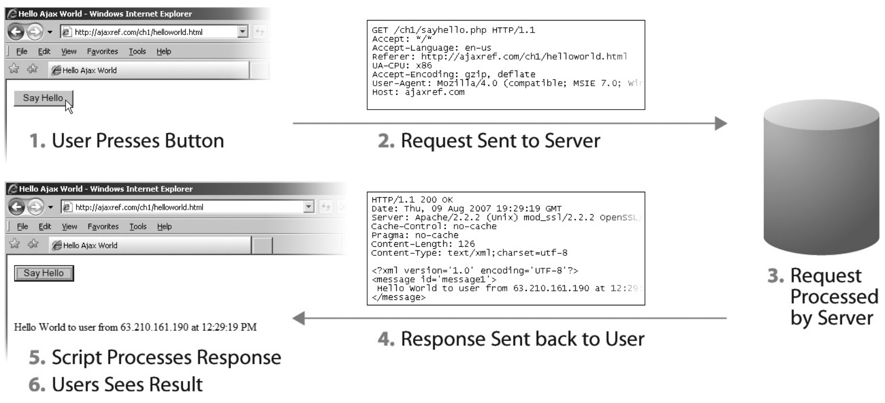
received content is typically performed using JavaScript in conjunction with Document Object Model (DOM) methods, though in some rare cases you see native XML facilities in the browser used. A graphic description of the wide variety of choices in implementing an Ajax-style Web application is shown in Figure 1-3.

## Hello Ajax World

With the basic concepts out of the way, like any good programming book we now jump right into coding with the ubiquitous "Hello World" example. In this version of the classic example, we will press a button and trigger an asynchronous communication request using an `XMLHttpRequest` (XHR) object and the Web server will issue an XML response which will be parsed and displayed in the page. The whole process is overviewed in Figure 1-4.

To trigger the action, a simple form button is used which, when clicked, calls a custom JavaScript function `sendRequest()` that will start the asynchronous communication. It might be tempting to just bind in a JavaScript call into an event handler attribute like so:

```
<form action="#">
 <input type="button" value="Say Hello" onclick="sendRequest();" />
</form>
```


```
GET /ch1/sayhello.php HTTP/1.1
Accept: */*
Accept-Language: en-us
Referer: http://ajaxref.com/ch1/helloworld.html
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Wi
Host: ajaxref.com
```

```
HTTP/1.1 200 OK
Date: Thu, 09 Aug 2007 19:29:19 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 126
Content-Type: text/xml;charset=utf-8

<?xml version='1.0' encoding='UTF-8'?>
<message id='message1'>
 Hello World to user from 63.210.161.190 at 12:29:
</message>
```

Hello World to user from 63.210.161.190 at 12:29:19 PM

Hello Ajax World - Windows Internet Explorer
http://ajaxref.com/ch1/helloworld.html
File Edit View Favorites Tools Help
Hello Ajax World
Say Hello
1. User Presses Button
2. Request Sent to Server
3. Request Processed by Server
4. Response Sent back to User
5. Script Processes Response
6. Users Sees Result

**FIGURE 1-4** Hello Ajax World in action

However, it is a much better idea to simply use `name` or `id` attributes for the form fields or other tags that trigger activity:

```
<form action="#">
 <input type="button" value="Say Hello" id="helloButton" />
</form>
```

and then bind the `onclick` event using JavaScript like so:

```
window.onload = function ()
{
 document.getElementById("helloButton").onclick = sendRequest;
};
```

A `<div>` tag named `responseOutput` is also defined. It will eventually be populated with the response back from the server by reference via DOM method, such as `getElementById()`.

```
<div id="responseOutput">&nbsp;</div>
```

When the `sendRequest` function is invoked by the user click, it will first try to instantiate an `XMLHttpRequest` object to perform the communication by invoking another custom function `createXHR`, which attempts to hide version and cross browser concerns. The function uses `try-catch` blocks to attempt to create an XHR object. It first tries to create it natively as supported in Internet Explorer 7.x, Safari, Opera, and Firefox. Then, if that fails, it tries using the `ActiveXObject` approach supported in the 5.x and 6.x versions of Internet Explorer.

```
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) {}
   alert("XMLHttpRequest not supported");
   return null;
}

function sendRequest()
{
    var xhr = createXHR();  // cross browser XHR creation

    if (xhr)
     {
      // use XHR
     }
}
```

If the `createXHR` function returns an XHR object, you begin your server communication by using the `open()` method to create an HTTP GET request to the URL http://ajaxref.com/ch1/sayhello.php. A true flag is specified to indicate that the request should proceed asynchronously.

```
 xhr.open("GET","http://ajaxref.com/ch1/sayhello.php",true);
```

This is just the briefest overview of the XHR object as we will study it in great depth in Chapter 3.

