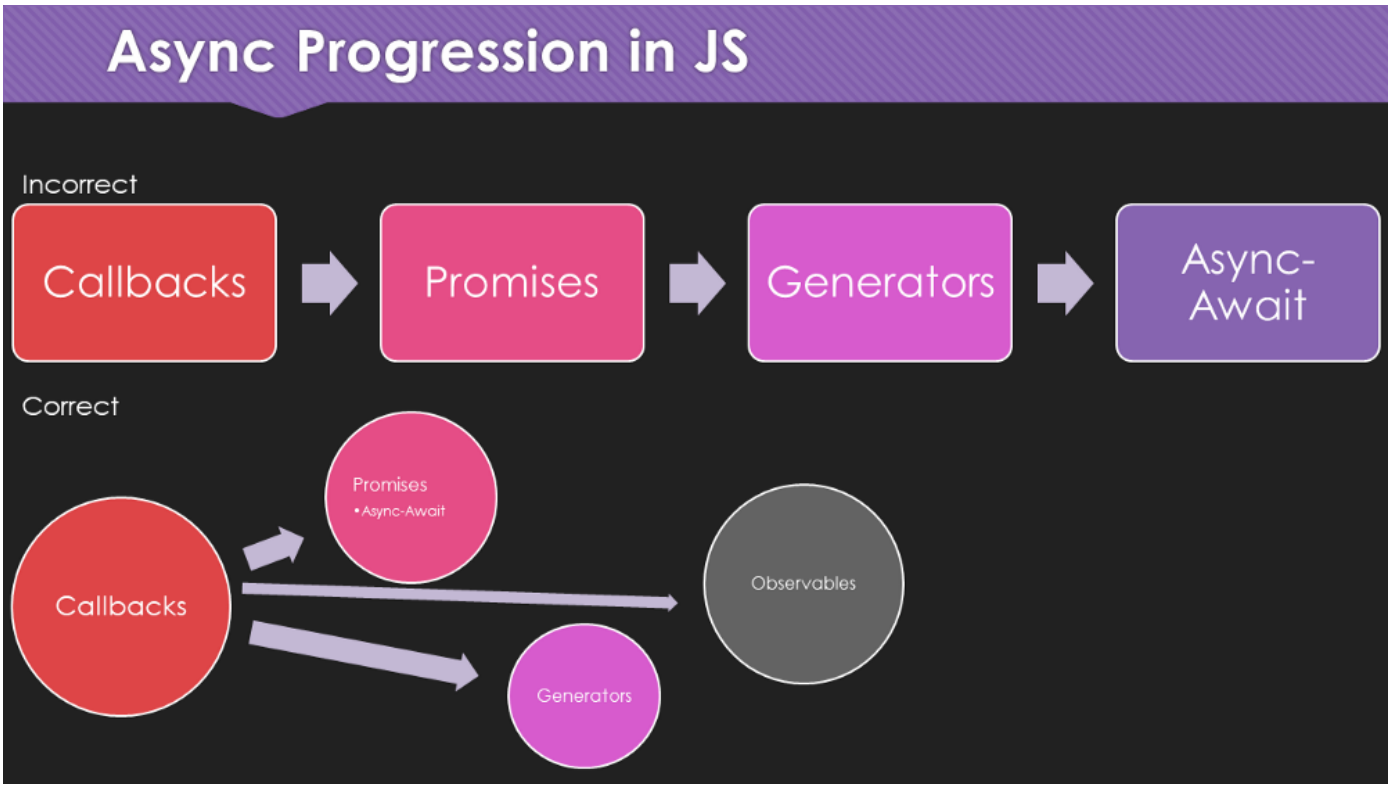

Callbacks: The Definitive Guide

Learn the Real Way to Async

Kevin Ghadyani Follow
Mar 2 · 8 min read

I don't want to name names, but a lot of articles I read about async in JavaScript, especially those that came out around the time ES2015 (ES6) was new, are just plain wrong. I'm going to clear up the falsehood by presenting a real understanding of the way async works in JavaScript.
This series of articles is based on a slide from a talk I did about Redux-Observable. I wanted to take it one step further and explain how async works and why it's a foundation to coding in JavaScript.

