INTRODUCTION TO NODE.JS

Notes And Remarks

MARCH 15, 2022

BERKTUG KAAN OZKAN
spaceymonk.github.io

Contents

Introduction to NodeJs

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

```
const http = require('http')

const hostname = '127.0.0.1'
const port = 3000

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World\n')
})

server.listen(port, hostname, () => {
  console.log(`Server running at
http://${hostname}:${port}/`)
})
```

Difference Between NodeJS and Browser

Browser	NodeJs
In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies.	all the nice APIs that Node.js provides through its modules, like the filesystem access functionality.
Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very convenient.	. Unless you are building an open source application that anyone can deploy anywhere, you know which version of Node.js you will run the application on.
in the browser we are starting to see the ES Modules standard being implemented. this means that `import` in the browser.	Node.js uses the CommonJS module system. this means that for the time being you use `require()` in Node.js

Exiting Programmatically

Solution #1

The `process` core module provides a handy method that allows you to programmatically exit from a Node.js program: `process.exit()`. When Node.js runs this line, the process is immediately forced to terminate. This means that any callback that's pending, any network request still being sent, any filesystem access, or processes writing to `stdout` or `stderr` - all is going to be ungracefully terminated right away. If this is fine for you, you can pass an integer that signals the operating system the exit code.

Solution #2

you need to send the command a SIGTERM signal, and handle that with the process signal handler:

```
process.on('SIGTERM', () => {
  server.close(() => {
    console.log('Process terminated')
  })
})
```

SIGKILL is the signal that tells a process to immediately terminate, and would ideally act like process.exit().

SIGTERM is the signal that tells a process to gracefully terminate. It is the signal that's sent from process managers like upstart or supervisord and many others.

You can send this signal from inside the program, in another function, or from another Node.js running program, or any other app running in your system that knows the PID of the process you want to terminate.:

```
process.kill(process.pid, 'SIGTERM')
```

Exit Codes

- 1 **Uncaught Fatal Exception**: There was an uncaught exception, and it was not handled by a domain or an 'uncaughtException' event handler.
- 2: Unused (reserved by Bash for builtin misuse)
- 3 **Internal JavaScript Parse Error**: The JavaScript source code internal in the Node.js bootstrapping process caused a parse error. This is extremely rare, and generally can only happen during development of Node.js itself.
- 4 **Internal JavaScript Evaluation Failure**: The JavaScript source code internal in the Node.js bootstrapping process failed to return a function value when evaluated. This is extremely rare, and generally can only happen during development of Node.js itself.
- 5 **Fatal Error**: There was a fatal unrecoverable error in V8. Typically a message will be printed to stderr with the prefix FATAL ERROR.
- 6 **Non-function Internal Exception Handler**: There was an uncaught exception, but the internal fatal exception handler function was somehow set to a non-function, and could not be called.
- 7 **Internal Exception Handler Run-Time Failure**: There was an uncaught exception, and the internal fatal exception handler function itself threw an error while attempting to handle it. This can happen, for example, if an 'uncaughtException' or domain.on('error') handler throws an error.
- 8: Unused. In previous versions of Node.js, exit code 8 sometimes indicated an uncaught exception.
- 9 **Invalid Argument**: Either an unknown option was specified, or an option requiring a value was provided without a value.

- 10 **Internal JavaScript Run-Time Failure**: The JavaScript source code internal in the Node.js bootstrapping process threw an error when the bootstrapping function was called. This is extremely rare, and generally can only happen during development of Node.js itself.
- 12 **Invalid Debug Argument**: The --inspect and/or --inspect-brk options were set, but the port number chosen was invalid or unavailable.
- 13 **Unfinished Top-Level Await**: await was used outside of a function in the top-level code, but the passed Promise never resolved.
- >128 **Signal Exits**: If Node.js receives a fatal signal such as SIGKILL or SIGHUP, then its exit code will be 128 plus the value of the signal code. This is a standard POSIX practice, since exit codes are defined to be 7-bit integers, and signal exits set the high-order bit, and then contain the value of the signal code. For example, signal SIGABRT has value 6, so the expected exit code will be 128 + 6, or 134.

Reading Environment Variables

The process core module of Node.js provides the env property which hosts all the environment variables that were set at the moment the process was started.

```
process.env.USER_ID // "239482"
process.env.USER_KEY // "foobar"
```

If you have multiple environment variables in your node project, you can also create an “`.env`” file in the root directory of your project, and then use the `dotenv` package to load them during runtime.

```
require('dotenv').config();

process.env.USER_ID // "239482"
process.env.USER_KEY // "foobar"
process.env.NODE_ENV // "development"
```

Reading Command Line Arguments

You can pass any number of arguments when invoking a Node.js application. Arguments can be standalone or have a key and a value. This changes how you will retrieve this value in the Node.js code. The way you retrieve it is using the process object built into Node.js.

It exposes an `argv` property, which is an array that contains all the command line invocation arguments:

- The first element is the full path of the node command.
- The second element is the full path of the file being executed.
- All the additional arguments are present from the third position going forward.

If key-value pairs given, `name=joe`, you need to parse it. The best way to do so is by using the `minimist` library, which helps dealing with arguments. This time you need to use double dashes before each argument name:

```
node app.js --name=joe

const args = require('minimist')(process.argv.slice(2))
args['name'] //joe
```

Outputting to Command Line

Node.js provides a console module, basically the same as the console object you find in the browser, which provides tons of very useful ways to interact with the command line.

