BLOG

How to write asynchronous code in JavaScript (and Meteor)

August 19, 2016 by Aryan Goharzad

Meteor Javascript Code

What’s this post about?

This post aims to introduce the reader to a few common ways of writing asynchronous code in Javascript and Meteor. The focus of this post is on the syntactical differences and the simplicity of the written code, rather than what goes on under the hood. We will look at the same code implemented in five ways:

- Using callbacks
- Using the caolan/async library
- Using promises
- Using async/await
- Using Fibers and wrapAsync (Meteor specific)

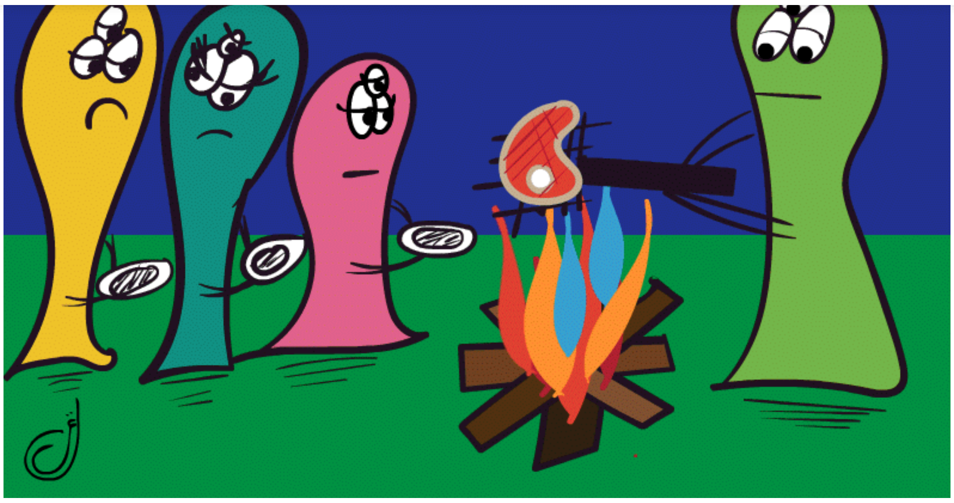
other Javascript frameworks.

Who is this post for?

This post is valuable to developers who are working with Meteor or plain JavaScript in either the client or the server. If you're already familiar with asynchronous programming, much of this article may be familiar to you. This post assumes some basic understanding of Javascript.

Why would I have to write asynchronous code?

Javascript is single-threaded, meaning that it can do only one thing at a time. This would imply a slow operation would "block", and everything else would have to wait for the slow operation to finish before they can continue.



All artworks in this post done by amazing Ghazal Tahernia

But in reality, this doesn't happen thanks to asynchronous calls. Asynchronous calls let us perform long-running I/O operations (Input/Output) without blocking. I/O operations consist of operations such as network requests and disk reads/writes.

To understand "blocking", consider the two functions below.

```
function printTwoAsync() {
  makeRequest(`posts?userId=2`, (err, results) => {
    console.log(results[0].userId);
  });
}

function printTwoSync() {
  console.log("2");
}
```

Note: If the `=>` *looks unfamiliar to you, these are ES6 standards for writing Javascript. I strongly recommend*

Note that the `printTwoSync()` just prints the number two, while `printTwoAsync()` grabs the number `2` from a network call, which means it is doing a non-blocking I/O operation that is slower than `printTwoSync()`.

Now consider a case where we want to print the numbers `1,2,3` using the functions above to print the `2`. Before reading the rest, try to guess the outcome of each of the functions below.

```
function blockIO(){
  console.log('1');
  printTwoSync();
  console.log('3');
}


function nonBlockIO(){
  console.log('1');
  printTwoAsync();
  console.log('3');
}
```

The output of the `blockIO()` function, is:

```
1
2
3
```

This is expected. Javascript waits for `printTwoSync()` to finish in order to run the next line.