Before moving on, you might want to call your test URL directly in your browser. It should return an XML response with a message indicating your IP address and the local time on the server, as shown in Figure 1-5.

It should be noted that it is not required to use XML in Ajax responses. Regardless of the 'x' in Ajax referencing XML, Chapter 4 will clearly show that the data format used in an Ajax application is up to the developer.

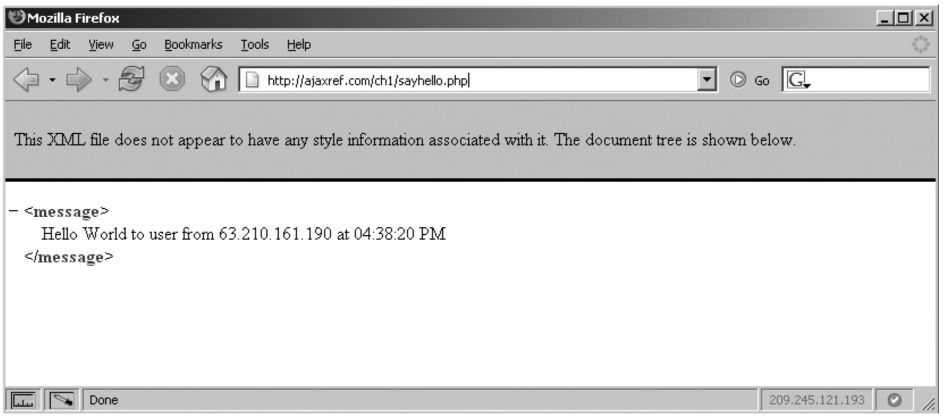
After creating the request, a callback function, `handleResponse`, is defined which will be invoked when data becomes available, as indicated by the `onreadystatechange` event handler. The callback function employs a closure that captures variable state so that the code has full access to the XHR object held in the variable `xhr` once `handleResponse` is finally called.

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

Closures might be unfamiliar to those readers newer to JavaScript, but they are fully covered in Chapter 3 as well as Appendix A.

Finally, the request is sent on its way using the `send()` method of the previously created XHR object. The complete `sendRequest` function is shown here:

```
function sendRequest()
{
    var xhr = createXHR();  // cross browser XHR creation

    if (xhr)  // if created run request
     {
      xhr.open("GET","http://ajaxref.com/ch1/sayhello.php",true);
      xhr.onreadystatechange = function(){handleResponse(xhr);};
      xhr.send(null);
     }
}
```


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://ajaxref.com/ch1/sayhello.php
Go
Done
209.245.121.193

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <message>
    Hello World to user from 63.210.161.190 at 04:38:20 PM
  </message>
```

**FIGURE 1-5** Returned XML packet shown directly in browser

Eventually, your server should receive our request and invoke the simple sayhello.php program shown here.

```
<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
header("Content-Type: text/xml");

$ip = $_SERVER['REMOTE_ADDR'];
$msg =  "Hello World to user from " . $ip . " at ". date("h:i:s A");

print "<?xml version='1.0' encoding='UTF-8'?>";
print "<message>$msg</message>";