- If you pass an object, it will render it as a string.
- You can pass multiple variables to console.log.
- We can also format pretty phrases by passing variables and a format specifier.
  - `%s` format a variable as a string
  - %d format a variable as a number
  - `%i` format a variable as its integer part only
  - %o format a variable as an object
- `console.clear()` clears the console (the behavior might depend on the console used)
- `console.count()` will count the number of times a string is printed, and print the count next to it.
- The `console.countReset()` method resets counter used with `console.count().`

```
const oranges = ['orange', 'orange']
const apples = ['just one apple']
oranges.forEach(fruit => {
  console.count(fruit)
})
apples.forEach(fruit => {
  console.count(fruit)
})

console.countReset('orange')

oranges.forEach(fruit => {
  console.count(fruit)
})
```

- There might be cases where it's useful to print the call stack trace of a function, maybe to answer the question how did you reach that part of the code? You can do so using `console.trace()`
- You can easily calculate how much time a function takes to run, using `time()` and `timeEnd()`

```
const doSomething = () => console.log('test')
const measureDoingSomething = () => {
  console.time('doSomething()')
  //do something, and measure the time it takes
  doSomething()
  console.timeEnd('doSomething()')
}
measureDoingSomething()
```

- As we saw console.log is great for printing messages in the Console. This is what's called the standard output, or stdout. `console.error` prints to the stderr stream. It will not appear in the console, but it will appear in the error log.
- You can color the output of your text in the console by using escape sequences. An escape sequence is a set of characters that identifies a color. **Chalk** is a library in addition to coloring it also helps with other styling facilities, like making text bold, italic or underlined.
- **Progress** is an awesome package to create a progress bar in the console.

```
const ProgressBar = require('progress')

const bar = new ProgressBar(':bar', { total: 10 })
const timer = setInterval(() => {
  bar.tick()
  if (bar.complete) {
    clearInterval(timer)
  }
}, 100)
```

Inputting from Command Line

`readline` module to perform exactly this: get input from a readable stream such as the `process.stdin` stream, which during the execution of a Node.js program is the terminal input, one line at a time.

```
const readline = require('readline').createInterface({
  input: process.stdin,
  output: process.stdout
})

readline.question(`What's your name?`, name => {
  console.log(`Hi ${name}!`)
  readline.close()
})
```

More detailed packages are:

- `Readline-sync`: If you need to require a password, it's best not to echo it back, but instead show a * symbol.
- `Inquirer.js`: lets you do many things like asking multiple choices, having radio buttons, confirmations, and more.

Expose Functionality Using Exports

Functionality must be exposed before it can be imported by other files. Any other object or variable defined in the file by default is private and not exposed to the outer world. This is what the `module.exports` API offered by the module system allows us to do. When you assign an object or a function as a new exports property, that is the thing that's being exposed, and as such, it can be imported in other parts of your app, or in other apps as well. Can be done in two ways:

Solution #1

```
// car.js
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}
module.exports = car

// index.js
const car = require('./car')
```

Solution #2

```
// car.js
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}
exports.car = car

// index.js
const items = require('./items')
const car = items.car
// or
const { car } = require('./items')
// or
const car = require('./items').car
```

What's the difference between `module.exports` and `exports`?

The first exposes the object it points to. The latter exposes the properties of the object it points to.

There is no magic. Your module code is sandwiched between the two items in this array (actual nodejs source code), and eval'd:

```
NativeModule.wrapper = [
  '(function (exports, require, module, __filename, __dirname) { ',
'\n});'
];
```

The magic variables you can use in modules - `exports`, `require`, `module`, `__filename`, and `__dirname` are not magic, they are just parameters to the function that is invoked when your module is loaded. Initially, `exports` and `module.exports` point at the same empty object. The `module` is a plain JavaScript Object representing the current module. It is local to each module and also it is private. It has `exports` property which is a plain JavaScript variable, set to `module.exports`. At the end of the file, Node.js return `module.exports` to the required function.

Module.exports	Exports

When we want to export a single class/variable/function from one module to another module, we use the module.exports way	When we want to export multiple variables/functions from one module to another, we use exports way.
It is the object reference that gets returned from the require() calls.	The exports is not returned by require()

By adding properties to exports you are effectively ensuring that you are returning a "typical" module export object. In contrast, by using module.exports you can return any value you want (primitive, array, function) and not just an object (which is the format most people expect). So module.exports offers more power but can also be used to have your module export atypical values (like a primitive). In contrast exports is more limiting but safer (so long as you simply add properties to it and don't reassign it).

