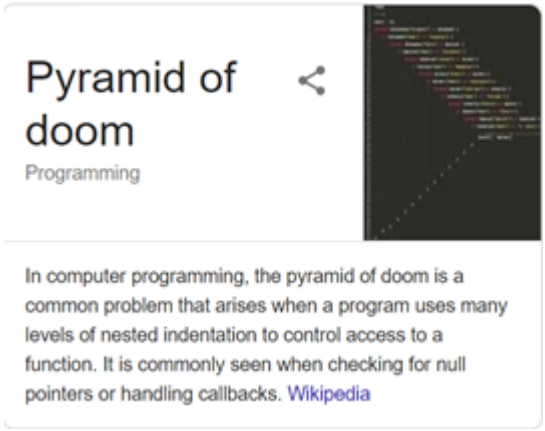
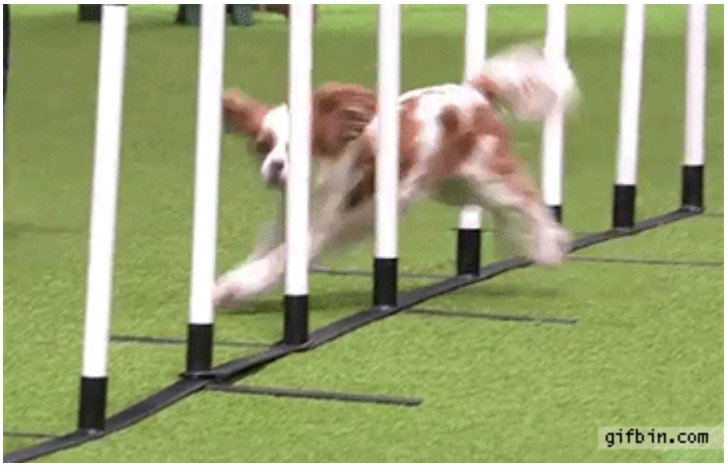
CALLBACK HELL

Callback Hell, also known as Pyramid of Doom, is an anti-pattern seen in code of asynchronous programming.

It is a slang term used to describe and unwieldy number of nested "if" statements or functions.



If you are not expecting your application logic to get too complex, a few callbacks seem harmless. But once your project requirements start to swell, you will quickly find yourself piling layers of nested callbacks. Congrats! Welcome to Callback Hell.



Callback Hell Node Js – JavaScript Callback

A Callback is a function “A” that is passed to another function “B” as a parameter. The function “B” executes the code “A” at some point. The invocation of “A” can be immediate, as in a synchronous callback, or, it can occur later as in an asynchronous callback. It’s actually a simple concept that will be well understood with an example:

```
var fs = require('fs');
fs.readFile('test.json', function(err, results) {
    if (err) {
        console.log(err);
    }
    console.log(JSON.parse(results).name);
});
```

In the code we can see how to make a call to **readFile** and we pass it as a second parameter function (Callback Hell). In this case, **readFile** is an asynchronous operation and when it’s done with the operation of reading the file, it will execute the callback by passing the results of the operation to parameters.

The use of callbacks makes the code difficult to write and maintain. It also increases the difficulty of identifying the flow of the application, which is an obstacle when it comes to making debug, hence the famous name to this problem: **Callback Hell.**

Q: What's worse than callback hell?

A: Not fixing it.

So it is definitely recommended to do it right from the get-go and avoid deeply-nested callbacks. My favourite solution for this will be the usage of the **Promise** object. I have been dealing with Node.js for my last few projects and **Promise** managed to keep my sanity in check. But if you are looking for something more *edgy*, you will love **Generators**. Another elegant approach to get rid call back hell, is to use async.waterfall. I will touch more in depth about all the approaches below.

JavaScript promises hell

A promise is the future result of an asynchronous operation and is represented as a JavaScript object with a public interface. If we apply the code above, we would be left with the following code:

```
var Promise = require('bluebird');
var fs = require('fs');
Promise.promisifyAll(fs);

fs.readFileAsync('test.json')   .then(function(results) {
    console.log(JSON.parse(results).name);
})   .catch(function(err) {
    console.log(err);
});
```

While it is true that by using promises the code is more readable, we still end up having a problem; we have an application that is difficult to debug and maintain. Again, it is very easy to lose track of what is happening in the application, since we are talking about "future" results. Therefore, we will first have to convert all these APIs based on callbacks to Promises. That is when the coroutines (Fibers) are quite helpful.

