How-To: Build a Web Application with Ajax Part 5

Displaying Results to Browser

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`.

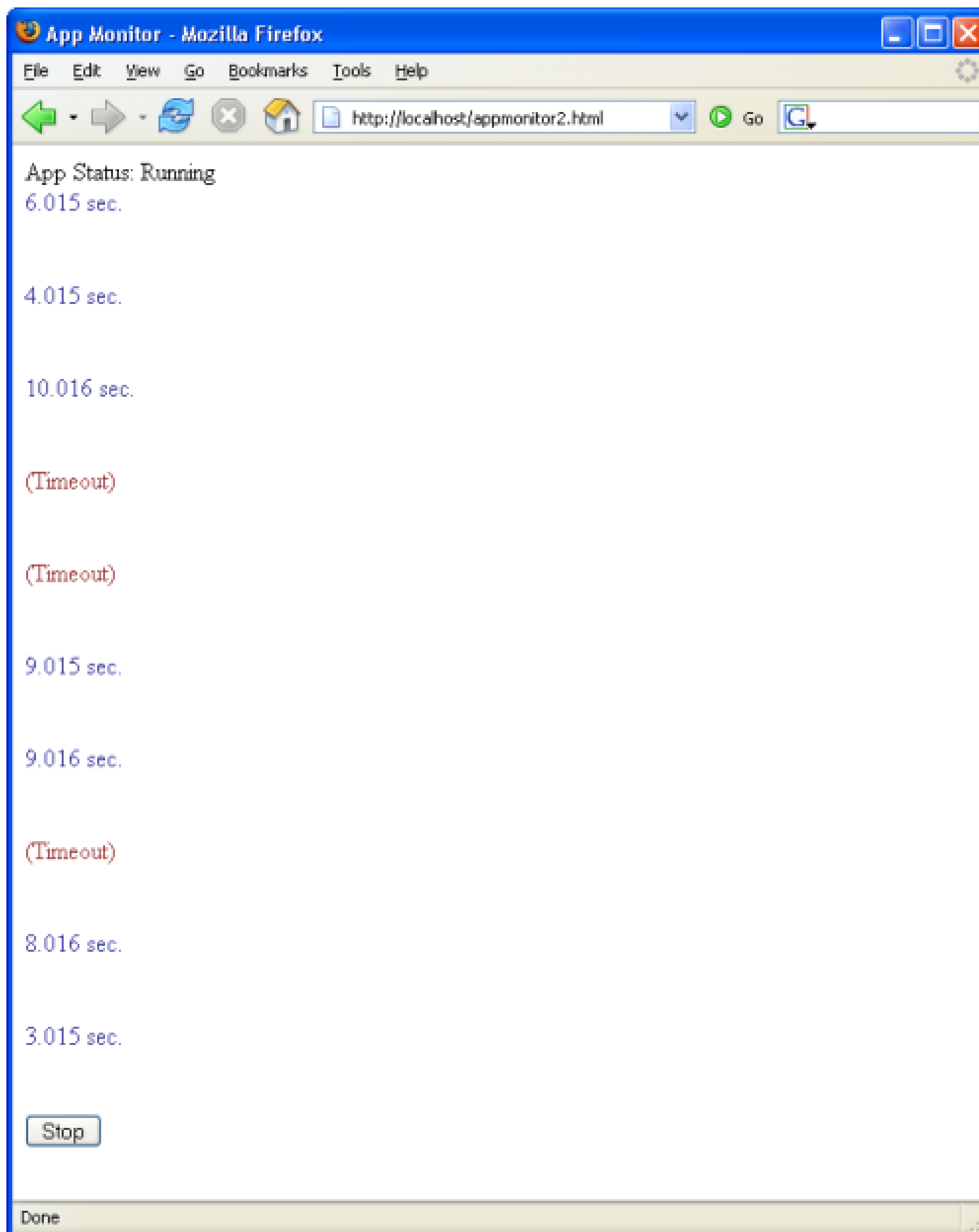

Figure 3.6. The application starting to take shape

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*).

