How-To: Build a Web Application with Ajax Part 3

Creating the Page

Now that we have created the `Ajax` object, and set up a simple handler function for the request, it's time to put our code into action.

The Fake Server Page

In the code above, you can see that the target URL for the request is set to a page called `fakeserver.php`.

To use this demonstration code, you'll need to serve both `ajaxtest.html` and `fakeserver.php` from the same PHP-enabled web server. You can do this from an IIS web server with some simple ASP, too. The fake server page is a super-simple page that simulates the varying response time of a web server using the PHP code below:

```
Example 2.19. fakeserver.php



<?php

header('Content-Type: text/plain');

sleep(rand(3, 12));

print 'ok';

?>
```

That's all this little scrap of code does: it waits somewhere between three and 12 seconds, then prints ok.

The `fakeserver.php` code sets the `Content-Type` header of the response to `text/plain`. Depending on the content of the page you pass back, you might choose another `Content-Type` for your response. For example, if you're passing an XML document back to the caller, you would naturally want to use `text/xml`.

This works just as well in ASP, although some features (such as sleep) are not as easily available, as the code below illustrates:

```
Example 2.20. fakeserver.asp



<%

Response.ContentType = "text/plain"

' There is no equivalent to sleep in ASP.

Response.Write "ok"

%>
```

Throughout this book, all of our server-side examples will be written in PHP, although they could just as easily be written in ASP, ASP.NET, Java, Perl, or just about any language that can serve content through a web server.

Use the `setMimeType` Method

Imagine that you have a response that you know contains a valid XML document that you want to parse as XML, but the server insists on serving it to you as text/plain. You can force that response to be parsed as XML in Firefox and Safari by adding an extra call to `setMimeType`, like so:

```
var ajax = new Ajax();
```

```
ajax.setMimeType('text/xml');

ajax.doGet('/fakeserver.php', hand, 'xml');
```
Naturally, you should use this approach only when you're certain that the response from the server will be valid XML, and you can be sure that the browser is Firefox or Safari.

Hitting the Page

Now comes the moment of truth! Hit your local web server, load up `ajaxtest.html`, and see what you get. If everything is working properly, there will be a few moments' delay, and then you'll see a standard JavaScript alert like the one in Figure 2.2 that says simply ok.

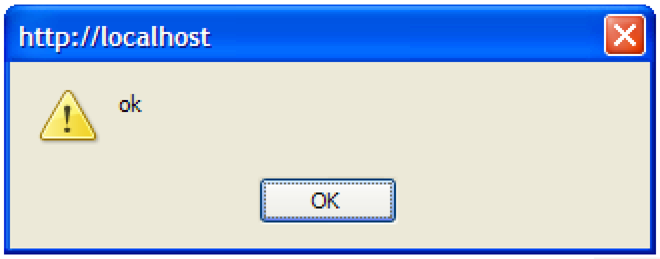


Figure 2.2. Confirmation that your `Ajax` class is working as expected

Now that all is well and our `Ajax` class is functioning properly, it's time to move to the next step.

Example: a Simple AJAX App

Okay, so using the awesome power of AJAX to spawn a tiny little JavaScript alert box that reads `"ok"` is probably not exactly what you had in mind when you bought this book. Let's implement some changes to our example code that will make this XMLHttpRequest stuff a little more useful. At the same time, we'll create that simple monitoring application I mentioned at the start of this chapter. The app will ping a web site and report the time it takes to get a response back.

Laying the Foundations

We’ll start off with a simple HTML document that links to two JavaScript files: `ajax.js`, which contains our library, and `appmonitor1.js`, which will contain the code for our application.

```
Example 2.21. appmonitor1.html



<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN"

    "https://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">

<html xmlns="https://www.w3.org/1999/xhtml">

  <head>

    <meta http-equiv="Content-Type"

        content="text/html; charset=iso-8859-1"
/>

    <title>App Monitor</title>

    <script type="text/javascript"
src="ajax.js"></script>

    <script type="text/javascript"
src="appmonitor1.js"></script>

  </head>

  <body>
```

```
    <div id="pollDiv"></div>

  </body>

</html>
```

You'll notice that there's virtually no content in the body of the page — there's just a single `div` element. This is fairly typical of web apps that rely on AJAX functions. Often, much of the content of AJAX apps is created by JavaScript dynamically, so we usually see a lot less markup in the body of the page source than we would in a non-AJAX web application for which all the content was generated by the server. However, where AJAX is not an absolutely essential part of the application, a plain HTML version of the application should be provided.

