Callback Hell is a Myth

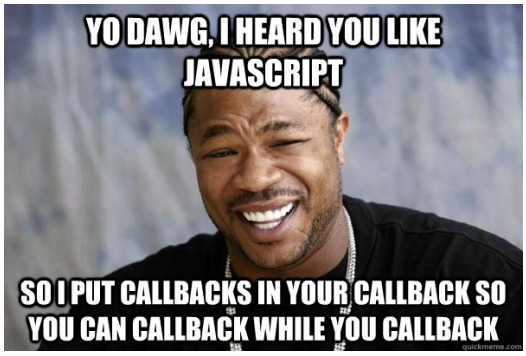
by Valeri Karpov
@code_barbarian (http://www.twitter.com/code_barbarian)
March 20, 2015

Hating "callback hell" is a favorite pastime for many JavaScript bloggers and open source devs. There are numerous modules out there, like async (http://npmjs.org/package/async), zone (http://npmjs.org/package/zone), and the various promise libraries, that promise to save you from the callback monster that's coming to eat your lunch. Even I've been experimenting with my own async framework (https://www.npmjs.com/package/wagner-core). Yet, in my experience, true callback hell is a symptom rather than a problem, and, unless you address the underlying problem, using a library like async will just postpone the inevitable.

What's the Underlying Problem?

When I first started using the async module (http://npmjs.org/package/async), it felt like a dream come true. The `waterfall()` function (https://www.npmjs.com/package/async#waterfall) became my new best friend. But, as the codebase grew and others started contributing, the code started becoming more and more tangled. Pretty soon I had `async.series()` and `async.parallel()` calls inside calls to `waterfall()` and the code was even more unmaintainable than before I started using async. But at least I didn't have callback hell, right?

Not exactly. My problem was not that I had nested callbacks, my problem was that my functions were doing too much (http://misko.hevery.com/code-reviewers-guide/flaw-class-does-too-much/). Async helped me sweep this problem under the rug for a time, but the poor code organization ended up coming back to bite me. Too often async and promises are used as a panacea to save codebases from bloated code and bad software design.



Why I Love Callbacks

People are always confused when I say I love NodeJS in spite of the fact that I agree (at least in spirit, if not the exact number) with Linus Torvalds' quip (http://en.wikiquote.org/wiki/Linus_Torvalds#1995-99) "if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program." Part of the reason why I loved NodeJS from the beginning was that callbacks are just enough of a pain to write.

Much like how the pain of your hands burning means you mistakenly picked up a hot pot in the kitchen, or a headache in the morning means you drank too much last night, the pain of dealing with 10 layers of nested callbacks in a single function means you desperately need to refactor. The small friction introduced by writing a callback is just enough to make you twice about "is this extra I/O really necessary, or can I design this in a better way?"" I still prefer to use a framework like async or promises to make code more readable. But, if you're counting on promises or `async.waterfall()` to save you from banana code, the problem is probably more with your software design than with your asynchronous programming framework.

How To Use Callbacks Badly

Let's take a look at a couple "pyramid of doom" examples I pulled down from Google images. Here's a typical callback hell example borrowed from a post on medium.com (https://medium.com/@mlfryman/to-survive-the-pyramid-of-doom-4e8ce4fb5d6b).

```
    describe('.totalValue', function(){
      it('should calculate the total value of items in a space', function(done){
      var table = new Item('table', 'dining room','07/23/2014', '1','3000');
      var chair = new Item('chair', 'living room','07/23/2014', '3','300');
      var couch = new Item('couch', 'living room','07/23/2014', '2','1100');
      var chair2 = new Item('chair', 'dining room','07/23/2014', '4','500');
      var bed = new Item('bed', 'bed room','07/23/2014', '1','2000');

      table.save(function(){
        chair.save(function(){
          couch.save(function(){
            chair2.save(function(){
              bed.save(function(){
                Item.totalValue({room: 'dining room'}, function(totalValue){
                  expect(totalValue).to.equal(5000);
                  done();
                  });
                });
              });
            });
          });
        });
      });
    });
  });
});
```

If you ignore this code's numerous other glaring issues, you'll see that callbacks aren't the problem here, the lack of sane abstractions is. Look carefully: does chair really need to be saved after table is done saving? Furthermore, there are programming constructs that enable you to not have to list out every Item explicitly; they're called `for` loops. If you're using mongoose (http://npmjs.org/package/mongoose), you would just use the `create()` function (http://mongoosejs.com/docs/api.html#model_Model.create). Even without mongoose, `async.parallel()`, or `promise.all()`, you can write a simple function to make this much cleaner.

```
function create(items, callback) {
  var numItems = items.length;
  var done = false;
  for (var i = 0; i < items.length; ++i) {
    items[i].save(function(error) {
      if (done) {
        return;
      }
      if (error) {
        done = true;
        return callback(error);
      }
      --numItems || callback();
    });
  }
}
```