However, the output of `nonBlockIO()` is:

This is because in the second case `printTwoAsync()` is a *slow* operation and Javascript doesn't wait for it to return in order to run the next line. However, once `printTwoAsync()` has finished its task, its callback is fired and `2` is printed. If we want to keep the order, `console.log(3)` would have to be in `printTwoAsync()`'s callback. We will get into what callbacks are below.

What are callbacks and why did Satan make them?

Let's say that your app is trying to find a user on a third party website, find their first post and the first comment from that post. Also, let's assume that you need to do the following steps in order to achieve that:

- Find the user's id
- Find all posts by that user and take the id of the first post
- Find the first comment on the given post and print it

One can observe that these operations can be different than, say, an addition operation because they take some time to come back with a result. However, these operations, like other I/O operations in Javascript, take advantage of a **callback** which is a function that is called once the HTTP request comes back. The callback *usually* has a potential error or `null` as the first parameter and the desired value as the second variable. The callback in

```
  makeRequest(`users/1`, (userErr, userResults) => {
    if (userErr) {
      throw new Meteor.Error(userErr);
    }
    const userId = userResults.id;
    makeRequest(`posts?userId=${userId}`, (postErr, postResults) => {
      if (postErr) {
        throw new Meteor.Error(postErr);
      }
      const postId = postResults[0].id;
      makeRequest(`posts/${postId}/comments`, (commentsErr, commen
        if (commentsErr) {
          throw new Meteor.Error(commentsErr);
        }
        console.log(commentsResults[0]);
      });
    });
  });
}
```

If you don't already see the problem, it's the pyramid of sad mustachioed winking faces `});`, aka the infamous callback hell. The code can get very messy very quickly. Just imagine having to do the above in a loop for each user in parallel, and limiting the number of concurrent calls so that your CPU doesn't die. Another big issue is error handling. Error handling can become very repetitive. And the final issue is that not all libraries follow the Node convention for how callbacks should be designed. We will get to the third point later in the post.

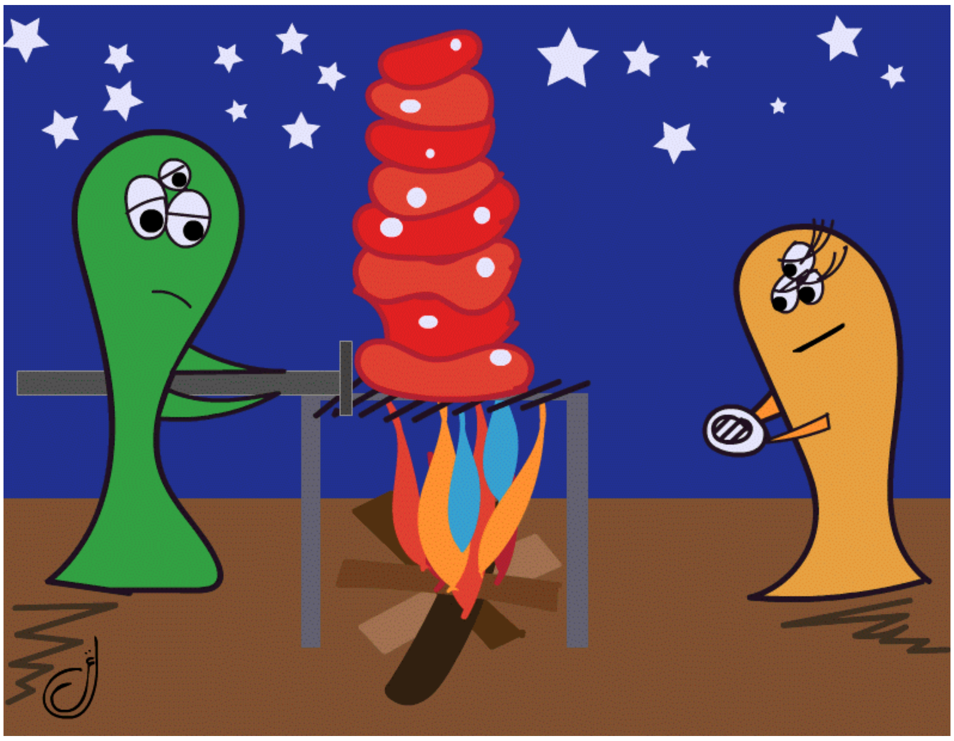
The mess in the code above can be (kind of) cleaned up by having an error handling function, and also flattening the pyramid by modularizing the functions. However, there are alternative solutions that produce more readable and maintainable code.

