Ajax Basics

An interesting misconception regarding Ajax is that, given all the cool features it has to offer, the JavaScript code must be extremely difficult to implement and maintain. The truth is, however, that beginning your experimentation with the technology could not be simpler. The structure of an Ajax-based server request is quite easy to understand and invoke. You must simply create an object of the XMLHttpRequest type, validate that it has been created successfully, point where it will go and where the result will be displayed, and then send it. That's really all there is to it. If that's all there is to it, then why is it causing such a fuss all of a sudden? It's because Ajax is less about the code required to make it happen and more about what's possible from a functionality, ergonomics, and interface perspective. The fact that Ajax is rather simple to implement from a development point of view is merely icing on a very fine cake. It allows developers to stop worrying about making the code work, and instead concentrate on imagining what might be possible when putting the concept to work. While Ajax can be used for very simple purposes such as loading HTML pages or performing mundane tasks such as form validation, its power becomes apparent when used in conjunction with a powerful server-side scripting language. As might be implied by this book's title, the scripting language I'll be discussing is PHP. When mixing a client side interactive concept such as Ajax with a server-side powerhouse such as PHP, amazing applications can be born. The sky is the limit when these two come together, and throughout this book I'll show you how they can be mixed for incredibly powerful results.

In order to begin making use of Ajax and PHP to create web applications, you must first gain a firm understanding of the basics. It should be noted that Ajax is a JavaScript tool, and so learning the basics of JavaScript will be quite important when attempting to understand Ajax-type applications. Let's begin with the basics.

HTTP Request and Response Fundamentals

In order to understand exactly how Ajax concepts are put together, it is important to know how a web site processes a request and receives a response from a web server. The current standard that browsers use to acquire information from a web server is the HTTP (HyperText Transfer Protocol) method (currently at version HTTP/1.1). This is the means a web browser uses to send out a request from a web site and then receive a response from the web server that is currently in charge of returning the response. HTTP requests work somewhat like e-mail. That is to say that when a request is sent, certain headers are passed along that allow the web server to know exactly what it is to be serving and how to handle the request. While most headers are optional, there is one header that is absolutely required (provided you want more than just the default page on the server): the host header. This header is crucial in that it lets the server know what to serve up.

Once a request has been received, the server then decides what response to return. There are many different response codes. Table 2-1 has a listing of some of the most common ones.

Table 2-1. *Common HTTP Response Codes*

Code	Description
200	OK This response code is returned if the document or file in question is

found and served correctly.

304 Not Modified This response code is returned if a browser has indicated that it has a local, cached copy, and the server's copy has not changed from this cached copy.

401 Unauthorized This response code is generated if the request in question requires authorization to access the requested document.

403 Forbidden This response code is returned if the requested document does not have proper permissions to be accessed by the requestor.

404 Not Found This response code is sent back if the file that is attempting to be accessed could not be found (e.g., if it doesn't exist).

500 Internal Server Error This code will be returned if the server that is being contacted has a problem.

503 Service Unavailable This response code is generated if the server is too overwhelmed to handle the request.

It should be noted that there are various forms of request methods available. A few of them, like GET and POST, will probably sound quite familiar. Table 2-2 lists the available request methods (although generally only the GET and POST methods are used).

Table 2-2. ***HTTP Request Methods***

Method	Description
GET	The most common means of sending a request; simply requests a specific resource from the server
HEAD	Similar to a GET request, except that the response will come back without the response body; useful for retrieving headers
POST	Allows a request to send along user-submitted data (ideal for web-based forms)
PUT	Transfers a version of the file request in question
DELETE	Sends a request to remove the specified document
TRACE	Sends back a copy of the request in order to monitor its progress
OPTIONS	Returns a full list of available methods; useful for checking on what methods a server supports
CONNECT	A proxy-based request used for SSL tunneling.

Now that you have a basic understanding of how a request is sent from a browser to a server and then has a response sent back, it will be simpler to understand how the XMLHttpRequest object works. It is actually quite similar, but operates in the background without the prerequisite page refresh.

The XMLHttpRequest Object

Ajax is really just a concept used to describe the interaction of the client-side XMLHttpRequest object with server-based scripts. In order to make a request to the server through Ajax, an object must be created that can be used for different forms of functionality. It should be noted that the XMLHttpRequest object is both instantiated and handled a tad differently across the browser gamut. Of particular note is that Microsoft Internet Explorer creates the object as an ActiveX control, whereas browsers such as Firefox and Safari use a basic JavaScript object. This is rather crucial in running cross-browser code as it is imperative to be able to run Ajax in any type of browser configuration.

XMLHttpRequest Methods

Once an instance of the XMLHttpRequest object has been created, there are a number of methods available to the user. These methods are expanded upon in further detail in Table 2-3. Depending on how you want to use the object, different methods may become more important than others.