An alternative solution using simple recursion makes for a more concise solution (12 SLOC, 4 branches) that maintains the behavior of the original code. Just another case where young developers should practice algorithms more (https://sites.google.com/site/codetrick/superprimerib) and spend less time building iPhone apps.

```
function create(items, callback, index) {
  index = index || 0;
  if (index >= items.length) {
    callback();
  }
  items[i].save(function(error) {
    if (error) {
      return callback(error);
    }
    create(items, callback, index + 1);
  });
}
```

Another example I pulled down that's a bit less trivial comes from this presentation on wrangling callback hell with async (http://slides.com/michaelholroyd/asyncnodejs#/). Click on the image to zoom in a little more, if you dare.

```
// get all the original jpegs, and create the edited jpegs.
function reserveWork()
{
  beanstalkd.reserve(function(err, jobid, payload){ reportError(err);
    beanstalkd.bury(jobid, 1024, function(err){ reportError(err);
      var spin = JSON.parse(payload.toString());
      console.dir(spin);
      spin.shortid = spin.short_id;
      var s3key = spin.shortid+"/spin.zip";

      console.log(("["+spin.shortid+"] STARTED (beanjob #"+jobid+")"));


      s3.getObject({Bucket: s3bucket, Key: s3key}, function(err, data) { if(err) console.error("Could not get "+s3key); reportError(err);
        fs.mkdirs(path.dirname(s3key),function(err){ reportError(err);
          fs.writeFile(s3key,data.Body,function(err){ reportError(err);
            // spin.zip is on the file system now
            var cmd = "unzip -o "+s3key+" -d "+path.dirname(s3key);
            exec(cmd,function(){
              console.log("Stuff is unzipped!");

              fs.mkdirs(path.dirname(s3key)+"/orig",function(){
                var vfs = ["null"];
                var rots = [null, "transpose=2", "transpose=2,transpose=2", "transpose=2,transpose=2,transpose=2"];
                var rotidx = parseInt(spin.rotation_angle,10)/90;
                if(rotidx) vfs.push(rots[rotidx]);
                var vf = "-vf "+vfs.join(", ");
                var ffmpeg_cmd = "ffmpeg -i "+path.dirname(s3key)+"/cap.mp4 -q:v 1 "+vf+" -pix_fmt yuv420p "+path.dirname(s3key)+"/orig/%03d.jpg";
                exec(ffmpeg_cmd,function(){
                  console.log("Done with ffmpeg");
                  // Upload everything to S3
                  Step( function(){
                    for (var i=1; i<=spin.frame_count; i++)
                    {
                      var s3key = spin.shortid + "/orig/" + ("00"+i).substr(-3) + ".jpg";
                      uploadOrig(s3key, this.parallel());
                    }
                  },
                  function(){
                    fs.readFile(spin.shortid+"/labels.txt",function(err,data){
                      if(err || !data)
                        data = new Buffer("{}");
                      s3.putObject({Bucket: s3bucket, Key: spin.shortid+"/labels.json", ACL: "public-read", ContentType: "text/plain", Body: data}, function(err, data){ reportError(err);
                        console.log("All files are uploaded");
                        beanstalkd.use("editor",function(err,tube){ reportError(err,jobid);
                          beanstalkd.put(1024,0,300,JSON.stringify(spin), function(err,new_jobid){ reportError(err,jobid);
                            console.log("Added new job to beanstalkd.");
                            beanstalkd.destroy(jobid, function(){
                              console.log(("["+spin.shortid+"] FINISHED (beanjob #"+jobid+")"));
                              reserveWork();
                            });
                          });
                        });
                      });
                    });
                  });
                });
              });
            });
          });
        });
      });

    });
  }
```

(http://i.imgur.com/EGGwaXP.png)

This case is a classic example where the single function is responsible for doing way too much, otherwise known as the God object anti-pattern (http://en.wikipedia.org/wiki/God_object). As written, this function does a lot of tangentially related tasks:

- registers a job in a work queue
- cleans up the job queue
- reads and writes from S3 with hard-coded options
- executes two shell commands
- etc. etc.

The function does way too much and has way too many points of failure. Furthermore, it skips error checks for many of these. Promises and async have mechanisms to help you check for errors in a more concise way, but odds are, if you're the type of person who ignores errors in callbacks, you'll also ignore them even if you're using `promise.catch()` or `async.waterfall()`. Callback hell is the least of this function's problems.

