How-To: Build a Web Application with Ajax Part 4

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.

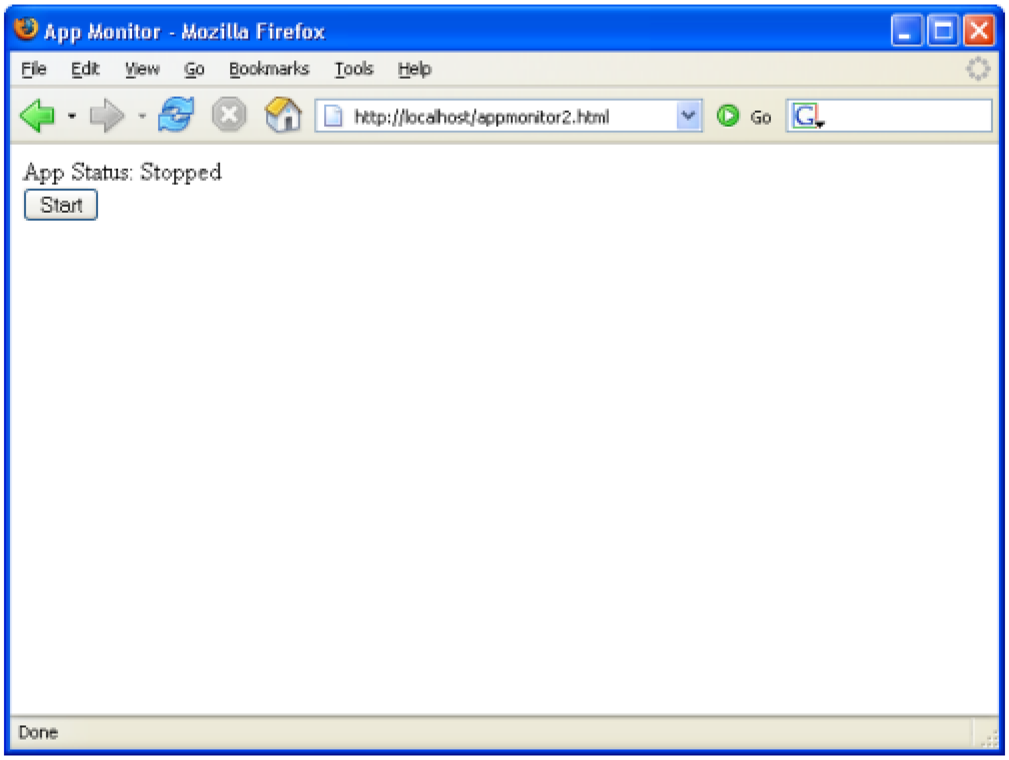
```
Example 3.11. appmonitor2.js (excerpt)



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.


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

App Status: Stopped

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:

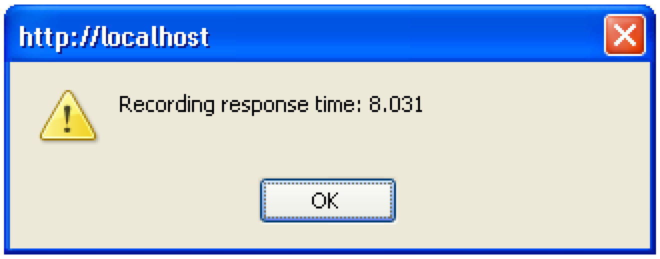
```
Example 3.17. appmonitor2.js (excerpt)



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.

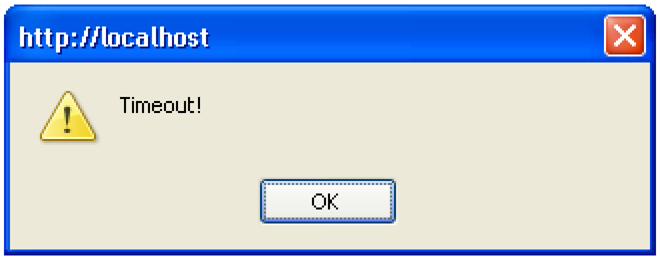
You can expect one of two outcomes: either a response is received by your page, and you see a dialog similar to the one shown in Figure 3.3, or you see the timeout message shown in Figure 3.4.



Recording response time: 8.031

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.



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.

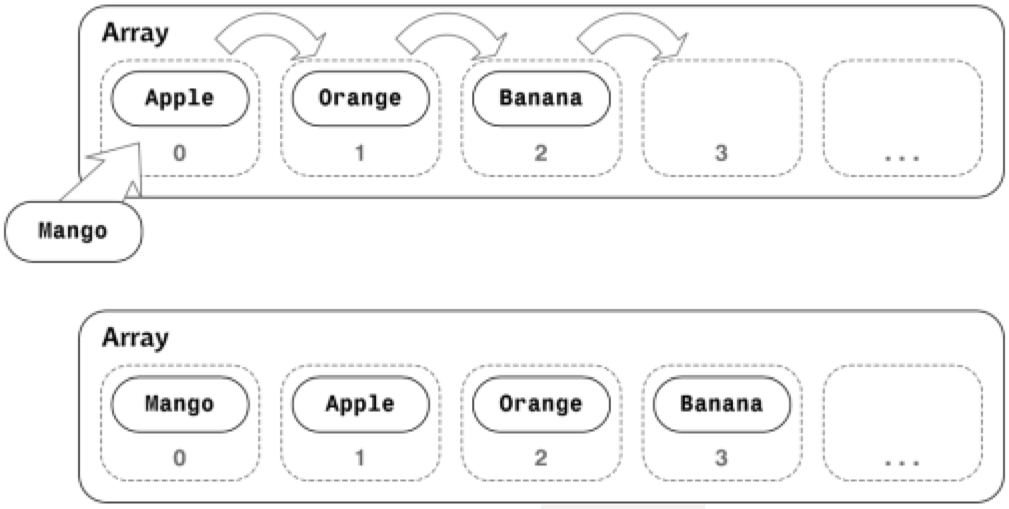


Figure 3.5. Inserting fruit using `unshift`

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.

Continue with Part 5: Displaying content to Browser

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

Modified: 2021.10.04.7.10.AM

Dököll Solutions,. Inc.