THE VALLEY OF CODE

NODE.JS HANDBOOK

FLAVIO COPES

Table of Contents

Preface

The Node.js Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

In particular, the goal is to get you up to speed quickly with Node.js.

This book is written by Flavio. I **publish programming tutorials** on my blog flaviocopes.com and The Valley Of Code.

You can reach me on Twitter @flaviocopes.

Enjoy!

The Node.js Handbook

Introduction to Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

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.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the

Node.js version, and you can also enable specific experimental features by running Node.js with flags.

Node.js has a vast number of libraries

`npm` with its simple structure helped the ecosystem of node.js proliferate and now the npm registry hosts almost 500.000 open source packages you can freely use.

An example Node.js application

The most common example Hello World of Node.js is a web server:

```
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}/`)
})
```

To run this snippet, save it as a `server.js` file and run `node server.js` in your terminal.

This code first includes the Node.js `http` module.

Node.js has an amazing standard library, including a first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and hostname. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the `request` event is called, providing two objects: a request (an `http.IncomingMessage` object) and a response (an `http.ServerResponse` object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with

```
res.statusCode = 200
```

we set the statusCode property to 200, to indicate a successful response.

We set the Content-Type header:

```
res.setHeader('Content-Type', 'text/plain')
```

and we end close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n')
```

Node.js frameworks and tools

Node.js is a low-level platform, and to make things easier and more interesting for developers thousands of libraries were built upon Node.js.

Many of those established over time as popular options. Here is a non-comprehensive list of the ones I consider very relevant and worth learning:

- **Express**, one of the most simple yet powerful ways to create a web server. Its minimalist approach, unopinionated, focused on the core features of a server, is key to its success.
- **Meteor**, an incredibly powerful full-stack framework, powering you with an isomorphic approach to building apps with JavaScript, sharing code on the client and the server. Once an off-the-shelf tool that provided everything, now integrates with frontend libs React, Vue and Angular. Can be used to create mobile apps as well.
- **koa**, built by the same team behind Express, aims to be even simpler and smaller, building on top of years of knowledge. The new project born out of the need to create incompatible changes without disrupting the existing community.
- **Next.js**, a framework to render server-side rendered React applications.
- **Micro**, a very lightweight server to create asynchronous HTTP microservices.
- **Socket.io**, a real-time communication engine to build network applications.
- **SvelteKit**: Sapper is a framework for building web applications of all sizes, with a beautiful development experience and flexible filesystem-based routing. Offers SSR and more!
- **Remix**: Remix is a fullstack web framework for building excellent user experiences for the web. It comes out of the box with everything you need to build modern web applications (both frontend and backend) and deploy them to any JavaScript-based runtime environment (including Node.js).
- **Fastify** a fast and efficient web framework highly focused on providing the best developer experience with the least overhead and a powerful plugin architecture, inspired by Hapi and Express.

A brief history of Node.js

Believe it or not, Node.js is only 13 years old.

In comparison, JavaScript is 26 years old and the Web is 33 years old.

13 years isn't a very long time in tech, but Node.js seems to have been around forever.

In this post, we draw the big picture of Node.js in its history, to put things in perspective.

A little bit of history JavaScript is a programming language that was created at Netscape as a scripting tool to manipulate web pages inside their browser, Netscape Navigator.

Part of the business model of Netscape was to sell Web Servers, which included an environment called Netscape LiveWire that could create dynamic pages using server-side JavaScript. Unfortunately, Netscape LiveWire wasn't very successful and server-side JavaScript wasn't popularized until recently, by the introduction of Node.js.

One key factor that led to the rise of Node.js was the timing. Just a few years earlier, JavaScript had started to be considered as a more serious language, thanks to "Web 2.0" applications (such as Flickr, Gmail, etc.) that showed the world what a modern experience on the web could be like.

JavaScript engines also became considerably better as many browsers competed to offer users the best performance. Development teams behind major browsers worked hard to offer better support for JavaScript and find ways to make JavaScript run faster. The engine that Node.js uses under the hood, V8 (also known as Chrome V8 for being the open-source JavaScript engine of The Chromium Project), improved significantly due to this competition.

Node.js happened to be built in the right place and right time, but luck isn't the only reason why it is popular today. It introduces a lot of innovative thinking and approaches for JavaScript server-side development that have already helped many developers.

2009

- Node.js is born

- The first version of npm

2010

- Express is born
- Socket.io is born

2011

- npm hits version 1.0
- Big companies start adopting Node: LinkedIn, Uber, etc

2012

- Adoption continues very rapidly

2013

- First big blogging platform using Node: Ghost
- Koa is born

2014

- The Big Fork: io.js is a major fork of Node.js, with the goal of introducing ES6 support and moving faster

2015

- The Node.js Foundation is born
- io.js is merged back into Node.js
- Node 4 (no 1, 2, 3 versions were previously released)

2016

- The leftpad incident

- Yarn is born
- Node 6

2017

- npm focuses more on security
- Node 8 - 9
- HTTP/2
- V8 introduces Node in its testing suite, officially making Node a target for the JS engine, in addition to Chrome
- 3 billion npm downloads every week

2018

- Node 10 - 11
- ES modules .mjs experimental support

2019

- Node 12 - 13

2020

- Node 14 - 15
- GitHub (owned by Microsoft) acquired NPM

2021

- Node.js 16
- Node.js 17

2022

- Node.js 18

How to install Node.js

Node.js can be installed in different ways.

Let me how you the most common and convenient ones.

Official packages for all the major platforms are available at https://nodejs.org/en/download/.

There you can choose to download an LTS version (LTS stands for Long Term Support) or the latest available release. As usual, the latest version contains the latest goodies.

On the site they have packages for Windows, Linux, macOS.

One very convenient way to install Node.js is through a package manager. In this case, every operating system has its own.

On macOS, Homebrew is the de-facto standard, and - once installed - allows to install Node.js very easily, by running this command in the CLI:

```
brew install node
```

Other package managers for Linux and Windows are listed in https://nodejs.org/en/download/package-manager/

`nvm` is a popular way to run Node. It allows you to easily switch the Node version, and install new versions to try and easily rollback if something breaks, for example.

It is also very useful to test your code with old Node versions.

See https://github.com/nvm-sh/nvm for more information about this option.

My suggestion is to use the official installer if you are just starting out and you don't use Homebrew already, otherwise, Homebrew is my favorite solution because I can easily update node by running `brew upgrade node`.

In any case, when Node is installed you'll have access to the `node` executable program in the command line.

How much JavaScript do you need to know to use Node?

As a beginner, it's hard to get to a point where you are confident enough in your programming abilities.

While learning to code, you might also be confused at where does JavaScript end, and where Node.js begins, and vice versa.

I would recommend you to have a good grasp of the main JavaScript concepts before diving into Node.js:

- Lexical Structure
- Expressions
- Types
- Variables
- Functions
- this
- Arrow Functions
- Loops
- Loops and Scope
- Arrays
- Template Literals
- Semicolons
- Strict Mode
- ECMAScript 6, 2016, 2017

With those concepts in mind, you are well on your road to become a proficient JavaScript developer, in both the browser and in Node.js.

The following concepts are also key to understand asynchronous programming, which is one fundamental part of Node.js:

- Asynchronous programming and callbacks
- Timers
- Promises
- Async and Await
- Closures
- The Event Loop

Differences between Node and the Browser

Both the browser and Node.js use JavaScript as their programming language.

Building apps that run in the browser is a completely different thing than building a Node.js application.

Despite the fact that it's always JavaScript, there are some key differences that make the experience radically different.

From the perspective of a frontend developer who extensively uses JavaScript, Node.js apps bring with them a huge advantage: the comfort of programming everything - the frontend and the backend - in a single language.

You have a huge opportunity because we know how hard it is to fully, deeply learn a programming language, and by using the same language to perform all your work on the web - both on the client and on the server, you're in a unique position of advantage.

What changes is the ecosystem.

In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies. Those do not exist in Node, of course. You don't have the `document`, `window` and all the other objects that are provided by the browser.

And in the browser, we don't have all the nice APIs that Node.js provides through its modules, like the filesystem access functionality.

Another big difference is that in Node.js you control the environment. 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. Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very convenient.

This means that you can write all the modern ES6-7-8-9 JavaScript that your Node version supports.

Since JavaScript moves so fast, but browsers can be a bit slow and users a bit slow to upgrade, sometimes on the web, you are stuck to use older JavaScript / ECMAScript releases.

You can use Babel to transform your code to be ES5-compatible before shipping it to the browser, but in Node, you won't need that.

Another difference is that Node originally used the CommonJS module system, while in the browser we use ES Modules, a standard that's recently landed in Node.js as well (https://nodejs.org/api/esm.html)

Going forward ES Modules (`import`) are the way to load modules across all JavaScript, frontend or backend, but Node.js still supports the `require` syntax.

The V8 JavaScript Engine

V8 is the name of the JavaScript engine that powers Google Chrome. It's the thing that takes our JavaScript and executes it while browsing with Chrome.

V8 provides the runtime environment in which JavaScript executes. The DOM, and the other Web Platform APIs are provided by the browser.

The cool thing is that the JavaScript engine is independent by the browser in which it's hosted. This key feature enabled the rise of Node.js. V8 was chosen to be the engine that powered Node.js back in 2009, and as the popularity of Node.js exploded, V8 became the engine that now powers an incredible amount of server-side code written in JavaScript.

The Node.js ecosystem is huge and thanks to it V8 also powers desktop apps, with projects like Electron.

Other JS engines

Other browsers have their own JavaScript engine:

- Firefox has **Spidermonkey**
- Safari has **JavaScriptCore** (also called Nitro)
- Edge was originally based on Chakra but has more recently been rebuilt using Chromium and the V8 engine.

and many others exist as well.

All those engines implement the ECMA ES-262 standard, also called ECMAScript, the standard used by JavaScript.

The quest for performance

V8 is written in C++, and it's continuously improved. It is portable and runs on Mac, Windows, Linux and several other systems.

In this V8 introduction, we will ignore the implementation details of V8: they can be found on more authoritative sites (e.g. the V8 official site), and they change over time, often radically.

V8 is always evolving, just like the other JavaScript engines around, to speed up the Web and the Node.js ecosystem.

On the web, there is a race for performance that's been going on for years, and we (as users and developers) benefit a lot from this competition because we get faster and more optimized machines year after year.

Compilation

JavaScript is generally considered an interpreted language, but modern JavaScript engines no longer just interpret JavaScript, they compile it.

This has been happening since 2009, when the SpiderMonkey JavaScript compiler was added to Firefox 3.5, and everyone followed this idea.

JavaScript is internally compiled by V8 with just-in-time (JIT) compilation to speed up the execution.

This might seem counter-intuitive, but since the introduction of Google Maps in 2004, JavaScript has evolved from a language that was generally executing a few dozens of lines of code to complete applications with thousands to hundreds of thousands of lines running in the browser.

Our applications can now run for hours inside a browser, rather than being just a few form validation rules or simple scripts.

In this new world, compiling JavaScript makes perfect sense because while it might take a little bit more to have the JavaScript ready, once done it's going to be much more performant than purely interpreted code.

Run Node.js scripts from the command line

The usual way to run a Node.js program is to run the globally available `node` command (once you install Node.js) and pass the name of the file you want to execute.

If your main Node.js application file is `app.js`, you can call it by typing:

```
node app.js
```

Above, you are explicitly telling the shell to run your script with `node`. You can also embed this information into your JavaScript file with a "shebang" line. The "shebang" is the first line in the file, and tells the OS which interpreter to use for running the script. Below is the first line of JavaScript:

```
#!/usr/bin/node
```

Above, we are explicitly giving the absolute path of interpreter. Not all operating systems have `node` in the bin folder, but all should have `env`. You can tell the OS to run `env` with node as parameter:

```
#!/usr/bin/env node

// your code
```

To use a shebang, your file should have executable permission. You can give `app.js` the executable permission by running:

```
chmod u+x app.js
```

While running the command, make sure you are in the same directory which contains the `app.js` file.

Restart the application automatically

The `node` command has to be re-executed in bash whenever there is a change in the application, to restart the application automatically, `nodemon` module is used.

Install the nodemon module globally to system path

```
npm i -g nodemon
```

You can also install nodemon as a development-dependency

```
npm i -D nodemon
```

This local installation of nodemon can be run by calling it from within npm script such as npm start or using npx nodemon.

Run the application using nodemon followed by application file name.

```
nodemon app.js
```

How to exit from a Node.js program

There are various ways to terminate a Node.js application.

When running a program in the console you can close it with `ctrl-C`, but what we want to discuss here is programmatically exiting.

Let's start with the most drastic one, and see why you're better off *not* using it.

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:

```
process.exit(1)
```

By default the exit code is `0`, which means success. Different exit codes have different meaning, which you might want to use in your own system to have the program communicate to other programs.

You can read more on exit codes at https://nodejs.org/api/process.html#process_exit_codes

You can also set the `process.exitCode` property:

```
process.exitCode = 1
```

and when the program ends, Node.js will return that exit code.

A program will gracefully exit when all the processing is done.

Many times with Node.js we start servers, like this HTTP server:

```
const express = require('express')

const app = express()

app.get('/', (req, res) => {
  res.send('Hi!')
})

app.listen(3000, () => console.log('Server ready'))
```

> Express is a framework that uses the http module under the hood, app.listen() returns an instance of http. You would use https.createServer if you needed to serve your app using HTTPS, as app.listen only uses the http module.

This program is never going to end. If you call `process.exit()`, any currently pending or running request is going to be aborted. This is *not nice*.

It is better to allow running request to complete before terminating. In this case you need to send the command a SIGTERM signal, and handle that with the process signal handler:

> Note: `process` does not require a "require", it's automatically available.

```
const express = require('express')

const app = express()

app.get('/', (req, res) => {
  res.send('Hi!')
})

const server = app.listen(3000, () => console.log('Server ready'))

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

> What are signals? Signals are a POSIX intercommunication system: a notification sent to a process in order to notify it of an event that occurred.

`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:

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

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.

How to read environment variables from Node.js

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.

The below code runs `app.js` and set `USER_ID` and `USER_KEY`.

```
USER_ID=239482 USER_KEY=foobar node app.js
```

That will pass the user `USER_ID` as **239482** and the `USER_KEY` as **foobar**. This is suitable for testing, however for production, you will probably be configuring some bash scripts to export variables.

> Note: `process` does not require a "require", it's automatically available.

Here is an example that accesses the `USER_ID` and `USER_KEY` environment variables, which we set in above code.

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

In the same way you can access any custom environment variable you set.

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.

```
# .env file
USER_ID="239482"
USER_KEY="foobar"
NODE_ENV="development"
```

In your js file

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

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

> You can also run your js file with `node -r dotenv/config index.js` command if you don't want to import the package in your code.

Where to host a Node.js app

Here is a non-exhaustive list of the options you can explore when you want to deploy your app and make it publicly accessible.

I will list the options from simplest and constrained to more complex and powerful.

Simplest option ever: local tunnel

Even if you have a dynamic IP, or you're under a NAT, you can deploy your app and serve the requests right from your computer using a local tunnel.

This option is suited for some quick testing, demo a product or sharing of an app with a very small group of people.

A very nice tool for this, available on all platforms, is **ngrok**.

Using it, you can just type `ngrok PORT` and the PORT you want is exposed to the internet. You will get a ngrok.io domain, but with a paid subscription you can get a custom URL as well as more security options (remember that you are opening your machine to the public Internet).

Another service you can use is https://github.com/localtunnel/localtunnel

Zero configuration deployments

Glitch

Glitch is a playground and a way to build your apps faster than ever, and see them live on their own glitch.com subdomain. You cannot currently have a a custom domain, and there are a few restrictions in place, but it's really great to prototype. It looks fun (and this is a plus), and it's not a dumbed down environment - you get all the power of Node.js, a CDN, secure storage for credentials, GitHub import/export and much more.

Provided by the company behind FogBugz and Trello (and co-creators of Stack Overflow).

I use it a lot for demo purposes.

Codepen

Codepen is an amazing platform and community. You can create a project with multiple files, and deploy it with a custom domain.

Serverless

A way to publish your apps, and have no server at all to manage, is Serverless. Serverless is a paradigm where you publish your apps as **functions**, and they respond on a network endpoint (also called FAAS - Functions As A Service).

To very popular solutions are

- Serverless Framework
- Standard Library

They both provide an abstraction layer to publishing on AWS Lambda and other FAAS solutions based on Azure or the Google Cloud offering.

PAAS

PAAS stands for Platform As A Service. These platforms take away a lot of things you should otherwise worry about when deploying your application.

Zeit Now

> Zeit is now called Vercel

Zeit is an interesting option. You just type `now` in your terminal, and it takes care of deploying your application. There is a free version with limitations, and the paid version is more powerful. You forget that there's a server, you just deploy the app.

Nanobox

Nanobox

Heroku

Heroku is an amazing platform.

This is a great article on getting started with Node.js on Heroku.

Microsoft Azure

Azure is the Microsoft Cloud offering.

Check out how to create a Node.js web app in Azure.

Google Cloud Platform

Google Cloud is an amazing structure for your apps.

They have a good Node.js Documentation Section

Virtual Private Server

In this section you find the usual suspects, ordered from more user friendly to less user friendly:

- Digital Ocean
- Linode
- Amazon Web Services, in particular I mention Amazon Elastic Beanstalk as it abstracts away a little bit the complexity of AWS.

Since they provide an empty Linux machine on which you can work, there is no specific tutorial for these.

There are lots more options in the VPS category, those are just the ones I used and I would recommend.

Bare metal

Another solution is to get a bare metal server, install a Linux distribution, connect it to the internet (or rent one monthly, like you can do using the Vultr Bare Metal service)

How to use the Node.js REPL

The `node` command is the one we use to run our Node.js scripts:

```
node script.js
```

If we run the `node` command without any script to execute or without any arguments, we start a REPL session:

```
node
```

> Note: REPL stands for Read Evaluate Print Loop, and it is a programming language environment (basically a console window) that takes single expression as user input and returns the result back to the console after execution. The REPL session provides a convenient way to quickly test simple JavaScript code.

If you try it now in your terminal, this is what happens:

```
❯ node
>
```

The command stays in idle mode and waits for us to enter something.

> Tip: if you are unsure how to open your terminal, google "How to open terminal on your-operating-system".

The REPL is waiting for us to enter some JavaScript code, to be more precise.

Start simple and enter

```
> console.log('test')
test
undefined
>
```

The first value, `test`, is the output we told the console to print, then we get `undefined` which is the return value of running `console.log()`. Node read this line of code, evaluated it, printed the result, and then went back to waiting for more lines of code. Node will loop through these three steps for every piece of code we execute in the REPL until we exit the session. That is where the REPL got its name.

Node automatically prints the result of any line of JavaScript code without the need to instruct it to do so. For example, type in the following line and press enter:

```
> 5 === '5'
false
>
```

Note the difference in the outputs of the above two lines. The Node REPL printed `undefined` after executed `console.log()`, while on the other hand, it just printed the result of `5 === '5'`. You need to keep in mind that the former is just a statement in JavaScript, and the latter is an expression.

In some cases, the code you want to test might need multiple lines. For example, say you want to define a function that generates a random number, in the REPL session type in the following line and press enter:

```
function generateRandom() {
...
```

The Node REPL is smart enough to determine that you are not done writing your code yet, and it will go into a multi-line mode for you to type in more code. Now finish your function definition and press enter:

```
function generateRandom() {
...return Math.random()
}
undefined
```

Node will get out of the multi-line mode, and print `undefined` since there is no value returned. This multi-line mode is limited. Node offers a more featured editor right inside the REPL. We discuss it below under Dot commands.