We'll begin our `appmonitor1.js` file with some simple content that makes use of our `Ajax` class:

```
Example 2.22. appmonitor1.js (excerpt)



var start = 0;

var ajax = new Ajax();



var doPoll = function() {

  start = new Date();

  start = start.getTime();

  ajax.doGet('/fakeserver.php?start=' + start,
showPoll);
}



window.onload = doPoll;
```

We'll use the start variable to record the time at which each request starts — this figure will be used to calculate how long each request takes. We make start a global variable so that we don't have to gum up the works of our `Ajax` class with extra code for timing requests — we can set the value of start immediately before and after our calls to the `Ajax` object.

The `ajax` variable simply holds an instance of our `Ajax` class.

The `doPoll` function actually makes the HTTP requests using the `Ajax` class. You should recognize the call to the `doGet` method from our original test page.

Notice that we've added to the target URL a query string that has the start value as a parameter. We're not actually going to use this value on the server; we're just using it as a random value to deal with Internet Explorer's overzealous caching. IE caches all `GET` requests made with `XMLHttpRequest`, and one way of disabling that "feature" is to append a random value into a query string. The milliseconds value in start can double as that random value. An alternative to this approach is to use the `setRequestHeader` method of the `XMLHttpRequest` class to set the `If-Modified-Since` header on the request.

Finally, we kick everything off by attaching `doPoll` to the `window.onload` event.

Handling the Result with `showPoll`

The second parameter we pass to `doGet` tells the `Ajax` class to pass responses to the function `showPoll`. Here's the code for that function:

```
Example 2.23. appmonitor1.js (excerpt)


var showPoll = function(str) {

  var pollResult = '';

  var diff = 0;

  var end = new Date();

  if (str == 'ok') {

    end = end.getTime();

    diff = (end - start) / 1000;

    pollResult = 'Server response time: ' + diff +
' seconds';

  }

  else {

    pollResult = 'Request failed.';

  }

  printResult(pollResult);

  var pollHand = setTimeout(doPoll, 15000);

}
```
This is all pretty simple: the function expects a single parameter, which should be the string `ok` returned from `fakeserver.php` if everything goes as expected. If the response is correct, the code does

the quick calculations needed to figure out how long the response took, and creates a message that contains the result. It passes that message to pollResult for display.

In this very simple implementation, anything other than the expected response results in a fairly terse and unhelpful message: Request failed. We'll make our handling of error conditions more robust when we upgrade this app in the next chapter.

Once `pollResult` is set, it's passed to the `printResult` function:

```
Example 2.24. appmonitor1.js (excerpt)



function printResult(str) {

  var pollDiv =
document.getElementById('pollDiv');

  if (pollDiv.firstChild) {

    pollDiv.removeChild(pollDiv.firstChild);

  }

  pollDiv.appendChild(document.createTextNode(str))
;

}
```

The `printResult` function displays the message that was sent from `showPoll` inside the lone `div` in the page.

Note the test in the code above, which is used to see whether our `div` has any child nodes. This checks for the existence of any text nodes, which could include text that we added to this `div` in previous iterations, or the text that was contained inside the `div` in the page markup, and then removes them. If you don't remove existing text nodes, the code will simply append the new result to the page as a new text node: you'll display a long string of text to which more text is continually being appended.

Why Not Use `innerHTML`?

You could simply update the `innerHTML` property of the `div`, like so:

```
document.getElementById('pollDiv').innerHTML = str;
```

The `innerHTML` property is not a web standard, but all the major browsers support it. And, as you can see from the fact that it's a single line of code (as compared with the four lines needed for DOM methods), sometimes it's just easier to use than the DOM methods. Neither way of displaying content on your page is inherently better.

In some cases, you may end up choosing a method based on the differences in rendering speeds of these two approaches (`innerHTML` can be faster than DOM methods). In other cases, you may base your decision on the clarity of the code, or even on personal taste.

Starting the Process Over Again

Finally, `showPoll` starts the entire process over by scheduling a call to the original `doPoll` function in 15 seconds time using `setTimeout`, as shown below:

```
Example 2.25. appmonitor1.js (excerpt)



var pollHand = setTimeout(doPoll, 15000);
```

The fact that the code continuously invokes the `doPoll` function means that once the page loads, the HTTP requests polling the `fakeserver.php` page will continue to do so until that page is closed. The `pollHand` variable is the interval ID that allows you to keep track of the pending operation, and cancel it using `clearTimeout`.

The first parameter of the `setTimeout` call, `doPoll`, is a pointer to the main function of the application; the second represents the length of time, in seconds, that must elapse between requests.