?>
```

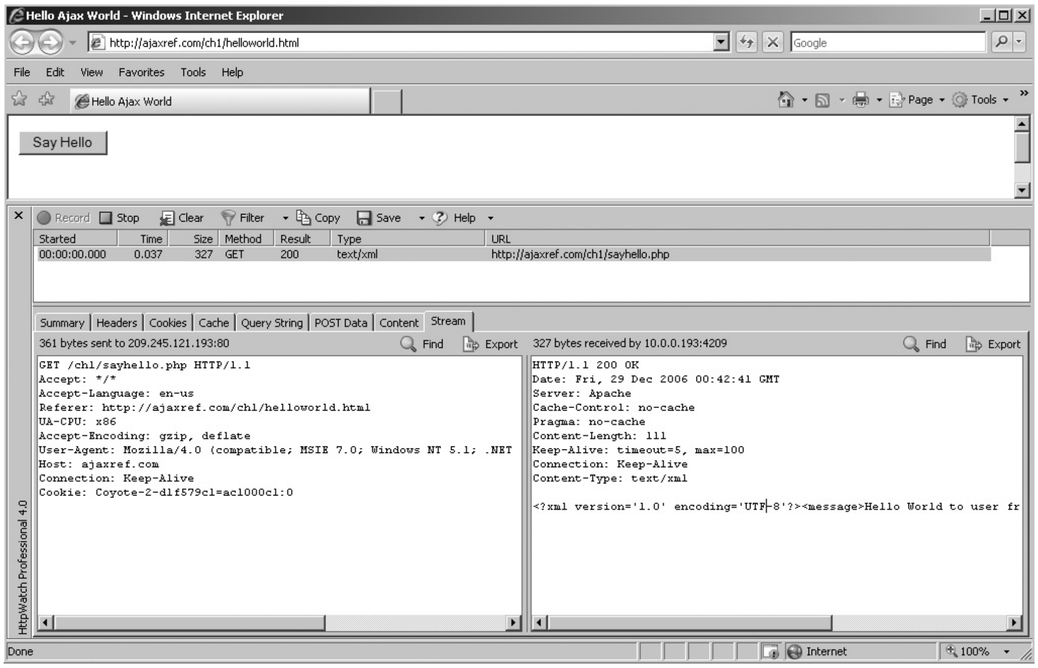
Ajax does not favor or require any particular server-side language or framework. The general idea should be the same in whatever environment you are comfortable. For example, sayhello.jsp looks quite similar to the PHP version.

```
<%
response.setHeader("Cache-Control","no-cache");
response.setHeader("Pragma","no-cache");
response.setContentType("text/xml");

String ip = request.getRemoteAddr();
String msg =  "Hello World to user from " + ip + " at " + new java.text
.SimpleDateFormat("h:m:s a").format(new java.util.Date());

out.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
out.print("<response>" + msg + "</response>");
%>
```

**NOTE** *PHP is used in most examples, given its simplicity and readability, but any server-side technology, such as Ruby, ASP.NET, and Java, is more than capable of servicing Ajax requests.*

On the server-side, you first emit some HTTP headers to indicate that the result should not be cached. Mixing Ajax and caching can be quite troubling and addressing it properly is a significant topic of Chapter 6. For now, the code simply indicates the result should never be cached. Next, the appropriate `Content-Type` HTTP header is set to `text/xml` indicating that XML will be returned. Finally, an XML packet is created containing a greeting for the user that also includes the user's IP address and local system time to prove that the request indeed went out over the network. However, it is much better to monitor the actual progress of the request directly, as shown in Figure 1-6.

Once the browser receives data from the network, it will signal such a change by modifying the value of the `readyState` property of the XHR object. Now, the event handler for `onreadystatechange` should invoke the function `handleResponse`. In that function, the state of the response is inspected to make sure it is completely available as indicated by a value of 4 in the `readyState` property. It is also useful to look at the HTTP status code returned by the request. Ensuring that the status code is 200 gives at least a basic indication that the response can be processed. Chapters 3, 5 and 6 will show that there is much more



| Started | Time | Size | Method | Result | Type | URL |
|---|---|---|---|---|---|---|
| 00:00:00.000 | 0.037 | 327 | GET | 200 | text/xml | http://ajaxref.com/ch1/sayhello.php |

361 bytes sent to 209.245.121.193:80

```
GET /chl/sayhello.php HTTP/1.1
Accept: */*
Accept-Language: en-us
Referer: http://ajaxref.com/chl/helloworld.html
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET
Host: ajaxref.com
Connection: Keep-Alive
Cookie: Coyote-2-d1f579c1=ac1000c1:0
```

327 bytes received by 10.0.0.193:4209

```
HTTP/1.1 200 OK
Date: Fri, 29 Dec 2006 00:42:41 GMT
Server: Apache
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 111
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/xml

<?xml version='1.0' encoding='UTF-8'?><message>Hello World to user fr
```

Hello Ajax World - Windows Internet Explorer
http://ajaxref.com/ch1/helloworld.html
File Edit View Favorites Tools Help
Hello Ajax World
Say Hello
Record Stop Clear Filter Copy Save Help
Summary Headers Cookies Cache Query String POST Data Content Stream
Find Export
HttpWatch Professional 4.0
Done
Internet
100%

**FIGURE 1-6** HTTP transaction details

that should be addressed than the `readyState` and status code in order to build a robust Ajax application, but this degree of detail is adequate for this simple example.

With the XML response received, it is now time to process it using standard DOM methods to pull out the message string. Once the message payload is extracted, it is output to the `<div>` tag named `responseOutput` mentioned at the beginning of the walk-through.