Use the tab to autocomplete

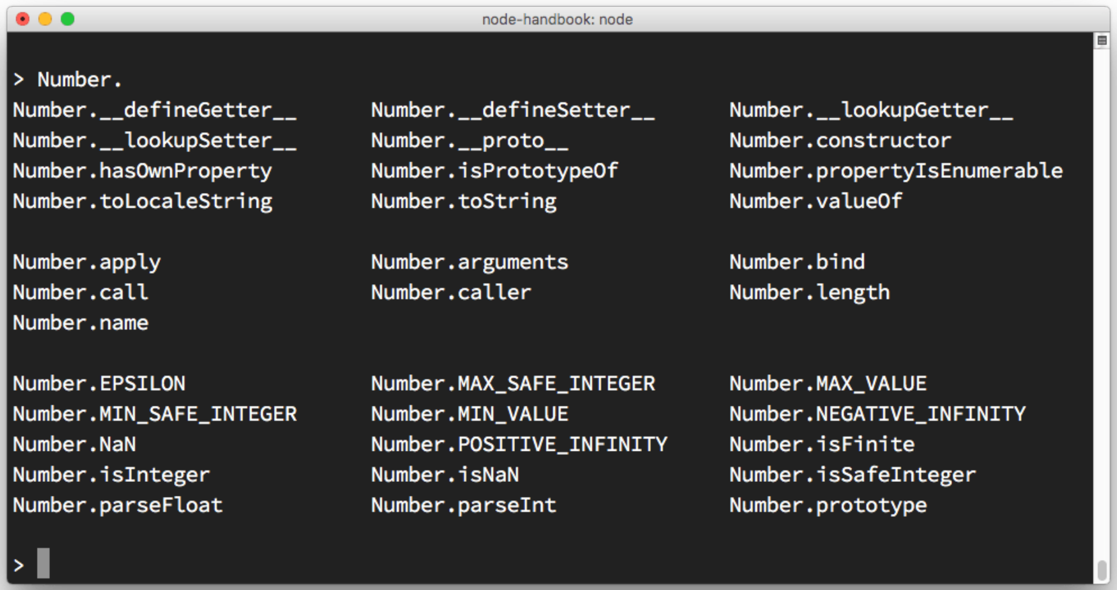
The cool thing about the REPL is that it's interactive.

As you write your code, if you press the `tab` key the REPL will try to autocomplete what you wrote to match a variable you already defined or a predefined one.

Exploring JavaScript objects

Try entering the name of a JavaScript class, like `Number`, add a dot and press `tab`.

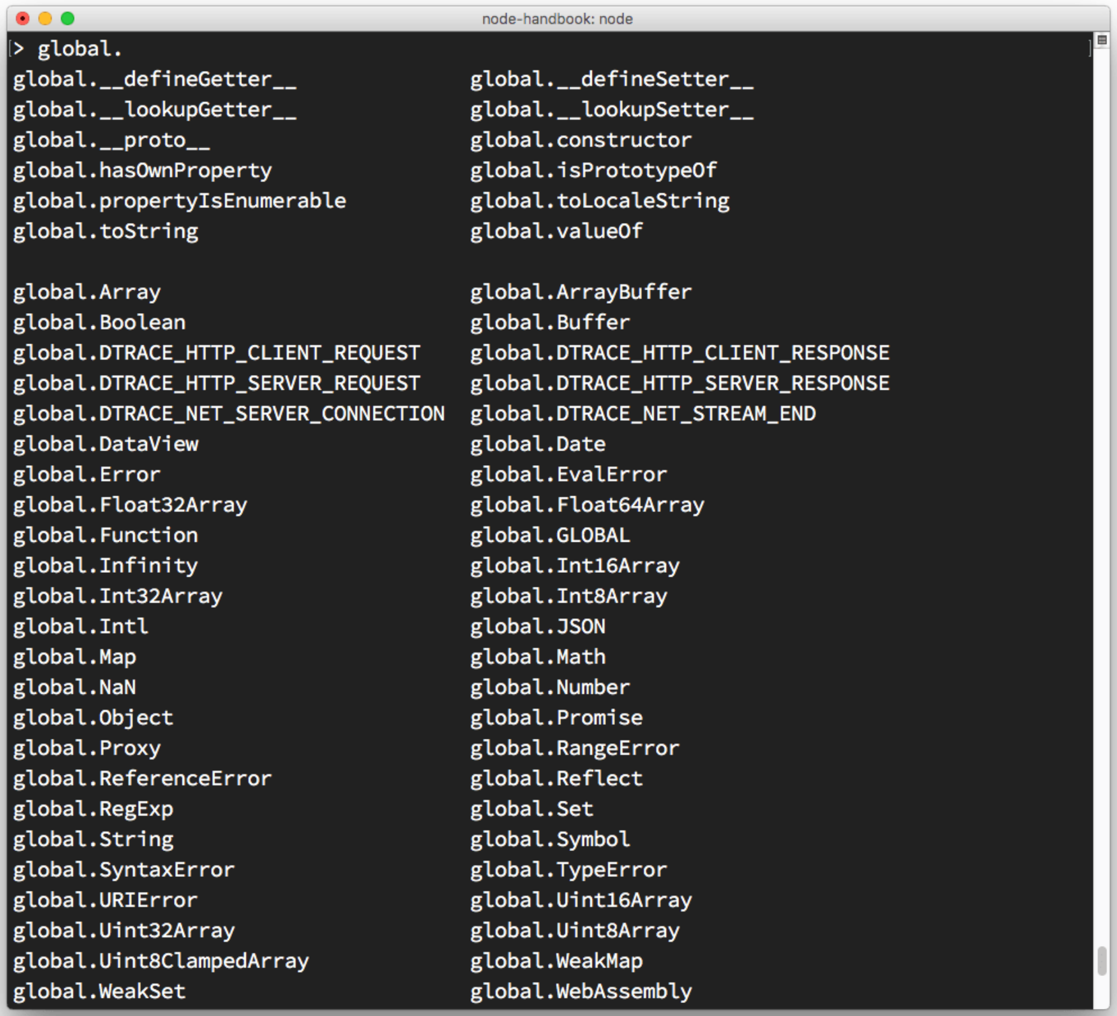
The REPL will print all the properties and methods you can access on that class:



```
> Number.
Number.__defineGetter__       Number.__defineSetter__       Number.__lookupGetter__
Number.__lookupSetter__       Number.__proto__              Number.constructor
Number.hasOwnProperty         Number.isPrototypeOf          Number.propertyIsEnumerable
Number.toLocaleString         Number.toString               Number.valueOf

Number.apply                  Number.arguments              Number.bind
Number.call                   Number.caller                 Number.length
Number.name

Number.EPSILON                Number.MAX_SAFE_INTEGER       Number.MAX_VALUE
Number.MIN_SAFE_INTEGER       Number.MIN_VALUE              Number.NEGATIVE_INFINITY
Number.NaN                    Number.POSITIVE_INFINITY      Number.isFinite
Number.isInteger              Number.isNaN                  Number.isSafeInteger
Number.parseFloat             Number.parseInt               Number.prototype

>
```

Explore global objects

You can inspect the globals you have access to by typing `global.` and pressing `tab`:



```
> global.
global.__defineGetter__              global.__defineSetter__
global.__lookupGetter__              global.__lookupSetter__
global.__proto__                     global.constructor
global.hasOwnProperty                global.isPrototypeOf
global.propertyIsEnumerable          global.toLocaleString
global.toString                      global.valueOf

global.Array                         global.ArrayBuffer
global.Boolean                       global.Buffer
global.DTRACE_HTTP_CLIENT_REQUEST    global.DTRACE_HTTP_CLIENT_RESPONSE
global.DTRACE_HTTP_SERVER_REQUEST    global.DTRACE_HTTP_SERVER_RESPONSE
global.DTRACE_NET_SERVER_CONNECTION  global.DTRACE_NET_STREAM_END
global.DataView                      global.Date
global.Error                         global.EvalError
global.Float32Array                  global.Float64Array
global.Function                      global.GLOBAL
global.Infinity                      global.Int16Array
global.Int32Array                    global.Int8Array
global.Intl                          global.JSON
global.Map                           global.Math
global.NaN                           global.Number
global.Object                        global.Promise
global.Proxy                         global.RangeError
global.ReferenceError                global.Reflect
global.RegExp                        global.Set
global.String                        global.Symbol
global.SyntaxError                   global.TypeError
global.URIError                      global.Uint16Array
global.Uint32Array                   global.Uint8Array
global.Uint8ClampedArray             global.WeakMap
global.WeakSet                       global.WebAssembly
```

The _ special variable

If after some code you type `_`, that is going to print the result of the last operation.

The Up arrow key

If you press the `up` arrow key, you will get access to the history of the previous lines of code executed in the current, and even previous REPL sessions.

Dot commands

The REPL has some special commands, all starting with a dot `.`. They are

- `.help`: shows the dot commands help
- `.editor`: enables editor mode, to write multiline JavaScript code with ease. Once you are in this mode, enter ctrl-D to run the code you wrote.
- `.break`: when inputting a multi-line expression, entering the .break command will abort further input. Same as pressing ctrl-C.
- `.clear`: resets the REPL context to an empty object and clears any multi-line expression currently being input.
- `.load`: loads a JavaScript file, relative to the current working directory
- `.save`: saves all you entered in the REPL session to a file (specify the filename)
- `.exit`: exits the repl (same as pressing ctrl-C two times)

The REPL knows when you are typing a multi-line statement without the need to invoke `.editor`.

For example if you start typing an iteration like this:

```
[1, 2, 3].forEach(num => {
```

and you press `enter`, the REPL will go to a new line that starts with 3 dots, indicating you can now continue to work on that block.

```
... console.log(num)
... })
```

If you type `.break` at the end of a line, the multiline mode will stop and the statement will not be executed.

Run REPL from JavaScript file

We can import the REPL in a JavaScript file using `repl`.

```
const repl = require('repl')
```

Using the repl variable we can perform various operations. To start the REPL command prompt, type in the following line

```
repl.start()
```

Run the file in the command line.

```
node repl.js
```

```
> const n = 10
```

You can pass a string which shows when the REPL starts. The default is '> ' (with a trailing space), but we can define custom prompt.

```
// a Unix style prompt
const local = repl.start('$ ')
```

You can display a message while exiting the REPL

```
local.on('exit', () => {
  console.log('exiting repl')
  process.exit()
})
```

Node, accept arguments from the command line

You can pass any number of arguments when invoking a Node.js application using

```
node app.js
```

Arguments can be standalone or have a key and a value.

For example:

```
node app.js joe
```

or

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

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.

You can iterate over all the arguments (including the node path and the file path) using a loop:

```
process.argv.forEach((val, index) => {
  console.log(`${index}: ${val}`)
})
```

You can get only the additional arguments by creating a new array that excludes the first 2 params:

```
const args = process.argv.slice(2)
```

If you have one argument without an index name, like this:

```
node app.js joe
```

you can access it using

```
const args = process.argv.slice(2)
args[0]
```

In this case:

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

`args[0]` is `name=joe`, and you need to parse it. The best way to do so is by using the `minimist` library, which helps dealing with arguments:

```
const args = require('minimist')(process.argv.slice(2))

args.name // joe
```

Install the required `minimist` package using `npm` (lesson about the package manager comes later on).

```
npm install minimist
```

This time you need to use double dashes before each argument name:

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

Output to the command line using Node

Basic output using the console module

Node.js provides a `console` module which provides tons of very useful ways to interact with the command line.

It is basically the same as the `console` object you find in the browser.

