How-To: Build a Web Application with Ajax

The term AJAX, originally coined by Jesse James Garrett of Adaptive Path in his essay AJAX: A New Approach To Web Applications, is an acronym for "Asynchronous JavaScript And XML." That's a bit of a mouthful, but it's simply describing a technique that uses JavaScript to refresh a page's contents from a web server without having to reload the entire page. This is different from the traditional method of updating web pages, which requires the browser to refresh the entire page in order to display any changes to the content.

Similar techniques have been around in one form or another (often achieved with the help of some clever hacks) for quite a while. But the increasing availability of the XMLHttpRequest class in browsers, the coining of the catchy term AJAX, and the advent of a number of high-profile examples such as Google Maps, Gmail, Backpack, and Flickr, have allowed these kinds of highly interactive web applications to begin to gain traction in the development world.

As the term AJAX has become more widespread, its definition has expanded to refer more generally to browser-based applications that behave much more dynamically than old-school web apps. This new crop of AJAX web applications make more extensive use of interaction techniques like edit-in-place text, drag-and-drop, and CSS animations or transitions to effect changes within the user interface. This tutorial will explain those techniques, and show you how to develop AJAX web applications of your own.

This tutorial is an excerpt from my new book, *Build Your Own AJAX Web Applications*. In the three chapters presented here, we'll discuss the basics of AJAX and learn how it ticks, before delving into the wonderful world of XMLHttpRequest. After we've played around with it, exploring its inner workings, making requests, and updating our application page asynchronously, we begin to develop our first true AJAX application.

It's going to be quite a ride, so I hope you're ready for some adventure! If you'd rather read these chapters to offline, download the .pdf version of them. But now, let's get a solid grounding in AJAX.

Chapter 1. AJAX: the Overview

He's escaping, idiot! Dispatch War Rocket Ajax! To bring back his body!

— General Kala, Flash Gordon

AJAX Web Applications

AJAX can be a great solution for many web development projects — it can empower web apps to step up and take over a lot of the ground that previously was occupied almost exclusively by desktop applications.

All the same, it's important to keep in mind that AJAX is not a sort of magic fairy dust that you can sprinkle on your app to make it whizzy and cool. Like any other new development technique, AJAX isn't difficult to mis-use, and the only thing worse than a horrible, stodgy, old-school web app is a horrible, poorly executed AJAX web app.

When you apply it to the right parts of your web application, in the right ways, AJAX can enhance users' experience of your application significantly. AJAX can improve the interactivity and speed of your app, ultimately making that application easier, more fun, and more intuitive to use.

Often, AJAX applications are described as being "like a desktop application in the browser." This is a fairly accurate description — AJAX web apps are significantly more responsive than traditional, old-fashioned web applications, and they can provide levels of interactivity similar to those of desktop applications.

But an AJAX web app is still a remote application, and behaves differently from a desktop application that has access to local storage. Part of your job as an AJAX developer is to craft applications

that feel responsive and easy to use despite the communication that must occur between the app and a distant server. Fortunately, the AJAX toolbox gives you a number of excellent techniques to accomplish exactly that.

The Bad Old Days

One of the first web development tasks that moved beyond serving simple, static HTML pages was the technique of building pages dynamically on the web server using data from a back-end data store.

Back in the "bad old days" of web development, the only way to create this dynamic, database-driven content was to construct the entire page on the server side, using either a CGI script (most likely written in Perl), or some server component that could interpret a scripting language (such as Microsoft's Active Server Pages). Even a single change to that page necessitated a round trip from browser to server — only then could the new content be presented to the user.

In those days, the normal model for a web application's user interface was a web form that the user would fill out and submit to the server. The server would process the submitted form, and send an entirely new page back to the browser for display as a result. So, for example, the completion of a multi-step, web-based "wizard" would require the user to submit a form — thereby prompting a round-trip between the browser and the server — for each step.

Granted, this was a huge advance on static web pages, but it was still a far cry from presenting a true "application" experience to end-users.

Prehistoric AJAX

Early web developers immediately began to look for tricks to extend the capabilities of that simple forms-based model, as they strove to create web applications that were more responsive and interactive. These hacks, while fairly ad hoc and crude, were the first steps web developers took toward the kind of interactivity we see in today's

AJAX applications. But, while these tricks and workarounds often provided serviceable, working solutions, the resulting code was not a pretty sight.

Nesting Framesets

One way to get around the problem of having to reload the entire page in order to display even the smallest change to its content was the hideous hack of nesting framesets within other framesets, often several levels deep. This technique allowed developers to update only selected areas of the screen, and even to mimic the behavior of tab-style navigation interfaces in which users' clicking on tabs in one part of the screen changed content in another area.

This technique resulted in horrible, unmaintainable code with profusions of pages that had names like EmployeeEditWizardMiddleLowerRight.asp.

The Hidden `iframe`

The addition of the `iframe` in browsers like Internet Explorer 4 made things much less painful. The ability to hide the iframe completely led to the development of another neat hack: developers would make HTTP requests to the server using a hidden iframe, then insert the content into the page using JavaScript and DHTML. This provided much of the same functionality that's available through modern AJAX, including the ability to submit data from forms without reloading the page — a feat that was achieved by having the form submit to the hidden iframe. The result was returned by the server to the `iframe`, where the page's JavaScript could access it.

The big drawback of this approach (beyond the fact that it was, after all, a hack) was the annoying burden of passing data back and forth between the main document and the document in the iframe.

Remote Scripting

Another early AJAX-like technique, usually referred to as remote scripting, involved setting the `src` attribute of a `<script>` tag to load pages that contained dynamically generated JavaScript.

This had the advantage of being much cleaner than the hidden `iframe` hack, as the JavaScript generated on the server would load right into the main document. However, only simple GET requests were possible using this technique.

What Makes AJAX Cool

This is why AJAX development is such an enormous leap forward for web development: instead of having to send everything to the server in a single, huge mass, then wait for the server to send back a new page for rendering, web developers can communicate with the server in smaller chunks, and selectively update specific areas of the page based on the server's responses to those requests. This is where the word asynchronous in the AJAX acronym originated.

It's probably easiest to understand the idea of an asynchronous system by considering its opposite — a synchronous system. In a synchronous system, everything occurs in order. If a car race was a synchronous system, it would be a very dull affair. The car that started first on the grid would be the first across the finish line, followed by the car that started second, and so on. There would be no overtaking, and if a car broke down, the traffic behind would be forced to stop and wait while the mechanics made their repairs.

Traditional web apps use a synchronous system: you must wait for the server to send you the first page of a system before you can request the second page, as shown in Figure 1.1.

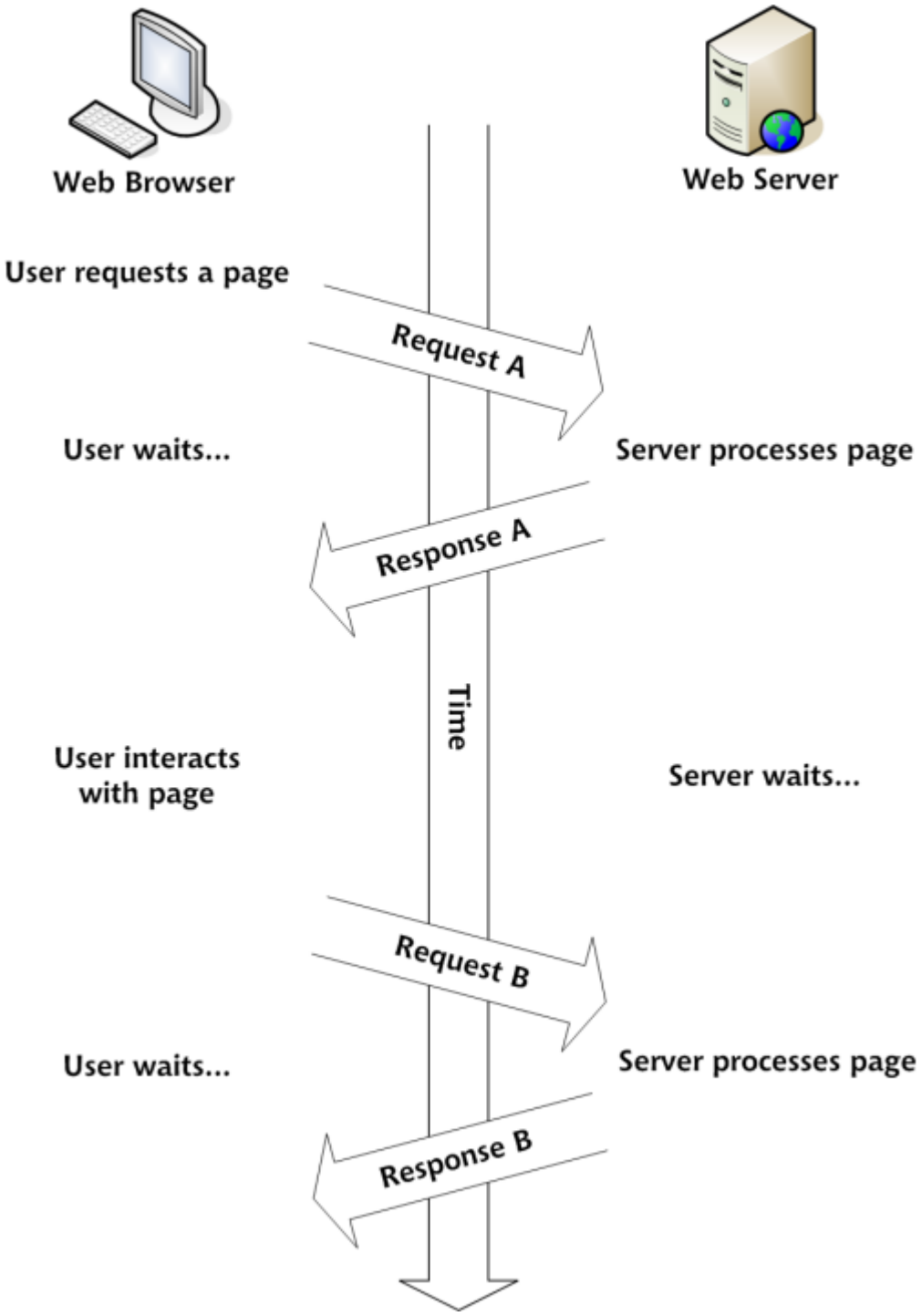

Figure 1.1. A traditional web app is a synchronous system

An asynchronous car race would be a lot more exciting. The car in pole position could be overtaken on the first corner, and the car that starts from the back of the grid could weave its way through the field and cross the finish line in third place. The HTTP requests from the

browser in an AJAX application work in exactly this way. It's this ability to make lots of small requests to the server on a needs-basis that makes AJAX development so cool. Figure 1.2 shows an AJAX application making asynchronous requests to a web server.

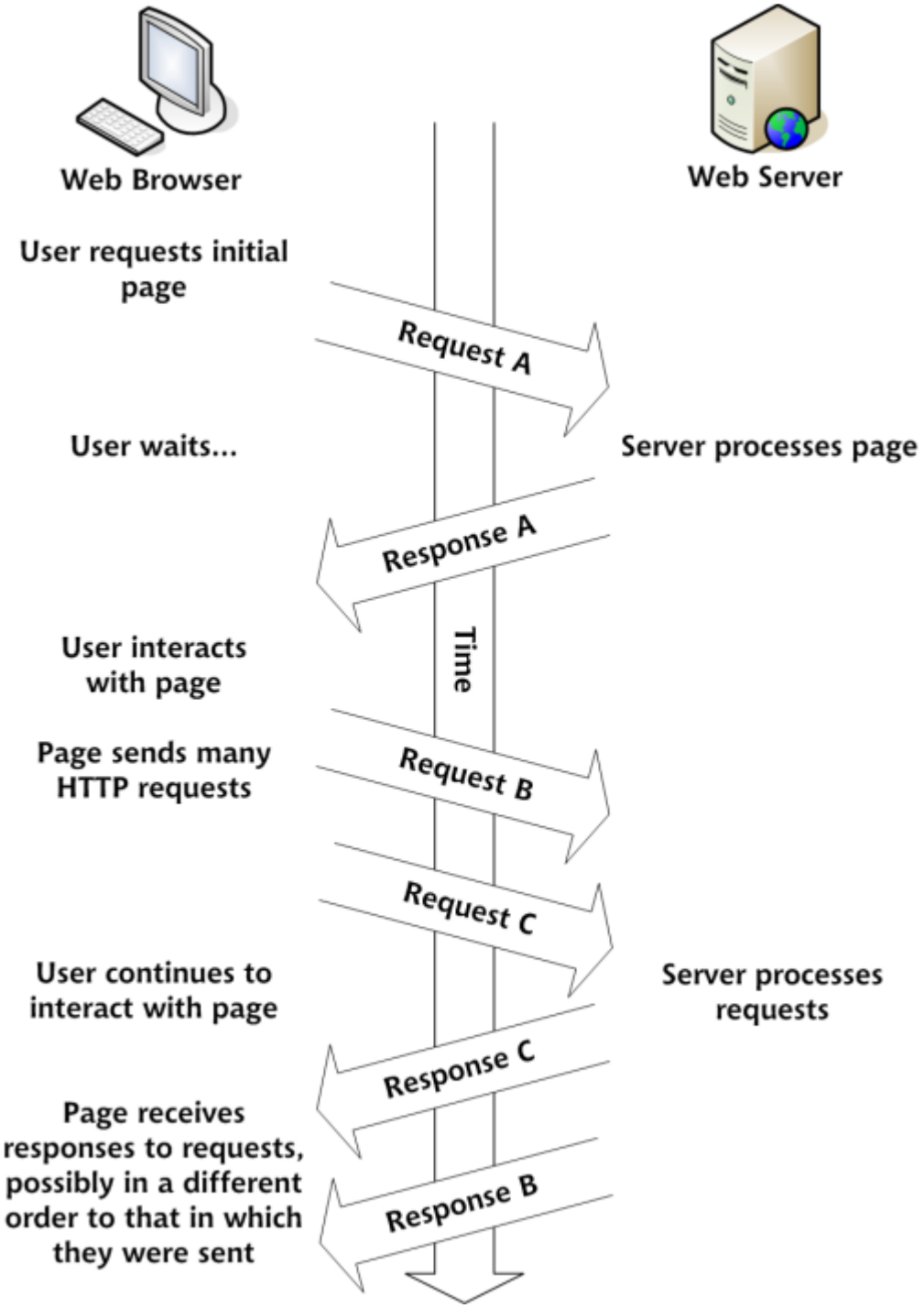

Figure 1.2. An AJAX web app is an asynchronous system

The end result is an application that feels much more responsive, as users spend significantly less time waiting for requests to process, and don't have to wait for an entire new web page to come across the wire, and be rendered by their browsers, before they can view the results.