Even if you wrote it in Python or Ruby, this function is a nightmare to test and debug. How would you improve this function to make it more durable and easier to test, debug, and understand?

- First you'd start by abstracting out the code that manages `beanstalkd` - the code that inserts tasks into a job queue has no business being in the same function as your business logic. Separation of concerns! The maximum indentation in the code as written is 30 (15x2). Creating a separate function that handles everything from L62-92 and making beanstalk call that function reduces maximum indentation to 20 (10x2).
- Adding an abstraction for "download this object in S3, unzip it, and put it into this directory, and create this directory if it doesn't exist" will reduce the maximum indentation to 14 (7x2). Since `mkdirs` creates directories recursively, you can simply save yourself the extra fs call and tell your S3 download code to create the `/orig` directory and bring it down to 12 (6x2).
- Finally, you can create a function that handles "read this file and upload it to S3" and reduce your maximum indentation to 10 (5x2).

By breaking this function into a few helper functions behind sane abstractions, you've tamed the callback hell. In other words, this callback hell example is a strawman - there's no excuse for a professional software engineer to write production server code like this.

Conclusion

Callbacks and callback hell get a bad rap from the NodeJS community. However, in practice, callback hell often ends up being helpful. It's JavaScript's built-in code reviewer that reminds you to break your code up into small, focused, reusable modules instead of giant monolithic blobs that will make your life miserable. It's the good angel on your shoulder (http://en.wikipedia.org/wiki/Shoulder_angel) reminding you to stop ignoring I/O errors, because, yes, they do happen. Next time, before you reach for `async.waterfall()`, take a second to think about whether you need to refactor first.

Found a typo or error? Open up a pull request! This post is available as markdown on Github (https://github.com/vkarpov15/thecodebarbarian.com/blob/master/lib/posts/20150320_callback_hell.md)

Homer Simpson • 3 years ago

' The small friction introduced
by writing a callback is just enough to make you (sic) twice about "is this extra I/O
really necessary, or can I design this in a better way?"" '

Or enough to make you stop using Node and use a better server-side language, like Golang.

1 • Reply • Share ›

Anders ↗ Homer Simpson • 3 years ago

Wouldn't mind a couple of reasons why Golang is always a better option. From what I read about Node vs Golang, what is best depends on your needs.

• Reply • Share ›

Homer Simpson ↗ Anders • 3 years ago

Go has better tooling, builtin concurrency primitives, performs well at scale, and compiles quickly and easily to a lot of different platforms and architectures.

In what case would someone use NodeJS over Golang?

1 • Reply • Share ›

vkarpov15 Mod ↗ Anders • 3 years ago

Golang is almost always the wrong choice, believe me, I learned that the hard way. Go is still several years away from being usable in a modern development paradigm. If you're really eager for cross-compilation and syntactic sugar on top of threads, just use Rust.

1 • Reply • Share ›

Homer Simpson ↗ vkarpov15 • 3 years ago

Rust hit 1.0 this year. Golang has been used for several years in a "modern development paradigm" by companies you might have heard of...like Google...come on bruh, Docker is written in Golang...

2 • Reply • Share ›

dan • 2 years ago

thank you for this post, in my experience this is something more people should be talking about.

maybe promises etc over callbacks are good for cleaning up what are essentially simple linear scripts? ironically in those cases, simple blocking, synchronous APIs or calls are probably more appropriate if you can

ironically in those cases, simple blocking, synchronous API is or calls are probably more appropriate if you can use them...

the outcome within non-trivial programs we need to maintain - e.g. production server code as you say - we may actually be using promises to tolerate and then suffer from less modularity, less meaning abstractions and more inappropriate coupling in our code.

• Reply • Share ›

vkarpov15 Mod ➚ dan • 2 years ago

Except for its every bit as easy to make broken unmaintainable spaghetti code using promises as it is with callbacks. For example, http://stackoverflow.com/qu... . People also write unmaintainable spaghetti in imperative languages, with no promises or callbacks.

Nowadays I use co or async/await a lot, and it definitely helps make code easier to read, but the point of this article is that often the reason why callbacks lead to bad code is because of bad design.

1 • Reply • Share ›

Homer Simpson • 3 years ago

Looks like the creator of Node is now a Golang programmer.

https://github.com/ry?tab=r...

• Reply • Share ›

Kyle Palko ➚ Homer Simpson • 2 years ago

salt

• Reply • Share ›

hellboy • 3 years ago

Is there some heuristcs, in which cases can I do refactoring to async? >=3 callbacks?

• Reply • Share ›

vkarpov15 Mod ➚ hellboy • 3 years ago

My heuristic is I should be able to describe what a function does in one sentence without using the word "and." Typically that ends up being 2-4 callbacks at most.

1 • Reply • Share ›