The most basic and most used method is `console.log()`, which prints the string you pass to it to the console.

If you pass an object, it will render it as a string.

You can pass multiple variables to `console.log`, for example:

```
const x = 'x'
const y = 'y'
console.log(x, y)
```

and Node.js will print both.

We can also format pretty phrases by passing variables and a format specifier.

For example:

```
console.log('My %s has %d ears', 'cat', 2)
```

- `%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

Example:

```
console.log('%o', Number)
```

Clear the console

`console.clear()` clears the console (the behavior might depend on the console used)

Counting elements

`console.count()` is a handy method.

Take this code:

```
const x = 1
const y = 2
const z = 3
console.count(
  'The value of x is ' + x + ' and has been checked .. how many times?'
)
console.count(
  'The value of x is ' + x + ' and has been checked .. how many times?'
)
console.count(
  'The value of y is ' + y + ' and has been checked .. how many times?'
)
```

What happens is that `console.count()` will count the number of times a string is printed, and print the count next to it:

You can just count apples and oranges:

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

Reset counting

The console.countReset() method resets counter used with console.count().

We will use the apples and orange example to demonstrate this.

```
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)
})
```

Notice how the call to `console.countReset('orange')` resets the value counter to zero.

Print the stack trace

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()` :

```
const function2 = () => console.trace()
const function1 = () => function2()
function1()
```

This will print the stack trace. This is what's printed if we try this in the Node.js REPL:

```
Trace
    at function2 (repl:1:33)
    at function1 (repl:1:25)
    at repl:1:1
    at ContextifyScript.Script.runInThisContext (vm.js:44:33)
    at REPLServer.defaultEval (repl.js:239:29)
    at bound (domain.js:301:14)
    at REPLServer.runBound [as eval] (domain.js:314:12)
    at REPLServer.onLine (repl.js:440:10)
    at emitOne (events.js:120:20)
    at REPLServer.emit (events.js:210:7)
```

Calculate the time spent

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()
```

stdout and stderr

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.

Color the output

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.

Example:

```
console.log('\x1b[33m%s\x1b[0m', 'hi!')
```

You can try that in the Node.js REPL, and it will print `hi!` in yellow.

However, this is the low-level way to do this. The simplest way to go about coloring the console output is by using a library. Chalk is such a library, and in addition to coloring it also helps with other styling facilities, like making text bold, italic or underlined.

You install it with `npm install chalk@4`, then you can use it:

```
const chalk = require('chalk')

console.log(chalk.yellow('hi!'))
```

Using `chalk.yellow` is much more convenient than trying to remember the escape codes, and the code is much more readable.

Check the project link posted above for more usage examples.

Create a progress bar

Progress is an awesome package to create a progress bar in the console. Install it using `npm install progress`

This snippet creates a 10-step progress bar, and every 100ms one step is completed. When the bar completes we clear the interval:

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

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

Accept input from the command line in Node

Node.js has a built-in module system.

A Node.js file can import functionality exposed by other Node.js files.

When you want to import something you use

```
const library = require('./library')
```

to import the functionality exposed in the `library.js` file that resides in the current file folder.

In this file, 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.

You can do so in 2 ways.

The first is to assign an object to `module.exports`, which is an object provided out of the box by the module system, and this will make your file export *just that object*:

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

module.exports = car
```

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

The second way is to add the exported object as a property of `exports`. This way allows you to export multiple objects, functions or data:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
}

exports.car = car
```

or directly

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

And in the other file, you'll use it by referencing a property of your import:

```
const items = require('./car')

const { car } = items
```

or you can use a destructuring assignment:

```
const { car } = require('./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.

`require` will always return the object that `module.exports` points to.

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

module.exports = {
  brand: 'Tesla',
  model: 'Model S',
}

// app.js
const tesla = require('./car')
const ford = require('./car').car

console.log(tesla, ford)
```

This will print `{ brand: 'Tesla', model: 'Model S' } undefined` since the `require` function's return value has been updated to the object that `module.exports` points to, so *the property* that `exports` added can't be accessed.

An introduction to the npm package manager

Introduction to npm

`npm` is the standard package manager for Node.js.

In January 2017 over 350000 packages were reported being listed in the npm registry, making it the biggest single language code repository on Earth, and you can be sure there is a package for (almost!) everything.

It started as a way to download and manage dependencies of Node.js packages, but it has since become a tool used also in frontend JavaScript.

There are many things that `npm` does.

> **Yarn** and **pnpm** are alternatives to npm cli. You can check them out as well.

Downloads

`npm` manages downloads of dependencies of your project.

Installing all dependencies

If a project has a `package.json` file, by running

```
npm install
```

it will install everything the project needs, in the `node_modules` folder, creating it if it's not existing already.

Installing a single package

You can also install a specific package by running

```
npm install <package-name>
```

Furthermore, 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:

- `-D` or `--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

Shorthands of the flags can also be used:

- -S: --save
- -D: --save-dev
- -O: --save-optional

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. Read more about optional dependencies.

Updating packages

Updating is also made easy, by running

```
npm update
```

`npm` will check all packages for a newer version that satisfies your versioning constraints.

You can specify a single package to update as well:

```
npm update <package-name>
```

Versioning

In addition to plain downloads, `npm` also manages **versioning**, so you can specify any specific version of a package, or require a version higher or lower than what you need.

Many times you'll find that a library is only compatible with a major release of another library.

Or a bug in the latest release of a lib, still unfixed, is causing an issue.

Specifying an explicit version of a library also helps to keep everyone on the same exact version of a package, so that the whole team runs the same version until the `package.json` file is updated.

In all those cases, versioning helps a lot, and `npm` follows the semantic versioning (semver) standard.

Running Tasks

The package.json file supports a format for specifying command line tasks that can be run by using

```
npm run <task-name>
```

For example:

```
{
  "scripts": {
    "start-dev": "node lib/server-development",
    "start": "node lib/server-production"
  }
}
```

It's very common to use this feature to run Webpack:

```
{
  "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"
  }
}
```

So instead of typing those long commands, which are easy to forget or mistype, you can run

```
$ npm run watch
$ npm run dev
$ npm run prod
```

Where does npm install the packages?

When you install a package using `npm` you can perform 2 types of installation:

- a local install
- a global install

By default, when you type an `npm install` command, like:

```
npm install lodash
```

the package is installed in the current file tree, under the `node_modules` subfolder.

As this happens, `npm` also adds the `lodash` entry in the `dependencies` property of the `package.json` file present in the current folder.

A global installation is performed using the `-g` flag:

```
npm install -g lodash
```

When this happens, npm won't install the package under the local folder, but instead, it will use a global location.

Where, exactly?

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

On macOS or Linux this location could be `/usr/local/lib/node_modules`. On Windows it could be `C:\Users\YOU\AppData\Roaming\npm\node_modules`

If you use `nvm` to manage Node.js versions, however, that location would differ.

For example, if your username is 'joe' and you use `nvm`, then packages location will show as `/Users/joe/.nvm/versions/node/v8.9.0/lib/node_modules`.

How to use or execute a package installed using npm

When you install a package into your `node_modules` folder using `npm`, or also globally, how do you use it in your Node.js code?

Say you install `lodash`, the popular JavaScript utility library, using

```
npm install lodash
```

This is going to install the package in the local `node_modules` folder.

To use it in your code, you just need to import it into your program using `require`:

```
const _ = require('lodash')
```

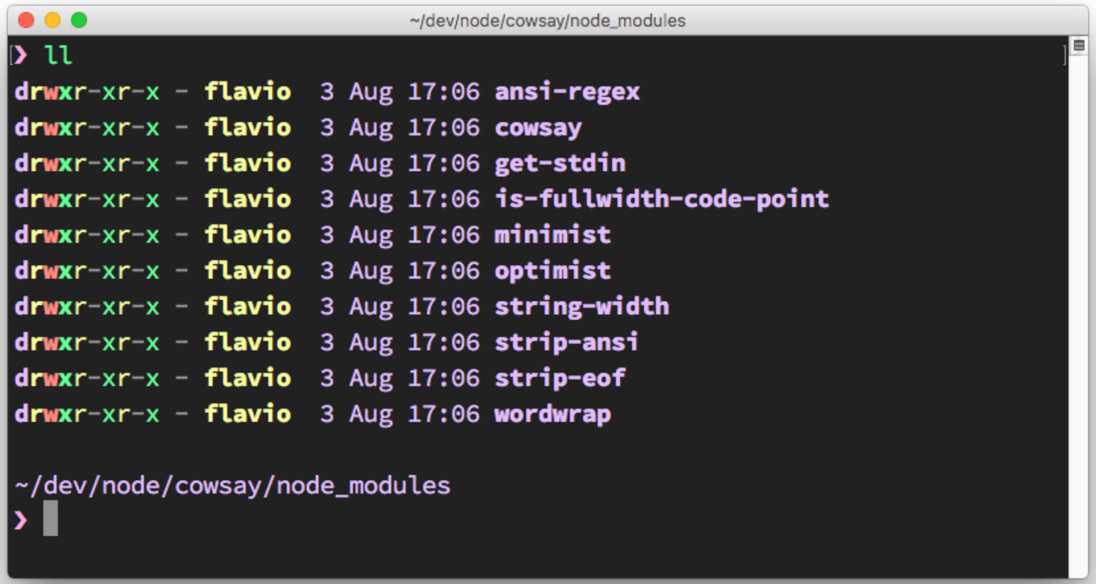
What if your package is an executable?

In this case, it will put the executable file under the `node_modules/.bin/` folder.

One easy way to demonstrate this is cowsay.

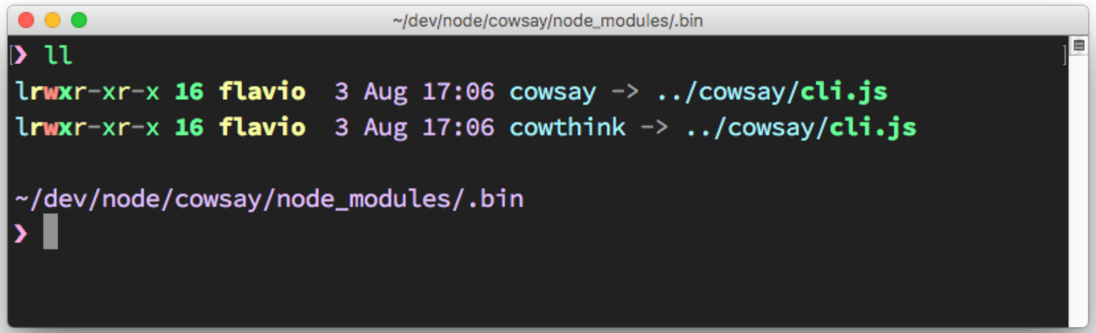
The cowsay package provides a command line program that can be executed to make a cow say something (and other animals as well 🦊).

When you install the package using `npm install cowsay`, it will install itself and a few dependencies in the `node_modules` folder:



```
~/dev/node/cowsay/node_modules
❯ ll
drwxr-xr-x - flavio  3 Aug 17:06 ansi-regex
drwxr-xr-x - flavio  3 Aug 17:06 cowsay
drwxr-xr-x - flavio  3 Aug 17:06 get-stdin
drwxr-xr-x - flavio  3 Aug 17:06 is-fullwidth-code-point
drwxr-xr-x - flavio  3 Aug 17:06 minimist
drwxr-xr-x - flavio  3 Aug 17:06 optimist
drwxr-xr-x - flavio  3 Aug 17:06 string-width
drwxr-xr-x - flavio  3 Aug 17:06 strip-ansi
drwxr-xr-x - flavio  3 Aug 17:06 strip-eof
drwxr-xr-x - flavio  3 Aug 17:06 wordwrap

~/dev/node/cowsay/node_modules
❯
```

There is a hidden `.bin` folder, which contains symbolic links to the cowsay binaries:



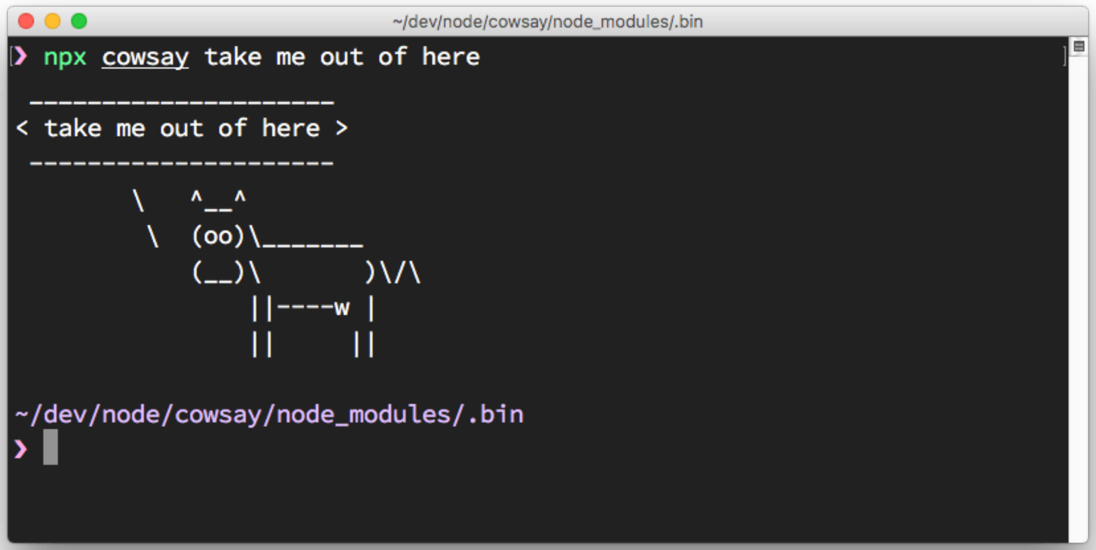
```
~/dev/node/cowsay/node_modules/.bin
❯ ll
lrwxr-xr-x 16 flavio  3 Aug 17:06 cowsay -> ../cowsay/cli.js
lrwxr-xr-x 16 flavio  3 Aug 17:06 cowthink -> ../cowsay/cli.js

~/dev/node/cowsay/node_modules/.bin
❯
```

How do you execute those?

You can of course type `./node_modules/.bin/cowsay` to run it, and it works, but `npx`, included in the recent versions of `npm` (since 5.2), is a much better option. You just run:

```
npx cowsay take me out of here
```

and `npx` will find the executable location.



```
❯ npx cowsay take me out of here
 ______________________
< take me out of here >
 ----------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

~/dev/node/cowsay/node_modules/.bin
❯
```

The package.json guide

If you work with JavaScript, or you've ever interacted with a JavaScript project, Node.js or a frontend project, you surely met the `package.json` file.

What's that for? What should you know about it, and what are some of the cool things you can do with it?

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 file structure

Here's an example package.json file:

```
{}
```

It's empty! There are no fixed requirements of what should be in a `package.json` file, for an application. The only requirement is that it respects the JSON format, otherwise it cannot be read by programs that try to access its properties programmatically.

If you're building a Node.js package that you want to distribute over `npm` things change radically, and you must have a set of properties that will help other people use it. We'll see more about this later on.

This is another package.json:

```
{
  "name": "test-project"
}
```

It defines a `name` property, which tells the name of the app, or package, that's contained in the same folder where this file lives.

Here's a much more complex example, which was extracted from a sample Vue.js application:

```
{
  "name": "test-project",
  "version": "1.0.0",
  "description": "A Vue.js project",
  "main": "src/main.js",
  "private": true,
  "scripts": {
    "dev": "webpack-dev-server --inline --progress --config build/webpack.
    "start": "npm run dev",
    "unit": "jest --config test/unit/jest.conf.js --coverage",
    "test": "npm run unit",
    "lint": "eslint --ext .js,.vue src test/unit",
    "build": "node build/build.js"
  },
  "dependencies": {
    "vue": "^2.5.2"
  },
  "devDependencies": {
    "autoprefixer": "^7.1.2",
    "babel-core": "^6.22.1",
    "babel-eslint": "^8.2.1",
    "babel-helper-vue-jsx-merge-props": "^2.0.3",
    "babel-jest": "^21.0.2",
    "babel-loader": "^7.1.1",
    "babel-plugin-dynamic-import-node": "^1.2.0",
    "babel-plugin-syntax-jsx": "^6.18.0",
    "babel-plugin-transform-es2015-modules-commonjs": "^6.26.0",
    "babel-plugin-transform-runtime": "^6.22.0",
    "babel-plugin-transform-vue-jsx": "^3.5.0",
    "babel-preset-env": "^1.3.2",
    "babel-preset-stage-2": "^6.22.0",
    "chalk": "^2.0.1",
    "copy-webpack-plugin": "^4.0.1",
    "css-loader": "^0.28.0",
    "eslint": "^4.15.0",
    "eslint-config-airbnb-base": "^11.3.0",
    "eslint-friendly-formatter": "^3.0.0",
    "eslint-import-resolver-webpack": "^0.8.3",
    "eslint-loader": "^1.7.1",
    "eslint-plugin-import": "^2.7.0",
    "eslint-plugin-vue": "^4.0.0",
    "extract-text-webpack-plugin": "^3.0.0",
    "file-loader": "^1.1.4",
    "friendly-errors-webpack-plugin": "^1.6.1",
    "html-webpack-plugin": "^2.30.1",
    "jest": "^22.0.4",
    "jest-serializer-vue": "^0.3.0",
    "node-notifier": "^5.1.2",
    "optimize-css-assets-webpack-plugin": "^3.2.0",
    "ora": "^1.2.0",
    "portfinder": "^1.0.13",
    "postcss-import": "^11.0.0",
    "postcss-loader": "^2.0.8",
    "postcss-url": "^7.2.1",
    "rimraf": "^2.6.0",
    "semver": "^5.3.0",
    "shelljs": "^0.7.6",
    "uglifyjs-webpack-plugin": "^1.1.1",
    "url-loader": "^0.5.8",
    "vue-jest": "^1.0.2",
    "vue-loader": "^13.3.0",
    "vue-style-loader": "^3.0.1",
    "vue-template-compiler": "^2.5.2",
    "webpack": "^3.6.0",
    "webpack-bundle-analyzer": "^2.9.0",
    "webpack-dev-server": "^2.9.1",
    "webpack-merge": "^4.1.0"
  },
  "engines": {
    "node": ">= 6.0.0",
    "npm": ">= 3.0.0"
  },
  "browserslist": ["> 1%", "last 2 versions", "not ie <= 8"]
}
```

there are *lots* of things going on here:

- `version` indicates the current version
- `name` sets the application/package name
- `description` is a brief description of the app/package
- `main` sets the entry point for the application
- `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

All those properties are used by either `npm` or other tools that we can use.

Properties breakdown

This section describes the properties you can use in detail. We refer to "package" but the same thing applies to local applications which you do not use as packages.

Most of those properties are only used on https://www.npmjs.com/, others by scripts that interact with your code, like `npm` or others.

name

Sets the package name.

Example:

```
"name": "test-project"
```

The name must be less than 214 characters, must not have spaces, it can only contain lowercase letters, hyphens (`-`) or underscores (`_`).

This is because when a package is published on `npm`, it gets its own URL based on this property.

If you published this package publicly on GitHub, a good value for this property is the GitHub repository name.

author

Lists the package author name

Example:

```
{
  "author": "Joe <joe@whatever.com> (https://whatever.com)"
}
```

Can also be used with this format:

```
{
  "author": {
    "name": "Joe",
    "email": "joe@whatever.com",
    "url": "https://whatever.com"
  }
}
```

contributors

As well as the author, the project can have one or more contributors. This property is an array that lists them.

Example:

```
{
  "contributors": ["Joe <joe@whatever.com> (https://whatever.com)"]
}
```

Can also be used with this format:

```
{
  "contributors": [
    {
      "name": "Joe",
      "email": "joe@whatever.com",
      "url": "https://whatever.com"
    }
  ]
}
```

bugs

Links to the package issue tracker, most likely a GitHub issues page

Example:

```
{
  "bugs": "https://github.com/whatever/package/issues"
}
```

homepage

Sets the package homepage

Example:

```
{
  "homepage": "https://whatever.com/package"
}
```

version

Indicates the current version of the package.

Example:

```
"version": "1.0.0"
```

This property follows the semantic versioning (semver) notation for versions, which means the version is always expressed with 3 numbers: `x.x.x`.

The first number is the major version, the second the minor version and the third is the patch version.

There is a meaning in these numbers: a release that only fixes bugs is a patch release, a release that introduces backward-compatible changes is a minor release, a major release can have breaking changes.

license

Indicates the license of the package.

Example:

```
"license": "MIT"
```

keywords

This property contains an array of keywords that associate with what your package does.

Example:

```
"keywords": [
  "email",
  "machine learning",
  "ai"
]
```

This helps people find your package when navigating similar packages, or when browsing the https://www.npmjs.com/ website.

description

This property contains a brief description of the package

Example:

```
"description": "A package to work with strings"
```

This is especially useful if you decide to publish your package to `npm` so that people can find out what the package is about.

repository

This property specifies where this package repository is located.

Example:

```
"repository": "github:whatever/testing",
```

Notice the `github` prefix. There are other popular services baked in:

```
"repository": "gitlab:whatever/testing",
```

```
"repository": "bitbucket:whatever/testing",
```

You can explicitly set the version control system:

```
"repository": {
  "type": "git",
  "url": "https://github.com/whatever/testing.git"
}
```

You can use different version control systems:

```
"repository": {
  "type": "svn",
  "url": "..."
}
```

main

Sets the entry point for the package.

When you import this package in an application, that's where the application will search for the module exports.

Example:

```
"main": "src/main.js"
```

private

if set to `true` prevents the app/package to be accidentally published on `npm`

Example:

```
"private": true
```

scripts

Defines a set of node scripts you can run

Example:

```
"scripts": {
  "dev": "webpack-dev-server --inline --progress --config build/webpack.de
  "start": "npm run dev",
  "unit": "jest --config test/unit/jest.conf.js --coverage",
  "test": "npm run unit",
  "lint": "eslint --ext .js,.vue src test/unit",
  "build": "node build/build.js"
}
```

These scripts are command line applications. You can run them by calling `npm run XXXX` or `yarn XXXX`, where `XXXX` is the command name. Example: `npm run dev`.

You can use any name you want for a command, and scripts can do literally anything you want.

dependencies

Sets a list of `npm` packages installed as dependencies.

When you install a package using npm or yarn:

```
npm install <PACKAGENAME>
yarn add <PACKAGENAME>
```

that package is automatically inserted in this list.

Example:

```
"dependencies": {
  "vue": "^2.5.2"
}
```

devDependencies

Sets a list of `npm` packages installed as development dependencies.

They differ from `dependencies` because they are meant to be installed only on a development machine, not needed to run the code in production.

When you install a package using npm or yarn:

```
npm install -D <PACKAGENAME>
yarn add --dev <PACKAGENAME>
```

that package is automatically inserted in this list.

Example:

```
"devDependencies": {
  "autoprefixer": "^7.1.2",
  "babel-core": "^6.22.1"
}
```

engines

Sets which versions of Node.js and other commands this package/app work on

Example:

```
"engines": {
  "node": ">= 6.0.0",
  "npm": ">= 3.0.0",
  "yarn": "^0.13.0"
}
```

browserslist

Is used to tell which browsers (and their versions) you want to support. It's referenced by Babel, Autoprefixer, and other tools, to only add the polyfills and fallbacks needed to the browsers you target.

Example:

```
"browserslist": [
  "> 1%",
  "last 2 versions",
  "not ie <= 8"
]
```

This configuration means you want to support the last 2 major versions of all browsers with at least 1% of usage (from the CanIUse.com stats), except IE8 and lower.

(see more)

Command-specific properties

The `package.json` file can also host command-specific configuration, for example for Babel, ESLint, and more.

Each has a specific property, like `eslintConfig` , `babel` and others. Those are command-specific, and you can find how to use those in the respective command/project documentation.

Package versions

You have seen in the description above version numbers like these: `~3.0.0` or `^0.13.0`. What do they mean, and which other version specifiers can you use?

That symbol specifies which updates your package accepts, from that dependency.

Given that using semver (semantic versioning) all versions have 3 digits, the first being the major release, the second the minor release and the third is the patch release, you have these "Rules".

You can combine most of the versions in ranges, like this: `1.0.0 || >=1.1.0 <1.2.0`, to either use 1.0.0 or one release from 1.1.0 up, but lower than 1.2.0.

The package-lock.json file

In version 5, npm introduced the `package-lock.json` file.

What's that? You probably know about the `package.json` file, which is much more common and has been around for much longer.

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.

This solves a very specific problem that `package.json` left unsolved. In package.json you can set which versions you want to upgrade to (patch or minor), using the **semver** notation, for example:

- if you write `~0.13.0`, you want to only update patch releases: `0.13.1` is ok, but `0.14.0` is not.
- if you write `^0.13.0`, you want to get updates that do not change the leftmost non-zero number: `0.13.1`, `0.13.2` and so on. If you write

`^1.13.0` , you will get patch and minor releases: `1.13.1` , `1.14.0` and so on up to `2.0.0` but not `2.0.0` .
- if you write `0.13.0` , that is the exact version that will be used, always

You don't commit to Git your node_modules folder, which is generally huge, and when you try to replicate the project on another machine by using the `npm install` command, if you specified the `~` syntax and a patch release of a package has been released, that one is going to be installed. Same for `^` and minor releases.

> If you specify exact versions, like `0.13.0` in the example, you are not affected by this problem.

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` .

This concept is not new, and other programming languages package managers (like Composer in PHP) have used a similar system for years.

The `package-lock.json` file needs to be committed to your Git repository, so it can be fetched by other people, if the project is public or you have collaborators, or if you use Git as a source for deployments.

The dependencies versions will be updated in the `package-lock.json` file when you run `npm update` .

An example

This is an example structure of a `package-lock.json` file we get when we run `npm install cowsay` in an empty folder:

```
{
  "requires": true,
  "lockfileVersion": 1,
  "dependencies": {
    "ansi-regex": {
      "version": "3.0.0",
      "resolved": "https://registry.npmjs.org/ansi-regex/-/ansi-regex-3.
0.0.tgz",
      "integrity": "sha1-7QMXwyIGT3lGbAKWa922Bas32Zg="
    },
    "cowsay": {
      "version": "1.3.1",
      "resolved": "https://registry.npmjs.org/cowsay/-/cowsay-1.3.1.tgz"
,
      "integrity": "sha512-3PVFe6FePVtPj1HTeLin9v8WyLl+VmM1l1H/5P+BTTDkM
Ajufp+0F9eLjzRnOHzVAYeIYFF5po5NjRrgefnRMQ==",
      "requires": {
        "get-stdin": "^5.0.1",
        "optimist": "~0.6.1",
        "string-width": "~2.1.1",
        "strip-eof": "^1.0.0"
      }
    },
    "get-stdin": {
      "version": "5.0.1",
      "resolved": "https://registry.npmjs.org/get-stdin/-/get-stdin-5.0.
1.tgz",
      "integrity": "sha1-Ei4WFZHiH/TFJTAwVpPyDmOTo5g="
    },
    "is-fullwidth-code-point": {
      "version": "2.0.0",
      "resolved": "https://registry.npmjs.org/is-fullwidth-code-point/-/
is-fullwidth-code-point-2.0.0.tgz",
      "integrity": "sha1-o7MKXE8ZkYMWeqq5O+764937ZU8="
    },
    "minimist": {
      "version": "0.0.10",
      "resolved": "https://registry.npmjs.org/minimist/-/minimist-0.0.10
.tgz",
      "integrity": "sha1-3j+YVD2/lggr5IrRoMfNqDYwHc8="
    },
    "optimist": {
      "version": "0.6.1",
      "resolved": "https://registry.npmjs.org/optimist/-/optimist-0.6.1.t
      "integrity": "sha1-2j6nRob6IaGaERwybpDrFaAZZoY=",
      "requires": {
        "minimist": "~0.0.1",
        "wordwrap": "~0.0.2"
      }
    },
    "string-width": {
      "version": "2.1.1",
      "resolved": "https://registry.npmjs.org/string-width/-/string-width-
      "integrity": "sha512-nOqH59deCq9SRHlxq1Aw85Jnt4w6KvLKqWVik6oA9ZklXL
      "requires": {
        "is-fullwidth-code-point": "^2.0.0",
        "strip-ansi": "^4.0.0"
      }
    },
    "strip-ansi": {
      "version": "4.0.0",
      "resolved": "https://registry.npmjs.org/strip-ansi/-/strip-ansi-4.0.
      "integrity": "sha1-qEeQIusaw2iocTibY1JixQXuNo8=",
      "requires": {
        "ansi-regex": "^3.0.0"
      }
    },
    "strip-eof": {
      "version": "1.0.0",
      "resolved": "https://registry.npmjs.org/strip-eof/-/strip-eof-1.0.0.
      "integrity": "sha1-u0P/VZim6wXYm1n80SnJgzE2Br8="
    },
    "wordwrap": {
      "version": "0.0.3",
      "resolved": "https://registry.npmjs.org/wordwrap/-/wordwrap-0.0.3.t
      "integrity": "sha1-o9XabNXAvAAI03I0u68b7WMFkQc="
    }
  }
}
```

We installed `cowsay`, which depends on

- `get-stdin`
- `optimist`
- `string-width`
- `strip-eof`

In turn, those packages require other packages, as we can see from the `requires` property that some have:

- `ansi-regex`
- `is-fullwidth-code-point`
- `minimist`
- `wordwrap`
- `strip-eof`

They are added in alphabetical order into the file, and each one has a `version` field, a `resolved` field that points to the package location, and an `integrity` string that we can use to verify the package.

Find the installed version of an npm package

To see the version of all installed npm packages, including their dependencies:

```
npm list
```

For example:

```
❯ npm list
/Users/joe/dev/node/cowsay
└─┬ cowsay@1.3.1
  ├── get-stdin@5.0.1
  ├─┬ optimist@0.6.1
  │ ├── minimist@0.0.10
  │ └── wordwrap@0.0.3
  ├─┬ string-width@2.1.1
  │ ├── is-fullwidth-code-point@2.0.0
  │ └─┬ strip-ansi@4.0.0
  │   └── ansi-regex@3.0.0
  └── strip-eof@1.0.0
```

You can also just open the `package-lock.json` file, but this involves some visual scanning.

`npm list -g` is the same, but for globally installed packages.

To get only your top-level packages (basically, the ones you told npm to install and you listed in the `package.json`), run `npm list --depth=0`:

```
❯ npm list --depth=0
/Users/joe/dev/node/cowsay
└── cowsay@1.3.1
```

You can get the version of a specific package by specifying its name:

```
❯ npm list cowsay
/Users/joe/dev/node/cowsay
└── cowsay@1.3.1
```

This also works for dependencies of packages you installed:

```
❯ npm list minimist
/Users/joe/dev/node/cowsay
└─┬ cowsay@1.3.1
  └─┬ optimist@0.6.1
    └── minimist@0.0.10
```

If you want to see what's the latest available version of the package on the npm repository, run `npm view [package_name] version`:

```
❯ npm view cowsay version

1.3.1
```

Install an older version of an npm package

You can install an old version of an npm package using the `@` syntax:

```
npm install <package>@<version>
```

Example:

```
npm install cowsay
```

installs version 1.3.1 (at the time of writing).

Install version 1.2.0 with:

```
npm install cowsay@1.2.0
```

The same can be done with global packages:

```
npm install -g webpack@4.16.4
```

You might also be interested in listing all the previous versions of a package. You can do it with `npm view <package> versions`:

```
❯ npm view cowsay versions

[ '1.0.0',
  '1.0.1',
  '1.0.2',
  '1.0.3',
  '1.1.0',
  '1.1.1',
  '1.1.2',
  '1.1.3',
  '1.1.4',
  '1.1.5',
  '1.1.6',
  '1.1.7',
  '1.1.8',
  '1.1.9',
  '1.2.0',
  '1.2.1',
  '1.3.0',
  '1.3.1' ]
```

Update all the Node dependencies to their latest version

How Packages Become Dependencies

When you install a package using `npm install <packagename>`, the latest version is downloaded to the `node_modules` folder. A corresponding entry is added to `package.json` and `package-lock.json` in the current folder.

npm determines the dependencies and installs their latest versions as well.

Let's say you install `cowsay`, a nifty command-line tool that lets you make a cow say *things*.

When you run `npm install cowsay`, this entry is added to the `package.json` file:

```
{
  "dependencies": {
    "cowsay": "^1.3.1"
  }
}
```

This is an extract of `package-lock.json` (nested dependencies were removed for clarity):

```
{
  "requires": true,
  "lockfileVersion": 1,
  "dependencies": {
    "cowsay": {
      "version": "1.3.1",
      "resolved": "https://registry.npmjs.org/cowsay/-/cowsay-1.3.1.tgz",
      "integrity": "sha512-3PVFe6FePVtPj1HTeLin9v8WyLl+VmM1l1H/5P+BTTDkMA
      "requires": {
        "get-stdin": "^5.0.1",
        "optimist": "~0.6.1",
        "string-width": "~2.1.1",
        "strip-eof": "^1.0.0"
      }
    }
  }
}
```

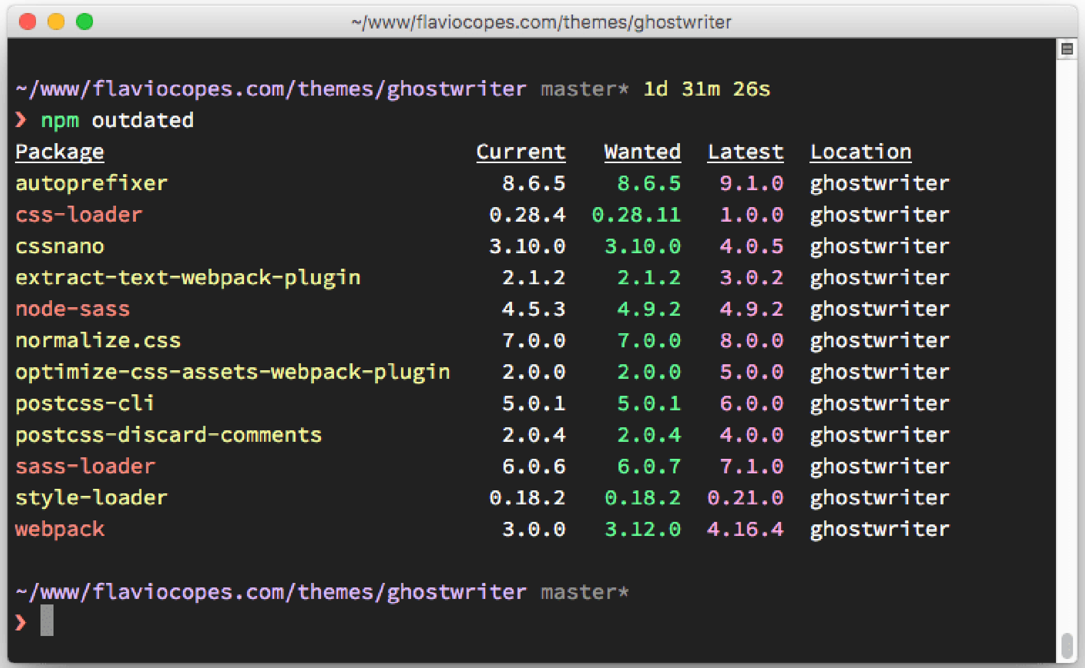
Now those 2 files tell us that we installed version `1.3.1` of cowsay, and our npm versioning rule for updates is `^1.3.1`. This means npm can update to patch and minor releases: `1.3.2`, `1.4.0` and so on.

If there is a new minor or patch release and we type `npm update`, the installed version is updated, and the `package-lock.json` file diligently filled with the new version.

Since npm version 5.0.0, `npm update` updates `package.json` with newer minor or patch versions. Use `npm update --no-save` to prevent modifying `package.json`.

To discover new package releases, use `npm outdated`.

Here's the list of a few outdated packages in a repository:



```
~/www/flaviocopes.com/themes/ghostwriter master* 1d 31m 26s
❯ npm outdated
Package                              Current  Wanted  Latest  Location
autoprefixer                           8.6.5   8.6.5   9.1.0  ghostwriter
css-loader                            0.28.4 0.28.11   1.0.0  ghostwriter
cssnano                               3.10.0  3.10.0   4.0.5  ghostwriter
extract-text-webpack-plugin            2.1.2   2.1.2   3.0.2  ghostwriter
node-sass                              4.5.3   4.9.2   4.9.2  ghostwriter
normalize.css                          7.0.0   7.0.0   8.0.0  ghostwriter
optimize-css-assets-webpack-plugin     2.0.0   2.0.0   5.0.0  ghostwriter
postcss-cli                            5.0.1   5.0.1   6.0.0  ghostwriter
postcss-discard-comments               2.0.4   2.0.4   4.0.0  ghostwriter
sass-loader                            6.0.6   6.0.7   7.1.0  ghostwriter
style-loader                          0.18.2  0.18.2  0.21.0  ghostwriter
webpack                                3.0.0  3.12.0  4.16.4  ghostwriter

~/www/flaviocopes.com/themes/ghostwriter master*
❯
```

Some of those updates are *major* releases. Running `npm update` won't help here. Major releases are *never* updated in this way because they (by definition) introduce breaking changes, and `npm` wants to save you trouble.

Update All Packages to the Latest Version

Leveraging npm-check-updates, you can upgrade all `package.json` dependencies to the latest version.

1. Install the `npm-check-updates` package globally:

```
npm install -g npm-check-updates
```

1. Now run `npm-check-updates` to upgrade all version hints in `package.json`, allowing installation of the new major versions:

```
ncu -u
```

1. Finally, run a standard install:

```
npm install
```

Semantic Versioning using npm

If there's one great thing in Node.js packages, it's that they all agreed on using Semantic Versioning for their version numbering.

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

The convention is adopted all across programming languages, and it is very important that every `npm` package adheres to it, because the whole system depends on that.

Why is that so important?

Because `npm` set some rules we can use in the `package.json` file to choose which versions it can update our packages to, when we run `npm update`.

The rules use these symbols:

- `^`

- `~`
- `>`
- `>=`
- `<`
- `<=`
- `=`
- `-`
- `||`

Let's see those 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`

You can combine some of those notations, for example use `1.0.0 || >=1.1.0 <1.2.0` to either use 1.0.0 or one release from 1.1.0 up, but lower than 1.2.0.

There are other rules, too:

- no symbol: you accept only that specific version you specify (`1.2.1`)
- `latest`: you want to use the latest version available

Uninstalling npm packages with `npm uninstall`

To uninstall a package you have previously installed **locally** (using `npm install <package-name>`), run

```
npm uninstall <package-name>
```

from the project root folder (the folder that contains the `node_modules` folder). This will update `dependencies`, `devDependencies`, `optionalDependencies`, and `peerDependencies` in both `package.json` and `package-lock.json` files.

Use `--no-save` option if you don't want to update the `package.json` and `package-lock.json` files.

If the package is installed **globally**, you need to add the `-g` / `--global` flag:

```
npm uninstall -g <package-name>
```

for example:

```
npm uninstall -g webpack
```

and you can run this command from anywhere you want on your system because, for global packages the current directory doesn't matter.

npm global or local packages

The main difference between local and global packages is this:

- **local packages** are installed in the directory where you run `npm install <package-name>`, and they are put in the `node_modules` folder under this directory

- **global packages** are all put in a single place in your system (exactly where depends on your setup), regardless of where you run `npm install -g <package-name>`

In your code you can only require local packages:

```
require('package-name')
```

so when should you install in one way or another?

In general, **all packages should be installed locally**.

This makes sure you can have dozens of applications in your computer, all running a different version of each package if needed.

Updating a global package would make all your projects use the new release, and as you can imagine this might cause nightmares in terms of maintenance, as some packages might break compatibility with further dependencies, and so on.

All projects have their own local version of a package, even if this might appear like a waste of resources, it's minimal compared to the possible negative consequences.

A package **should be installed globally** when it provides an executable command that you run from the shell (CLI), and it's reused across projects.

You can also install executable commands locally and run them using npx, but some packages are just better installed globally.

Great examples of popular global packages which you might know are

- `npm`
- `vue-cli`
- `grunt-cli`
- `mocha`
- `react-native-cli`
- `gatsby-cli`

- `forever`
- `nodemon`

You probably have some packages installed globally already on your system. You can see them by running

```
npm list -g --depth 0
```

on your command line.

npm dependencies and devDependencies

When you install an npm package using `npm install <package-name>`, you are installing it as a **dependency**.

The package is automatically listed in the package.json file, under the `dependencies` list (as of npm 5. previously, you had to manually specify `--save`).

When you add the `-D` flag, or `--save-dev`, you are installing it as a development dependency, which adds it to the `devDependencies` list.

Development dependencies are intended as development-only packages, that are unneeded in production. For example testing packages, webpack or Babel.

When you go in production, if you type `npm install` and the folder contains a `package.json` file, they are installed, as npm assumes this is a development deploy.

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

The npx Node Package Runner

`npx` is a very powerful command that's been available in **npm** starting version 5.2, released in July 2017.

> If you don't want to install npm, you can install npx as a standalone package

`npx` lets you run code built with Node.js and published through the npm registry.

Easily run local commands

Node.js developers used to publish most of the executable commands as global packages, in order for them to be in the path and executable immediately.

This was a pain because you could not really install different versions of the same command.

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

A typical demonstration of using `npx` is through the `cowsay` command. `cowsay` will print a cow saying what you wrote in the command. For example:

`cowsay "Hello"` will print

```
 _______
< Hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

This only works if you have the `cowsay` command globally installed from npm previously. Otherwise you'll get an error when you try to run the command.