Generators

Generators, like Promise, is also one of the features in ES6 that can manage asynchronous programming. The great thing about Generators is that it can work hand-in-hand with Promise to bring us closer to synchronous programming.

```
/* Using the same Serial Flow example */
Q.spawn(function* (){
    try {
        if(yield checkUrlPromise(url)){
            var str = yield lowerCasePromise(url);
            var link = yield generateLinkPromise(name,str);
            console.log("Solve with Generators: ",link);
        } else {
            console.log("Invalid Url")
        }
    }

    catch(e){
        console.log(e);
    }
});
```

Pro-tip: If you

*cannot see how this might look synchronous, strip away the function *() and yield.*
From the above example, we are coding in a synchronous manner with Generators- using the try and catch block and writing the code as if the result is returned immediately. There are also no verbosity with the then handler.

Run-to-completion

Javascript function are expected to run-to-completion - This means once the function starts running, it will run to the end of the function. However, Generators allow us to interrupt this execution and switch to other tasks before returning back to the last interrupted task.

Async Waterfall

Async Waterfall is amazing simple and powerful technique to get out of callback hell. On top of that it makes code readable, easy to understand and even easy to maintain and debug. Let's take an example of reading a json file.

Identify the steps

First divide the code into simple asynchronous step functions that need to be executed to perform a given task. In the example of reading json file, the steps could be reading the json file and processing the file once read. Remember each step function takes a callback as argument. The first parameter to callback is error object. If error object is not null the waterfall stops further processing and error handler is called with error object. which takes parameters to next step function as argument along with arguments required.

Read the file:

```
function readFile(readFileCallback) {
    fs.readFile('test.json', function (error, file) {
        if (error) {
            readFileCallback(error);
        } else {
            readFileCallback(null, file);
        }
    });
}
```

Process the file:

```
function processFile(file, processFileCallback) {
    var json = JSON.parse(file);
    if (json['data'] != null) {
        processData(json['data'], function (error) {
            if (err) {
                console.error(error);
                processFileCallback(error);
            } else {
                console.log("Processed successfully");
                processFileCallback(null);
            }
        });
    }
    else {
        console.log("No data to process");
        processFileCallback(null); //callback should
        always be called once (and only one time)
    }
}
```

Note that I did no specific error handling here, I'll take benefit of async.waterfall to centralize error handling at the same place.

Also be careful that if you have (if/else/switch/...) branches in an asynchronous function, it always call the callback one (and only one) time.

Plug everything with async.waterfall

```
async.waterfall([
    readFile,
    processFile
], function (error) {
    if (error) {
        //handle readFile error or processFile error here
    }
});
```

Note how easy to understand and maintain the code is.

The Top 10 Most Common Mistakes That Node.js Developers Make

Node.js is a free, open source runtime environment for executing JavaScript code that runs on various platforms. It is used for highly scalable, data-intensive and real time apps due to its non-blocking/asynchronous nature.

Like any other platform, Node.js is also vulnerable to developer problems and issues. Some of these mistakes degrade performance, while others make Node.js appear straight out unusable for whatever you are trying to achieve.

- **Mistake #1: Blocking the event loop (or the Worker Pool)**
  Node.js runs JavaScript code in the Event Loop (initialization and callbacks), and offers a Worker Pool to handle expensive tasks like file I/O. Nodejs application runs in single threaded environment. To achieve concurrency, I/O bound operations needs to be handled/run asynchronously.

- **Mistake #2: Invoking a Call-back More Than Once**
  One common node.js issue related to using callbacks is calling them more than once. Callback doesn't automatically end the execution of the current function until it is last statement, so we need to have return statement if we want to return post callback.
  Example:

```
module.export.verifyPassword = function(user, password, callback){
if(typeof password !== 'string') {
done(new Error('password should be a string'))
return
}computeHash(password, user.passwordHashOpts, function(err, hash) {
if(err) {
done(err)
return
}
done(null, hash === user.passwordHash)
})
}
```