Table 2-3. ***XMLHttpRequest Object Methods***

Method	Description
abort()	Cancels the current request
getAllResponseHeaders()	Returns all HTTP headers as a String type variable
getResponseHeader()	Returns the value of the HTTP header specified in the method
open()	Specifies the different attributes necessary to make a connection to the server; allows you to make selections such as GET or POST (more on that later), whether to connect asynchronously, and which URL to connect to
setRequestHeader()	Adds a label/value pair to the header when sent
send()	Sends the current request

While the methods shown in Table 2-3 may seem somewhat daunting, they are not all that complicated. That being said, let's take a closer look at them.

abort()

The abort method is really quite simple—it stops the request in its tracks. This function can be handy if you are concerned about the length of the connection. If you only want a request to fire for a certain length of time, you can call the abort method to stop the request prematurely.

getAllResponseHeaders()

You can use this method to obtain the full information on all HTTP headers that are being passed. An example set of headers might look like this:

Date: Sun, 13 Nov 2005 22:53:06 GMT
Server: Apache/2.0.53 (Win32) PHP/5.0.3
X-Powered-By: PHP/5.0.3
Content-Length: 527
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: text/html

getResponseHeader("headername")

You can use this method to obtain the content of a particular piece of the header. This method can be useful to retrieve one part of the generally large string obtained from a set of headers. For example, to retrieve the size of the document requested, you could simply call getResponseHeader ("Content-Length").

open ("method","URL","async","username","pswd")

Now, here is where we start to get into the meat and potatoes of the XMLHttpRequest object. This is the method you use to open a connection to a particular file on the server. It is where you pass in the method to open a file (GET or POST), as well as define how the file is to be opened. Keep in mind that not all of the arguments in this function are required and can be customized depending on the situation.

setRequestHeader("label","value")

With this method, you can give a header a label of sorts by passing in a string representing both the label and the value of said label. An important note is that this method may only be invoked after the open() method has been used, and must be used before the send function is called.

send("content")

This is the method that actually sends the request to the server. If the request was sent asynchronously, the response will come back immediately; if not, it will come back after the response is received. You can optionally specify an input string as an argument, which is helpful for processing forms, as it allows you to pass the values of form elements.

XMLHttpRequest Properties

Of course, any object has a complete set of properties that can be used and manipulated in order for it work to its fullest. A complete list of the XMLHttpRequest object properties is presented in Table 2-4. It is important to take note of these properties—you will be making use of them as you move into the more advanced functionality of the object.

Table 2-4. ***XMLHttpRequest Object Properties***

Property	Description
onreadystatechange	Used as an event handler for events that trigger upon state changes
readyState	Contains the current state of the object (0: uninitialized, 1: loading, 2: loaded, 3: interactive, 4: complete)

responseText Returns the response in string format

responseXML Returns the response in proper XML format status Returns the status of the request in numerical format (regular page errors are returned, such as the number 404, which refers to a not found error)

statusText Returns the status of the request, but in string format (e.g., a 404 error would return the string Not Found)

onreadystatechange

The onreadystatechange property is an event handler that allows you to trigger certain blocks of code, or functions, when the state (referring to exactly where the process is at any given time) changes. For example, if you have a function that handles some form of initialization, you could get the main set of functionality you want to fire as soon as the state changes to the complete state.

readyState

The readyState property gives you an in-depth description of the part of the process that the current request is at. This is a highly useful property for exception handling, and can be important when deciding when to perform certain actions. You can use this property to create individual actions based upon how far along the request is. For example, you could have a set of code execute when readyState is loading, or stop executing when readyState is complete.

responseText

The responseText property will be returned once a request has gone through. If you are firing a request to a script of some sort, the output of the script will be returned through this property. With that in mind, most scripts will make use of this property by dumping it into an innerHTML property of an element, thereby asynchronously loading a script or document into a page element.

responseXML

This works similarly to responseText, but is ideal if you know for a fact that the response will be returned in XML format—especially if you plan to use built-in XML-handling browser functionality.

status

This property dictates the response code (a list of common response codes is shown in Table 2-1) that was returned from the request. For instance, if the file requested could not be found, the status will be set to 404 because the file could not be found.

statusText

This property is merely a textual representation of the status property. Where the status property might be set to 404, the statusText would return Not Found. By using both the status and statusText properties together, you can give your user more in-depth knowledge of what has occurred. After all, not many users understand the significance of the number 404.