Introduction to NPM

Installing all dependencies You need to set the --production flag (npm install --production) to avoid installing those development dependencies.	`npm install`

Installing a single package
Since npm 5, this command adds <package-name> to the package.json file dependencies. Before version 5, you needed to add the flag --save.
Often you'll see more flags added to this command:
• --save-dev installs and adds the entry to the package.json file devDependencies
• --no-save installs but does not add the entry to the package.json file dependencies
• --save-optional installs and adds the entry to the package.json file optionalDependencies
• --no-optional will prevent optional dependencies from being installed
	`npm install <package-name>`
	`npm install <package>@<version>`

Updating packages	`npm update`
Updating a single package	`npm update <package-name>`

Running Tasks
{
"scripts": {
"watch": "webpack --watch --progress --colors --config webpack.conf.js",
"dev": "webpack --progress --colors --config webpack.conf.js",
"prod": "NODE_ENV=production webpack -p --config webpack.conf.js",
}
}
	`npm run <task-name>`
	`$ npm run watch`
	`$ npm run dev`
	`$ npm run prod`

see the version of all installed npm packages, including their dependencies	`npm list`
listing all the previous versions of a package	`npm view <package> versions`
Uninstalling npm packages Using the -S flag, or --save, this operation will also remove the reference in the package.json file. If the package is installed globally, you need to add the -g / --global flag	`npm uninstall <package-name>`

The difference between devDependencies and dependencies is that the former contains development tools, like a testing library, while the latter is bundled with the app in production.

As for the optionalDependencies the difference is that build failure of the dependency will not cause installation to fail. But it is your program's responsibility to handle the lack of the dependency. Running `npm install --no-optional` will prevent these dependencies from being installed. It is still your program's responsibility to handle the lack of the dependency. For example, something like this:

```
try {
  var foo = require('foo')
  var fooVersion = require('foo/package.json').version
} catch (er) {
  foo = null
}
if ( notGoodFooVersion(fooVersion) ) {
  foo = null
}

// .. then later in your program ..

if (foo) {
  foo.doFooThings()
}
```

Package Installation Location

By default, when you type an npm install command, the package is installed in the current file tree, under the `node_modules` subfolder.

A global installation is performed using the `-g` flag. When this happens, npm won't install the package under the local folder, but instead, it will use a global location.

The `npm root -g` command will tell you where that exact location is on your machine.

In your code you can only require local packages.

Executing Packages

If package is executable, it will put the executable file under the `node_modules/.bin/` folder. You can of course type `./node_modules/.bin/<app>` to run it, and it works, but npx, included in the recent versions of `npm` (since 5.2), is a much better option.

The package.json guide

The package.json file is kind of a manifest for your project. It can do a lot of things, completely unrelated. It's a central repository of configuration for tools, for example. It's also where npm and yarn store the names and versions for all the installed packages.

The only requirement is that it respects the JSON format, otherwise it cannot be read by programs that try to access its properties programmatically.

- `version` indicates the current version
  This property follows the semantic versioning (semver) notation for versions, which means the version is always expressed with 3 numbers: x.x.x.
- `name` sets the application/package name
- `description` is a brief description of the app/package
- `main` sets the entry point for the application. When you import this package in an application, that's where the application will search for the module exports.
- `private` if set to true prevents the app/package to be accidentally published on npm
- `scripts` defines a set of node scripts you can run
- `dependencies` sets a list of npm packages installed as dependencies
- `devDependencies` sets a list of npm packages installed as development dependencies
- `engines` sets which versions of Node.js this package/app works on
- `browserslist` is used to tell which browsers (and their versions) you want to support
- `author` Lists the package author name
- `contributors` As well as the author, the project can have one or more contributors. This property is an array that lists them.
- `bugs` Links to the package issue tracker, most likely a GitHub issues page
- `homepage` Sets the package homepage
- `license` Indicates the license of the package.
- `keywords` This property contains an array of keywords that associate with what your package does.
  This helps people find your package when navigating similar packages, or when browsing the https://www.npmjs.com/ website.

The package-lock.json file

The goal of package-lock.json file is to keep track of the exact version of every package that is installed so that a product is 100% reproducible in the same way even if packages are updated by their maintainers.

In package.json you can set which versions you want to upgrade to (patch or minor), using the semver notation.

It could be you, or another person trying to initialize the project on the other side of the world by running npm install. So your original project and the newly initialized project are actually different. Even if a patch or minor release should not introduce breaking changes, we all know bugs can (and so, they will) slide in.

The package-lock.json sets your currently installed version of each package in stone, and npm will use those exact versions when running npm ci.

Semantic Versioning (Semver)

The Semantic Versioning concept is simple: all versions have 3 digits: x.y.z.

- the first digit is the major version
- the second digit is the minor version
- the third digit is the patch version

When you make a new release, you don't just up a number as you please, but you have rules:

- you up the major version when you make incompatible API changes
- you up the minor version when you add functionality in a backward-compatible manner
- you up the patch version when you make backward-compatible bug fixes

How npm uses Semver?

rules in detail:

- ^: It will only do updates that do not change the leftmost non-zero number i.e there can be changes in minor version or patch version but not in major version. If you write ^13.1.0, when running npm update, it can update to 13.2.0, 13.3.0 even 13.3.1, 13.3.2 and so on, but not to 14.0.0 or above.
- ~: if you write ~0.13.0 when running npm update it can update to patch releases: 0.13.1 is ok, but 0.14.0 is not.
- >: you accept any version higher than the one you specify
- >=: you accept any version equal to or higher than the one you specify
- <=: you accept any version equal or lower to the one you specify
- <: you accept any version lower than the one you specify
- =: you accept that exact version
- -: you accept a range of versions. Example: 2.1.0 - 2.6.2
- ||: you combine sets. Example: < 2.1 || > 2.6
- no symbol: you accept only that specific version you specify (1.2.1)
- latest: you want to use the latest version available

NodeJs Package Runner (npx)

Easily run local commands: Running `npx commandname` automatically finds the correct reference of the command inside the node_modules folder of a project, without needing to know the exact path, and without requiring the package to be installed globally and in the user's path.