```
function handleResponse(xhr)
{
  if (xhr.readyState == 4  && xhr.status == 200)
    {
     var parsedResponse = xhr.responseXML;
     var msg = parsedResponse.getElementsByTagName("message")[0].firstChild.nodeValue;
     var responseOutput = document.getElementById("responseOutput");
     responseOutput.innerHTML = msg;
    }
}
```

The complete example can be found at http://ajaxref.com/ch1/helloworld.html. It is possible for this example to be run locally, but a number of issues must be noted and some changes potentially made. For now the code hosted online is presented for inspection, while the next section covers the issues required to run the code from your desktop.

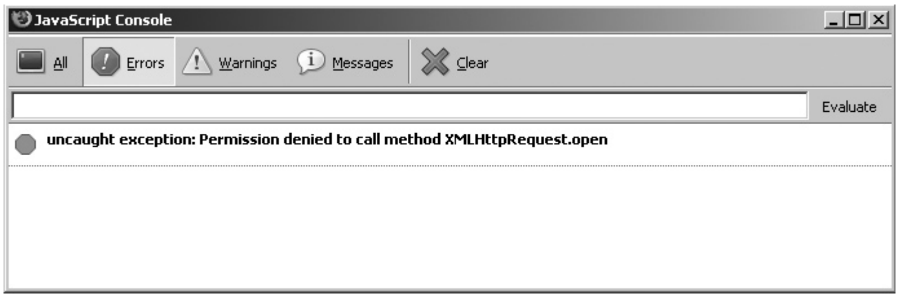
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Hello Ajax World</title>
<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) {}
   alert("XMLHttpRequest not supported");
   return null;
}

function sendRequest()
{
    var xhr = createXHR();

    if (xhr)
     {
      xhr.open("GET","http://ajaxref.com/ch1/sayhello.php",true);
      xhr.onreadystatechange = function(){handleResponse(xhr);};
      xhr.send(null);
     }
}

function handleResponse(xhr)
{
  if (xhr.readyState == 4  && xhr.status == 200)
    {
     var parsedResponse = xhr.responseXML;
     var msg = parsedResponse.getElementsByTagName("message")[0].firstChild.nodeValue;
     var responseOutput = document.getElementById("responseOutput");
     responseOutput.innerHTML = msg;
    }
}

window.onload = function ()
{
 document.getElementById("helloButton").onclick = sendRequest;
};

</script>
</head>
<body>
<form action="#">
 <input type="button" value="Say Hello" id="helloButton" />
</form>

<br /><br />
<div id="responseOutput">&nbsp;</div>

</body>
</html>
```

**NOTE** *If you are stickler for separation, you should also put all the JavaScript code in an external JS file referenced by a* `<script>` *tag, but our purpose here is to quickly illustrate Ajax. However, be assured the majority of the book strives for the cleanest separation of concerns possible and always aim to reinforce best practices in coding, markup and style as long it does not get in the way of illustrating the new concepts being presented.*

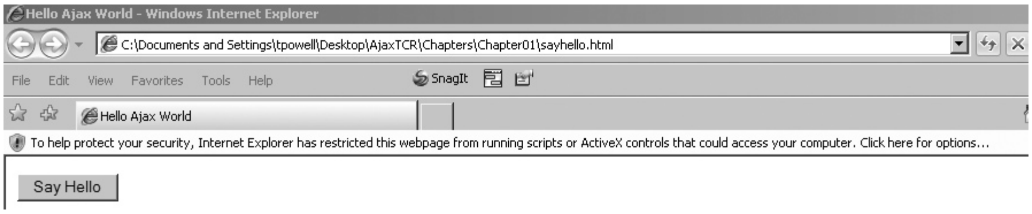
## The Challenges of Running Ajax Examples Locally

Ajax is, at its heart, fundamentally networked use of JavaScript, and because of that you will likely have problems running examples locally from your system. The main issues have to do with the security concerns of a locally saved JavaScript invoking communication. For example, if you simply copy the previous example and run it from your local disk, the code will not work, with Firefox failing behind the scenes, as shown here:



```
uncaught exception: Permission denied to call method XMLHttpRequest.open
```

Internet Explorer will prompt you to allow or deny the script.