Full Example Code

Here's all the code from our first trial run with this simple monitoring application.

```
Example 2.26. appmonitor1.js



var start = 0;

var ajax = new Ajax();



var doPoll = function() {

  start = new Date();

  start = start.getTime();

  ajax.doGet('/fakeserver.php?start=' + start,
showPoll);

}



window.onload = doPoll;
```

```
var showPoll = function(str) {

  var pollResult = '';

  var diff = 0;

  var end = new Date();

  if (str == 'ok') {

    end = end.getTime();

    diff = (end - start)/1000;

    pollResult = 'Server response time: ' + diff +
' seconds';

  }

  else {

    pollResult = 'Request failed.';

  }

  printResult(pollResult);

  var pollHand = setTimeout(doPoll, 15000);

}


function printResult(str) {
  var pollDiv =
document.getElementById('pollDiv');

  if (pollDiv.firstChild) {

    pollDiv.removeChild(pollDiv.firstChild);

  }

  pollDiv.appendChild(document.createTextNode(str))
;

}
```

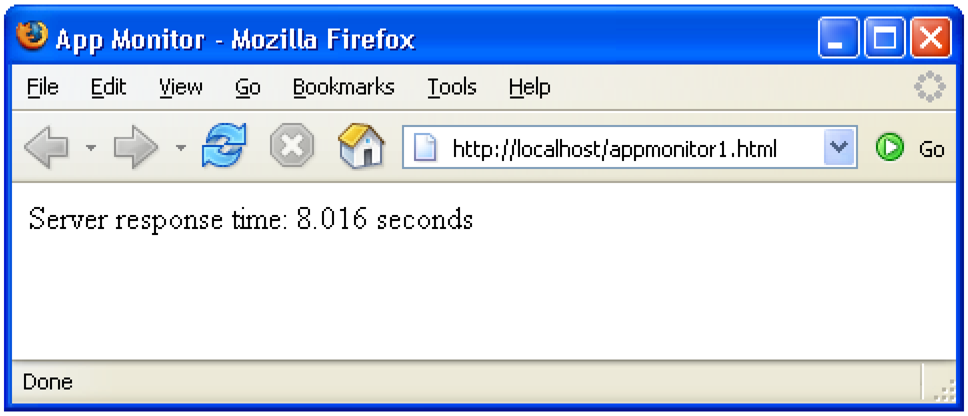
In a bid to follow good software engineering principles, I've separated the JavaScript code from the markup, and put them in two different files.

I'll be following a similar approach with all the example code for this book, separating each example's markup, JavaScript code, and CSS into separate files. This little monitoring app is so basic that it has no CSS file. We'll be adding a few styles to make it look nicer in the next chapter.

Running the App

Try loading the page in your browser. Drop it into your web server's root directory, and open the page in your browser.

If the `fakeserver.php` page is responding properly, you'll see something like the display shown in Figure 2.3.


App Monitor - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/appmonitor1.html
Go
Done

Server response time: 8.016 seconds

Figure 2.3. Running the simple monitoring application

Further Reading

Here are some online resources that will help you learn more about the techniques and concepts in this chapter.

JavaScript's Object Model

- http://docs.sun.com/source/816-6409-10/obj.htm
- http://docs.sun.com/source/816-6409-10/obj2.htm

Check out these two chapters on objects from the Client-Side JavaScript Guide for version 1.3 of JavaScript, hosted by Sun Microsystems. The first chapter explains all the basic concepts you need to understand how to work with objects in JavaScript. The second goes into more depth about JavaScript's prototype-based inheritance model, allowing you to leverage more of the power of object-oriented coding with JavaScript.

This is a brief introduction to creating private instance variables with JavaScript objects. It will help you get a deeper understanding of JavaScript's prototype-based inheritance scheme.

`XMLHttpRequest`

Here's a good reference page from the Apple Developer Connection. It gives a nice overview of the XMLHttpRequest class, and a reference table of its methods and properties.

This article, originally posted in 2002, continues to be updated with new information. It includes information on making HEAD requests (instead of just GET or POST), as well as JavaScript Object Notation (JSON), and SOAP.

This is XULPlanet's exhaustive reference on the `XMLHttpRequest` implementation in Firefox.

Here's another nice overview, which also shows some of the lesser-used methods of the `XMLHttpRequest` object, such as `overrideMimeType`, `setRequestHeader`, and `getResponseHeader`. Again, this reference is focused on implementation in Firefox.

This is Microsoft's documentation on MSDN of its implementation of `XMLHttpRequest`.