Installation-less command execution: There is another great feature of npx, which is allowing to run commands without first installing them. This is pretty useful, mostly because:

1. you don't need to install anything
2. you can run different versions of the same command, using the syntax @version

npx allows you to run that npm command without installing it first. If the command isn't found, npx will install it into a central cache

Run arbitrary code snippets directly from a URL: You can run code that sits in a GitHub gist. For example, npx <URL>

The Node.js Event Loop

explains how Node.js can be **asynchronous** and have **non-blocking I/O**,

The Node.js JavaScript code runs on a single thread. There is just **one thing happening at a time**.

You just need to pay attention to how you write your code and avoid anything that could block the thread, like **synchronous network calls** or **infinite loops**.

In general, in most browsers there is an **event loop** for every browser tab, to make every process isolated and avoid a web page with infinite loops or heavy processing to block your entire browser.

Any JavaScript code that takes too long to return back control to the event loop will **block the execution** of any JavaScript code in the page, even block the UI thread, and the user cannot click around, scroll the page, and so on. Almost all the I/O **primitives in JavaScript are non-blocking**. Network requests, filesystem operations, and so on.

The call stack	**The Message Queue (callback queue)**
The call stack is a LIFO (Last In, First Out) stack. The event loop continuously checks the call stack to see if there's any function that needs to run.	The Message Queue is also where user-initiated events like click or keyboard events, or fetch responses are queued before your code has the opportunity to react to them. Or also DOM events like onload.
The loop gives priority to the call stack, and it first processes everything it finds in the call stack, and once there's nothing in there, it goes to pick up things in the message queue.	

We don't have to wait for functions like setTimeout, fetch or other things to do their own work, because they are provided by the browser, and they live on their own threads. For example, if you set the setTimeout timeout to 2 seconds, you don't have to wait 2 seconds - the wait happens elsewhere.

ES6 Job Queue

ECMAScript 2015 introduced the concept of the Job Queue, which is used by Promises (also introduced in ES6/ES2015). It's a way to execute the result of an async function as soon as possible, rather than being put at the end of the call stack.

ES6 has 2 queues

1. CallBack queue
2. Job queue (Micro-task queue)

Both setTimeout and a promise are async code.

Once the timer has done its job it pushes the function onto the callback queue and has to wait until all the sync code of js has completed

The `.then()` specifies which function to be run once the promise has been resolved , but not immediately. The function specified inside the `.then()` gets pushed inside the Job queue on task completion.

Once all the sync code in JS is completed, the event loop **checks the job queue first and then the callback queue.**

your asynchronous code runs after all the synchronous code is done executing

In summary:

- Tasks execute in order, and the browser may render between them
- Microtasks execute in order, and are executed:
    - after every callback, as long as no other JavaScript is mid-execution
    - at the end of each task

process.nextTick()

Every time the event loop takes a full trip, we call it a tick. When we pass a function to `process.nextTick()`, we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts.

It's the way we can tell the JS engine to process a function asynchronously (after the current function), but as soon as possible, not queue it.

```
process.nextTick(() => {
  //do something
})
```

Calling `setTimeout(() => {}, 0)` will execute the function at the end of next tick

Using `nextTick()` executes it just before the beginning of the next tick.

setImmediate()

Any function passed as the setImmediate() argument is a callback that's executed in the next iteration of the event loop.

How is setImmediate() different from setTimeout(() => {}, 0) (passing a 0ms timeout), and from process.nextTick()?

- A function passed to process.nextTick() is going to be **executed on the current iteration** of the event loop, after the current operation ends. This means it will always execute before setTimeout and setImmediate.
- A setTimeout() callback with a 0ms delay is very similar to setImmediate(). The **execution order will depend on various factors**, but they will be both **run in the next iteration** of the event loop.

setInterval()

similar to setTimeout, with a difference: instead of running the callback function once, it will run it forever, at the specific time interval you specify (in milliseconds).

The function above runs every 2 seconds unless you tell it to stop, using clearInterval, passing it the interval id that setInterval returned. It's common to call clearInterval inside the setInterval callback function, to let it auto-determine if it should run again or stop:

```
const interval = setInterval(() => {
  if (App.somethingIWait === 'arrived') {
    clearInterval(interval)
    return
  }
  // otherwise do things
}, 100)
```

If the function takes different execution times, depending on network conditions for example and maybe one long execution overlaps the next one, then to avoid this, you can schedule a recursive setTimeout to be called when the callback function finishes:

```
const myFunction = () => {
  // do something
  setTimeout(myFunction, 1000)
}
setTimeout(myFunction, 1000)
```

Callbacks

A callback is a simple function that's passed as a value to another function, and will only be executed when the event happens. We can do this because JavaScript has first-class functions.

A programming language is said to have First-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable. A function that returns a function is called a Higher-Order Function.

It's common to wrap all your client code in a load event listener on the window object, which runs the callback function only when the page is ready.

```
window.addEventListener('load', () => {
  //window loaded
  //do what you want
})
```

Usually callbacks are only used when doing I/O, e.g. downloading things, reading files, talking to databases, etc. 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).

Handling errors in callbacks

One very common strategy is to use what Node.js adopted: the first parameter in any callback function is the error object: **error-first callbacks**.