Hello Ajax World - Windows Internet Explorer
C:\Documents and Settings\tpowell\Desktop\AjaxTCR\Chapters\Chapter01\sayhello.html
File Edit View Favorites Tools Help
SnagIt
Hello Ajax World
Say Hello

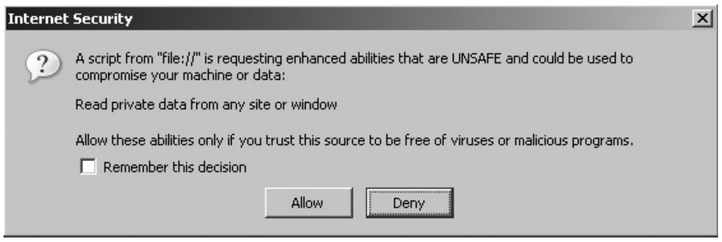
To help protect your security, Internet Explorer has restricted this webpage from running scripts or ActiveX controls that could access your computer. Click here for options...

If you accept the security changes it should run properly. However, be aware that this may not be the case in future versions of Internet Explorer as it is locked down more, and a similar solution to the one discussed next may be required.

It is possible to modify the simple example to allow Firefox to run the code by using the `netscape.security.PrivilegeManager` object. You can then use this object's `enablePrivilege` method to allow `UniversalBrowserRead` privileges so the XHR can work from the local drive. Adding try-catch blocks like so:

```
try {
 netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
  }
catch (e) {}
```

to your code will then allow it to work. However, during the run of the program you will be prompted to allow the extended permissions by the browser.



A script from "file://" is requesting enhanced abilities that are UNSAFE and could be used to compromise your machine or data:

Read private data from any site or window

Allow these abilities only if you trust this source to be free of viruses or malicious programs.

☐ Remember this decision

A complete version of the code that can be run locally in either Firefox or Internet Explorer can be found at http://ajaxref.com/ch1/helloworldlocal.html.

**NOTE** *Other Ajax-aware browsers may have no way to run JavaScript code that utilizes the XHR object from the desktop. You should run the examples from the book site or, more appropriately, set up your own development environment to address this limitation.*

To avoid this concern, you may decide instead to host the file on a server, but then you will run into another JavaScript security issue called the same origin policy. In this sense you run into a problem where the domain that issues the script—your domain—is different from the place you call, in this case ajaxref.com. The JavaScript same origin policy keeps this communication from happening to keep you more secure. The main way to solve this is to simply copy the same type of server-side code (as used in the example) to your server and adjust the URL to call your local system, likely using a relative URL. There are a few other ways you will be able to get around the same origin policy, but you really shouldn't be trying to get around it in the first place unless you very carefully consider the security implications. With the rise of mash-ups and Service Oriented Architecture (SOA), such advice may seem restrictive, but readers really should heed some of the warnings found in Chapter 7 before advocating extreme loosening of cross-domain limitations.

Like any good "Hello World" example, you should get the idea of the demonstrated technology without all the details. Unfortunately, as shown by issues such as trying to run examples locally, it may not be as easy to develop Ajax applications as we might like. However, from the example you should also see that Ajax is really just a special usage of client-side JavaScript that allows you to make and respond to HTTP requests and does not have any particular affinity for one Web server-side programming environment or another.

Yet since this is just "Hello World," we have omitted all the gory details of advanced JavaScript, HTTP, networking problems, XML, security, usability, integration with server-side frameworks, and proper use of Ajax within Web applications. That's what the rest of the book is for. However, before getting there, let's put Ajax in context with an understanding of its historical rise and by previewing some of its possible implications.

## The Rise of Ajax

The name Ajax is new, but the ideas behind it are not. The term was first coined by Jesse James Garrett in an article written in February 2005. However, undoubtedly Jesse would be the first to acknowledge that the idea of Ajax goes back much farther than his application of the new moniker. Microsoft first added the XHR ActiveX object to Internet Explorer 5 in 1999 in support of Outlook Web Access. Numerous developers from around the same time used a variety of techniques such as hidden inline frames to create Web applications that follow what looks like the Ajax pattern. It enjoyed names like "remote scripting," "innerbrowsing" (courtesy of Netscape), and "Rich Internet Applications (RIAs)" (from Macromedia and others). However, whatever it was called, for some reason this approach to Web development did not really excite most Web professionals.

Why this technology was ignored for years suddenly to be rediscovered is cause for great speculation and debate. Very likely, conservative industry conditions stemming from the dotcom meltdown around the turn of the century slowed adoption, but what changed this is less clear. It is the author's opinion that Google's Gmail, Yahoo's purchase of Ajax pioneer Oddpost, and Microsoft's Outlook Web Access for Exchange 2000 demonstrated to the world that a JavaScript-based Web application using partial page updates really could work for a large scale, public facing, mission critical application, in this particular case, Web based e-mail. The introduction of other rich Web applications such as Google Maps helped to demonstrate this to be a viable design pattern for arbitrary applications. Once the pattern was successfully demonstrated multiple times, add in appropriate hype and chatter from the blogging classes and the rest, as they say, is history.