The Most Powerful Method

Callbacks — or callback functions — are hated by many who don't really understand async in JavaScript. They're by-far the most-powerful and amazing parts of the language; in fact, all other

complete opposite, but that's because, once I learned how to use callbacks, they became increasingly powerful.

And then there's the lack of a return value; another problematic situation. Regular functions sometimes lack return values, but with async callbacks, if you return a value, that value usually goes nowhere. It's actually possible to grab a hold of these return values, but only when your callbacks are synchronous. You'll see this kind of synchronous callback behavior all over the place in promises, observables, and even `Array.prototype` functions like `map`, `filter`, and `reduce`.

Callbacks are Also Synchronous

While callbacks are commonly thought of as asynchronous, they're also able to be executed synchronously:

```
const executeSynchronously = (
  callback,
) => {
  // Do some processing...
  callback()
}

executeSynchronously(() => {
  console
  .log('Hello World!')
})
```

executeSynchronously.js hosted with ❤ by GitHub view raw

If you've figured it out, this could really be made easier by calling another function at the end of your code instead of using a callback:

```
const logToConsole = () => {
  console
  .log('Hello World!')
}

const executeSynchronously = () => {
  // Do some processing...
  logToConsole()
}
```

executeSynchronously.js hosted with ❤ by GitHub view raw

So why would you even use a callback if it's not asynchronous? In this simpler example, the non-callback example looks easier to read, but there *are* times you might want to generalize some code and that's where data comes into play.

Callbacks With Data

```
  console
  .log(data)
}

const logDataSynchronously = () => {
  // Do some processing...
  logToConsole(data)
}
```

logDataSynchronously.js hosted with ❤ by GitHub view raw

Notice in `logDataSynchronously`, your function can only log data. There's absolutely nowhere to go after that. We could improve on this function by moving our logging operation out and returning the data. Then we can call it and log the info we received:

```
const getDataSynchronously = () => {
  // Do some processing...
  return data
}

console
.log(
  getDataSynchronously()
)
```

getDataSynchronously.js hosted with ❤ by GitHub view raw

We've made it significantly better, but we're writing how our function gives us data rather than have it do that for us. I used to write quite a bit of code like this in the past, and it's got some limitations as you'll see.

Callbacks are More Powerful

In older JavaScript applications, you wouldn't want to make functions available globally so the best way you could pass them around was using callbacks. In that same vein, if you do any unit testing, **dependency injection** is made significantly easier with callbacks.
That's why I think `getDataSynchronously` could be improved by being the callback-based `giveDataSynchronously`:

```
const giveDataSynchronously = (
  callback,
) => {
  // Do some processing...
  callback(data)
}
```

```
  .log(data)
})
```

giveDataSynchronously.js hosted with ❤ by GitHub view raw

Other than the fact that we're now passing `data`, our code has been nearly unchanged from the original callback example.

While it may seem strange to think about, `giveDataSynchronously` could be synchronous or asynchronous, and you wouldn't even need to know. That's the beauty of it. Your code is the same either way. Not every developer agrees though. JS beginners suffer the most because of the differences in how sync and async callbacks are handled. These are the cause of some very frustrating "it works sometimes and not others" situations.

If you're used to thinking in terms of callbacks, there's a lot less to worry about in terms of gotcha's because synchronous callbacks usually return values and asynchronous callbacks don't. It's that simple.

And in a completely different world, if you've ever developed a plugin for Google's Chrome browser, you'll notice some APIs accept callbacks that both return a synchronous value and call other callback functions asynchronously. Ridiculous! It's a nightmare to work with and makes your code unnecessarily complex.

Callback Hell!