A third party async library - caolan/async

Thanks to Caolan's very popular async library, the code above can be simplified by using async.waterfall like below:

```
function getFirstCommentCaolonAsync() {
  async.waterfall([
    (callback) => makeRequest(`users/1`, callback),
    (userResults, callback) =>
      makeRequest(`posts?userId=${userResults.id}`, callback),
    (postResults, callback) =>
      makeRequest(`posts/${postResults.id}/comments`, callback),
  ], (err, comments) => {
    if (err) {
      throw new Meteor.Error(err);
```

The `async` package makes the code more readable and maintainable. Each of the three `makeRequest` calls is on a separate line, and thanks to `async.waterfall()` they pass the result of each call to the next step, as the first parameter.



The async package is extremely powerful for much more complicated situations by providing some of the following features:

- `async.parallel` : For parallel jobs, for example if you want make the call to get multiple users' ids
- `limits` : You can limit the number of concurrently running instances of your method depending on how powerful your server is

documentation as well as the awesome “What is Async.js solving?” article.

ES6 - Promises

An alternative solution to the pyramid of frowny faces is using promises, which were introduced in ECMAScript 6. One of the main differences between callbacks and promises is that callbacks are built with a pattern that the Javascript community has agreed to follow (e.g. Node.js convention), but it is not enforced in the language. Whereas Promises have a defined structure built in the language.

Callbacks *usually* have either an Error object or `null` in the first parameter depending on the outcome of the function, and a value in the second parameter. But some libraries use them differently by emitting the error in an event instead, or firing the callback with multiple values.

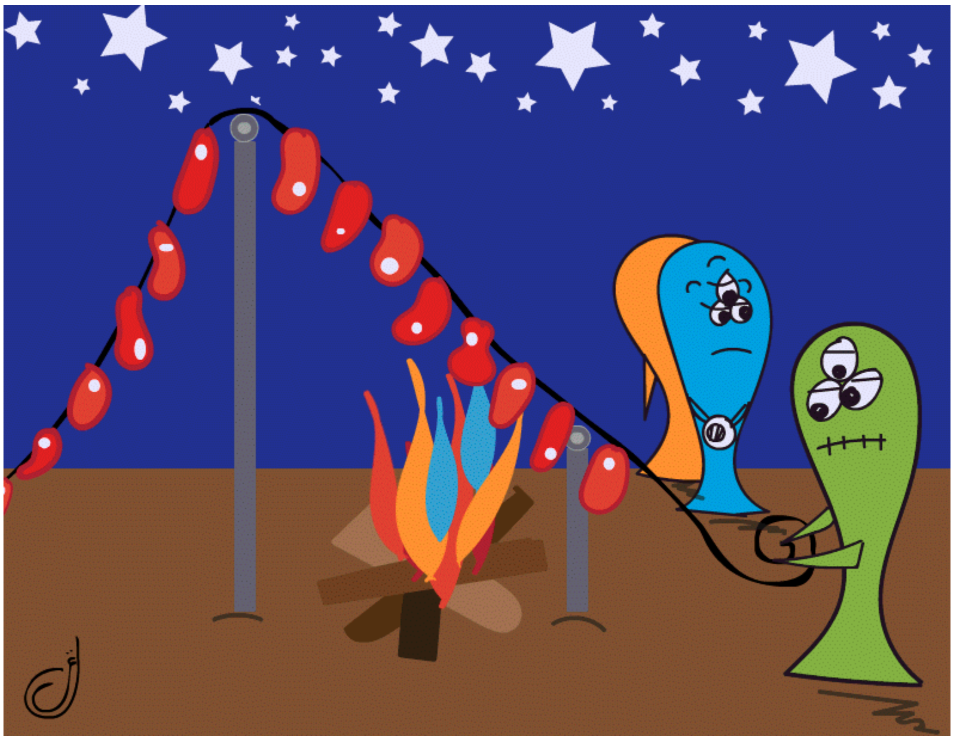
On the other hand, Promises are structured to only have the following states:

- fulfilled/rejected : Indicating if the chain of promises should continue
- pending/settled : Indicating whether or not the promise has been given the fulfilled or rejected status

To understand how promises work, observe the following example which is very similar to the

```
    .then(results => makeRequestPromise(`posts?userId=${results.id}`))
    .then(results => makeRequestPromise(`posts/${results[0].id}/commer
    .then(results => console.log(results[0]))
    .catch(err => {
      throw new Meteor.Error(err);
    });
  }
```

But how does this work? Notice that in this example we are using `makeRequestPromise()` rather than `makeRequest()`. The trick here is that your request function needs to have certain features such as being "thenable" (I did not make that word up!). This means that your function has to return an object with a `.then()` method, which connects each task it to what has to happen next.



```
function makeRequest(url, callback) {
  HTTP.call('GET', `http://jsonplaceholder.typicode.com/${url}`, {},
    (err, response) => callback(err, response.data)
  );
};
```

Note that the function fires a callback with appropriate values once the call has finished. The callback will either be fired with an error as its first argument, or `null` in the first argument and the desired value as the second argument. This is because `HTTP.call()` follows the Node.js convention for callbacks when it fires with either an error or the response.

However, in the promises world `makeRequest()` would be structured slightly different. Rather than calling a callback when it has finished, it will either `reject` or `resolve` the promise. Rejecting is akin to calling the callback with an error in the first argument, whereas resolving is similar to calling the callback with `null` in its first argument and the response in the second argument.

```
function makeRequestPromise(url, callback){
  return new Promise(function(resolve, reject) {
    HTTP.call('GET', `http://jsonplaceholder.typicode.com/${url}`, {},
    (err, response) => {
      if(err) {
        reject(err);
      } else {
        resolve(response.data);
      }
```

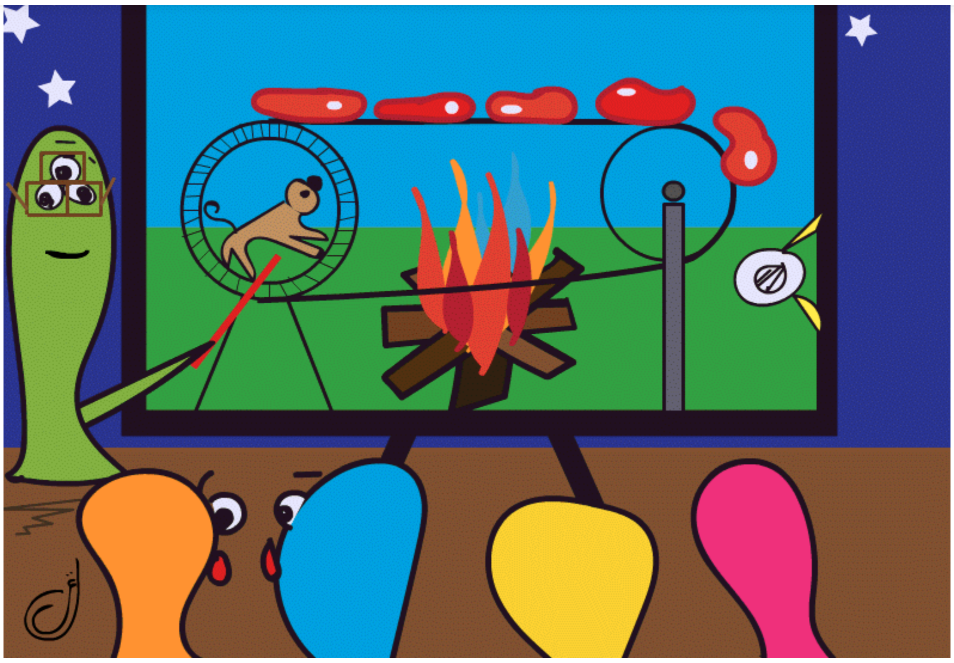
We had to jump through hoops in `makeRequestPromise()` in the example above only because we used `HTTP.call()` which works with callbacks rather than promises (at least for now). In reality, instead of above we can use libraries that work with promises such as request-promise. Using request-promise, `makeRequestPromise()` will be sexy like below:

```
const makeRequestPromise = (url) => {
  const options = {
    uri: `http://jsonplaceholder.typicode.com/${url}`,
    json: true,
  };
  return rp(options);
};
```