If there is no error, the object is null. If there is an error, it contains some description of the error and other information.

```
fs.readFile('/file.json', (err, data) => {
  if (err) {
    //handle error
    console.log(err)
    return
  }

  //no errors, process data
  console.log(data)
})
```

How do I fix callback hell?

Callback hell is caused by poor coding practices.

Solution #1: Keep your code shallow

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

Solution #2: Modularize

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."*

Code has the following benefits:

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

Solution #3: Handle every single error

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.

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 its 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.

JavaScript function hoisting

When JS is parsed, a first pass is done over each scope, and function definitions are immediately discovered. When a function is declared like this, with a name, that name becomes available to the entire scope when the code in that scope is executed.

A crude timeline of how JS gets executed:

1. Parse the scope and detect all function definitions
2. Execute the code top-to-bottom with all functions found in step 1 available as variables

This behavior is called 'hoisting' because it is almost like the function definitions have been 'hoisted' up to the top of the function. A function must not be associated with an assignment in order for it to be hoisted.

Wrapping a function in parenthesis (()) is a quick way to convert a function definition into a function expression, which means it does not get hoisted (similar to assigning the function to a variable).

`function sayHi() {`
`  console.log('hi!')`
`}`
`sayHi()`
	works	`sayHi()`
		`function sayHi() {`
		`  console.log('hi!')`
		`}`
			works

`var sayHi = function() {`
`  console.log('hi!')`
`}`
`sayHi()`
	works	`sayHi()`
		`var sayHi = function() {`
		`  console.log('hi!')`
		`}`
			does not work

`(function sayHi() {`
`  console.log('hi!')`
`})`
`sayHi()`
	does not work	`sayHi()`
		`(function sayHi() {`
		`  console.log('hi!')`
		`})`
			does not work

`(function sayHi() {`
`  console.log('hi!')`
`})()`
	works	`(function() {`
		`  console.log('hi!')`
		`})()`
			works

`var sayHi = (function() {`
`  console.log('hi!')`
`})()`
	works	`var sayHi = (function() {`
		`  console.log('hi!')`
		`})`
		`sayHi()`
			works

`sayHi()`
`var sayHi = (function() {`
`  console.log('hi!')`
`})`
	does not work	`sayHi()`
		`var x = function sayHi() {`
		`  console.log('hi!')`
		`}`
			does not work

`var x = (function sayHi() {`
`  console.log('hi!')`
`})`
`x() // 1`
`sayHi() // 2`
	works (1)
	does not work (2)		

Promises

A promise is commonly defined as **a proxy for a value that will eventually become available**. How it works:

1. Once a promise has been called, it will start in a **pending state**. This means that the calling function continues executing, while the promise is pending until it resolves, giving the calling function whatever data was being requested.
2. The created promise will eventually end in a **resolved state**, or in a **rejected state**, calling the respective callback functions (passed to `then` and `catch`) upon finishing.

Creating a promise

The Promise API exposes a Promise constructor, which you initialize using `new Promise():`

```
let done = true

const isItDoneYet = new Promise((resolve, reject) => {
  if (done) {
    const workDone = 'Here is the thing I built'
    resolve(workDone)
  } else {
    const why = 'Still working on something else'
    reject(why)
  }
})
```

Using resolve and reject, we can communicate back to the caller what the resulting promise state was, and what to do with it. In the above snippet a string returned, but it could be an object or null as well.

Promisifying

A more common example you may come across is a technique called Promisifying. This technique is a way to be able to use a classic JavaScript function that takes a callback, and have it return a promise:

```
const fs = require('fs')

const getFile = (fileName) => {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, (err, data) => {
      if (err) {
        reject(err)  // calling `reject` will cause the promise to fail
with or without the error passed as an argument
        return        // and we don't want to go any further
      }
      resolve(data)
    })
  })
}

getFile('/etc/passwd')
.then(data => console.log(data))
.catch(err => console.error(err))
```

In recent versions of Node.js, you won't have to do this manual conversion for a lot of the API. There is a promisifying function available in the util module that will do this for you, given that the function you're promisifying has the correct signature:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.').then((stats) => {
  // Do something with `stats`
}).catch((error) => {
  // Handle the error.
});
```

Consuming a promise

```
const isItDoneYet = new Promise(/* ... as above ... */)
//...

const checkIfItsDone = () => {
  isItDoneYet
    .then(ok => {
      console.log(ok)
    })
    .catch(err => {
      console.error(err)
    })
}
```

Chaining promises

A promise can be returned to another promise, creating a chain of promises.

A great example of chaining promises is the Fetch API, which we can use to get a resource and queue a chain of promises to execute when the resource is fetched.

```
const status = response => {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response)
  }
  return Promise.reject(new Error(response.statusText))
}

const json = response => response.json()

fetch('/todos.json')
  .then(status)    // note that the `status` function is actually **called** here,
and that it **returns a promise***
  .then(json)      // likewise, the only difference here is that the `json` function
here returns a promise that resolves with `data`
  .then(data => {  // ... which is why `data` shows up here as the first parameter to
the anonymous function
    console.log('Request succeeded with JSON response', data)
  })
  .catch(error => {
    console.log('Request failed', error)
  })
```

Handling errors

In the example, in the previous section, we had a catch that was appended to the chain of promises.

When anything in the chain of promises fails and raises an error or rejects the promise, the control goes to the nearest catch() statement down the chain.

If **inside the catch() you raise an error**, you can append a second catch() to handle it, and so on.

Orchestrating promises

Promise.all()

If you need to synchronize different promises, Promise.all() helps you define a list of promises, and execute something when they are all resolved.