## Implications of Ajax

It should go without saying that the implications of Ajax are significant. A few of the major considerations are presented here for thought. More actionable responses to the network, architecture, and interface challenges caused by Ajax will be presented throughout later chapters.

### Software Market Disruption

If we could truly and effectively deliver a desktop application experience via a Web browser, it would fundamentally change the software industry. Why distribute applications via download if a user can just visit the application and run the latest code? If a Web application just needs a browser, why would I care what operating system is running? Why would I need to save files locally if I could just keep everything online? If these questions seem familiar, they should; they are the same ones posed by Sun and Netscape during the mid-1990s when Java first came about.

While Java never really delivered upon its promises with public facing Web applications, things are much different now. First, the Web market is a bit more mature. Second, software as a service has been demonstrated as a viable business model. Finally, unlike Java, JavaScript is already ubiquitously supported. These conditions suggest a bright future for Ajax-powered Web applications and given the reaction from Microsoft with the introduction of Office Live and other Ajax-based initiatives, there must be some cause for concern in Redmond that the software industry just might change.

### Significant Emphasis on JavaScript

Long underestimated as a significant programming language, with Ajax, JavaScript has finally been recognized as the powerful tool it always has been. Unfortunately, given the past misunderstandings of JavaScript's capabilities, few developers are actually true experts in the language. While this is changing rapidly, it is common for many new Ajax developers to spend a great deal of time first mastering JavaScript language features such as loose typing, reference types, advanced event handling, DOM methods, and prototype OOP style programming before really dealing with Ajax. Or, more often they don't and they write poor applications.

Readers needing more background in JavaScript are encouraged to read Appendix A, the companion book *JavaScript: The Complete Reference 2nd Edition*, and numerous online JavaScript tutorials. Do note the book you are currently reading assumes more than passing knowledge of the JavaScript language, so make sure to brush up on your JavaScript if necessary.

### Increased Network Concerns

Traditional Web applications have a predictable network pattern. Users clicking links and submitting forms are accustomed to clicking the browser's back or reload button to mitigate a momentary network problem. This "layer 8" form of error correction just isn't possible in an Ajax-style Web application where network activity may be happening at any moment. If you do not account for network failures, the Ajax application will appear fragile and certainly not deliver on the promise of the desktop-like experience. Chapter 6 discusses such network concerns in great detail and wraps the solutions into a communications library that will be used to build example Ajax applications.

### Effects upon Traditional Web Architecture

Typical Ajax applications will very likely sit on a single URL with updates happening within the page. This is very different from the architecture traditionally used on the Web where one URL is equated to one page or a particular state within the application. With URLs uniquely identifying page or state, it is easy to provide this address to others, record it as a bookmark, index it as part of a search result, and move around the site or application with the Web browser's history mechanism. In an Ajax application, the URL is typically not uniquely tied to the site or application state. You may be required to either engineer around this or to give up some interface aspects of the Web experience that users are accustomed to. We'll spend plenty of time talking about each and every one of these considerations, particularly in Chapter 9.

### User Interface Effects

Ajax applications afford developers much richer forms of interactions. The typical design patterns of Web applications will need to be extended to take advantage of the technology. We also should build constructs to show network activity, since browser features such as the status bar, loading bar, and pulsating logo do not consistently or necessarily work at all in an Ajax-based Web application. We may further need to add features to ensure that our Web applications continue to be accessible to those with lesser technology and to support access by the disabled. There certainly will be many changes, both visual and nonvisual, from the application of Ajax which we explore throughout the book and especially in Chapter 8.

## Summary

Ajax (Asynchronous JavaScript and XML) is more than an acronym; it is a whole new way to approach Web application design. By sending data behind the scenes using JavaScript, typically in conjunction with the `XMLHttpRequest` object, you can get data incrementally and make partial page updates rather than redrawing the whole window. When applied properly, the effect of Ajax can be wondrous to users producing Web applications that begin to rival desktop software applications in terms of richness and responsiveness. However, Ajax is often misunderstood and misapplied and is not nearly as new and unique as some might have you believe. The next chapter will show that there is in fact a multitude of ways to accomplish Ajax-like magic with JavaScript.