AUG 20 2014 — *by Tom Coleman* in **TUTORIALS**

Get the door, Javes! Understanding Sync & Async in JavaScript

Sync. Async. Single-threaded. Callback hell. Fibers. If you use JavaScript – whether it's on the client or server – you've probably ran across these terms before.

It's easy to realize that these concepts are all pretty important. But it can be harder to see how the pieces all fit together.

So in this article, we'll take a look at how these concepts apply to JavaScript and Node.js. Then, in the second part of this series we'll focus on Meteor more specifically. Let's get started!

Meet Javes

You rang, sir?

In the browser, JavaScript is used to provide interactivity: you click a button, your click triggers a bit of JavaScript code (known as an **event handler function**), and an alert pops up.

You can think of JavaScript in the browser as your own personal **Jeeves**, a valet who executes various tasks – i.e. event handlers – for you. We'll call this "JavaScript Jeeves" **Javes** for short.

You only have one valet, so while he'll faithfully follow every one of your orders, he crucially can only take care of *one task at a time*. In programming terms, we say JavaScript is **single-threaded**.

Single-Threading

In a single-threaded environment, only one section of code can be running at any one time. Consider the following example:

```
$('.button').click(function () {
  var answer = prompt("What’s your name, sir?");
  console.log(answer);
});

$('.button').click(function () {
  console.log("The guests have arrived.");
});
```

Even though both event handlers run on the `click` event, the second handler won't run until you've provided your name to the prompt and the first handler has completed.

Moreover, the browser won't respond to *any* events whatsoever while it completes the first handler. After all, Javes can't answer the phone while he's opening the door.

Tasks like these that occupy Javes' full attention are called **blocking** or **synchronous**.

Asynchronous Functions

Synchronous functions are generally not a big problem because we have fast computers, and most event handlers are executed instantly.

But what happens when you need more time though? For example, you might need to query third-party servers for data, or make API calls. In other words, send Javes to the post office to post a telegram and get a reply.

Having the browser lock up while we wait for a call to finish would be like having Javes wait at the post office for the reply to every telegram he sends. We can't very well have that, after all he's got work to do!

Primitive APIs (credit: Mike Cattell)

This is where **asynchronous** functions come in: whereas a synchronous task will occupy Javes continuously until its completion, an *asynchronous* task can be initiated and then put aside until a later date while our valet gets started on the next task on his to-do list.

You can think of it as Javes asking the post office to call him back when the reply arrives, at which point Javes can stroll back down there and get the mail.

The Event Loop

Your new secretary

So how does Javes keep track of which task to work on next? In JavaSscript, Javes' to-do list is called the **event loop**.

There's just one problem: Javes can only handle one task at a time, so how will he answer the post office's phone call if he's already busy polishing the silverware?

There's no two ways around it: we'll need to hire a secretary to help manage Javes' to-do list and keep it up to date as new tasks appear.

In turns out that this is exactly what happens with JavaScript, too: the JavaScript engine actually sports an additional background thread which takes care of managing the event loop.

Still, your own code will always run in a single main thread. So from the perspective of the programmer, it's safe to say that JavaScript is single-threaded.

Ajax & Callbacks

In the browser, asynchronous code usually takes the form of **Ajax**. Ajax is most commonly used through a **jQuery wrapper**:

```
$('button').click(function() {
  $.get('http://ask.com', function(data) {
    console.log("Ma’am, your data has arrived:", data);
  });
})
```

You’ll notice we now have *two* event handlers: one `function(){}` block for the click, and another passed as the second argument of the `$.get()` function.

The first handler does two things: send a request off to Ask.com, and schedule the *second* handler to run when that request comes back. We can think of this as Javes sending our telegram, and asking the post office to let him know when a reply comes back.

The second handler is known as a *callback*, and instead of responding to the `click` event, it will respond to a special `ajaxDone` event fired by the browser.