```
const f1 = fetch('/something.json')
const f2 = fetch('/something2.json')

Promise.all([f1, f2])
  .then(res => {
    console.log('Array of results', res)
  })
  .catch(err => {
    console.error(err)
  })

// or
Promise.all([f1, f2]).then(([res1, res2]) => {
  console.log('Results', res1, res2)
})
```

Promise.race()

Promise.race() runs when the first of the promises you pass to it settles (resolves or rejects), and it runs the attached callback just once, with the result of the first promise settled.

```
const first = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'first')
})
const second = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'second')
})

Promise.race([first, second]).then(result => {
  console.log(result) // second
})
```

Promise.any()

Promise.any() settles when any of the promises you pass to it fulfill or all of the promises get rejected. It returns a single promise that resolves with the value from the first promise that is fulfilled. If all promises are rejected, then the returned promise is rejected with an AggregateError.

```
const first = new Promise((resolve, reject) => {
  setTimeout(reject, 500, 'first')
})
const second = new Promise((resolve, reject) => {
  setTimeout(reject, 100, 'second')
})

Promise.any([first, second]).catch(error => {
  console.log(error) // AggregateError
})
```

Promise.race	Promise.any	Promise.all
is settled as soon as any of the promises you feed it settle, whether they are fulfilled or rejected.	is settled as soon as any of the promises you feed it is fulfilled or they	Is rejected as soon as any of the promises you feed it is rejected.

	are all rejected, in which case it's rejected with an AggregateError.	Is fulfilled as soon as all of the promises you feed is fulfilled.

Common errors

- If you get the `Uncaught TypeError: undefined is not a promise` error in the console, make sure you use `new Promise()` instead of just `Promise()`
- `UnhandledPromiseRejectionWarning` This means that a promise you called rejected, but there was no catch used to handle the error. Add a catch after the offending then to handle this properly.

Async and Await

An async function returns a promise,

```
const doSomethingAsync = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}
```

When you want to call this function you prepend await, and the calling code will stop until the promise is resolved or rejected. **One caveat: the client function must be defined as async.** Here's an example:

```
const doSomething = async () => {
  console.log(await doSomethingAsync())
}
```

Prepending the async keyword to any function means that the function will return a promise. Even if it's not doing so explicitly, it will internally make it return a promise.

`const aFunction = async () => {`
`  return 'test'`
`}`
`aFunction().then(alert) // This will alert 'test'`
	`const aFunction = () => {`
	`  return Promise.resolve('test')`
	`}`
	`aFunction().then(alert) // This will alert 'test'`

How to use?

Smelly	Classy

`const getFirstUserData = () => {`
`  return fetch('/users.json') // get users list`
`    .then(response => response.json()) // parse JSON`
`    .then(users => users[0]) // pick first user`
`    .then(user => fetch(`/users/${user.name}`)) // get user data`
`    .then(userResponse => userResponse.json()) // parse JSON`
`}`
 	
`getFirstUserData()`
	`const getFirstUserData = async () => {`
	`  const response = await fetch('/users.json') // get users list`
	`  const users = await response.json() // parse JSON`
	`  const user = users[0] // pick first user`
	`  const userResponse = await fetch(`/users/${user.name}`) // get user data`
	`  const userData = await userResponse.json() // parse JSON`
	`  return userData`
	`}`
	
	`getFirstUserData()`

The Node.js Event emitter

Events module, EventEmitter class.

```
const EventEmitter = require('events')
const eventEmitter = new EventEmitter()
eventEmitter.on('start', () => {
  console.log('started')
})
eventEmitter.emit('start')
```

This object exposes, among many others, the on and emit methods.

- **emit** is used to trigger an event
- **on** is used to add a callback function that's going to be executed when the event is triggered
- **once**(): add a one-time listener
- **removeListener**() / **off**(): remove an event listener from an event
- **removeAllListeners**(): remove all listeners for an event

File Operations

fs.open()

- r+ open the file for reading and writing, if file doesn't exist it won't be created.
- w+ open the file for reading and writing, positioning the stream at the beginning of the file. The file is created if not existing.
- a open the file for writing, positioning the stream at the end of the file. The file is created if not existing.
- a+ open the file for reading and writing, positioning the stream at the end of the file. The file is created if not existing.

```
const fs = require('fs')

fs.open('/Users/joe/test.txt', 'r', (err,
fd) => {
  //fd is our file descriptor
})
```

fs.stat()

- if the file is a directory or a file, using stats.isFile() and stats.isDirectory()
- if the file is a symbolic link using stats.isSymbolicLink()
- the file size in bytes using stats.size.

```
const fs = require('fs')
fs.stat('/Users/joe/test.txt', (err,
stats) => {
  if (err) {
    console.error(err)
    return
  }
  //we have access to the file stats in
`stats`
})
```

fs.readFile()

read the full content of the file in memory before returning the data. This means that big files are going to have a major impact on your memory consumption and speed of execution of the program.
In this case, a better option is to read the file content using streams.