AJAX Technologies

The technologies that are used to build AJAX web applications encompass a number of different programming domains, so AJAX development is neither as straightforward as regular applications development, nor as easy as old-school web development.

On the other hand, the fact that AJAX development embraces so many different technologies makes it a lot more interesting and fun. Here's a brief listing of the technologies that work together to make an AJAX web application:

- XML
- the W3C DOM
- CSS
- XMLHttpRequest
- JavaScript

Through the rest of this chapter, we'll meet each of these technologies and discuss the roles they play in an AJAX web application.

Data Exchange and Markup: XML

XML (XML stands for Extensible Markup Language — not that anyone ever calls it that outside of textbooks.) is where AJAX gets its letter "X." This is fortunate, because tech acronyms are automatically seen as being much cooler if they contain the letter "X." (Yes, I am kidding!)

Data Exchange Lingua Franca

XML often serves as the main data format used in the asynchronous HTTP requests that communicate between the browser and the server

in an AJAX application. This role plays to XML's strengths as a neutral and fairly simple data exchange format, and also means that it's relatively easy to reuse or reformat content if the need arises.

There are, of course, numerous other ways to format your data for easy exchange between the browser and the server (such as CSV (comma separated values), JSON (JavaScript object notation), or simply plain text), but XML is one of the most common.

XML as Markup

The web pages in AJAX applications consist of XHTML markup, which is actually just a flavor of XML. XHTML, as the successor to HTML, is very similar to it. It's easily picked up by any developer who's familiar with old-school HTML, yet it boasts all the benefits of valid XML. There are numerous advantages to using XHTML:

- It offers lots of standard tools and script libraries for viewing, editing, and validating XML.
- It's forward-compatible with newer, XML-compatible browsers.
- It works with either the HTML Document Object Model (DOM) or the XML DOM.
- It's more easily repurposed for viewing in non-browser agents.

Some of the more pedantic folks in the development community insist that people should not yet be using XHTML. They believe very strongly that XHTML, since it is actual XML, should not be used at all unless it can be served with a proper HTTP `Content-Type` header of `application/xhtml+xml` (`text/xml` and `application/xml` would also be okay, though they're less descriptive) for which, at present, there is still limited browser support. (Internet Explorer 6 and 7 do not support it at all.)

In practice, you can serve XHTML to the browser with a `Content-Type` of `text/html`, as all the mainstream browsers render correctly all XHTML documents served as text/html. Although browsers will treat your code as plain old HTML, other programs can still interpret it

as XML, so there's no practical reason not to "future-proof" your markup by using it.

If you happen to disagree with me, you can choose instead to develop using the older HTML 4.01 standard. This is still a viable web standard, and is a perfectly legitimate choice to make in developing your web application.

XHTML and this Book

Most of the code examples in this book will use XHTML 1.0 Strict. The iframe element is not available in Strict, so the few code examples we show using the iframe will be XHTML 1.0 Transitional.

The World Wide Web Consortium maintains an FAQ on the differences between HTML and XHTML.

W3C Document Object Model

The Document Object Model (DOM) is an object-oriented representation of XML and HTML documents, and provides an API for changing the content, structure, and style of those documents.

Originally, specific browsers like Netscape Navigator and Internet Explorer provided differing, proprietary ways to manipulate HTML documents using JavaScript. The DOM arose from efforts by the World Wide Web Consortium (W3C) to provide a platform- and browser-neutral way to achieve the same tasks.

The DOM represents the structure of an XML or HTML document as an object hierarchy, which is ideal for parsing by standard XML tools.

DOM Manipulation Methods

JavaScript provides a large API for dealing with these DOM structures, in terms of both parsing and manipulating the document. This is one of the primary ways to accomplish the smaller, piece-by-piece changes to a web page that we see in an AJAX application. (Another

method is simply to change the `innerHTML` property of an element. This method is not well documented in any standard, though it's widely supported by mainstream browsers.)

DOM Events

The other important function of the DOM is that it provides a standard means for JavaScript to attach events to elements on a web page. This makes possible much richer user interfaces, because it allows you to give users opportunities to interact with the page beyond simple links and form elements.

A great example of this is drag-and-drop functionality, which lets users drag pieces of the page around on the screen, and drop them into place to trigger specific pieces of functionality. This kind of feature used to exist only in desktop applications, but now it works just as well in the browser, thanks to the DOM.

Presentation: CSS

CSS (Cascading Style Sheets) provides a unified method for controlling the appearance of user interface elements in your web application. You can use CSS to change almost any aspect of the way the page looks, from font sizes, colors, and spacing, to the positioning of elements.

In an AJAX application, one very good use of CSS is to provide user-interface feedback (with CSS-driven animations and transitions), or to indicate portions of the page with which the user can interact (with changes to color or appearance triggered, for example, by mouseovers). For example, you can use CSS transitions to indicate that some part of your application is waiting for an HTTP request that's processing on the server.

CSS manipulation figures heavily in the broader definition of the term AJAX — in various visual transitions and effects, as well as in drag-and-drop and edit-in-place functionality.

Communication: `XMLHttpRequest`

`XMLHttpRequest`, a JavaScript class with a very easy-to-use interface, sends and receives HTTP requests and responses to and from web servers. The `XMLHttpRequest` class is what makes true AJAX application development possible. The HTTP requests made with `XMLHttpRequest` work just as if the browser were making normal requests to load a page or submit a form, but without the user ever having to leave the currently loaded web page.

Microsoft first implemented `XMLHttpRequest` in Internet Explorer 5 for Windows as an ActiveX object. The Mozilla project provided a JavaScript-native version with a compatible API in the Mozilla browser, starting in version 1.0. (It's also available in Firefox, of course.) Apple has added `XMLHttpRequest` to Safari since version 1.2.

The response from the server — either an XML document or a string of text — can be passed to JavaScript to use however the developer sees fit — often to update some piece of the web application's user interface.

Putting it All Together: JavaScript

JavaScript is the glue that holds your AJAX application together. It performs multiple roles in AJAX development:

- controlling HTTP requests that are made using `XMLHttpRequest`
- parsing the result that comes back from the server, using either DOM manipulation methods, XSLT, or custom methods, depending on the data exchange format used
- presenting the resulting data in the user interface, either by using DOM manipulation methods to insert content into the web page, by updating an element's `innerHTML` property, or by changing elements' CSS properties

Because of its long history of use in lightweight web programming (and at the hands of inexperienced programmers), JavaScript has not been seen by many traditional application developers as a "serious programming language," despite the fact that, in reality, it's a fully-featured, dynamic language capable of supporting object-oriented programming methodologies.

The misperception of JavaScript as a "toy language" is now changing rapidly as AJAX development techniques expand the power and functionality of browser-based applications. As a result of the advent of AJAX, JavaScript now seems to be undergoing something of a renaissance, and the explosive growth in the number of JavaScript toolkits and libraries available for AJAX development is proof of the fact.

Summary

In this chapter, we had a quick overview of AJAX and the technologies that make it tick. We looked at some of the horrible coding contortions that developers had to endure back in the bad old days to create something resembling an interactive UI, and we saw how AJAX offers a huge improvement on those approaches. With a decent command of the building blocks of AJAX — XML, the DOM, CSS, XMLHttpRequest, and JavaScript, which ties them all together — you have everything you need to start building dynamic and accessible AJAX sites.

Chapter 2. Basic XMLHttpRequest

I can't wait to share this new wonder, The people will all see its light, Let them all make their own music, The priests praise my name on this night.

— Rush, Discovery

It's `XMLHttpRequest` that gives AJAX its true power: the ability to make asynchronous HTTP requests from the browser and pull down content in small chunks.

Web developers have been using tricks and hacks to achieve this for a long time, while suffering annoying limitations: the invisible iframe hack forced us to pass data back and forth between the parent document and the document in the `iframe`, and even the "remote scripting" method was limited to making GET requests to pages that contained JavaScript.

Modern AJAX techniques, which use XMLHttpRequest, provide a huge improvement over these kludgy methods, allowing your app to make both GET and POST requests without ever completely reloading the page.

In this chapter, we'll jump right in and build a simple AJAX web application — a simple site-monitoring application that pings a page on a web server to a timed schedule. But before we start making the asynchronous HTTP requests to poll the server, we'll need to simplify the use of the XMLHttpRequest class by taking care of all of the little browser incompatibilities, such as the different ways XMLHttpRequest objects are instantiated, inside a single, reusable library of code.

A Simple AJAX Library

One approach to simplifying the use of the `XMLHttpRequest` class would be to use an existing library of code. Thanks to the increasing popularity of AJAX development, there are literally dozens of libraries, toolkits, and frameworks available that make `XMLHttpRequest` easier to use.

But, as the code for creating an instance of the `XMLHttpRequest` class is fairly simple, and the API for using it is easy to understand, we'll just write a very simple JavaScript library that takes care of the basic stuff we need.

Stepping through the process of creating your own library will ensure you know how the `XMLHttpRequest` class works, and will help you get more out of those other toolkits or libraries when you do decide to use them.

Starting our `Ajax` Class

We'll start by creating a basic class, called `Ajax`, in which we'll wrap the functionality of the `XMLHttpRequest` class.

I've Never done Object Oriented Programming in JavaScript — Help!

In this section, we'll start to create classes and objects in JavaScript. If you've never done this before, don't worry — it's quite simple as long as you know the basics of object oriented programming.

In JavaScript, we don't declare classes with complex syntax like we would in Java, C++ or one of the .NET languages; we simply write a constructor function to create an instance of the class. All we need to do is:

- provide a constructor function — the name of this function is the name of your class
- add properties to the object that's being constructed using the keyword this, followed by a period and the name of the property
- add methods to the object in the same way we'd add properties, using JavaScript's special function constructor syntax

Here's the code that creates a simple class called `HelloWorld`:

```
function HelloWorld() {

  this.message = 'Hello, world!';

  this.sayMessage = function() {

    window.alert(this.message);

  };

}
```

JavaScript's framework for object oriented programming is very lightweight, but functions surprisingly well once you get the hang of it.

More advanced object oriented features, such as inheritance and polymorphism, aren't available in JavaScript, but these features are rarely needed on the client side in an AJAX application. The complex business logic for which these features are useful should always be on the web server, and accessed using the `XMLHttpRequest` class.

In this example, we create a class called `HelloWorld` with one property (`message`) and one method (`sayMessage`). To use this class, we simply call the constructor function, as shown below:

```
var hw = new HelloWorld();

hw.sayMessage();

hw.message = 'Goodbye';

hw.sayMessage();
```

Here, we create an instance of `HelloWorld` (called `hw`), then use this object to display two messages. The first time we call `sayMessage`, the default "Hello, world!" message is displayed. Then, after changing our object's `message` property to "Goodbye," we call `sayMessage` and "Goodbye" is displayed.

Don't worry if this doesn't make too much sense at the moment. As we progress through the building of our `Ajax` class, it will become clearer.

Here are the beginnings of our `Ajax` class's constructor function:

```
Example 2.1. ajax.js (excerpt)



function Ajax() {

  this.req = null;
```

```
  this.url = null;

  this.method = 'GET';

  this.async = true;

  this.status = null;

  this.statusText = '';

  this.postData = null;

  this.readyState = null;

  this.responseText = null;

  this.responseXML = null;

  this.handleResp = null;

  this.responseFormat = 'text', // 'text', 'xml',
or 'object'

  this.mimeType = null;

}
```

This code just defines the properties we'll need in our `Ajax` class in order to work with `XMLHttpRequest` objects. Now, let's add some methods to our object. We need some functions that will set up an `XMLHttpRequest` object and tell it how to make requests for us.

Creating an `XMLHttpRequest` Object

First, we'll add an `init` method, which will create an `XMLHttpRequest` object for us.
Unfortunately, `XMLHttpRequest` is implemented slightly differently in Firefox (in this book, whenever I explain how something works in

Firefox, I'm referring to all Mozilla-based browsers, including Firefox, Mozilla, Camino, and SeaMonkey), Safari, and Opera than it was in Internet Explorer's original implementation (interestingly, Internet Explorer version 7 now supports the same interface as Firefox, which promises to simplify AJAX development in the future), so you'll have to try instantiating the object in a number of different ways if you're not targeting a specific browser. Firefox and Safari create `XMLHttpRequest` objects using a class called `XMLHttpRequest`, while Internet Explorer versions 6 and earlier use a special class called `ActiveXObject` that's built into Microsoft's scripting engine. Although these classes have different constructors, they behave in the same way.

Cross-browser Code

Fortunately, most modern browsers (Internet Explorer 6, Firefox 1.0, Safari 1.2, and Opera 8, or later versions of any of these browsers) adhere to web standards fairly well overall, so you won't have to do lots of browser-specific branching in your AJAX code.

This usually makes a browser-based AJAX application faster to develop and deploy cross-platform than a desktop application. As the power and capabilities available to AJAX applications increase, desktop applications offer fewer advantages from a user-interface perspective.

The `init` method looks like this:

Example 2.2. ajax.js (excerpt)

```
this.init = function() {
  if (!this.req) {
```

```
    try {
      // Try to create object for Firefox, Safari,
IE7, etc.
      this.req = new XMLHttpRequest();
    }
    catch (e) {
      try {
        // Try to create object for later versions
of IE.
        this.req = new
ActiveXObject('MSXML2.XMLHTTP');
      }
      catch (e) {
        try {
          // Try to create object for early
versions of IE.
          this.req = new
ActiveXObject('Microsoft.XMLHTTP');
        }
        catch (e) {
          // Could not create an XMLHttpRequest
object.
```

```
            return false;

          }

        }

      }

    }

    return this.req;

};
```

The `init` method goes through each possible way of creating an `XMLHttpRequest` object until it creates one successfully. This object is then returned to the calling function.

Degrading Gracefully

Maintaining compatibility with older browsers (by "older" I mean anything older than the "modern browsers" I mentioned in the previous note) requires a lot of extra code work, so it's vital to define which browsers your application should support.