Any callback chain has the possibility of becoming callback-hell. This occurs when you create a lot of anonymous — unnamed — functions as callbacks which call functions that require another callback and so on:

```
createServer((
  server,
) => {
  server
  .listen((
    connection,
  ) => {
    connection
    .listen((
      message,
    ) => {
      // Do some processing...
      console
      .log(message)
    })
  })
})
```

```
) => {
  // Do some processing...
  console
  .log(message)
}

const handleConnection = (
  connection,
) => {
  connection
  .listen(
    handleMessage
  )
}

const handleServerCreation = (
  server,
) => {
  server
  .listen(
    handleConnection
  )
}

createServer(
  handleServerCreation
)
```

namedCallbacks.js hosted with ❤ by GitHub view raw

That's one way of putting it. Callback-hell definitely makes things harder to read because they can tab out really far. I think of it differently though. Having a bunch of indirection is even worse. Without all that recursive scoping, you have no way of knowing which server or which connection your message came from. Now you have to use functional closures to retain state:

```
const handleMessage = (
  server,
  connection,
) => (
  message,
) => {
  // Do some processing...
```

```
  server,
) => (
  connection,
) => {
  connection
  .listen(
    handleMessage(
      server,
      connection,
    )
  )
}

const handleServerCreation = (
  server,
) => {
  server
  .listen(
    handleConnection(
      server,
    )
  )
}

createServer(
  handleServerCreation
)
```

namedCallbackHell.js hosted with ❤ by GitHub view raw

Congratulations. You made your application significantly harder for anyone to maintain (including you 6 months later). Good luck! I've done similar things like this before when my computational logic was just too much for callback-hell, but even better, I refactored it to use observables instead and got rid of that problem entirely.

The issue is one of scope. Because callback-hell allows for easy-scoping, it removes a lot of the indirection caused by named-callback-hell. Now you have to look all around the codebase to figure out where these functions are coming from and what they do. And in a lot of projects, most I've worked in (including my own), callbacks like this tend to be spread out among a bunch of files. The less of those you need to look at and the more-generic they are (meaning you see them more often), the better.

is why observables are the better version of callbacks. They make your code more readable and easier-to-follow as well as allow functional programming.

Asynchronous Callbacks

As stated earlier, synchronous callbacks are asynchronous callbacks. But what do those look like? What's happening in that `// Do some processing...` comment that makes a callback async?

```
setTimeout
```

Yep! Just about anything async has to do with `setTimeout`. Even `setInterval` can be built with `setTimeout`. They're referred to as schedulers because they allow you specify their execution time as "sometime after this function" all the way up to the integer max.

Another way to think about schedulers is concurrency. By asynchronously executing tasks, you're able to run many tasks concurrently even in a single thread. Pretty rad!

In Node.js, this is referred to as the Event Loop. In fact, Node.js has ways of manipulating the order in which callbacks are processed with `process.nextTick` which can either prioritize or significantly slow down your app depending on how you use it.

In JavaScript, almost anything that has to pull data into your app or push data out will always be asynchronous because it's not going to be running in the same thread; thus, it's speed of execution cannot be determined. If it's synchronous though, such as rendering to the DOM or dealing with `localStorage`, then yes, your app's performance will suffer.

Just Try and Catch Errors

Another issue is that callbacks do not work with `try - catch`. That's not entirely true, you can use them in your callback functions, but if you wrap a function that accepts a callback in a `try - catch`, it won't catch an error in your callback or the accepting function. This is a huge beginner sand-trap and one you should be aware.