The result is that our code is now *asynchronous*, and will not lock the browser up no matter how long it takes Ask.com to answer our call.

In other words, instead of staying at the post office all day, Javes now can come home and handle other household tasks while waiting for the post office to call.

Node.js: JavaScript on the Server

On the server, the Node.js runtime is single-threaded as well.

But in Node.js, you’ll likely come across many more asynchronous operations: calling APIs, reading and writing files, executing operations on the server, etc.

For instance, let’s say you want to maintain a record of each new user log-ins in your database:

```
trackUser =  function(userId) {
  users.findOne({userId: userId}, function(err, user) {
    var logIn = {userName: user.name, when: new Date};
    logIns.insert(logIn, function(err, done) {
      console.log('wrote log-in!');
      });
  });
});
```

This time, we've set up **three** nested functions: our main `trackUser` function, one for looking up the user in the database, and one for inserting a new record.

As you can see, even a relatively simple operation requires three levels of nested functions. So you can imagine that more complex operations tend to produce even more levels and sub-levels, which is what is poetically known as **callback hell**.

An early depiction of Node.js callback hell

Dealing With Callback Hell

In the browser, a common pattern to deal with excessive callbacks is to use **promises**. jQuery ships with a simple built-in promise library that enables us to chain callbacks and deal with errors.

But in Node.js, we can use a different approach. By extending the runtime, we can abstract away the asynchronicity and write code that *looks* sychronous.

For example, this is what Meteor does behind the scenes using the **Fibers** package. So in Meteor, our database code would look a lot simpler:

```
trackUser = function(userId) {
  var user = Meteor.users.findOne({userId: userId});
  logins.insert({userName: user.name, when: new Date});
  console.log(‘wrote login!’);
}
```

Introducing Fibers

The simplest way to use Fibers directly is the **Future** sub-library.

Here’s how you would take an *asynchronous* function (`setTimeout`) and build a *synchronous* equivalent (`wait`):

```
var app = require('express')();
var Fiber = require('fibers')
var Future = require('fibers/future')

// a function that "synchronously" waits for X ms
var wait = function(ms) {
  var future = new Future;

  setTimeout(function() {
    future.return();
  }, ms);

  return future.wait();
}

app.get('/hello', function() {
  Fiber(function() {
    console.log('waiting');
    wait(10000);
    console.log('done')
  }).run();
});

var server = app.listen(3000);
```

If you run this simple app and ping that route (by typing `curl http://localhost:3000/hello` in your terminal), you’ll see that the app waits ten seconds before logging the second message, as expected.

Simulated Synchronicity

But wait. We’ve seen previously that because of JavaScript’s single-threaded nature, synchronous code will lock up the browser for the duration of its execution.

Locking up the browser was already bad enough, but locking up the *server* would be disastrous: it would mean that no other client could connect to our app during our 10-second timeout.

Thankfully, Fibers doesn't *really* make asynchronous code synchronous. It just **lets us write code in a synchronous *style***, with synchronous *control flow*.

To test this out, try `curl` ing your Node.js app in rapid succession from different terminal tabs. You'll see the `waiting` message appear immediately, proving that your server's single JavaScript thread is not actually blocked.

Conclusion

That being said, how does Meteor fit into all this? As we've briefly hinted, Meteor stands firmly in the "synchronous style" camp. Yet Meteor is still part of the generally asynchronous Node.js ecosystem. And unlike Node.js, Meteor *also* runs on the client, which brings yet another set of sync/async considerations.

So stay tuned for the next part of this series, where we'll see how these concepts apply to Meteor and finally make sense of all this!

Share this article: **Tweet**

ABOUT THE AUTHOR

Tom Coleman is part of **Chroma**, a product company with ideas on code, design, and digital collaboration. Tom has been in the Meteor community since the early days and was responsible for the first **Router** and **Package System**. As a member of the Meteor Development Group, he co-wrote the **Meteor Guide** and was involved in the framework's development.

You can **follow him on Twitter.**