If you know your application will receive significant traffic via older browsers that don't support the `XMLHtmlRequest` class (e.g., Internet Explorer 4 and earlier, Netscape 4 and earlier), you will need either to leave it out completely, or write your code so that it degrades gracefully. That means that instead of allowing your functionality simply to disappear in less-capable browsers, you code to ensure that users of those browsers receive something that's functionally equivalent, though perhaps in a less interactive or easy-to-use format.

It's also possible that your web site will attract users who browse with JavaScript disabled. If you want to cater to these users, you should provide an alternative, old-school interface by default, which you can then modify on-the-fly — using JavaScript — for modern browsers.

Sending a Request

We now have a method that creates an `XMLHttpRequest`. So let's write a function that uses it to make a request. We start the doReq method like this:

```
Example 2.3. ajax.js (excerpt)



this.doReq = function() {

  if (!this.init()) {

    alert('Could not create XMLHttpRequest
object.');

    return;

  }

};
```

This first part of `doReq` calls `init` to create an instance of the `XMLHttpRequest` class, and displays a quick alert if it's not successful.

Setting Up the Request

Next, our code calls the `open` method on `this.req` — our new instance of the `XMLHttpRequest` class — to begin setting up the HTTP request:

```
Example 2.4. ajax.js (excerpt)



this.doReq = function() {
```

```
  if (!this.init()) {

    alert('Could not create XMLHttpRequest
object.');

    return;

  }

  this.req.open(this.method, this.url,
this.async);

};
```

The `open` method takes three parameters:

1. Method – This parameter identifies the type of HTTP request method we'll use. The most commonly used methods are GET and POST.

Methods are Case-sensitive

According to the HTTP specification (RFC 2616), the names of these request methods are case-sensitive. And since the methods described in the spec are defined as being all uppercase, you should always make sure you type the method in all uppercase letters.

2. URL – This parameter identifies the page being requested (or posted to if the method is POST).

Crossing Domains

Normal browser security settings will not allow you to send HTTP requests to another domain. For example, a page served from ajax.net would not be able to send a request to remotescripting.com unless the user had allowed such requests.

3. Asynchronous Flag – If this parameter is set to `true`, your JavaScript will continue to execute normally while waiting for a

response to the request. As the state of the request changes, events are fired so that you can deal with the changing state of the request.

If you set the parameter to `false`, JavaScript execution will stop until the response comes back from the server. This approach has the advantage of being a little simpler than using a callback function, as you can start dealing with the response straight after you send the request in your code, but the big disadvantage is that your code pauses while the request is sent and processed on the server, and the response is received. As the ability to communicate with the server asynchronously is the whole point of an AJAX application, this should be set to `true`.

In our `Ajax` class, the method and async properties are initialized to reasonable defaults (GET and true), but you'll always have to set the target URL, of course.

Setting Up the `onreadystatechange` Event Handler

As the HTTP request is processed on the server, its progress is indicated by changes to the readyState property. This property is an integer that represents one of the following states, listed in order from the start of the request to its finish:

- `0`: uninitialized – `open` has not been called yet.
- `1`: loading – `send` has not been called yet.
- `2`: loaded – `send` has been called, but the response is not yet available.
- `3`: interactive – The response is being downloaded, and the responseText property holds partial data.
- `4`: completed – The response has been loaded and the request is completed.

An `XMLHttpRequest` object tells you about each change in state by firing a `readystatechange` event. In the handler for this event, check the `readyState` of the request, and when the request

completes (i.e., when the `readyState` changes to `4`), you can handle the server's response.

A basic outline for our `Ajax` code would look like this:

Example 2.5. ajax.js (excerpt)

```
this.doReq = function() {

  if (!this.init()) {

    alert('Could not create XMLHttpRequest
object.');

    return;

  }

  this.req.open(this.method, this.url,
this.async);

  var self = this; // Fix loss-of-scope in inner
function

  this.req.onreadystatechange = function() {

    if (self.req.readyState == 4) {

      // Do stuff to handle response

    }

  };

};
```

We'll discuss how to "do stuff to handle response" in just a bit. For now, just keep in mind that you need to set up this event handler before the request is sent.

Sending the Request

Use the `send` method of the `XMLHttpRequest` class to start the HTTP request, like so:

Example 2.6. ajax.js (excerpt)

```
this.doReq = function() {

  if (!this.init()) {

    alert('Could not create XMLHttpRequest
object.');

    return;

  }

  this.req.open(this.method, this.url,
this.async);

  var self = this; // Fix loss-of-scope in inner
function

  this.req.onreadystatechange = function() {

    if (self.req.readyState == 4) {

      // Do stuff to handle response

    }
```

```
  };

  this.req.send(this.postData);

};
```

The send method takes one parameter, which is used for `POST` data. When the request is a simple `GET` that doesn't pass any data to the server, like our current request, we set this parameter to null.

Loss of Scope and `this`

You may have noticed that `onreadystatechange` includes a weird-looking variable assignment:

```
Example 2.7. ajax.js (excerpt)



var self = this; // Fix loss-of-scope in inner
function
```

This new variable, `self`, is the solution to a problem called "loss of scope" that's often experienced by JavaScript developers using asynchronous event handlers. Asynchronous event handlers are commonly used in conjunction with `XMLHttpRequest`, and with functions like `setTimeout` or `setInterval`.

The `this` keyword is used as shorthand in object-oriented JavaScript code to refer to "the current object." Here's a quick example — a class called `ScopeTest`:

```
function ScopeTest() {

  this.message = 'Greetings from ScopeTest!';

  this.doTest = function() {
```

```
    alert(this.message);

  };

}

var test = new ScopeTest();

test.doTest();
```

This code will create an instance of the `ScopeTest` class, then call that object's `doTest` method, which will display the message "Greetings from ScopeTest!" Simple, right?

Now, let's add some simple `XMLHttpRequest` code to our `ScopeTest` class. We'll send a simple `GET` request for your web server's home page, and, when a response is received, we'll display the content of both `this.message` and `self.message`.

```
function ScopeTest() {

  this.message = 'Greetings from ScopeTest!';

  this.doTest = function() {

    // This will only work in Firefox, Opera and
Safari.

    this.req = new XMLHttpRequest();

    this.req.open('GET', '/index.html', true);

    var self = this;

    this.req.onreadystatechange = function() {

      if (self.req.readyState == 4) {
```

```
        var result = 'self.message is ' +
self.message;

        result += 'n';

        result += 'this.message is ' +
this.message;

        alert(result);

      }

    }

    this.req.send(null);

  };

}

var test = new ScopeTest();

test.doTest();
```

So, what message is displayed? The answer is revealed in Figure 2.1.

We can see that `self.message` is the greeting message that we're expecting, but what's happened to `this.message`?

Using the keyword `this` is a convenient way to refer to "the object that's executing this code." But this has one small problem — its meaning changes when it's called from outside the object. This is the result of something called execution context. All of the code inside the object runs in the same execution context, but code that's run from other objects — such as event handlers — runs in the calling object's execution context. What this means is that, when you're writing object-oriented JavaScript, you won't be able to use the `this` keyword to refer to the object in code for event handlers

(like `onreadystatechange` above). This problem is called loss of scope.

If this concept isn't 100% clear to you yet, don't worry too much about it. We'll see an actual demonstration of this problem in the next chapter. In the meantime, just kind of keep in mind that if you see the variable self in code examples, it's been included to deal with a loss-of-scope problem.

Figure 2.1. Message displayed by ScopeTest class

Processing the Response

Now we're ready to write some code to handle the server's response to our HTTP request. Remember the "do stuff to handle response" comment that we left in the `onreadystatechange` event handler? We'll, it's time we wrote some code to do that stuff! The function needs to do three things:

1. Figure out if the response is an error or not.
2. Prepare the response in the desired format.
3. Pass the response to the desired handler function.

Include the code below in the inner function of our `Ajax` class:

```
Example 2.8. ajax.js (excerpt)



this.req.onreadystatechange = function() {
```

```
  var resp = null;
  if (self.req.readyState == 4) {
    switch (self.responseFormat) {
      case 'text':
        resp = self.req.responseText;
        break;
      case 'xml':
        resp = self.req.responseXML;
        break;
      case 'object':
        resp = req;
        break;
    }
    if (self.req.status >= 200 && self.req.status
<= 299) {
      self.handleResp(resp);
    }
    else {
      self.handleErr(resp);
```

```
    }

  }

};
```

When the response completes, a code indicating whether or not the request succeeded is returned in the status property of our `XMLHttpRequest` object. The status property contains the HTTP status code of the completed request. This could be code 404 if the requested page was missing, 500 if an error occurred in the server-side script, 200 if the request was successful, and so on. A full list of these codes is provided in the HTTP Specification (RFC 2616).

No Good with Numbers?

If you have trouble remembering the codes, don't worry: you can use the statusText property, which contains a short message that tells you a bit more detail about the error (e.g., "Not Found," "Internal Server Error," "OK").

Our `Ajax` class will be able to provide the response from the server in three different formats: as a normal JavaScript string, as an XML document object accessible via the W3C XML DOM, and as the actual `XMLHttpRequest` object that was used to make the request. These are controlled by the `Ajax` class's `responseFormat` property, which can be set to `text`, `xml` or `object`.

The content of the response can be accessed via two properties of our `XMLHttpRequest` object:

- `responseText` – This property contains the response from the server as a normal string. In the case of an error, it will contain the web server's error page HTML. As long as a response is returned (that is, `readyState` becomes 4), this property will contain data, though it may not be what you expect.
- `responseXML` – This property contains an XML document object. If the response is not XML, this property will be empty.

Our `Ajax` class initializes its `responseFormat` property to text, so by default, your response handler will be passed the content from the server as a JavaScript string. If you're working with XML content, you can change the `responseFormat` property to `xml`, which will pull out the XML document object instead.

There's one more option you can use if you want to get really fancy: you can return the actual `XMLHttpRequest` object itself to your handler function. This gives you direct access to things like the status and `statusText` properties, and might be useful in cases in which you want to treat particular classes of errors differently — for example, completing extra logging in the case of 404 errors.

Setting the Correct `Content-Type`

Implementations of `XMLHttpRequest` in all major browsers require the HTTP response's `Content-Type` to be set properly in order for the response to be handled as XML. Well-formed XML, returned with a content type of `text/xml` (or `application/xml`, or even `application/xhtml+xml`), will properly populate the `responseXML` property of an `XMLHttpRequest` object; non-XML content types will result in values of `null` or `undefined` for that property.

However, Firefox, Safari, and Internet Explorer 7 provide a way around `XMLHttpRequest`'s pickiness over XML documents: the `overrideMimeType` method of the `XMLHttpRequest` class. Our simple `Ajax` class hooks into this with the `setMimeType` method:

```
Example 2.9. ajax.js (excerpt)



this.setMimeType = function(mimeType) {
```

```
  this.mimeType = mimeType;

};
```

This method sets the `mimeType` property.

Then, in our `doReq` method, we simply call `overrideMimeType` inside a `try ... catch` block, like so:

```
Example 2.10. ajax.js (excerpt)



req.open(this.method, this.url, this.async);

if (this.mimeType) {

  try {

    req.overrideMimeType(this.mimeType);

  }

  catch (e) {

    // couldn't override MIME type  --  IE6 or
Opera?

  }

}

var self = this; // Fix loss-of-scope in inner
function
```

Being able to override `Content-Type` headers from uncooperative servers can be very important in environments in which you don't have control over both the front and back ends of your web application. This is especially true since many of today's apps access services and

content from a lot of disparate domains or sources. However, as this technique won't work in Internet Explorer 6 or Opera 8, you may not find it suitable for use in your applications today.

Response Handler

According to the HTTP 1.1 specification, any response that has a code between 200 and 299 inclusive is a successful response.

The `onreadystatechange` event handler we've defined looks at the status property to get the status of the response. If the code is within the correct range for a successful response,
the `onreadystatechange` event handler passes the response to the response handler method (which is set by the `handleResp` property).

The response handler will need to know what the response was, of course, so we'll pass it the response as a parameter. We'll see this process in action later, when we talk about the doGet method.

Since the handler method is user-defined, the code also does a cursory check to make sure the method has been set properly before it tries to execute the method.

Error Handler

If the status property indicates that there's an error with the request (i.e., it's outside the 200 to 299 code range), the server's response is passed to the error handler in the handleErr property. Our Ajax class already defines a reasonable default for the error handler, so we don't have to make sure it's defined before we call it.

The `handleErr` property points to a function that looks like this:

```
Example 2.11. ajax.js (excerpt)
```

```
this.handleErr = function() {

  var errorWin;

  try {

    errorWin = window.open('', 'errorWin');

    errorWin.document.body.innerHTML =
this.responseText;

  }

  catch (e) {

    alert('An error occurred, but the error message
cannot be '

      + 'displayed. This is probably because of
your browser's '

      + 'pop-up blocker.n'

      + 'Please allow pop-ups from this web site if
you want to '

      + 'see the full error messages.n'

      + 'n'

      + 'Status Code: ' + this.req.status + 'n'

      + 'Status Description: ' +
this.req.statusText);

  }

};
```

This method checks to make sure that pop-ups are not blocked, then tries to display the full text of the server's error page content in a new browser window. This code uses a `try ... catch` block, so if users have blocked pop-ups, we can show them a cut-down version of the error message and tell them how to access a more detailed error message.

This is a decent default for starters, although you may want to show less information to the end-user — it all depends on your level of paranoia. If you want to use your own custom error handler, you can use `setHandlerErr` like so:

Example 2.12. ajax.js (excerpt)

```
this.setHandlerErr = function(funcRef) {
  this.handleErr = funcRef;
}
```

Or, the One True Handler

It's possible that you might want to use a single function to handle both successful responses and errors. `setHandlerBoth`, a convenience method in our `Ajax` class, sets this up easily for us:

Example 2.13. ajax.js (excerpt)

```
this.setHandlerBoth = function(funcRef) {
  this.handleResp = funcRef;
  this.handleErr = funcRef;
```

```
};
```
Any function that's passed as a parameter to `setHandlerBoth` will handle both successful responses and errors.

This setup might be useful to a user who sets your class's `responseFormat` property to object, which would cause the `XMLHttpRequest` object that's used to make the request — rather than just the value of the `responseText` or `responseXML` properties — to be passed to the response handler.