Much like caolan/async which provides async.parallel for running tasks in parallel promises can be run in parallel using Promise.all. Promises and the async package also provide async.race and Promise.race for cases where only the fastest task to finish should fire the callback. Note that promises can also be used in the caolan/async library using asyncify.

ES7 Async/Await:

With ES7, async/await was introduced, which along with promises make life much easier. The good news is that this already works in Meteor 1.3 so you don't need any extra libraries for it to run. This method allows the



The syntax for async/await is such that you have to add the term async before the function, and await before the thenable function like below. When a function is declared with async, it will yield execution to the calling function while it awaits for a promise to be resolved. This allows us to write synchronous-looking code, and also use the standard try-catch methods rather than then() and catch() which were used in promises. The example above can be re-written using async/await like below:

```
async function getFirstCommentAsyncAwait() {
 try {
  userResults = await makeRequestPromise(`users/1`);
  postResults = await makeRequestPromise(`posts?userId=${userResu
  comments = await makeRequestPromise(`posts/${postResults[0].id}
  console.log(comments[0]);
 } catch (err) {
  throw new Meteor.Error(err);
 }
};
```

function. Being able to write asynchronous code that looks synchronous is a big relief for the developer (as opposed to having to deal with callback hell). Note that syntactically this is very similar to Meteor's approach with Meteor.wrapAsync. Async/Await is the recommended method because it's the most readable and maintainable format compared to all the other methods discussed.

What about Meteor and Fibers?

Meteor is built on Node.js, so the issue with being single threaded and callback hells would be inherited if it wasn't for Fibers. Fibers provides an abstraction layer for Node's eventloop so that we don't have to deal with callbacks and other related issues.

Let's see how our example would look like if we took advantage of this Meteor feature. You can wrap you asynchronous function using Meteor.wrapAsync, which makes it possible to use it like a synchronous function like below:

```
const makeRequestSync = Meteor.wrapAsync(makeRequest);

function getFirstCommentWrapAsync() {
  try {
    userResults = makeRequestSync(`users/1`);
    postResults = makeRequestSync(`posts?userId=${userResults.id}`);
    comments = makeRequestSync(`posts/${postResults[0].id}/commen
    console.log(comments[0]);
  } catch (err) {
    throw new Meteor.Error(err);
```

It can be observed that the syntax is almost identical to the async/await example. However, it seems that "in the future, once async/await is more supported, Meteor will simply get rid of fibres. They are migrating to using the async/await abstraction, rather than using Fibers directly". Therefore, I personally suggest using async/await, because it will be ES7 standard very soon and your piece of code will be compatible in any other Javascript environment.

The good thing about both async/await and fibers is that they allow you to write code in a synchronous style, i.e. directly using the returned value, without having to care that the single-threaded loop interrupted the function partway through and then resumed. This is much more readable because it's closer to what you would write in any kind of pseudo-code.

This wraps up our analysis of various methods to write asynchronous code in Javascript. Check out the examples repo: https://github.com/arrygoo/async-examples. It has working code that you can play with.

Also check out Ghazal's portfolio or youtube page, she created all the cool artwork.

We'll share what we've learned, get tips and info in your inbox occasionally