Cross-Browser Usage

Although at the time of this writing, Microsoft's Internet Explorer continues to dominate the browser market, competitors such as Firefox have been making

significant headway. Therefore, it is as important as ever to make sure your Ajax applications are crossbrowser compatible. One of the most important aspects of the Ajax functionality is that it can be deployed across browsers rather seamlessly, with only a small amount of work required to make it function across most browsers (the exception being rather old versions of the current browsers). Consider the following code snippet, which instantiates an instance of the XMLHttpRequest object, and works within any browser that supports XMLHttpRequest. Figure 2-1 shows the difference between the Internet Explorer and non–Internet Explorer outcomes.

```
//Create a boolean variable to check for a valid Internet Explorer instance.
var xmlhttp = false;
//Check if we are using IE.
try {
//If the Javascript version is greater than 5.
xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
alert ("You are using Microsoft Internet Explorer.");
} catch (e) {
//If not, then use the older active x object.
try {
//If we are using Internet Explorer.
xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
alert ("You are using Microsoft Internet Explorer");
} catch (E) {
//Else we must be using a non-IE browser.
xmlhttp = false;
}
}
//If we are using a non-IE browser, create a javascript instance of the object.
if (!xmlhttp && typeof XMLHttpRequest != 'undefined') {
xmlhttp = new XMLHttpRequest();
alert ("You are not using Microsoft Internet Explorer");
}
```

Figure 2-1. *This script lets you know which browser you are currently using to perform an Ajax-based request.*

As you can see, the process of creating an XMLHttpRequest object may differ, but the end result is always the same; you have a means to create a usable

XMLHttpRequest object. Microsoft becomes a little more complicated in this respect than most other browsers, forcing you to check on which version of Internet Explorer (and, subsequently, JavaScript) the current user is running. The flow of this particular code sample is quite simple. Basically, it checks whether the user is using a newer version of Internet Explorer (by attempting to create the ActiveX Object); if not, the script will default to the older ActiveX Object. If it's determined that neither of these is the case, then the user must be using a non–Internet Explorer browser, and the standard XMLHttpRequest object can thus be created as an actual JavaScript object.
Now, it is important to keep in mind that this method of initiating an XMLHttpRequest object is not the only way to do so. The following code snippet will do largely the same thing, but is quite a bit simpler:

```
var xmlhttp;
//If, the activexobject is available, we must be using IE.
if (window.ActiveXObject){
xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
} else {
//Else, we can use the native Javascript handler.
xmlhttp = new XMLHttpRequest();
}
```

As you can see, this case is a much less code-intensive way to invoke the XMLHttpRequest object. Unfortunately, while it does the job, I feel it is less thorough, and since you are going to be making use of some object-oriented technologies, it makes sense to use the first example for your coding. A large part of using Ajax is making sure you take care of as many cases as possible.

Sending a Request to the Server

Now that you have your shiny, new XMLHttpRequest object ready for use, the natural next step is to use it to submit a request to the server. This can be done in a number of ways, but the key aspect to remember is that you must validate for a proper response, and you must decide whether to use the GET or POST method to do so. It should be noted that if you are using Ajax to retrieve information from the server, the GET method is likely the way to go. If you are sending information to the server, POST is the best way to handle this. I'll go into more depth with this later in the book, but for now, note that GET does not serve very well to send information due to its inherent size limitations.

In order to make a request to the server, you need to confirm a few basic functionality based questions. First off, you need to decide what page (or script) you want to connect to, and then what area to load the page or script into. Consider the following function, which receives as arguments the page (or script) that you want to load and the div (or other object) that you want to load the content into.

```
function makerequest(serverPage, objID) {
var obj = document.getElementById(objID);
xmlhttp.open("GET", serverPage);
xmlhttp.onreadystatechange = function() {
if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
obj.innerHTML = xmlhttp.responseText;
```

```
}
}
xmlhttp.send(null);
}
```

Basically, the code here is taking in the HTML element ID and server page. It then attempts to open a connection to the server page using the open() method of the XMLHttpRequest object. If the readyState property returns a 4 (complete) code and the status property returns a 200 (OK) code, then you can load the response from the requested page (or script) into the innerHTML element of the passed-in object after you send the request.

Basically, what is accomplished here is a means to create a new XMLHttpRequest object and then use it to fire a script or page and load it into the appropriate element on the page. Now you can begin thinking of new and exciting ways to use this extremely simple concept.

Basic Ajax Example

As Ajax becomes an increasingly widely used and available technique, one of the more common uses for it is navigation. It is a rather straightforward process to dynamically load content into a page via the Ajax method. However, since Ajax loads in the content exactly where you ask it to, without refreshing the page, it is important to note exactly where you are loading content.

You should be quite used to seeing pages load from scratch whenever a link is pressed, and you've likely become dependent on a few of the features of such a concept. With Ajax, however, if you scroll down on a page and dynamically load content in with Ajax, it will not move you back to the top of the page. The page will sit exactly where it is and load the content in without much notification.