Aborting the Request

Sometimes, as you'll know from your own experience, a web page will take a very long time to load. Your web browser has a Stop button, but what about your `Ajax` class? This is where the `abort` method comes into play:

Example 2.14. ajax.js (excerpt)

```
this.abort = function() {
  if (this.req) {
    this.req.onreadystatechange = function() { };
    this.req.abort();
    this.req = null;
  }
};
```

This method changes the `onreadystate` event handler to an empty function, calls the `abort` method on your instance of

the `XMLHttpRequest` class, then destroys the instance you've created. That way, any properties that have been set exclusively for the request that's being aborted are reset. Next time a request is made, the `init` method will be called and those properties will be reinitialized.

So, why do we need to change the `onreadystate` event handler? Many implementations of `XMLHttpRequest` will fire the onreadystate event once abort is called, to indicate that the request's state has been changed. What's worse is that those events come complete with a `readyState` of 4, which indicates that everything completed as expected (which is partly true, if you think about it: as soon as we call abort, everything should come to a stop and our instance of `XMLHttpRequest` should be ready to send another request, should we so desire). Obviously, we don't want our response handler to be invoked when we abort a request, so we remove the existing handler just before we call `abort`.

Wrapping it Up

Given the code we have so far, the Ajax class needs just two things in order to make a request:

- a target URL
- a handler function for the response

Let's provide a method called `doGet` to set both of these properties, and kick off the request:

Example 2.15. ajax.js (excerpt)

```
this.doGet = function(url, hand, format) {

  this.url = url;
```

```
  this.handleResp = hand;

  this.responseFormat = format || 'text';

  this.doReq();

};
```

You'll notice that, along with the two expected parameters, `url` and `hand`, the function has a third parameter: `format`. This is an optional parameter that allows us to change the format of the server response that's passed to the handler function.

If we don't pass in a value for format, the `responseFormat` property of the `Ajax` class will default to a value of text, which means your handler will be passed the value of the `responseText` property. You could, instead, pass `xml` or `object` as the format, which would change the parameter that's being passed to the response handler to an XML DOM or `XMLHttpRequest` object.

Example: a Simple Test Page

It's finally time to put everything we've learned together! Let's create an instance of this `Ajax` class, and use it to send a request and handle a response.

Now that our class's code is in a file called `ajax.js`, any web pages in which we want to use our `Ajax` class will need to include the Ajax code with a `<script type="text/javascript" src="ajax.js">` tag. Once our page has access to the Ajax code, we can create an `Ajax` object.

```
Example 2.16. ajaxtest.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>A Simple AJAX Test</title>

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

    <script type="text/javascript">

      var ajax = new Ajax();

    </script>

  </head>

  <body>

  </body>

</html>
```

This script gives us a shiny, new instance of the `Ajax` class. Now, let's make it do something useful.

To make the most basic request with our `Ajax` class, we could do something like this:

```
Example 2.17. ajaxtest.html (excerpt)



<script type="text/javascript">

  var hand = function(str) {

    alert(str);

  }

  var ajax = new Ajax();

  ajax.doGet('/fakeserver.php', hand);

</script>
```

This creates an instance of our `Ajax` class that will make a simple `GET` request to a page called `fakeserver.php`, and pass the result back as text to the hand function. If `fakeserver.php` returned an XML document that you wanted to use, you could do so like this:

```
Example 2.18. ajaxtest.html (excerpt)



<script type="text/javascript">

  var hand = function(str) {

    // Do XML stuff here

  }
```

```
  var ajax = new Ajax();

  ajax.doGet('/fakeserver.php', hand);

</script>
```

You would want to make absolutely sure in this case that somepage.php was really serving valid XML and that its `Content-Type` HTTP response header was set to `text/xml` (or something else that was appropriate).

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.

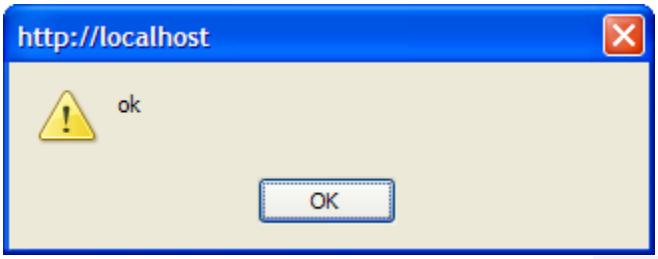

Figure 2.2. Confirmation that your `Ajax` class is working as expected

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.

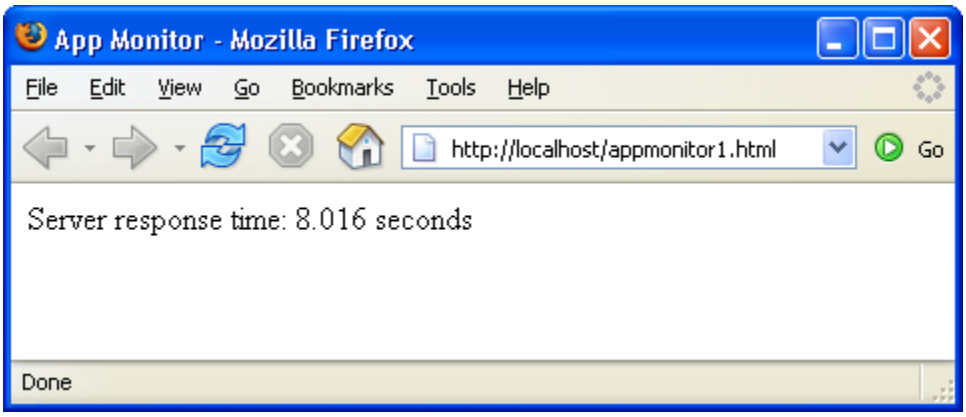

Figure 2.3. Running the simple monitoring application

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>
```

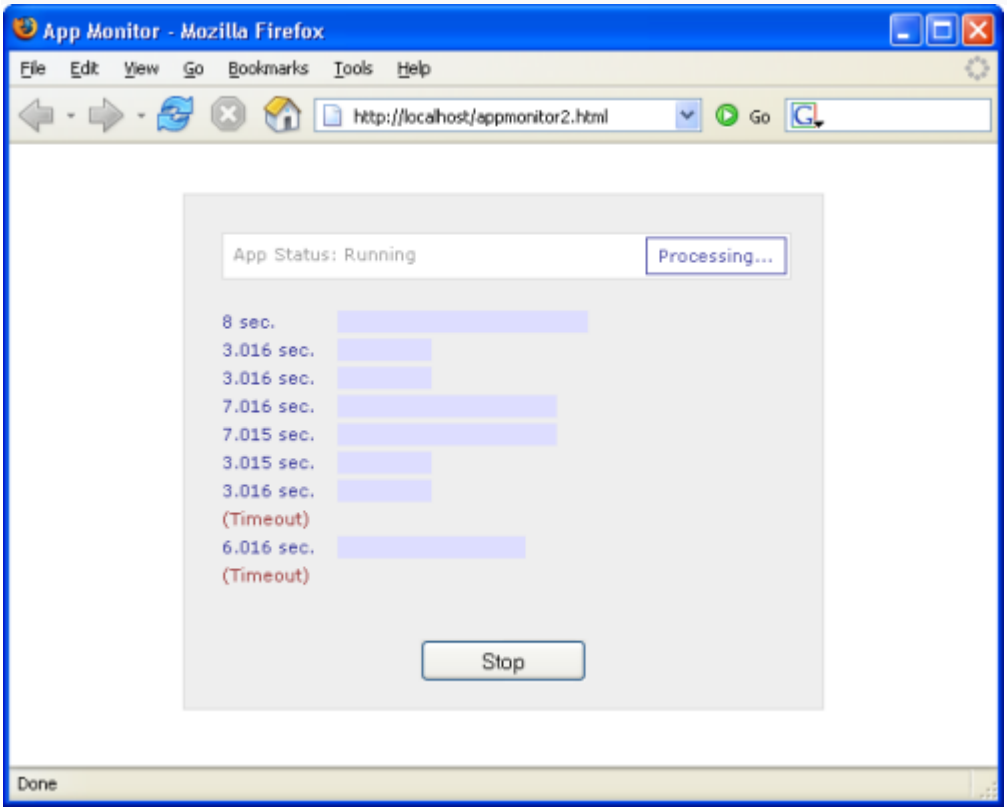

Figure 3.1. The running application

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.

Singletons with JavaScript

A “singleton” is called that because only a “single” instance of that class exists at any time. Making a class into a singleton is surprisingly easy:

```
var ScopeTest = new function() {
```

```
  this.message = "Greetings from ScopeTest!";

  this.doTest = function() {

    var onTimeout = function() {

      alert(this.message);

    };

    setTimeout(onTimeout, 1000);

  };

}
```

Using the keyword new before function creates a “one-shot” constructor. It creates a single instance of `ScopeTest`, and it’s done: you can’t use it to create any more `ScopeTest` objects.

To call the `doTest` method of this singleton object, you must use the actual name of the class (since there’s only the one instance of it):

```
ScopeTest.doTest();
```

That’s all well and good, but we haven’t solved our loss of scope problem. If you were to try the code now, you’d get the same “undefined” message you saw before, because this doesn’t refer to an instance of `ScopeTest`. However, using a singleton gives us an easy way to fix the problem. All we have to do is use the actual name of the object — instead of the keyword `this` — inside `onTimeout`:

```
var ScopeTest = new function() {

  this.message = "Greetings from ScopeTest!";

  this.doTest = function() {
```

```
    var onTimeout = function() {

      alert(ScopeTest.message);

    };

    setTimeout(onTimeout, 1000);

  };

}
```

There's only one instance of `ScopeTest`, and we're using its actual name instead of `this`, so there's no confusion about which instance of `ScopeTest` is being referred to here.

When you execute this code, you'll see the expected value of "Greetings from ScopeTest!" in the JavaScript alert box.

Now, I get tired of using the actual object name throughout my object code, and I like to use a shortcut keyword like this wherever I possibly can. So, usually I create a variable `self` that I can use in place of `this`, and point it to the object name at the top of each method, like so:

```
var onTimeout = function() {

  var self = ScopeTest;

  alert(self.message);

};
```

This looks a bit silly in a method that's as short as that, but in longer chunks of code it's nice to have a shorthand solution similar to this that you can use to refer to your object. I use `self`, but you could use `me`, or `heyYou`, or `darthVader` if you wanted to.

Creating the Monitor Object

Now that we have a plan for code organization that will fix the loss-of-scope problem from `setTimeout`, it's time to create our base `Monitor` class:

```
Example 3.2. appmonitor2.js (excerpt)



var Monitor = new function(){

  this.targetURL = null;

  this.pollInterval = null;

  this.maxPollEntries = null;

  this.timeoutThreshold = null;

  this.ajax = new Ajax();

  this.start = 0;

  this.pollArray = [];

  this.pollHand = null;

  this.timeoutHand = null;

  this.reqStatus = Status;

}
```

The first four properties, `targetURL`, `pollInterval`, `maxPollEntries`, and `timeoutThreshold`, will be initialized as part of the class's initialization. They will take on the values defined in the application's configuration, which we'll look at in the next section.

Here's a brief rundown on the other properties:

- `ajax` – The instance of our Ajax class that makes the HTTP requests to the server we're monitoring.
- `start` – Used to record the time at which the last request was sent.
- `pollArray` – An array that holds the server response times; the constant `MAX_POLL_ENTRIES` determines the number of items held in this array.
- `pollHand`, `timeoutHand` – Interval IDs returned by the `setTimeout` calls for two different processes — the main polling process, and the timeout watcher, which controls a user-defined timeout period for each request.
- `reqStatus` – Used for the status animation that notifies the user when a request is in progress. The code that achieved this is fairly complicated, so we'll be writing another singleton class to take care of it. The `reqStatus` property points to the single instance of that class.

Configuring and Initializing our Application

A webmaster looking at this application may think that it was quite cool, but one of the first things he or she would want is an easy way to configure the app's polling interval, or the time that elapses between requests the app makes to the site it's monitoring. It's easy to configure the polling interval using a global constant.

To make it very simple for any user of this script to set the polling interval, we'll put this section of the code in a script element within the head of `appmonitor2.html`:

```
Example 3.3. appmonitor2.html (excerpt)



<script type="text/javascript">
```

```
  // URL to monitor

  var TARGET_URL = '/fakeserver.php';

  // Seconds between requests

  var POLL_INTERVAL = 5;

  // How many entries bars to show in the bar
graph

  var MAX_POLL_ENTRIES = 10;

  // Seconds to wait for server response

  var TIMEOUT_THRESHOLD = 10;

</script>
```

You'll notice that these variable names are written in all-caps. This is an indication that they should act like constants — values that are set early in the code, and do not change as the code executes. Constants are a feature of many programming languages but, unfortunately, JavaScript is not one of them. (Newer versions of JavaScript allow you to set real constants with the constkeyword, but this facility isn't widely supported (even by many modern browsers).) Note that these constants relate directly to the first four properties of our class: `targetURL`, `pollInterval`, `maxPollEntries`, and `timeoutThreshold`. These properties will be initialized in our class's `init` method:

```
Example 3.4. appmonitor2.js (excerpt)



this.init = function() {
```

```
  var self = Monitor;

  self.targetURL = TARGET_URL;

  self.pollInterval = POLL_INTERVAL;

  self.maxPollEntries = MAX_POLL_ENTRIES;

  self.timeoutThreshold = TIMEOUT_THRESHOLD;

  self.toggleAppStatus(true);

  self.reqStatus.init();

};
```

As well as initializing some of the properties of our class, the `init` method also calls two methods: `toggleAppStatus`, which is responsible for starting and stopping the polling, and the `init` method of the `reqStatus` object. reqStatus is the instance of the `Status` singleton class that we discussed a moment ago.

This `init` method is tied to the `window.onload` event for the page, like so:

```
Example 3.5. appmonitor2.js (excerpt)



window.onload = Monitor.init;
```

Setting Up the UI

The first version of this application started when the page loaded, and ran until the browser window was closed. In this version, we want to give users a button that they can use to toggle the polling process on or off. The `toggleAppStatus` method handles this for us:

```
Example 3.6. appmonitor2.js (excerpt)
```

```
this.toggleAppStatus = function(stopped) {

  var self = Monitor;

  self.toggleButton(stopped);

  self.toggleStatusMessage(stopped);

};
```

Okay, so `toggleAppStatus` doesn't really do the work, but it calls the methods that do: `toggleButton`, which changes Start buttons into Stop buttons and vice versa, and `toggleStatusMessage`, which updates the application's status message. Let's take a closer look at each of these methods.

The `toggleButton` Method

This method toggles the main application between its "Stop" and "Start" states. It uses DOM-manipulation methods to create the appropriate button dynamically, assigning it the correct text and an onclick event handler:

```
Example 3.7. appmonitor2.js (excerpt)



this.toggleButton = function(stopped) {

  var self = Monitor;

  var buttonDiv =
document.getElementById('buttonArea');

  var but = document.createElement('input');
```

```
  but.type = 'button';
  but.className = 'inputButton';
  if (stopped) {
    but.value = 'Start';
    but.onclick = self.pollServerStart;
  }
  else {
    but.value = 'Stop';
    but.onclick = self.pollServerStop;
  }
  if (buttonDiv.firstChild) {
    buttonDiv.removeChild(buttonDiv.firstChild);
  }
  buttonDiv.appendChild(but);
  buttonDiv = null;
};
```

The only parameter to this method, `stopped`, can either be `true`, indicating that the polling has been stopped, or `false`, indicating that polling has started.

As you can see in the code for this method, the button is created, and is set to display Start if the application is stopped, or Stop if the

application is currently polling the server. It also assigns either `pollServerStart` or `pollServerStop` as the button's onclick event handler. These event handlers will start or stop the polling process respectively.

When this method is called from `init` (via `toggleAppStatus`), `stopped` is set to `true` so the button will display Start when the application is started.

As this code calls for a `div` with the ID `buttonArea`, let's add that to our markup now:

Example 3.8. appmonitor2.html (excerpt)

```
<body>

  <div id="buttonArea"></div>

</body>
```

The `toggleStatusMessage` Method

Showing a button with the word "Start" or "Stop" on it might be all that programmers or engineers need to figure out the application's status, but most normal people need a message that's a little clearer and more obvious in order to work out what's going on with an application.

This upgraded version of the application will display a status message at the top of the page to tell the user about the overall state of the application (stopped or running), and the status of the polling process. To display the application status, we'll place a nice, clear message in the application's status bar that states App Status: Stopped or App Status: Running.

In our markup, let's insert the status message above where the button appears. We'll include only the "App Status" part of the message in our

markup. The rest of the message will be inserted into a span with the ID `currentAppState`:

Example 3.9. appmonitor2.html (excerpt)

```
<body>
  <div id="statusMessage">App Status:
      <span id="currentAppState"></span>
  </div>
  <div id="buttonArea"></div>
</body>
```

The `toggleStatusMessage` method toggles between the words that can display inside the `currentAppState` span:

Example 3.10. appmonitor2.js (excerpt)

```
this.toggleStatusMessage = function(stopped) {
  var statSpan =
document.getElementById('currentAppState');
  var msg;
  if (stopped) {
    msg = 'Stopped';
  }
```

```
  else {

    msg = 'Running';

  }

  if (statSpan.firstChild) {

    statSpan.removeChild(statSpan.firstChild);

  }

  statSpan.appendChild(document.createTextNode(msg)
);

};
```

Once the UI is set up, the application is primed and ready to start polling and recording response times.

Checking your Work In Progress

Now that you've come this far, it would be nice to be able to see your work in action, right? Well, unfortunately, we've still got a lot of loose ends in our application — we've briefly mentioned a singleton class called Status but we haven't created it yet, and we still have event handlers left to write. But never fear! We can quickly get the application up and running with a few class and function stubs.

We'll start by creating that Status singleton class with one empty method.

```
Example 3.11. appmonitor2.js (excerpt)



var Status = new function() {
```

```
  this.init = function() {

    // don't mind me, I'm just a stub ...

  };

}
```

Since the `Status` class is used by the `Monitor` class, we must declare `Status` before `Monitor`.

Then, we'll add our button's `onclick` event handlers to the `Monitor` class. We'll have them display alert dialogs so that we know what would be going on if there was anything happening behind the scenes.

```
Example 3.12. appmonitor2.js (excerpt)



this.pollServerStart = function() {

  alert('This will start the application polling
the server.');

};

this.pollServerStop = function() {

  alert('This will stop the application polling the
server.');

};
```

With these two simple stubs in place, your application should now be ready for a test-drive.

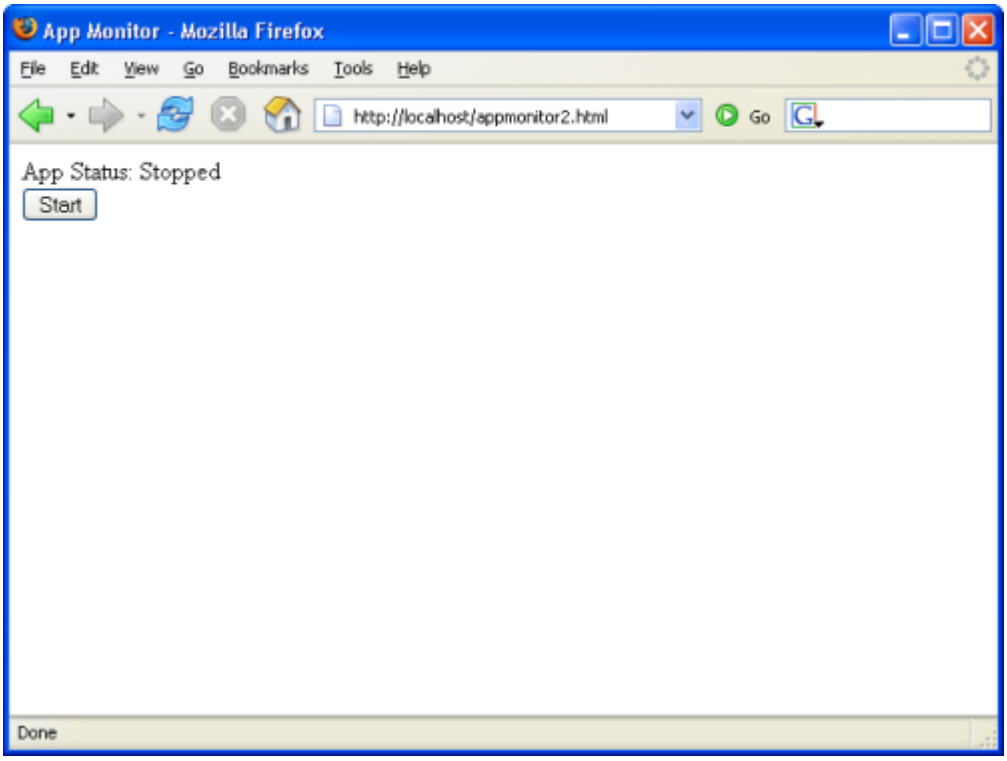

Figure 3.2. Humble beginnings

When you click the Start button in the display shown in Figure 3.2 you're presented with an alert box that promises greater things to come. Let's get started making good on those promises.

Polling the Server

The first step is to flesh out the Start button's `onclick` event handler, `pollServerStart`:

Example 3.13. appmonitor2.js (excerpt)

```
this.pollServerStart = function() {
  var self = Monitor;
  self.doPoll();
```

```
  self.toggleAppStatus(false);

};
```

This code immediately calls the `doPoll` method, which, like the app monitor we built in Chapter 2, Basic XMLHttpRequest, will be responsible for making an HTTP request to poll the server. Once the request has been sent, the code calls `toggleAppStatus`, passing it false to indicate that polling is underway.

Where's the Poll Interval?

You might wonder why, after all this talk about setting a poll interval, our code jumps right in with a request to the server; where's the time delay? The answer is that we don't want a time delay on the very first request. If users click the button and there's a ten-second delay before anything happens, they'll think the app is broken. We want delays between all the subsequent requests that occur once the application is running, but when the user first clicks that button, we want the polling to start right away.

The only difference between `doPoll` in this version of our app monitor and the one we saw in the last chapter is the use of `self` to prefix the properties of the class, and the call to `setTimeout`. Take a look:

Example 3.14. appmonitor2.js (excerpt)

```
this.doPoll = function() {

  var self = Monitor;

  var url = self.targetURL;

  var start = new Date();
```

```
  self.reqStatus.startProc();

  self.start = start.getTime();

  self.ajax.doGet(self.targetURL + '?start=' +
self.start,

      self.showPoll);

  self.timeoutHand =
setTimeout(self.handleTimeout,

      self.timeoutThreshold * 1000);

};
```

Our call to `setTimeout` instructs the browser to call `handleTimeout` once the timeout threshold has passed. We're also keeping track of the interval ID that's returned, so we can cancel our call to `handleTimeout` when the response is received by `showPoll`.

Here's the code for the `showPoll` method, which handles the response from the server:

```
Example 3.15. appmonitor2.js (excerpt)



this.showPoll = function(str) {

  var self = Monitor;

  var diff = 0;

  var end = new Date();
```

```
  clearTimeout(self.timeoutHand);

  self.reqStatus.stopProc(true);

  if (str == 'ok') {

    end = end.getTime();

    diff = (end - self.start) / 1000;

  }

  if (self.updatePollArray(diff)) {

    self.printResult();

  }

  self.doPollDelay();

};
```

The first thing this method does is cancel the delayed call to `handleTimeout` that was made at the end of `doPoll`. After this, we tell our instance of the `Status` class to stop its animation (we'll be looking at the details of this a little later).

After these calls, `showPoll` checks to make sure that the response is ok, then calculates how long that response took to come back from the server. The error handling capabilities of the `Ajax` class should handle errors from the server, so our script shouldn't return anything other than `ok` … though it never hurts to make sure!

Once it has calculated the response time, `showPoll` records that response time with `updatePollArray`, then displays the result with `printResult`. We'll look at both of these methods in the next section.

Finally, we schedule another poll in `doPollDelay` — a very simple method that schedules another call to `doPoll` once the poll interval has passed:

Example 3.16. appmonitor2.js (excerpt)

```
this.doPollDelay = function() {
  var self = Monitor;
  self.pollHand = setTimeout(self.doPoll,
      self.pollInterval * 1000);
};
```

To check our progress up to this point, we'll need to add a few more stub methods. First, let's add `startProc` and `stopProc` to the `Status` class:

Example 3.17. appmonitor2.js (excerpt)

```
var Status = new function() {
  this.init = function() {
    // don't mind me, I'm just a stub ...
  };
  this.startProc = function() {
    // another stub function
```

```
  };

  this.stopProc = function() {

    // another stub function

  };

}
```

Let's also add a few stub methods to our `Monitor` class:

```
Example 3.18. appmonitor2.js (excerpt)



this.handleTimeout = function() {

  alert("Timeout!");

};

this.updatePollArray = function(responseTime) {

  alert("Recording response time: " +
responseTime);

};

this.printResult = function() {

  // empty stub function

};
```

Now we're ready to test our progress. Open `appmonitor2.html` in your web browser, click Start, and wait for `fakeserver.php` to wake from its sleep and send ok back to your page.

You can expect one of two outcomes: either a response is received by your page, and you see a dialog similar to the one shown in Figure 3.3, or you see the timeout message shown in Figure 3.4.

Figure 3.3. A response received by your AJAX application

Don't worry if you receive the timeout message shown in Figure 3.4. Keep in mind that in our AJAX application, our timeout threshold is currently set to ten seconds, and that `fakeserver.php` is currently sleeping for a randomly selected number of seconds between three and 12. If the random number is ten or greater, the AJAX application will report a timeout.

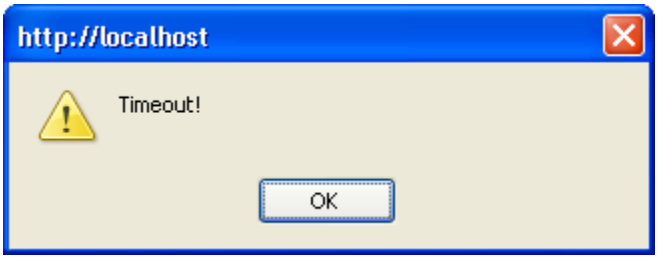

Figure 3.4. Your AJAX application giving up hope

At the moment, we haven't implemented a way to stop the polling, so you'll need to stop it either by reloading the page or closing your browser window.

Handling Timeouts

If you've run the code we've written so far, you've probably noticed that even when a timeout is reported, you see a message reporting the request's response time soon afterward. This occurs because handleTimeout is nothing but a simple stub at the moment. Let's look at building on that stub so we don't get this side-effect.

handleTimeout is basically a simplified version of showPoll: both methods are triggered by an asynchronous event (a call to setTimeout and an HTTP response received by an XMLHttpRequest object respectively), both methods need to record the response time (in a timeout's case, this will be 0), both methods need to update the user interface, and both methods need to trigger the next call to doPoll. Here's the code for handleTimeout:

Example 3.19. appmonitor2.js (excerpt)

```
this.handleTimeout = function() {
  var self = Monitor;
  if (self.stopPoll()) {
    self.reqStatus.stopProc(true);
    if (self.updatePollArray(0)) {
      self.printResult();
    }
    self.doPollDelay();
  }
};
```

Here, `handleTimeout` calls stopPoll to stop our application polling the server. It records that a timeout occurred, updates the user interface, and finally sets up another call to `doPoll` via `doPollDelay`. We moved the code that stops the

polling into a separate function because we'll need to revisit it later and beef it up. At present, the `stopPoll` method merely aborts the HTTP request via the `Ajax` class's `abort` method; however, there are a few scenarios that this function doesn't handle. We'll address these later, when we create the complete code to stop the polling process, but for the purposes of handling the timeout, `stopPoll` is fine.

Example 3.20. appmonitor2.js (excerpt)

```
this.stopPoll = function() {
  var self = Monitor;
  if (self.ajax) {
    self.ajax.abort();
  }
  return true;
};
```

Now, when we reload our application, the timeouts perform exactly as we expect them to.

The Response Times Bar Graph

Now, to the meat of the new version of our monitoring app! We want the application to show a list of past response times, not just a single entry of the most recent one, and we want to show that list in a way that's quickly and easily readable. A running bar graph display is the perfect tool for the job.

The Running List in `pollArray`

All the response times will go into an array that's stored in the `pollArray` property of the `Monitor` class. We keep this array updated with the intuitively named `updatePollArray` method. It's a very simple method that looks like this:

```
Example 3.21. appmonitor2.js (excerpt)



this.updatePollArray = function(pollResult) {

  var self = Monitor;

  self.pollArray.unshift(pollResult);

  if (self.pollArray.length > self.maxPollEntries)
{

    self.pollArray.pop();

  }

  return true;