Summary

```
XMLHttpRequest is at the heart of AJAX. It gives
scripts within the browser the ability to make
their own requests and get content from the server.
The simple AJAX library we built in this chapter
provided a solid understanding of how
XMLHttpRequest works, and that understanding will
help you when things go wrong with your AJAX code
(whether you're using a library you've built
yourself, or one of the many pre-built toolkits and
libraries listed in Appendix A, AJAX Toolkits). The
sample app we built in this chapter gave us a
chance to dip our toes into the AJAX pool -- now
it's time to dive in and learn to swim.
```

Chapter 3. The "A" in AJAX

It's flying over our heads in a million pieces.

-- *Willy Wonka, Willy Wonka & the Chocolate Factory*

The "A" in AJAX stands for "asynchronous," and while it's not nearly as cool as the letter "X," that "A" is what makes AJAX development so powerful. As we discussed in Chapter 1, AJAX: the Overview, AJAX's ability to update sections of an interface asynchronously has given developers a much greater level of control over the interactivity of the apps we build, and a degree of power that's driving web apps into what was previously the domain of desktop applications alone.

Back in the early days of web applications, users interacted with data by filling out forms and submitting them. Then they'd wait a bit, watching their browser's "page loading" animation until a whole new page came back from the server. Each data transaction between the browser and server was large and obvious, which made it easy for users to figure out what was going on, and what state their data was in.

As AJAX-style development becomes more popular, users can expect more interactive, "snappy" user interfaces. This is a good thing for users, but presents new challenges for the developers working to deliver this increased functionality. In an AJAX application, users alter data in an ad hoc fashion, so it's easy for both the user and the application to become confused about the state of that data.

The solution to both these issues is to display the application's status, which keeps users informed about what the application is doing. This makes the application seem very responsive, and gives users important guidance about what's happening to their data. This critical part of AJAX web application development is what separates the good AJAX apps from the bad.

Planned Application Enhancements

To create a snappy user interface that keeps users well-informed of the application's status, we'll take the monitoring script we developed in the previous chapter, and add some important functionality to it. Here's what we're going to add:

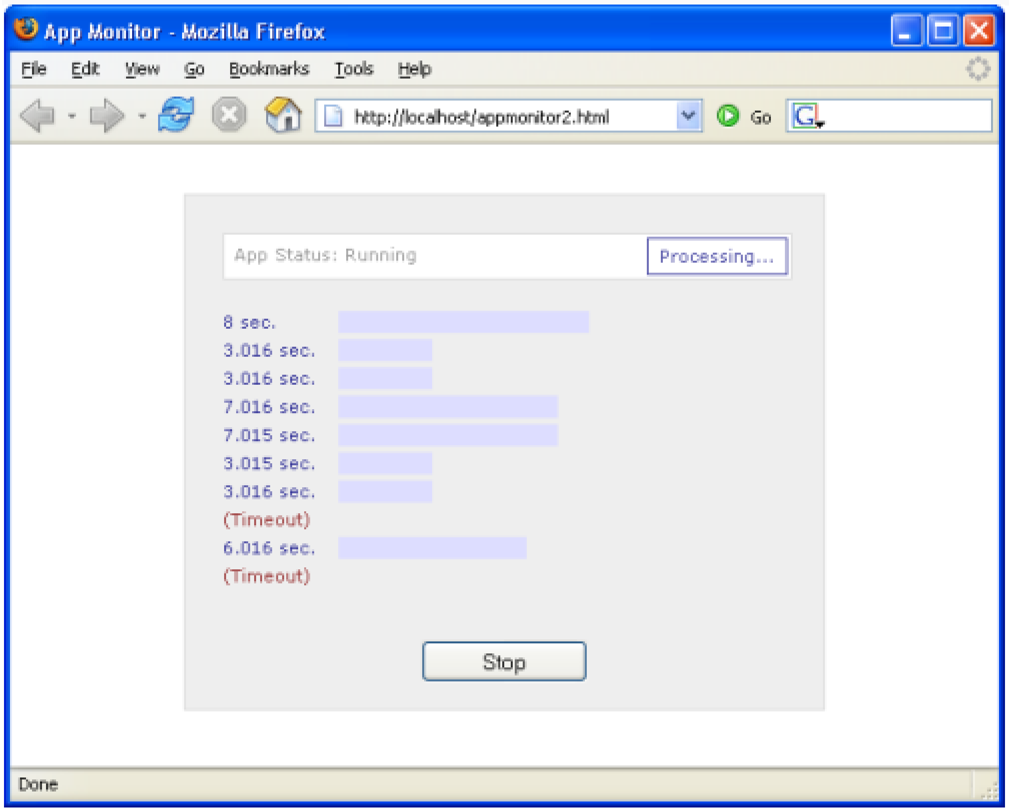
- a way for the system administrator to configure the interval between polls and the timeout threshold
- an easy way to start and stop the monitoring process

- a bar graph of response times for previous requests; the number of entries in the history list will be user-configurable
- user notification when the application is in the process of making a request
- graceful handling of request timeouts

Figure 3.1 shows what the running application will look like once we're done with all the enhancements.