Take this function example:

```
const throwError = () => {
  throw "Who made this function?"
}
```

throwError.js hosted with ❤ by GitHub view raw

It throws an error. You could wrap it in a `try - catch` no problem. Let's put this same function in an async callback:

```
const someAsyncListener = (
  callback,
) => {
  setTimeout(
    callback,
  )
}
```

```
}
catch (error) {
  console
  .log(error)
}

console
.log("I'm alive!")

// I'm alive!
// ERROR: Who made this function?
```

someAsyncListener.js hosted with ❤ by GitHub view raw

To catch an error, you have to move your `try-catch` to the callback function itself:

```
const throwError = () => {
  try {
    throw "Who made this function?"
  }
  catch (error) {
    console
    .log(error)
  }
}
```

throwError.js hosted with ❤ by GitHub view raw

Although, if your callback is synchronous, then you ***can*** catch errors using `try-catch`:

```
const someSyncListener = (
  callback,
) => {
  callback()
}

// This catches
try {
  someSyncListener(
    throwError
  )
}
catch (error) {
  console
```

```
20
21  // Who made this function?
22  // I'm alive!
```

someSyncListener.js hosted with ❤ by GitHub view raw

This is another beginner pitfall, but also one that could trip up even JS vets because it's so simple to screw up. Like I've shown, if you're used to synchronous callbacks, then when you write asynchronous callbacks, you might not realize your old method of using `try` - `catch` just won't work. Suddenly, you've created an anti-pattern.

Conclusion

Compared to any other async handlers, these are the benefits of callbacks:

- Can receive zero or more values over time.
- Can receive zero or more arguments.
- Allows returning nothing or data and errors.
- Somewhat easy to cancel (it depends on the implementation).

These pros allow a lot of versatility, the same versatility that allows all forms of async to be based on callbacks.

On the other hand, most callbacks require some boilerplate like those in Node.js, but since they're inherently functional, you can compose around that boilerplate like so:

```
1   const doTheThing = (
2     ...args
3   ) => {
4     // Do some processing...
5   }
6
7   const handleError = (
8     callback,
9   ) => (
10    error,
11    ...args
12  ) => {
13    error
14    && (
15      console
16      error(error)
17    )
18
19    callback(...args)
20  }
21
22  someListener(
```

In this way, we're first calling `handleError` by passing in our function `doTheThing` as a callback. It returns a new function that listens for `error` and `...args`. When a function returns another function like `handleError`, it's referred to as a closure and allows us to compose `handleError` and `doTheThing` as part of what happens when our `'event'` fires. While never referred to as callback'ception, this allows us to manage boilerplate — in this case handling errors — without having to copy-paste a bunch of code.

Lack of Standards (by design)

It can also be very confusing, as a developer, what kind of callback you're working with. If it's one you made yourself, you need to standardize how it works since in Node.js, the error is the first argument and with DOM event callbacks, they have a specific way they cancel.

Other callbacks, like those found in `Array.prototype.map` pass multiple arguments such as the index and the originating array. You have to be aware those are being passed in, or your `...args` will cause problems.

So while they're everywhere, I would still limit their use. There are only two methods which surpass callbacks: Observables and Generators. Native JS promises are another beast and while convenient, since they're part of the language, they're currently not in a state where they're that much better than callbacks and, in fact, can be worse.

More Reads

I've got more of these async articles coming! The next one is on Promises:

Promises: The Definitive Guide

If you liked what you read, please checkout my other articles on similar eye-opening topics:

- Feature Flags: Be Truly Agile
- Make Poop from Food Emojis
- Async React using React Router & Suspense
- Using Transducers to Speed Up JavaScript Arrays

Thanks to Kiarash Irandoust.

JavaScript Async Asynchronous Callback Callback Hell

WRITTEN BY

Kevin Ghadyani

Always Living in the Future

Follow

ITNEXT is a platform for IT developers & software engineers to share knowledge, connect, collaborate, learn and experience next-gen technologies.

See responses (8)

More From Medium

More from ITNEXT

More from ITNEXT

More from ITNEXT

The Top 10 Most Common Mistakes I've Seen in Go Projects

Teiva... 1.8K

Test Driven Development Is Dumb. Fight Me.

Mike... Jul 18 ... 543

JavaScript Fundamentals: Mastering Classes

Timot... Jul 17 ·... 149