```
const fs = require('fs')

fs.readFile('/Users/joe/test.txt', 'utf8'
, (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(data)
})
```

fs.writeFile()

By default, this API will replace the contents of the file if it does already exist. Use flag parameter to change

```
fs.writeFile('/Users/joe/test.txt', content, { flag:
'a+' }, err => {})
```

write the full content to the file before returning the control.

```
const fs = require('fs')

const content = 'Some content!'

fs.writeFile('/Users/joe/test.txt',
content, err => {
  if (err) {
    console.error(err)
    return
  }
  //file written successfully
})
```

fs.appendFile()

append content to the end of a file

write the full content to the file before returning the control.

```
const content = 'Some content!'

fs.appendFile('file.log', content, err =>
{
  if (err) {
    console.error(err)
    return
  }
  //done!
})
```

path.[dir/base/ext]name()

- dirname: get the parent folder of a file
- basename: get the filename part
- extname: get the file extension

```
const path = require('path')
const notes = '/users/joe/notes.txt'

path.dirname(notes) // /users/joe
path.basename(notes) // notes.txt
path.extname(notes) // .txt
path.basename(notes, path.extname(notes))
//notes
```

path.join()

join two or more parts of a path

```
const name = 'joe'
path.join('/', 'users', name,
'notes.txt') //'/users/joe/notes.txt'
```

<table>
<tr><td><b>path.resolve()</b>
get the absolute path calculation of a relative path. Default, it simply append /joe.txt to the current working directory
If you specify a second parameter folder, resolve will use the first as a base for the second
If the first parameter starts with a slash, that means it's an absolute path:</td><td><pre>
path.resolve('joe.txt')
//'/Users/joe/joe.txt' if run from my home folder

path.resolve('tmp', 'joe.txt')
//'/Users/joe/tmp/joe.txt' if run from my home folder

path.resolve('/etc', 'joe.txt')
//'/etc/joe.txt'
</pre></td></tr>
<tr><td><b>path.normalize()</b>
try and calculate the actual path, when it contains relative specifiers like . or .., or double slashes</td><td><pre>
path.normalize('/users/joe/..//test.txt')
//'/users/test.txt'
</pre></td></tr>
<tr><td><b>fs.access()</b>
Check if a folder exists</td><td></td></tr>
<tr><td><b>fs.mkdir()</b>
Create a new folder</td><td><pre>
const fs = require('fs')

const folderName = '/Users/joe/test'

try {
  if (!fs.existsSync(folderName)) {
    fs.mkdirSync(folderName)
  }
} catch (err) {
  console.error(err)
}
</pre></td></tr>
<tr><td><b>fs.readdir()</b>
Read the content of a directory
both files and subfolders, and returns their relative path:</td><td><pre>
const fs = require('fs')

const folderPath = '/Users/joe'

fs.readdirSync(folderPath)
</pre></td></tr>
<tr><td><b>fs.rename()</b>
Rename a folder</td><td><pre>
const fs = require('fs')

fs.rename('/Users/joe', '/Users/roger',
err => {
  if (err) {
    console.error(err)
    return
  }
  //done
})
</pre></td></tr>
<tr><td><b>fs.rmdir()</b>
Remove a folder

<b>[DEPRECATED]</b> Removing a folder that has content can be more complicated than you need. You can pass the option <code>{ recursive: true }</code> to recursively remove the contents.</td><td><pre>
const fs = require('fs')

fs.rmdir(dir, { recursive: true }, (err)
=> {
    if (err) {
        throw err;
    }

    console.log(`${dir} is deleted!`);
});
</pre></td></tr>
<tr><td><b>fs.rm</b>
delete folders that have content in them</td><td><pre>
const fs = require('fs')

fs.rm(dir, { recursive: true, force: true
}, (err) => {
  if (err) {
    throw err;
  }

  console.log(`${dir} is deleted!`)
});
</pre></td></tr>
</table>

Neither `resolve` nor `normalize` will check if the path exists. They just calculate a path based on the information they got.

Or you can install and make use of the *`fs-extra`* module, which is very popular and well maintained. It's a drop-in replacement of the fs module, which provides more features on top of it.

Node.js Streams

The Node.js stream module provides the foundation upon which all streaming APIs are built. All streams are instances of EventEmitter.

An example file serving both block and stream approach:

Block	Stream

const http = require('http')
const fs = require('fs')
 	
const server = http.createServer(function(req, res) {
  fs.readFile(__dirname + '/data.txt', (err, data) => {
    res.end(data)
  })
})
server.listen(3000)
	const http = require('http')
	const fs = require('fs')
	
	const server = http.createServer((req, res) => {
	  const stream = fs.createReadStream(__dirname +
	'/data.txt')
	  stream.pipe(res)
	})
	server.listen(3000)

pipe()

Takes the source, and pipes it into a destination.

The return value of the pipe() method is the destination stream, which is a very convenient thing that lets us chain multiple pipe() calls.

Readable Stream

a stream you can pipe from, but not pipe into (you can receive data, but not send data to it). When you push data into a readable stream, it is buffered, until a consumer starts to read the data.

We get the Readable stream from the stream module, and we initialize it and implement the readable._read() method.

```
const Stream = require('stream')
const readableStream = new Stream.Readable()
readableStream._read = () => {}
//or
const readableStream = new Stream.Readable({
  read() {}
})

//send data
readableStream.push('hi!')
readableStream.push('ho!')
```

Writable Stream

a stream you can pipe into, but not pipe from (you can send data, but not receive from it)

To create a writable stream we extend the base Writable object, and we implement its _write() method.