The code for this application is broken up into three files: the markup in `appmonitor2.html`, the JavaScript code in `appmonitor2.js`, and the styles in `appmonitor2.css`. To start with, we'll link all the required files in to `appmonitor2.html`:

```
Example 3.1. appmonitor2.html (excerpt)



<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN"

    "https://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">

<html xmlns="https://www.w3.org/1999/xhtml">

  <head>

    <meta http-equiv="Content-Type"

        content="text/html; charset=iso-8859-1"
/>

    <title>App Monitor</title>
    <script type="text/javascript"
src="ajax.js"></script>

    <script type="text/javascript"
src="appmonitor2.js"></script>

    <link rel="stylesheet" href="appmonitor2.css"

        type="text/css" />

  </head>

  <body>

  </body>

</html>
```


App Monitor - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/appmonitor2.html
Go
Stop
Done

```
App Status: Running    Processing...

8 sec.
3.016 sec.
3.016 sec.
7.016 sec.
7.015 sec.
3.015 sec.
3.016 sec.
(Timeout)
6.016 sec.
(Timeout)
```

Figure 3.1. The running application

Organizing the Code

All this new functionality will add a lot more complexity to our app, so this is a good time to establish some kind of organization within our code (a much better option than leaving everything in the global scope). After all, we're building a fully functional AJAX application, so we'll want to have it organized properly.

We'll use object-oriented design principles to organize our app. And we'll start, of course, with the creation of a base class for our application — the `Monitor` class.

Typically, we'd create a class in JavaScript like this:

```
function Monitor() {

  this.firstProperty = 'foo';

  this.secondProperty = true;

  this.firstMethod = function() {

    // Do some stuff here

  };

}
```

This is a nice, normal constructor function, and we could easily use it to create a `Monitor` class (or a bunch of them if we wanted to).

Loss of Scope with `setTimeout`

Unfortunately, things will not be quite so easy in the case of our application. We're going to use a lot of calls to `setTimeout` (as well as `setInterval`) in our app, so the normal method of creating JavaScript classes may prove troublesome for our `Monitor` class.

The `setTimeout` function is really handy for delaying the execution of a piece of code, but it has a serious drawback: it runs that code in an execution context that's different from that of the object. (We talked a little bit about this problem, called loss of scope, in the last chapter.)

This is a problem because the object keyword `this` has a new meaning in the new execution context. So, when you use it within your class, it suffers from a sudden bout of amnesia — it has no idea what it is!

This may be a bit difficult to understand; let's walk through a quick demonstration so you can actually see this annoyance in action. You might remember the `ScopeTest` class we looked at in the last chapter. To start with, it was a simple class with one property and one method:

```
function ScopeTest() {

  this.message = "Greetings from ScopeTest!";

  this.doTest = function() {

    alert(this.message);

  };

}

var test = new ScopeTest();

test.doTest();
```

The result of this code is the predictable JavaScript alert box with the text "Greetings from ScopeTest!"

Let's change the doTest method so that it uses `setTimeout` to display the message in one second's time.

```
function ScopeTest() {

  this.message = "Greetings from ScopeTest!";

  this.doTest = function() {

    var onTimeout = function() {

      alert(this.message);

    };

    setTimeout(onTimeout, 1000);

  };

}

var test = new ScopeTest();

test.doTest();
```

Instead of our greeting message, the alert box that results from this version of the code will read "undefined." Because we called `onTimeout` with `setTimeout`, `onTimeout` is run within a new execution context. In that execution context, this no longer refers to an instance of `ScopeTest`, so `this.message` has no meaning.

The simplest way to deal with this problem of loss of scope is by making the `Monitor` class a special kind of class, called a singleton.

Continue with Part 4: Singletons in JavaScript…

Courtesy: https://www.sitepoint.com/build-your-own-ajax-web-apps/

Modified: 2021.10.04.7.10.AM

Dököll Solutions,. Inc.