  If the first "return" was commented out, passing a non-string password `to this function will still result in "computeHash" being called. Depending on how "computeHash" deals with such a scenario, "done" may be called multiple times, which result in unknown issues.
- **Mistake #3: Deeply Nesting Callbacks ("callback hell")**
  Callback hell is a phenomenon that afflicts a JavaScript developer when he tries to execute multiple asynchronous operations one after the other

```
doSomething(param1, param2, function(err, paramx){
    doMore(paramx, function(err, result){
        insertRow(result, function(err){
            yetAnotherOperation(someparameter, function(s){
                somethingElse(function(x){
                });
            });
        });
    });
});
```

  By nesting callbacks in such a way, we easily end up with error-prone, hard to read, and hard to maintain code.Soln: Best code practice to handle it

    - Keep your code shallow
    - Give your functions names
    - Modularize
    - Handle every single error
    - Declare your functions beforehand

  Techniques to fix callback hell

    - Using Async.js
    - Using Promises
    - Using Async-Await
    - Coroutine (Promise + Generator)

- **Mistake #4: Expecting Callbacks to Run Synchronously**In Nodejs, Callback functions does not always run synchronously.With callbacks a particular function may not run well until the task it is waiting on is finished. The execution of the current function will run until the end without any stop:
```
function testTimeout() {
        console.log("Begin")
        setTimeout(function() {
                console.log("Done!")
        }, duration * 1000)
        console.log("Waiting..")
}
```
Calling the "testTimeout" function will first print "Begin", then print "Waiting.." followed by the the message "Done!" after about a second.
- **Mistake #5: Assigning to "exports", Instead of "module.exports"**
In Nodejs, there is difference between "module.exports" and "exports", which developers think are same.Node.js treats each file as a small isolated module. If your package has two files, perhaps "a.js" and "b.js", then for "b.js" to access "a.js"'s functionality, "a.js" must export it by adding properties to the exports object:
```
// a.js
exports.verifyPassword = function(user, password, done) { ... }
```
When this is done, anyone requiring "a.js" will be given an object with the property function "verifyPassword":
```
// b.js
require('a.js') // { verifyPassword: function(user, password, done) { ... } }
```
However, what if we want to export this function directly, and not as the property of some object? We can overwrite exports to do this, but we must not treat it as a global variable then:
```
// a.js
module.exports = function(user, password, done) { ... }
```
Notice how we are treating "exports" as a property of the module object
- **Mistake #6: Throwing Errors from Inside Callbacks**
Handling exceptions (try-catch) does not behave as you might expect it to in asynchronous situations. For example, if you wanted to protect a large chunk of code with lots of asynchronous activity with one big try-catch block, it wouldn't necessarily work:

```
try {
        db.User.get(userId, function(err, user) {
                if(err) {
                        throw err
                }
                // ...
                usernameSlug = slugifyUsername(user.username)
                // ...
        })
} catch(e) {
        console.log('Oh no!')
}
```

If the callback to "db.User.get" fired asynchronously, the scope containing the try-catch block would have long gone out of context for it to still be able to catch those errors thrown from inside the callback.

- **Mistake #7: Assuming Number to Be an Integer Datatype**
  Numbers in JavaScript are floating points - there is no integer data type. This will be problem if we try to access large integer value.The following evaluates to true in Node.js:

  ```
  Math.pow(2, 53)+1 === Math.pow(2, 53)
  ```

  Soln: Use big integer libraries that implement the important mathematical operations on large precision numbers, such as node-bigint.
- **Mistake #8: Ignoring the Advantages of Streaming APIs**
  Streaming apis like using pipe etc can make nodejs application muct more performanent and easy to handle.
- **Mistake #9: Using Console.log for Debugging Purposes**
  In Node.js, "console.log" allows you to print almost anything to the console. However, it is strongly recommended that you avoid "console.log" in real code. Instead, use one of the amazing libraries that are built just for this, such as debug (provide convenient ways of enabling and disabling certain debug lines when you start the application)
- **Mistake #10: Not Using Supervisor Programs**
  Nodejs application should use power of Supervisor programs like pm2, forever, nodemon etc. They give plenty of benefits like application will fail fast i.e. If an unexpected error occurs, do not try to handle it, rather let your program crash and have a supervisor restart it in a few seconds and many more.

References:

- V8/Event Loop/call stack – explained very well
  https://www.youtube.com/watch?v=8aGhZQkoFbQ
- https://www.toptal.com/nodejs/top-10-common-nodejs-developer-mistakes