```
const Stream = require('stream')
const writableStream = new Stream.Writable()
writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}

// pipe readable stream in
process.stdin.pipe(writableStream)
```

Duplex Stream

a stream you can both pipe into and pipe from, basically a combination of a Readable and Writable stream

Transform Stream

a Transform stream is similar to a Duplex, but the output is a transform of its input.

We get the Transform stream from the stream module, and we initialize it and implement the transform._transform() method.

```
const { Transform } = require('stream')
const transformStream = new Transform();
transformStream._transform = (chunk, encoding, callback) => {
  transformStream.push(chunk.toString().toUpperCase());
  callback();
}
process.stdin.pipe(transformStream).pipe(process.stdout);
```

How to get data from a readable stream

```
const Stream = require('stream')

const readableStream = new Stream.Readable({
  read() {}
})

/////
const writableStream = new Stream.Writable()
writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}
readableStream.pipe(writableStream)

// or

readableStream.on('readable', () => {
  console.log(readableStream.read())
})
/////

readableStream.push('hi!')
readableStream.push('ho!')
```

How to send data to a writable stream

```
writableStream.write('hey!\n')
```

Signaling a writable stream that you ended writing

```
const Stream = require('stream')

const readableStream = new Stream.Readable({
  read() {}
})
const writableStream = new Stream.Writable()

writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}

readableStream.pipe(writableStream)

readableStream.push('hi!')
readableStream.push('ho!')

readableStream.on('close', () => writableStream.end())
writableStream.on('close', () => console.log('ended'))

readableStream.destroy()
```

end() is called within a listener to the close event on the readable stream
to ensure it is not called before all write events have passed through the pipe,
as doing so would cause an error event to be emitted.

Calling destroy() on the readable stream causes the close event to be emitted.

The listener to the close event on the writable stream
demonstrates the completion of the process
as it is emitted after the call to end().

Node.js Buffers

Buffers were introduced to help developers deal with binary data, in an ecosystem that traditionally only dealt with strings rather than binaries.

Create

A buffer is created using the Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe() methods.

```
const buf = Buffer.from('Hey!')
const buf = Buffer.alloc(1024)
const buf = Buffer.allocUnsafe(1024)
```

While both alloc and allocUnsafe allocate a Buffer of the specified size in bytes, the Buffer created by alloc will be initialized with zeroes and the one created by allocUnsafe will be uninitialized. This means that while allocUnsafe would be quite fast in comparison to alloc, the allocated segment of memory may contain old data which could potentially be sensitive.

Older data, if present in the memory, can be accessed or leaked when the Buffer memory is read. This is what really makes allocUnsafe unsafe and extra care must be taken while using it.

Access

A buffer, being an array of bytes, can be accessed like an array.

You can print the full content of the buffer using the toString() method

```
const buf = Buffer.from('Hey!')
console.log(buf[0]) //72
console.log(buf[1]) //101
console.log(buf[2]) //121
console.log(buf.toString())
```

Length of a buffer	`const buf = Buffer.from('Hey!')`
	`console.log(buf.length)`

Iterate over the contents	`const buf = Buffer.from('Hey!')`
	`for (const item of buf) {`
	`  console.log(item) //72 101 121 33`
	`}`

Changing the content	`const buf = Buffer.alloc(4)`
	`buf.write('Hey!')`
	`//or`
	`const buf = Buffer.from('Hey!')`
	`buf[1] = 111 //o in UTF-8`
	`console.log(buf.toString()) //Hoy!`

Slice a buffer
If you want to create a partial visualization of a buffer, you can create a slice. A slice is not a copy: the original buffer is still the source of truth. If that changes, your slice changes.
	`const buf = Buffer.from('Hey!')`
	`buf.subarray(0).toString() //Hey!`
	`const slice = buf.subarray(0, 2)`
	`console.log(slice.toString()) //He`
	`buf[1] = 111 //o`
	`console.log(slice.toString()) //Ho`

Copy a buffer (whole)	`const buf = Buffer.from('Hey!')`
	`const bufcopy = Buffer.alloc(4) //allocate 4 bytes`
	`bufcopy.set(buf)`

Copy a buffer (partial)	`const buf = Buffer.from('Hey?')`
	`const bufcopy = Buffer.from('Moo!')`
	`bufcopy.set(buf.subarray(1, 3), 1)`
	`bufcopy.toString() //'Mey!'`

Error handling in Node.js

An exception is created using the throw keyword.

As soon as JavaScript executes this line, the normal program flow is halted and the control is held back to the nearest exception handler.

Usually in client-side code value can be any JavaScript value including a string, a number or an object. In Node.js, we don't throw strings, we just throw **Error objects**.

An error object is an object that is either an instance of the Error object, or extends the Error class, provided in the Error core module:

```
throw new Error('Ran out of coffee')
//or
class NotEnoughCoffeeError extends Error
{
  //...
}
throw new NotEnoughCoffeeError()
```

Catching uncaught exceptions

If an uncaught exception gets thrown during the execution of your program, your program will crash.

To solve this, you listen for the uncaughtException event on the process object:

```
process.on('uncaughtException', err => {
  console.error('There was an uncaught error', err)
  process.exit(1) //mandatory (as per the Node.js docs)
})
```