};
```

The code is very straightforward, although some of the functions we've used in it have slightly confusing names.

The `unshift` method of an `Array` object puts a new item in the very first element of the array, and shifts the rest of the array's contents over by one position, as shown in Figure 3.5.

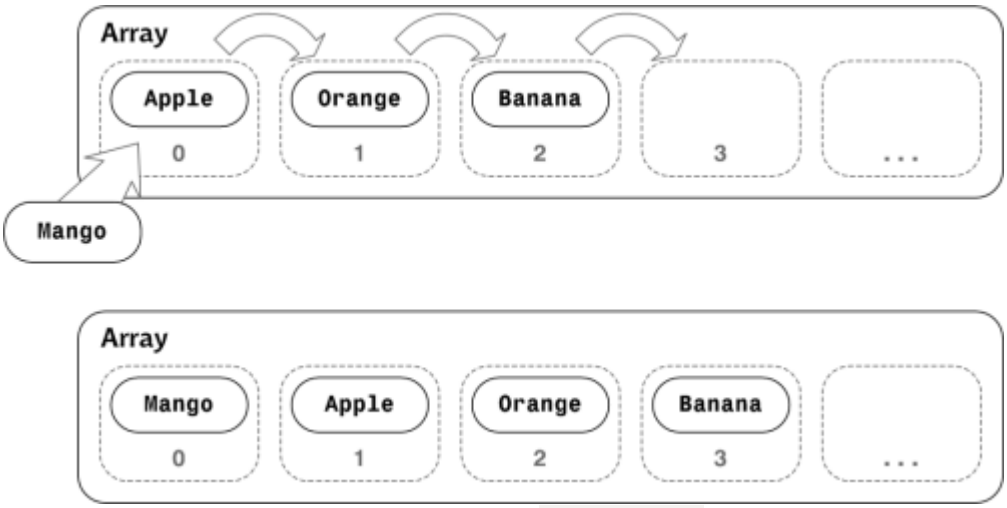

Figure 3.5. Inserting fruit using `unshift`

When the array exceeds the user-defined maximum length, `updatePollArray` truncates it by "popping" an item off the end. This is achieved by the `pop` method, which simply deletes the last item of an array. (Note that the method name `pop` may seem quite odd, but it makes more sense once you understand a data structure called a stack, which stores a number of items that can be accessed only in the reverse of the order in which they were added to the stack. We "push" an item onto a stack to add it, and "pop" an item from a stack to retrieve it. The `pop` method was originally designed for developers who were using arrays as stacks, but here we've repurposed it simply to delete the last item in an array.) The reason why we append items to the top and remove items from the bottom of the array is that, in our display, we want the most recent entries to appear at the top, and older entries to gradually move down to the bottom.

Displaying the Results

Once we've updated the results in `pollArray`, we can display them using the `printResult` method. This is actually the cool part: the user will experience first-hand the difference between our AJAX application and an older-style app that requires an entire page refresh to update content.

Rendering Page Partials

In AJAX jargon, the chunk of the page that holds the list of response times is called a page partial. This refers to an area of a web page that's updated separately from the rest of the page.

Updating a chunk of a web page in response to an asynchronous request to the server is called "rendering a page partial."

The `printResult` method iterates through `pollArray`, and uses DOM methods to draw the list of poll results inside a `div` with the ID `pollResults`. We'll start by adding that `div` to our markup:

```
Example 3.22. appmonitor2.html (excerpt)



<body>

  <div id="statusMessage">App Status:

    <span id="currentAppState"></span>

  </div>

  <div id="pollResults"></div>

  <div id="buttonArea"></div>

</body>
```

Now we're ready for the `printResult` method:

```
Example 3.23. appmonitor2.js (excerpt)



this.printResult = function() {
```

```
  var self = Monitor;

  var polls = self.pollArray;

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

  var entryDiv = null;

  var messageDiv = null;

  var barDiv = null;

  var clearAll = null;

  var msgStr = '';

  var txtNode = null;

  while (pollDiv.firstChild) {

    pollDiv.removeChild(pollDiv.firstChild);

  }

  for (var i = 0; i < polls.length; i++) {

    if (polls[i] == 0) {

      msgStr = '(Timeout)';

    }

    else {

      msgStr = polls[i] + ' sec.';
```

```
    }
    entryDiv = document.createElement('div');
    messageDiv = document.createElement('div');
    barDiv = document.createElement('div');
    clearAll = document.createElement('br');
    entryDiv.className = 'pollResult';
    messageDiv.className = 'time';
    barDiv.className = 'bar';
    clearAll.className = 'clearAll';
    if (polls[i] == 0) {
      messageDiv.style.color = '#933';
    }
    else {
      messageDiv.style.color = '#339';
    }
    barDiv.style.width = (parseInt(polls[i] * 20))
+ 'px';
    messageDiv.appendChild(document.createTextNode(
msgStr));
```

```
    barDiv.appendChild(document.createTextNode('u00
A0'));

    entryDiv.appendChild(messageDiv);

    entryDiv.appendChild(barDiv);

    entryDiv.appendChild(clearAll);

    pollDiv.appendChild(entryDiv);

  }

};
```

There's quite a bit here, so let's look at this method step by step.

```
Example 3.24. appmonitor2.js (excerpt)



while (pollDiv.firstChild) {

  pollDiv.removeChild(pollDiv.firstChild);

}
```

After initializing some variables, this method removes everything from `pollDiv`: the `while` loop uses `removeChild` repeatedly to delete all the child nodes from `pollDiv`.

Next comes a simple for loop that jumps through the updated array of results and displays them.

We generate a message for the result of each item in this array. As you can see below, timeouts (which are recorded as a 0) generate a message of `(Timeout)`.

```
Example 3.25. appmonitor2.js (excerpt)
```

```
if (polls[i] == 0) {

  msgStr = '(Timeout)';

}

else {

  msgStr = polls[i] + ' sec.';

}
```

Next, we use DOM methods to add the markup for each entry in the list dynamically. In effect, we construct the following HTML in JavaScript for each entry in the list:

```
<div class="pollResult">

    <div class="time" style="color: #339;">8.031
sec.</div>

    <div class="bar" style="width:
160px;"> </div>

    <br class="clearAll"/>

</div>
```

The width of the bar `div` changes to reflect the actual response time, and timeouts are shown in red, but otherwise all entries in this list are identical. Note that you have to put something in the `div` to cause its background color to display. Even if you give the `div` a fixed width, the background color will not show if the `div` is empty. This is annoying, but it's easy to fix: we can fill in the `div` with a non-breaking space character.

Let's take a look at the code we'll use to insert this markup:

Example 3.26. appmonitor2.js (excerpt)

```
entryDiv = document.createElement('div');
messageDiv = document.createElement('div');
barDiv = document.createElement('div');
clearAll = document.createElement('br');
entryDiv.className = 'pollResult';
messageDiv.className = 'time';
barDiv.className = 'bar';
clearAll.className = 'clearAll';
if (polls[i] == 0) {
  messageDiv.style.color = '#933';
}
else {
  messageDiv.style.color = '#339';
}
barDiv.style.width = (parseInt(polls[i] * 20)) +
'px';
messageDiv.appendChild(document.createTextNode(msgS
tr));
```

```
barDiv.appendChild(document.createTextNode('u00A0')
);

entryDiv.appendChild(messageDiv);

entryDiv.appendChild(barDiv);

entryDiv.appendChild(clearAll);

pollDiv.appendChild(entryDiv);
```

This code may seem complicated if you've never used DOM manipulation functions, but it's really quite simple. We use the well-named `createElement` method to create elements; then we assign values to the properties of each of those element objects.

Just after the `if` statement, we can see the code that sets the pixel width of the bar `div` according to the number of seconds taken to generate each response. We multiply that time figure by 20 to get a reasonable width, but you may want to use a higher or lower number depending on how much horizontal space is available on the page.

To add text to elements, we use `createTextNode` in conjunction with `appendChild`, which is also used to place elements inside other elements.

`createTextNode` and Non-breaking Spaces

In the code above, we create a non-breaking space using `u00A0`. If we try to use the normal ` ` entity here, `createTextNode` will attempt to be "helpful" by converting the ampersand to `&`; the result of this is that ` ` is displayed on your page. The workaround is to use the escaped unicode non-breaking space: `u00A0`.

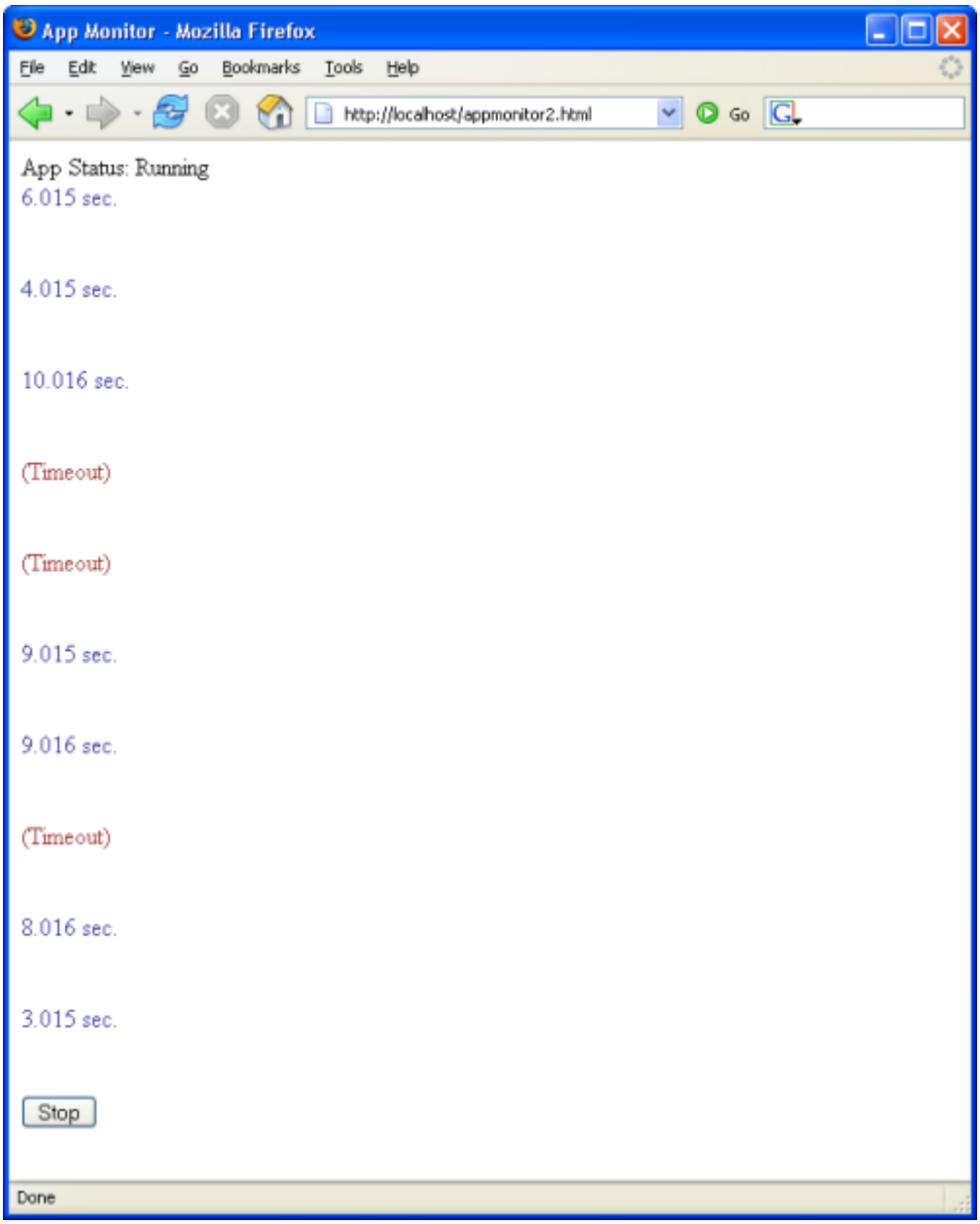

Figure 3.6. The application starting to take shape

The last piece of the code puts all the div elements together, then places the pollResult div inside the pollResults div. Figure 3.6 shows the running application.

“Hold on a second,” you may well be thinking. “Where’s the bar graph we’re supposed to be seeing?”

The first bar is there, but it's displayed in white on white, which is pretty useless. Let's make it visible through our application's CSS:

Example 3.27. appmonitor2.css (excerpt)

```
.time {
  width: 6em;
  float: left;
}
.bar {
  background: #ddf;
  float: left;
}
.clearBoth {
  clear: both;
}
```

The main point of interest in the CSS is the `float: left` declarations for the `time` and `bar div` elements, which make up the time listing and the colored bar in the bar graph. Floating them to the left is what makes them appear side by side. However, for this positioning technique to work, an element with the `clearBoth` class must appear immediately after these two `div`s.

This is where you can see AJAX in action. It uses bits and pieces of all these different technologies — `XMLHttpRequest`, the W3C DOM, and

CSS — wired together and controlled with JavaScript. Programmers often experience the biggest problems with CSS and with the practicalities of building interface elements in their code.

As an AJAX programmer, you can either try to depend on a library to take care of the CSS for you, or you can learn enough to get the job done. It's handy to know someone smart who's happy to answer lots of questions on the topic, or to have a good book on CSS (for example, SitePoint's *The CSS Anthology: 101 Essential Tips, Tricks & Hacks*).

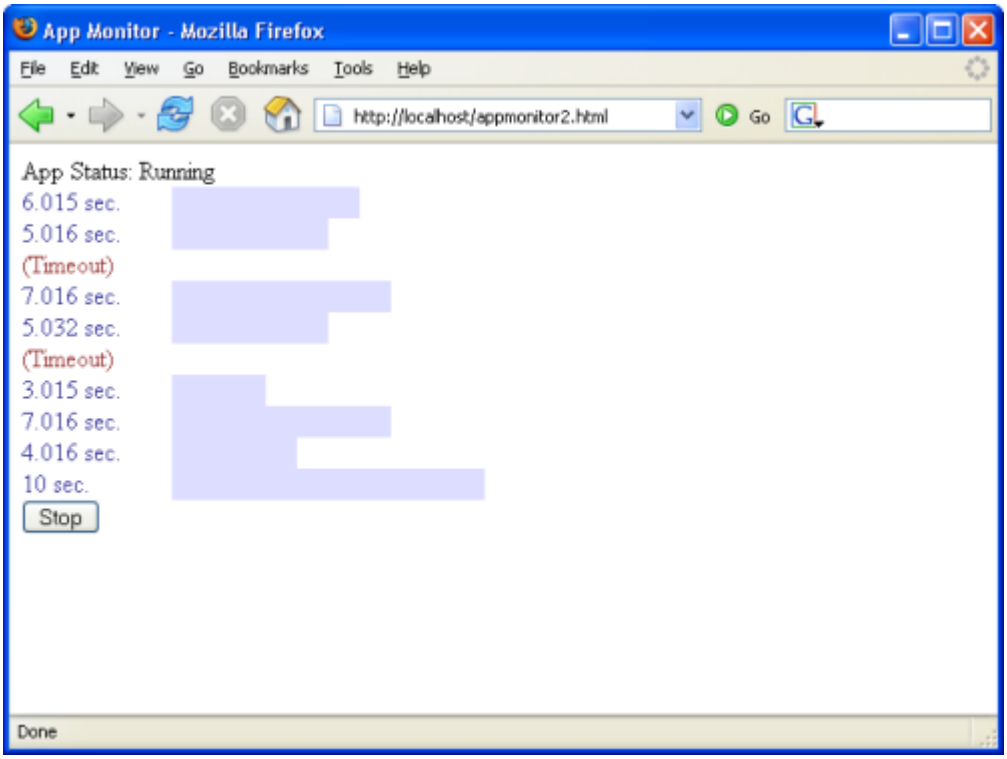

Figure 3.7. The beginnings of our bar graph

Now that our CSS is in place, we can see the bar graph in our application display, as Figure 3.7 illustrates.

Stopping the Application

The final action of the `pollServerStart` method, after getting the app running, is to call `toggleAppStatus` to toggle the appearance of the application. `toggleAppStatus` changes the status display to App Status: Running, switches the Start button to a Stop button, and

attaches the `pollServerStop` method to the button's `onclick` event.

The `pollServerStop` method stops the ongoing polling process, then toggles the application back so that it looks like it's properly stopped:

```
Example 3.28. appmonitor2.js (excerpt)



this.pollServerStop = function() {

  var self = Monitor;

  if (self.stopPoll()) {

    self.toggleAppStatus(true);

  }

  self.reqStatus.stopProc(false);

};
```

This code reuses the `stopPoll` method we added earlier in the chapter. At the moment, all that method does is abort the current HTTP request, which is fine while we're handling a timeout. However, this method needs to handle two other scenarios as well.

The first of these scenarios occurs when the method is called during the poll interval (that is, after we receive a response to an HTTP request, but before the next request is sent). In this scenario, we need to cancel the delayed call to `doPoll`.

The second scenario that this method must be able to handle arises when `stopPoll` is called after it has sent a request, but before it

receives the response. In this scenario, the timeout handler needs to be canceled.

As we keep track of the interval IDs of both calls, we can modify `stopPoll` to handle these scenarios with two calls to `clearTimeout`:

Example 3.29. appmonitor2.js (excerpt)

```
this.stopPoll = function() {
  var self = Monitor;
  clearTimeout(self.pollHand);
  if (self.ajax) {
    self.ajax.abort();
  }
  clearTimeout(self.timeoutHand);
  return true;
};
```

Now, you should be able to stop and start the polling process just by clicking the Start/Stop button beneath the bar graph.

Status Notifications

The ability of AJAX to update content asynchronously, and the fact that updates may affect only small areas of the page, make the display of status notifications a critical part of an AJAX app's design and development. After all, your app's users need to know what the app is doing.

Back in the old days of web development, when an entire page had to reload in order to reflect any changes to its content, it was perfectly clear to end users when the application was communicating with the server. But our AJAX web apps can talk to the server in the background, which means that users don't see the complete page reload that would otherwise indicate that something was happening.

So, how will users of your AJAX app know that the page is communicating with the server? Well, instead of the old spinning globe or waving flag animations that display in the browser chrome, AJAX applications typically notify users that processing is under way with the aid of small animations or visual transitions. Usually achieved with CSS, these transitions catch users' eyes — without being distracting! — and provide hints about what the application is doing. An important aspect of the good AJAX app design is the development of these kinds of notifications.

The Status Animation

Since we already have at the top of our application a small bar that tells the user if the app is running or stopped, this is a fairly logical place to display a little more status information.

Animations like twirling balls or running dogs are a nice way to indicate that an application is busy — generally, you'll want to display an image that uses movement to indicate activity. However, we don't want to use a cue that's going to draw users' attention away from the list, or drive people to distraction as they're trying to read the results, so we'll just go with the slow, pulsing animation shown in Figure 3.8.

This animation has the added advantages of being lightweight and easy to implement in CSS — no Flash player is required, and there's no bulky GIF image to download frame by tedious frame.

The far right-hand side of the white bar is unused space, which makes it an ideal place for this kind of notification: it's at the top of the user

interface, so it's easy to see, but it's off to the right, so it's out of the way of people who are trying to read the list of results.

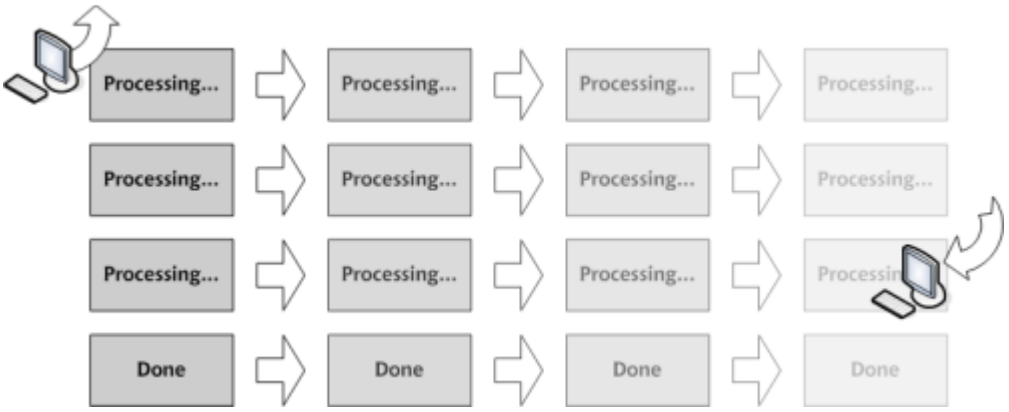

Figure 3.8. Our pulsing status animation

To host this animation, we'll add a `div` with the ID `pollingMessage` just below the status message `div` in our document:

```
Example 3.30. appmonitor2.html (excerpt)



<body>

  <div id="statusMessage">App Status:

    <span id="currentAppState"></span>

  </div>

  <div id="pollingMessage"></div>

  <div id="pollResults"></div>

  <div id="buttonArea"></div>

</body>
```

Add a CSS rule to your style sheet to position this `div`:

```
Example 3.31. appmonitor2.css (excerpt)



#pollingMessage {

  float: right;

  width: 80px;

  padding: 0.2em;

  text-align: center;

}
```

This animation is now positioned to the right of the page.

When you open the page in your browser, you won't be able to see the animation — it's nothing but a white box on a white background at the moment. If you'd like to, add some content to `pollingMessage` to see where it's positioned.

`setInterval` and Loss of Scope

The JavaScript setInterval is an obvious and easy way to handle a task that occurs repeatedly — for instance, to control a pulsing animation.

All the CSS gyrations with `setInterval` result in some fairly interesting and bulky code. So, as I mentioned before, it makes sense to put the code for the status animation into its own class — `Status` — that we can reference and use from the `Monitor` class.

Some of the clever developers reading this may already have guessed that `setInterval` suffers from the same loss-of-scope problems as `setTimeout`: the object keyword `this` becomes lost. Since we have to deal with only one status animation in our monitoring application, it makes sense to take the expedient approach, and make

our `Status` class a singleton class, just as we did for the `Monitor` class.

Setting Up `Status`

Let's start by adding some properties to the Status stub we've already written, in order to get the previous code working:

Example 3.32. appmonitor2.js (excerpt)

```
var Status = new function() {

  this.currOpacity = 100;

  this.proc = 'done'; // 'proc', 'done' or
'abort'

  this.procInterval = null;

  this.div = null;

  this.init = function() {

    // don't mind me, I'm just a stub ...

  };

  this.startProc = function() {

    // another stub function

  };

  this.stopProc = function() {
```

```
    // another stub function

  };

}
```

The `Status` object has four properties:

- The `currOpacity` property tracks the opacity of the `pollingMessage div`. We use `setInterval` to change the opacity of this `div` rapidly, which produces the pulsing and fading effect.
- The `proc` property is a three-state switch that indicates whether an HTTP request is currently in progress, has been completed successfully, or was aborted before completion.
- The `procInterval` property is for storing the interval ID for the `setInterval` process that controls the animation. We'll use it to stop the running animation.
- The `div` property is a reference to the `pollingMessage div`. The `Status` class manipulates the `pollingMessage div`'s CSS properties to create the animation.

Initialization

An `init` method is needed to bind the `div` property to `pollingMessage`:

```
Example 3.33. appmonitor2.js (excerpt)



this.init = function() {

  var self = Status;

  self.div =
document.getElementById('pollingMessage');
```

```
  self.setAlpha();

};
```

The `init` method also contains a call to a method named `setAlpha`, which is required for an IE workaround that we'll be looking at a bit later.

Internet Explorer Memory Leaks

DOM element references (variables that point to `div`, `td`, or `span` elements and the like) that are used as class properties are a notorious cause of memory leaks in Internet Explorer. If you destroy an instance of a class without clearing such properties (by setting them to `null`), memory will not be reclaimed.

Let's add to our `Monitor` class a cleanup method that handles the `window.onunload` event, like so:

```
Example 3.34. appmonitor2.js (excerpt)



window.onunload = Monitor.cleanup;
```

This method cleans up the `Status` class by calling that class's `cleanup` method and setting the `reqStatus` property to `null`:

```
Example 3.35. appmonitor2.js (excerpt)



this.cleanup = function() {

  var self = Monitor;

  self.reqStatus.cleanup();
```

```
  self.reqStatus = null;

};
```

The `cleanup` method in the `Status` class does the IE housekeeping:

```
Example 3.36. appmonitor2.js (excerpt)



this.cleanup = function() {

  Status.div = null;

};
```

If we don't set that `div` reference to `null`, Internet Explorer will keep the memory it allocated to that variable in a death grip, and you'll see memory use balloon each time you reload the page.

In reality, this wouldn't be much of a problem for our tiny application, but it can become a serious issue in large web apps that have a lot of DHTML. It's good to get into the habit of cleaning up DOM references in your code so that this doesn't become an issue for you.

The `displayOpacity` Method

The central piece of code in the `Status` class lives in the `displayOpacity` method. This contains the browser-specific code that's necessary to change the appropriate CSS properties of the `pollingMessage div`. Here's the code:

```
Example 3.37. appmonitor2.js (excerpt)



this.displayOpacity = function() {
```

```
  var self = Status;

  var decOpac = self.currOpacity / 100;

  if (document.all && typeof window.opera ==
'undefined') {

    self.div.filters.alpha.opacity =
self.currOpacity;

  }

  else {

    self.div.style.MozOpacity = decOpac;

  }

  self.div.style.opacity = decOpac;

};
```

The `currOpacity` property of the object represents the opacity to which the `pollingMessage div` should be set. Our implementation uses an integer scale ranging from 0 to 100, which is employed by Internet Explorer, rather than the fractional scale from zero to one that's expected by Mozilla and Safari. This choice is just a personal preference; if you prefer to use fractional values, by all means do.

In the method, you'll see a test for `document.all` — a property that's supported only by IE and Opera — and a test for `window.opera`, which, unsurprisingly, is supported only by Opera. As such, only IE should execute the if clause of this if statement. Inside this IE branch of the `if` statement, the proprietary `alpha.opacity` property is used to set opacity, while in the `else` clause, we use the older `MozOpacity` property, which is supported by older Mozilla-based browsers.

Finally, this method sets the opacity in the standards-compliant way: using the `opacity` property, which should ultimately be supported in all standards-compliant browsers.

IE Gotchas

Internet Explorer version 6, being an older browser, suffers a couple of issues when trying to render opacity-based CSS changes.

Fortunately, the first of these is easily solved by an addition to our `pollingMessage` CSS rule:

Example 3.38. appmonitor2.css (excerpt)

```
#pollingMessage {
  float: right;
  width: 80px;
  padding: 0.2em;
  text-align: center;
  background: #fff;
}
```

The addition of the background property fixes the first specific problem with Internet Explorer. We must set the background color of an element if we want to change its opacity in IE, or the text will display with jagged edges. Note that setting background to transparent will not work: it must be set to a specific color.

The second problem is a little trickier if you want your CSS files to be valid. IE won't let you change the `style.alpha.opacity` unless it's

declared in the style sheet first. Now, if you don't mind preventing your style sheets from being passed by the W3C validator, it's easy to fix this problem by adding another declaration:

Example 3.39. appmonitor2.css (excerpt)

```
#pollingMessage {
  float: right;
  width: 80px;
  padding: 0.2em;
  text-align: center;
  background: #fff;
  filter: alpha(opacity = 100);
}
```

Unfortunately, this approach generates CSS warnings in browsers that don't support that proprietary property, such as Firefox 1.5, which displays CSS warnings in the JavaScript console by default. A solution that's better than inserting IE-specific style information into your global style sheet is to use JavaScript to add that declaration to the `pollingMessage` `div`'s `style` attribute in IE only. That's what the `setAlpha` method that's called in `init` achieves. Here's the code for that method:

Example 3.40. appmonitor2.js (excerpt)

```
this.setAlpha = function() {
  var self = Status;
  if (document.all && typeof window.opera ==
      'undefined') {
    var styleSheets = document.styleSheets;
    for (var i = 0; i < styleSheets.length; i++)
{
      var rules = styleSheets[i].rules;
      for (var j = 0; j < rules.length; j++) {
        if (rules[j].selectorText ==
            '#pollingMessage') {
          rules[j].style.filter =
              'alpha(opacity = 100)';
          return true;
        }
      }
    }
  }
  return false;
};
```

This code, which executes only in Internet Explorer, uses the `document.styleSheets` array to iterate through each style sheet that's linked to the current page. It accesses the rules in each of those style sheets using the `rules` property, and finds the style we want by looking at the `selectorText` property. Once it has the right style in the `rules` array, it gives the `filter` property the value it needs to change the opacity.

Opacity in Opera?

Unfortunately, at the time of writing, even the latest version of Opera (version 8.5) doesn't support CSS opacity, so such an animation does not work in that browser. However, this feature is planned for Opera version 9.

Running the Animation

The code for the processing animation consists of five methods: the first three control the "Processing …" animation, while the remaining two control the "Done" animation. The three methods that control the "Processing …" animation are:

- `startProc`, which sets up the "Processing …" animation and schedules repeated calls to `doProc` with `setInterval`
- `doProc`, which monitors the properties of this class and sets the current frame of the "Processing …" animation appropriately
- `stopProc`, which signals that the "Processing …" animation should cease

The two that control the "Done" animation are:

- `startDone` sets up the "Done" animation and schedules repeated calls to `doDone` with `setInterval`
- `doDone` sets the current frame of the "Done" animation and terminates the animation once it's completed

Starting it Up

Setting the animation up and starting it are jobs for the `startProc` method:

```
Example 3.41. appmonitor2.js (excerpt)



this.startProc = function() {

  var self = Status;

  self.proc = 'proc';

  if (self.setDisplay(false)) {

    self.currOpacity = 100;

    self.displayOpacity();

    self.procInterval = setInterval(self.doProc,
90);

  }

};
```

After setting the `proc` property to `proc` (processing), this code calls the `setDisplay` method, which sets the color and content of the `pollingMessage div`. We'll take a closer look at `setDisplay` next.

Once the code sets the color and content of the `pollingMessage div`, it initializes the `div`'s opacity to 100 (completely opaque) and calls `displayOpacity` to make this setting take effect.

Finally, this method calls `setInterval` to schedule the next step of the animation process. Note that, as with `setTimeout`,

the `setInterval` call returns an interval ID. We store this in the `procInterval` property so we can stop the process later.

Both the “Processing …” and “Done” animations share the `setDisplay` method:

Example 3.42. appmonitor2.js (excerpt)

```
this.setDisplay = function(done) {
  var self = Status;
  var msg = '';
  if (done) {
    msg = 'Done';
    self.div.className = 'done';
  }
  else {
    msg = 'Processing...';
    self.div.className = 'processing';
  }
  if (self.div.firstChild) {
    self.div.removeChild(self.div.firstChild);
  }
```

```
  self.div.appendChild(document.createTextNode(msg)
);

  return true;

};
```

Since the only differences between the “Processing …” and “Done” states of the `pollingMessage div` are its color and text, it makes sense to use this common function to toggle between the two states of the `pollingMessage div`. The colors are controlled by assigning classes to the `pollingMessage div`, so we’ll need to add CSS class rules for the `done` and `processing` classes to our style sheet:

```
Example 3.43. appmonitor2.css (excerpt)



.processing {

  color: #339;

  border: 1px solid #339;

}

.done {

  color:#393;

  border:1px solid #393;

}
```

Making it Stop

Stopping the animation smoothly requires some specific timing. We don’t want the animation to stop abruptly right in the middle of a

pulse. We want to stop it in the natural break, when the “Processing …” image’s opacity is down to zero.

So the `stopProc` method for stopping the animation doesn’t actually stop it per se — it just sets a flag to tell the animation process that it’s time to stop when it reaches a convenient point. This is a lot like the phone calls received by many programmers at the end of the day from wives and husbands reminding them to come home when they get to a logical stopping point in their code.

Since very little action occurs here, the method is pretty short:

Example 3.44. appmonitor2.js (excerpt)

```
this.stopProc = function(done) {
  var self = Status;
  if (done) {
    self.proc = 'done';
  }
  else {
    self.proc = 'abort';
  }
};
```

This method does have to distinguish between two types of stopping: a successfully completed request (`done`) and a request from the user to stop the application (`abort`).

The `doProc` method uses this flag to figure out whether to display the "Done" message, or just to stop.

Running the Animation with `doProc`

The `doProc` method, which is invoked at 90 millisecond intervals, changes the opacity of the `pollingMessage` `div` to produce the pulsing effect of the processing animation. Here's the code:

Example 3.45. appmonitor2.js (excerpt)

```
this.doProc = function() {
  var self = Status;
  if (self.currOpacity == 0) {
    if (self.proc == 'proc') {
      self.currOpacity = 100;
    }
    else {
      clearInterval(self.procInterval);
      if (self.proc == 'done') {
        self.startDone();
      }
      return false;
```

```
    }

  }

  self.currOpacity = self.currOpacity - 10;

  self.displayOpacity();

};
```

This method is dead simple — its main purpose is simply to reduce the opacity of the `pollingMessage div` by 10% every time it's called.

The first if statement looks to see if the `div` has completely faded out. If it has, and the animation is still supposed to be running, it resets the opacity to 100 (fully opaque). Executing this code every 90 milliseconds produces a smooth effect in which the `pollingMessage div` fades out, reappears, and fades out again — the familiar pulsing effect that shows that the application is busy doing something.

If the animation is not supposed to continue running, we stop the animation by calling `clearInterval`, then, if the proc property is done, we trigger the "Done" animation with a call to `startDone`.

Starting the "Done" Animation with `startDone`

The `startDone` method serves the same purpose for the "Done" animation that the `startProc` method serves for the "Processing …" animation. It looks remarkably similar to `startProc`, too:

```
Example 3.46. appmonitor2.js (excerpt)



this.startDone = function() {
```

```
  var self = Status;

  if (self.setDisplay(true)) {

    self.currOpacity = 100;

    self.displayOpacity();

    self.procInterval = setInterval(self.doDone,
90);

  }

};
```

This time, we pass `true` to `setDisplay`, which will change the text to “Done” and the color to green.

We then set up calls to `doDone` with `setInterval`, which actually performs the fadeout.

The Final Fade

The code for `doDone` is significantly simpler than the code for `doProc`. It doesn’t have to process continuously until told to stop, like `doProc` does. It just keeps on reducing the opacity of the `pollingMessage` `div` by 10% until it reaches zero, then stops itself. Pretty simple stuff:

```
Example 3.47. appmonitor2.js (excerpt)



this.doDone = function() {

  var self = Status;
```

```
  if (self.currOpacity == 0) {

    clearInterval(self.procInterval);

  }

  self.currOpacity = self.currOpacity - 10;

  self.displayOpacity();

};
```

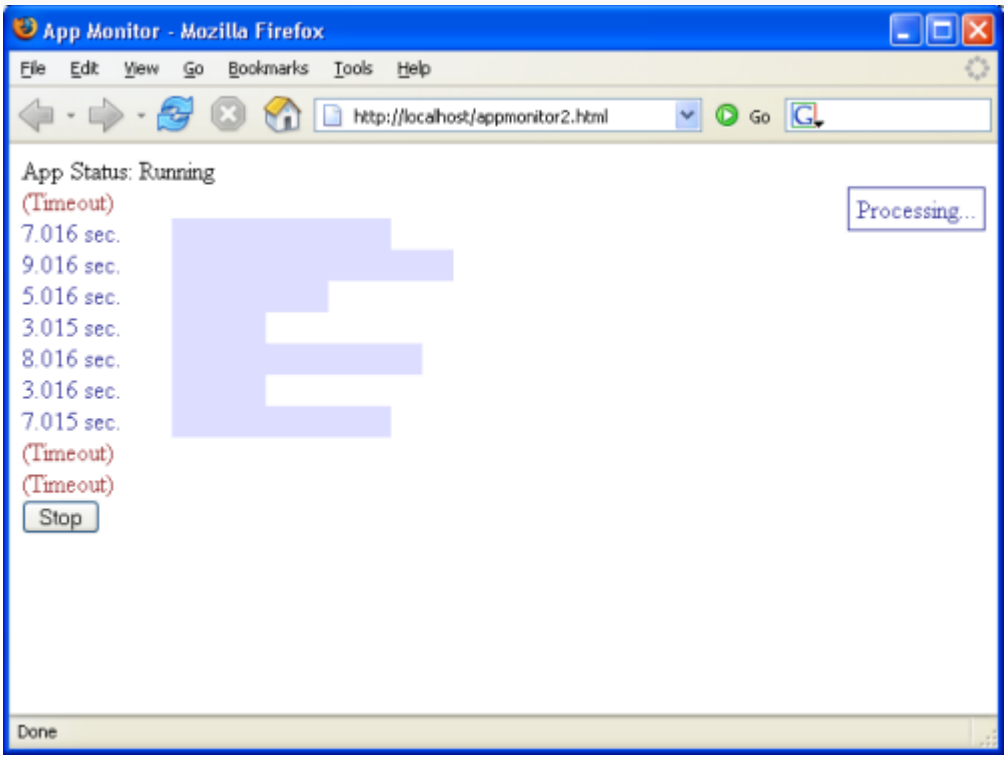

Figure 3.9. The application with a pulsing status indicator

Finally, we're ready to test this code in our browser. Open `appmonitor2.html` in your browser, click the Start button, and you should see a pulsing Processing … message near the top right-hand corner of the browser's viewport, like the one shown in Figure 3.9.

Be Careful with that Poll Interval!

Now that we have an animation running in the page, we need to be careful that we don't start the animation again before the previous one stops. For this reason, it's highly recommended that you don't set `POLL_INTERVAL` to anything less than two seconds.

Styling the Monitor

Now that we've got our application up and running, let's use CSS to make it look good. We'll need to add the following markup to achieve our desired layout:

Example 3.48. appmonitor2.html (excerpt)

```
<body>
  <div id="wrapper">
    <div id="main">
      <div id="status">
        <div id="statusMessage">App Status:
          <span id="currentAppState"></span>
        </div>
        <div id="pollingMessage"></div>
        <br class="clearBoth" />
      </div>
      <div id="pollResults"></div>
      <div id="buttonArea"></div>
```

```
    </div>

  </div>

</body>
```

As you can see, we've added three `div`s from which we can hang our styles, and a line break to clear the floated application status message and animation. The completed CSS for this page is as follows; the styled interface is shown in Figure 3.10:

Example 3.49. appmonitor2.css

```
body, p, div, td, ul {

  font-family: verdana, arial, helvetica, sans-
serif;

  font-size:12px;

}

#wrapper {

  padding-top: 24px;

}

#main {

  width: 360px;

  height: 280px;

  padding: 24px;
```

```
  text-align: left;
  background: #eee;
  border: 1px solid #ddd;
  margin:auto;
}
#status {
  width: 358px;
  height: 24px;
  padding: 2px;
  background: #fff;
  margin-bottom: 20px;
  border: 1px solid #ddd;
}
#statusMessage {
  font-size: 11px;
  float: left;
  height: 16px;
  padding: 4px;
  text-align: left;
```

```
  color: #999;
}
#pollingMessage {
  font-size: 11px;
  float: right;
  width: 80px;
  height: 14px;
  padding: 4px;
  text-align: center;
  background: #fff;
}
#pollResults {
  width: 360px;
  height: 210px;
}
#buttonArea {
  text-align: center;
}
.pollResult {
```

```
  padding-bottom: 4px;
}
.time {
  font-size: 11px;
  width: 74px;
  float: left;
}
.processing {
  color: #339;
  border: 1px solid #333399;
}
.done {
  color: #393;
  border: 1px solid #393;
}
.bar {
  background: #ddf;
  float: left;
}
```

```
.inputButton {
  width: 8em;
  height: 2em;
}
.clearBoth {
  clear: both;
}
```

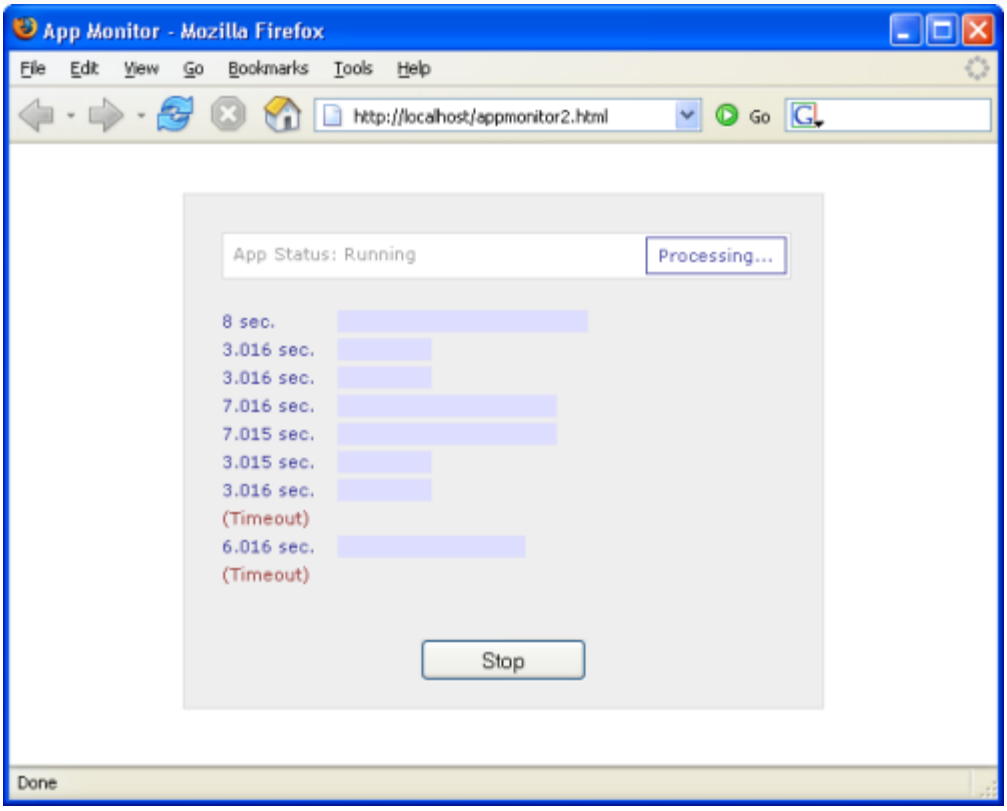

Figure 3.10. The completed App Monitor

Summary

Our first working application showed how AJAX can be used to make multiple requests to a server without the user ever leaving the currently loaded page. It also gave a fairly realistic picture of the kind of complexity we have to deal with when performing multiple tasks

asynchronously. A good example of this complexity was our use of `setTimeout` to time the XMLHttpRequest requests. This example provided a good opportunity to explore some of the common problems you'll encounter as you develop AJAX apps, such as loss of scope and connection timeouts, and provided practical solutions to help you deal with them.

That's it for this excerpt from *Build Your Own AJAX Web Applications* — don't forget, you can download this article in .pdf format. The book has eight chapters in total, and by the end of it, readers will have built numerous fully functioning web apps including an online chess game that multiple players can play in real time — the book's Table of Contents has the full details.

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

Modified: 2021.10.04.7.10.AM

Dököll Solutions,. Inc.