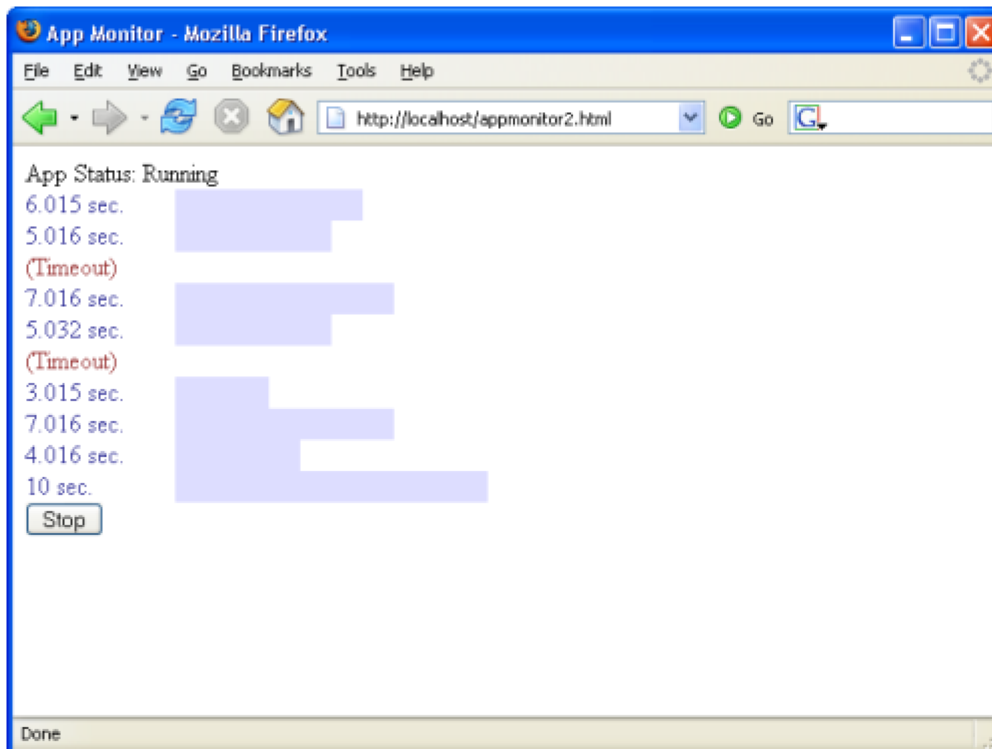

Figure 3.7. The beginnings of our bar graph

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.

