Callback Hell

A guide to writing asynchronous JavaScript programs

What is "*callback hell*"?

Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this:

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

See the pyramid shape and all the `})` at the end? Eek! This is affectionately known as **callback hell**.

The cause of callback hell is when people try to write JavaScript in a way where execution happens visually from top to bottom. Lots of people make this mistake! In other languages like C, Ruby or Python there is the expectation that whatever happens on line 1 will finish before the code on line 2 starts running and so on down the file. As you will learn, JavaScript is different.

What are callbacks?

Callbacks are just the name of a convention for using JavaScript functions. There isn't a special thing called a 'callback' in the JavaScript language, it's just a convention. Instead of immediately returning some result like most functions, functions that use callbacks take some time to produce a result. The word 'asynchronous', aka 'async' just means 'takes some time' or 'happens in the future, not right now'. Usually callbacks are only used when doing I/O, e.g. downloading things, reading files, talking to databases, etc.

When you call a normal function you can use its return value:

```
var result = multiplyTwoNumbers(5, 10)
console.log(result)
// 50 gets printed out
```

However, functions that are async and use callbacks don't return anything right away.

```
var photo = downloadPhoto('http://coolcats.com/cat.gif')
// photo is 'undefined'!
```

In this case the gif might take a very long time to download, and you don't want your program to pause (aka 'block') while waiting for the download to finish.

Instead, you store the code that should run after the download is complete in a function. This is the callback! You give it to the `downloadPhoto` function and it will run your callback (e.g. 'call you back later') when the download is complete, and pass in the photo (or an error if something went wrong).

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error)
  else console.log('Download finished', photo)
}

console.log('Download started')
```

The biggest hurdle people have when trying to understand callbacks is understanding the order that things execute as a program runs. In this example three major things happen. First the `handlePhoto` function is declared, then the `downloadPhoto` function is invoked and passed the `handlePhoto` as its callback, and finally `'Download started'` is printed out.

Note that the `handlePhoto` is not invoked yet, it is just created and passed as a callback into `downloadPhoto`. But it won't run until `downloadPhoto` finishes doing its task, which could take a long time depending on how fast the Internet connection is.

This example is meant to illustrate two important concepts:

- The `handlePhoto` callback is just a way to store some things to do at a later time
- The order in which things happen does not read top-to-bottom, it jumps around based on when things complete

How do I fix callback hell?

Callback hell is caused by poor coding practices. Luckily writing better code isn't that hard!

You only need to follow **three rules**:

1. Keep your code shallow

Here is some messy browser JavaScript that uses **browser-request** to make an AJAX request to a server:

```
var form = document.querySelector('form')
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

This code has two anonymous functions. Let's give em names!

```
var form = document.querySelector('form')
form.onsubmit = function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function postResponse (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

As you can see naming functions is super easy and has some immediate benefits:

- makes code easier to read thanks to the descriptive function names

- when exceptions happen you will get stacktraces that reference actual function names instead of "anonymous"
- allows you to move the functions and reference them by their names

Now we can move the functions to the top level of our program:

```
document.querySelector('form').onsubmit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

Note that the `function` declarations here are defined at the bottom of the file. This is thanks to **function hoisting**.

2. Modularize

This is the most important part: **Anyone is capable of creating modules** (aka libraries). To quote **Isaac Schlueter** (of the node.js project): *"Write small modules that each do one thing, and assemble them into other modules that do a bigger thing. You can't get into callback hell if you don't go there."*

Let's take out the boilerplate code from above and turn it into a module by splitting it up into a couple of files. I'll show a module pattern that works for either browser code or server code (or code that works in both):

Here is a new file called `formuploader.js` that contains our two functions from before:

```
module.exports.submit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

The `module.exports` bit is an example of the node.js module system which works in node, Electron and the browser using **browserify**. I quite like this style of modules because it works everywhere, is very simple to understand and doesn't require complex configuration files or scripts.

Now that we have `formuploader.js` (and it is loaded in the page as a script tag after being browserified) we just need to require it and use it! Here is how our application specific code looks now:

```
var formUploader = require('formuploader')
document.querySelector('form').onsubmit = formUploader.submit
```

Now our application is only two lines of code and has the following benefits:

- easier for new developers to understand -- they won't get bogged down by having to read through all of the `formuploader` functions
- `formuploader` can get used in other places without duplicating code and can easily be shared on github or npm

3. Handle every single error

There are different types of errors: syntax errors caused by the programmer (usually caught when you try to first run the program), runtime errors caused by the programmer (the code ran but had a bug that caused something to mess up), platform errors caused by things like invalid file permissions, hard drive failure, no network connection etc. This section is only meant to address this last class of errors.

The first two rules are primarily about making your code readable, but this one is about making your code stable. When dealing with callbacks you are by definition dealing with tasks that get dispatched, go off and do something in the background, and then complete successfully or abort due to failure. Any experienced developer will tell you that you can never know when these errors happen, so you have to plan on them always happening.

With callbacks the most popular way to handle errors is the Node.js style where the first argument to the callback is always reserved for an error.

```
var fs = require('fs')

fs.readFile('/Does/not/exist', handleFile)

function handleFile (error, file) {
  if (error) return console.error('Uhoh, there was an error', error)
  // otherwise, continue on and use `file` in your code
}
```

Having the first argument be the `error` is a simple convention that encourages you to remember to handle your errors. If it was the second argument you could write code like `function handleFile (file) { }` and more easily ignore the error.

Code linters can also be configured to help you remember to handle callback errors. The simplest one to use is called **standard**. All you have to do is run `$ standard` in your code folder and it will show you every callback in your code with an unhandled error.

Summary

1. Don't nest functions. Give them names and place them at the top level of your program
2. Use function hoisting to your advantage to move functions 'below the fold'
3. Handle **every single error** in every one of your callbacks. Use a linter like standard to help you with this.
4. Create reusable functions and place them in a module to reduce the cognitive load required to understand your code. Splitting your code into small pieces like this also helps you handle errors, write tests, forces you to create a stable and documented public API for your code, and helps with refactoring.

The most important aspect of avoiding callback hell is **moving functions out of the way** so that the programs flow can be more easily understood without newcomers having to wade through all the detail of the functions to get to the meat of what the program is trying to do.

You can start by moving the functions to the bottom of the file, then graduate to moving them into another file that you load in using a relative require like `require('./photo-helpers.js')` and then finally move them into a standalone module like `require('image-resize')`.

Here are some rules of thumb when creating a module:

- Start by moving repeatedly used code into a function
- When your function (or a group of functions related to the same theme) get big enough, move them into another file and expose them using `module.exports`. You can load this using a relative require
- If you have some code that can be used across multiple projects give it it's own readme, tests and `package.json` and publish it to github and npm. There are too many awesome benefits to this specific approach to list here!
- A good module is small and focuses on one problem
- Individual files in a module should not be longer than around 150 lines of JavaScript
- A module shouldn't have more than one level of nested folders full of JavaScript files. If it does, it is probably doing too many things

- Ask more experienced coders you know to show you examples of good modules until you have a good idea of what they look like. If it takes more than a few minutes to understand what is happening, it probably isn't a very good module.

More reading

Try reading my **longer introduction to callbacks**, or try out some of the **nodeschool** tutorials.

Also check out the **browserify-handbook** for examples of writing modular code.

What about promises/generators/ES6 etc?

Before looking at more advanced solutions, remember that callbacks are a fundamental part of JavaScript (since they are just functions) and you should learn how to read and write them before moving on to more advanced language features, since they all depend on an understanding of callbacks. If you can't yet write maintainable callback code, keep working at it!

If you *really* want your async code to read top-to-bottom, there are some fancy things you can try. Note that **these may introduce performance and/or cross platform runtime compatibility issues**, so make sure to do your research.

Promises are a way to write async code that still appears as though it is executing in a top-down way, and handles more types of errors due to encouraged use of `try/catch` style error handling.

Generators let you 'pause' individual functions without pausing the state of the whole program, which at the cost of slightly more complex to understand code lets your async code appear to execute in a top-down fashion. Check out **watt** for an example of this approach.

Async functions are a proposed ES7 feature that will further wrap generators and promises in a higher level syntax. Check them out if that sounds interesting to you.

Personally I use callbacks for 90% of the async code I write and when things get complicated I bring in something like **run-parallel** or **run-series**. I don't think callbacks vs promises vs whatever else really make a difference for me, the biggest impact comes from keeping code simple, not nested and split up into small modules.

Regardless of the method you choose, always **handle every error** and **keep your code simple**.

Remember, only *you* can prevent callback hell and forest fires

You can find the source for this **on github**.