A common problem with Ajax is that users simply don't understand that anything has happened on the page. Therefore, if Ajax is to be used as a navigational tool, it is important to note that not all page layouts will react well to such functionality. In my experience, pages that rely upon navigational menus on the top of the screen (rather than at the bottom, in the content, or on the sides) and then load in content below it seem to function the best, as content is quite visible and obvious to the user.

Consider the following example, which shows a generic web page that loads in content via Ajax to display different information based on the link that has been clicked.

```
<!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>Sample 2_1</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<script type="text/javascript">
<!—
//Create a boolean variable to check for a valid Internet Explorer instance.
var xmlhttp = false;
```

```
//Check if we are using IE.
try {
//If the Javascript version is greater than 5.
xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
alert ("You are using Microsoft Internet Explorer.");
} catch (e) {
//If not, then use the older active x object.
try {
//If we are using Internet Explorer.
xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
alert ("You are using Microsoft Internet Explorer");
} catch (E) {
//Else we must be using a non-IE browser.
xmlhttp = false;
}
}
//If we are using a non-IE browser, create a javascript instance of the object.
if (!xmlhttp && typeof XMLHttpRequest != 'undefined') {
xmlhttp = new XMLHttpRequest();
alert ("You are not using Microsoft Internet Explorer");
}
function makerequest(serverPage, objID) {
var obj = document.getElementById(objID);
xmlhttp.open("GET", serverPage);
xmlhttp.onreadystatechange = function() {
if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
obj.innerHTML = xmlhttp.responseText;
}
}
xmlhttp.send(null);
}
//-->
</script>
<body onload="makerequest ('content1.html','hw')">
<div align="center">
<h1>My Webpage</h1>
<a href="content1.html" onclick="makerequest('content1.html','hw'); ➥
return false;"> Page 1</a> | <a href="content2.html"➥
onclick="makerequest('content2.html','hw'); ➥
return false;">Page 2</a> | <a href="content3.html" onclick=➥
"makerequest('content3.html','hw'); return false;">Page 3</a> | ➥
<a href="content4.html" onclick="makerequest('content4.html','hw'); return
false;">➥
Page 4</a>
<div id="hw"></div>
</div>
</body>
</html>
<!-- content1.html -->
<div style="width: 770px; text-align: left;">
<h1>Page 1</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod➥
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, ➥
```

```
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.➥
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu ➥
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in➥
culpa qui officia deserunt mollit anim id est laborum.</p>
</div>
<!-- content2.html -->
<div style="width: 770px; text-align: left;">
<h1>Page 2</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod ➥
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, ➥
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.➥
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu ➥
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in ➥
culpa qui officia deserunt mollit anim id est laborum.</p>
</div>
<!-- content3.html -->
<div style="width: 770px; text-align: left;">
<h1>Page 3</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod➥
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,➥
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.➥
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu➥
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in➥
culpa qui officia deserunt mollit anim id est laborum.</p>
</div>
<!-- content4.html -->
<div style="width: 770px; text-align: left;">
<h1>Page 4</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod ➥
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, ➥
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.➥
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu ➥
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in ➥
culpa qui officia deserunt mollit anim id est laborum.</p>
</div>
```

As you can see in Figure 2-2, by making use of Ajax, you can create a fully functional, Ajax navigation–driven site in a manner of minutes. You include the JavaScript required to process the links into <script> tags in the head, and can then make use of the makerequest() function at any time to send a server-side request to the web server without refreshing the page. You can call the makerequest() function on any event (you are using onclick() here) to load content into the respective object that is passed to the function.

Figure 2-2. *An Ajax-based application in full effect.Note the address bar, which shows whether you have refreshed the page as you navigate.*

Using this method to handle navigation is a very nice way to produce a solid break between content and design, as well as create a fast-loading web site. Because the design wrapper only needs to be created once (and content can be loaded on the fly), users will find less lag when viewing the web site, and have a seamless page in front of them at all times. While those users without a fast Internet connection typically have to wait while a site loads using traditional linking methods, they won't have to wait with Ajax. Using the Ajax method allows the content being retrieved from the server to be loaded with little to no obtrusive maneuvering of the web page that the user is viewing.

Summary

To summarize, Ajax can efficiently produce seamless requests to the server while retrieving and manipulating both external scripts and internal content on the fly. It is quite simple to set up, very easy to maintain, and quite portable across platforms. With the right amount of exception handling, you can ensure that most of your site users will see and experience your web site or application exactly as you had envisioned it. You are well on our way to integrating the concept of Ajax into robust PHP applications. In Chapter 3, you'll begin to bring the two web languages together into seamless, powerful web-based applications.