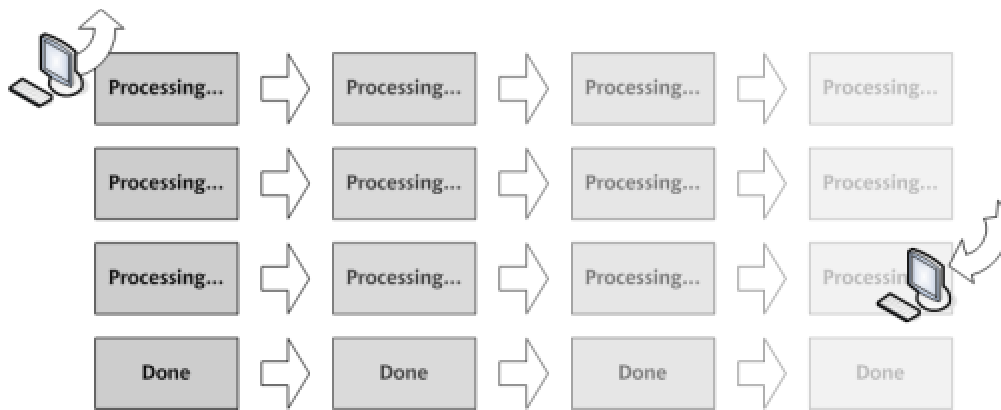

Figure 3.8. Our pulsing status animation

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();
};
```

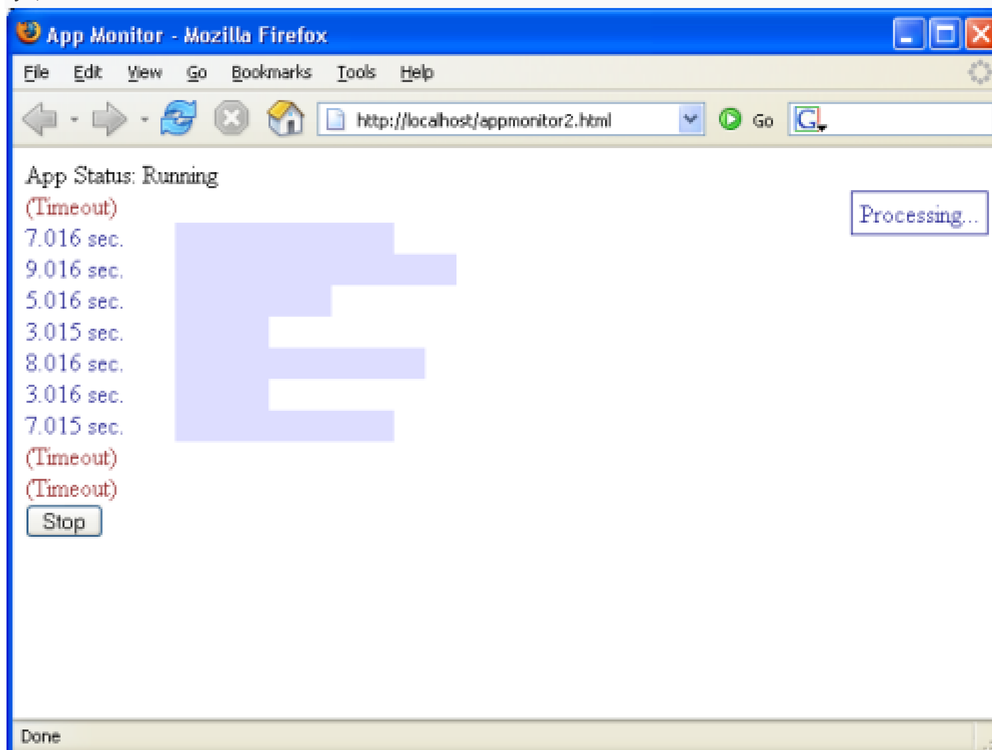

Figure 3.9. The application with a pulsing status indicator

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;
}
```

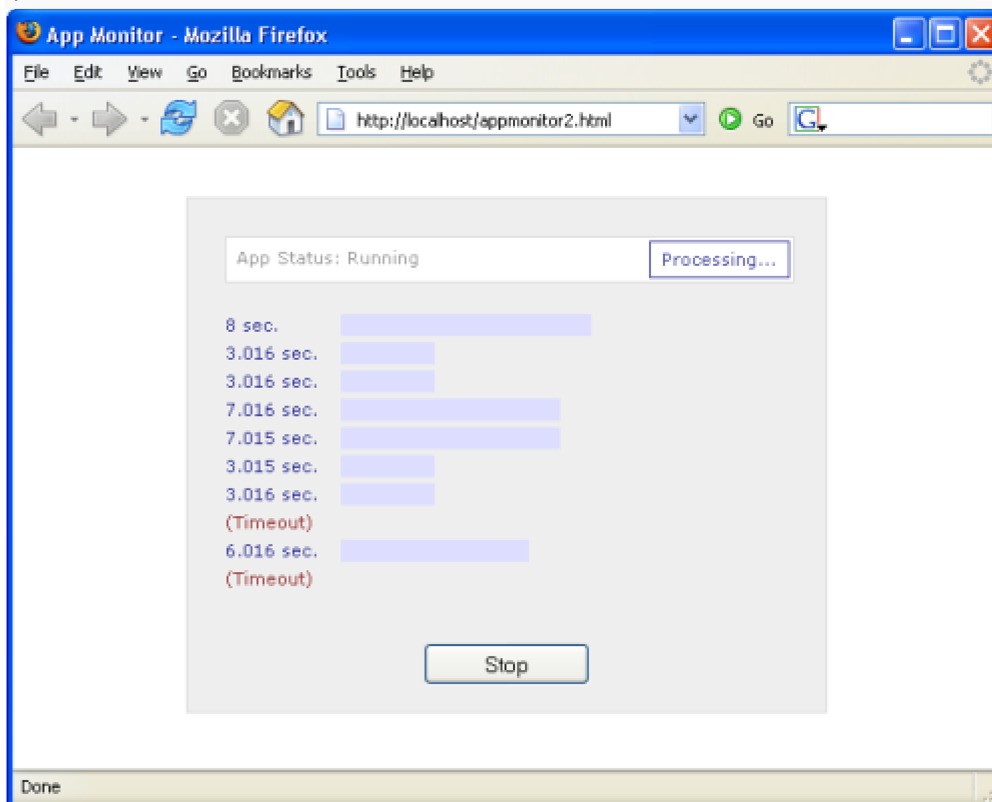

Figure 3.10. The completed App Monitor

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.

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

Modified: 2021.10.04.7.10.AM

Dököll Solutions,. Inc.