`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:

```
npx cowsay "Hello"
```

will do the job.

Now, this is a funny useless command. Other scenarios include:

- running the `vue` CLI tool to create new applications and run them: `npx @vue/cli create my-vue-app`
- creating a new React app using `create-react-app`: `npx create-react-app my-react-app`

and many more.

Run some code using a different Node.js version

Use the `@` to specify the version, and combine that with the `node` npm package:

```
npx node@10 -v #v10.18.1
npx node@12 -v #v12.14.1
```

This helps to avoid tools like `nvm` or the other Node.js version management tools.

Run arbitrary code snippets directly from a URL

`npx` does not limit you to the packages published on the npm registry.

You can run code that sits in a GitHub gist, for example:

```
npx https://gist.github.com/zkat/4bc19503fe9e9309e2bfaa2c58074d32
```

Of course, you need to be careful when running code that you do not control, as with great power comes great responsibility.

The Node.js Event Loop

Introduction

The **Event Loop** is one of the most important aspects to understand about Node.js.

Why is this so important? Because it explains how Node.js can be asynchronous and have non-blocking I/O, and so it explains basically the "killer feature" of Node.js, the thing that made it this successful.

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

This is a limitation that's actually very helpful, as it simplifies a lot how you program without worrying about concurrency issues.

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.

The environment manages multiple concurrent event loops, to handle API calls for example. Web Workers run in their own event loop as well.

You mainly need to be concerned that *your code* will run on a single event loop, and write code with this thing in mind to avoid blocking it.

Blocking the event loop

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. Being blocking is the exception, and this is why JavaScript is based so much on callbacks, and more recently on promises and async/await.

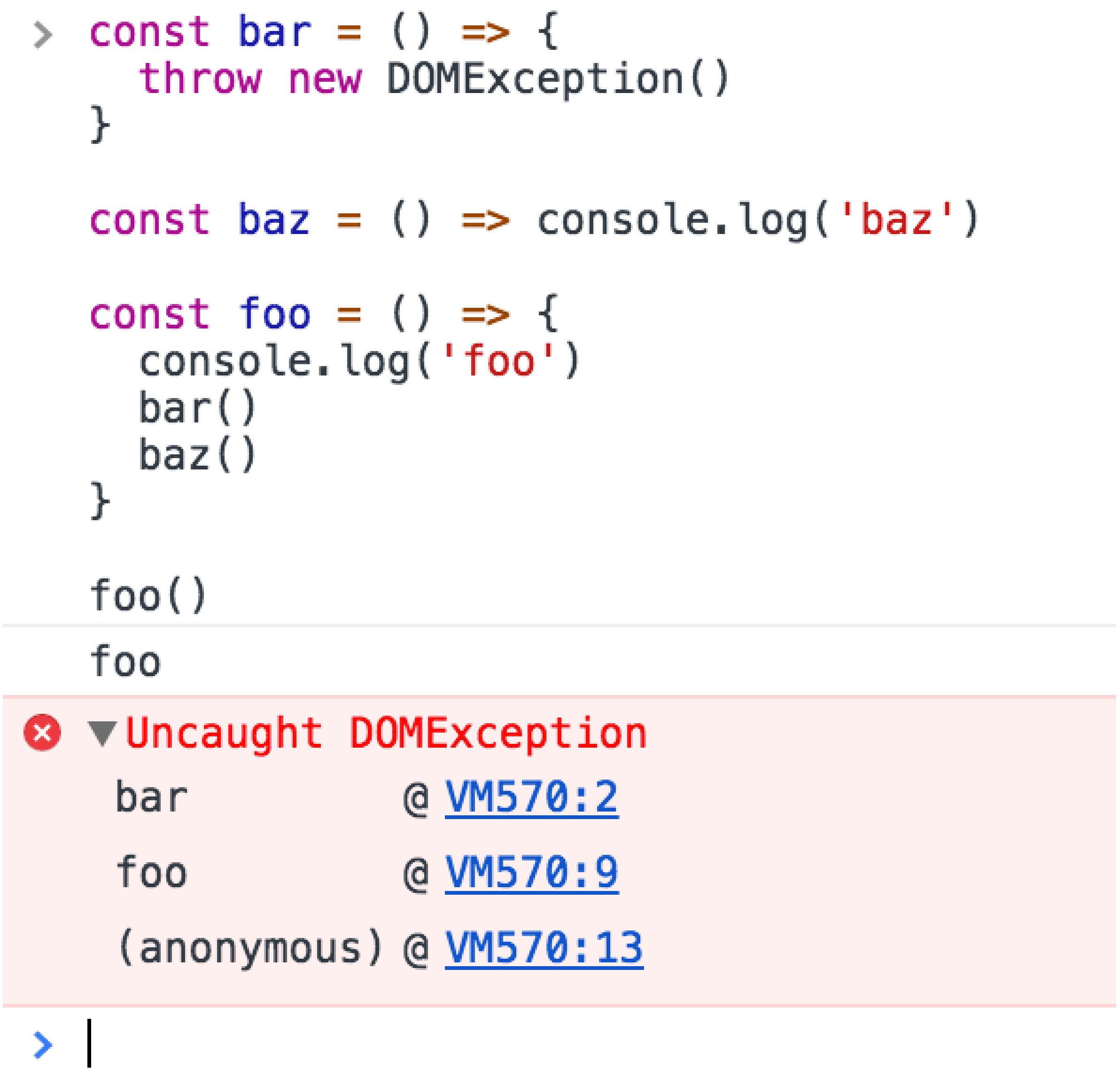
The call stack

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.

While doing so, it adds any function call it finds in the call stack and executes each one in order.

You know the error stack trace you might be familiar with, in the debugger or in the browser console? The browser looks up the function names in the call stack to inform you which function originates the current call:



```
> const bar = () => {
    throw new DOMException()
  }

  const baz = () => console.log('baz')

  const foo = () => {
    console.log('foo')
    bar()
    baz()
  }

  foo()
  foo
✖ ▼Uncaught DOMException
    bar         @ VM570:2
    foo         @ VM570:9
    (anonymous) @ VM570:13
> |
```

A simple event loop explanation

Let's pick an example:

```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  bar()
  baz()
}

foo()
```

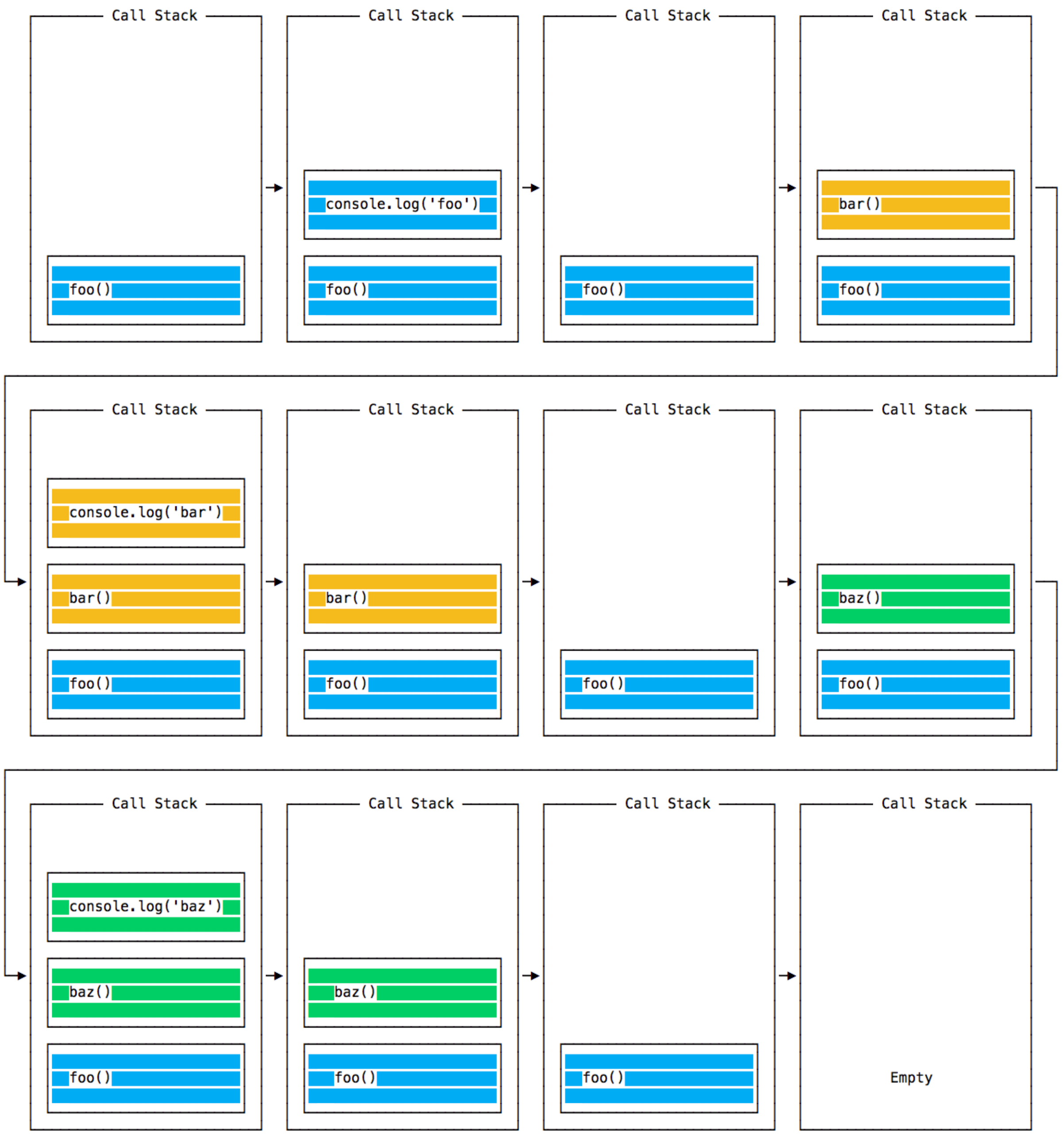
This code prints

```
foo
bar
baz
```

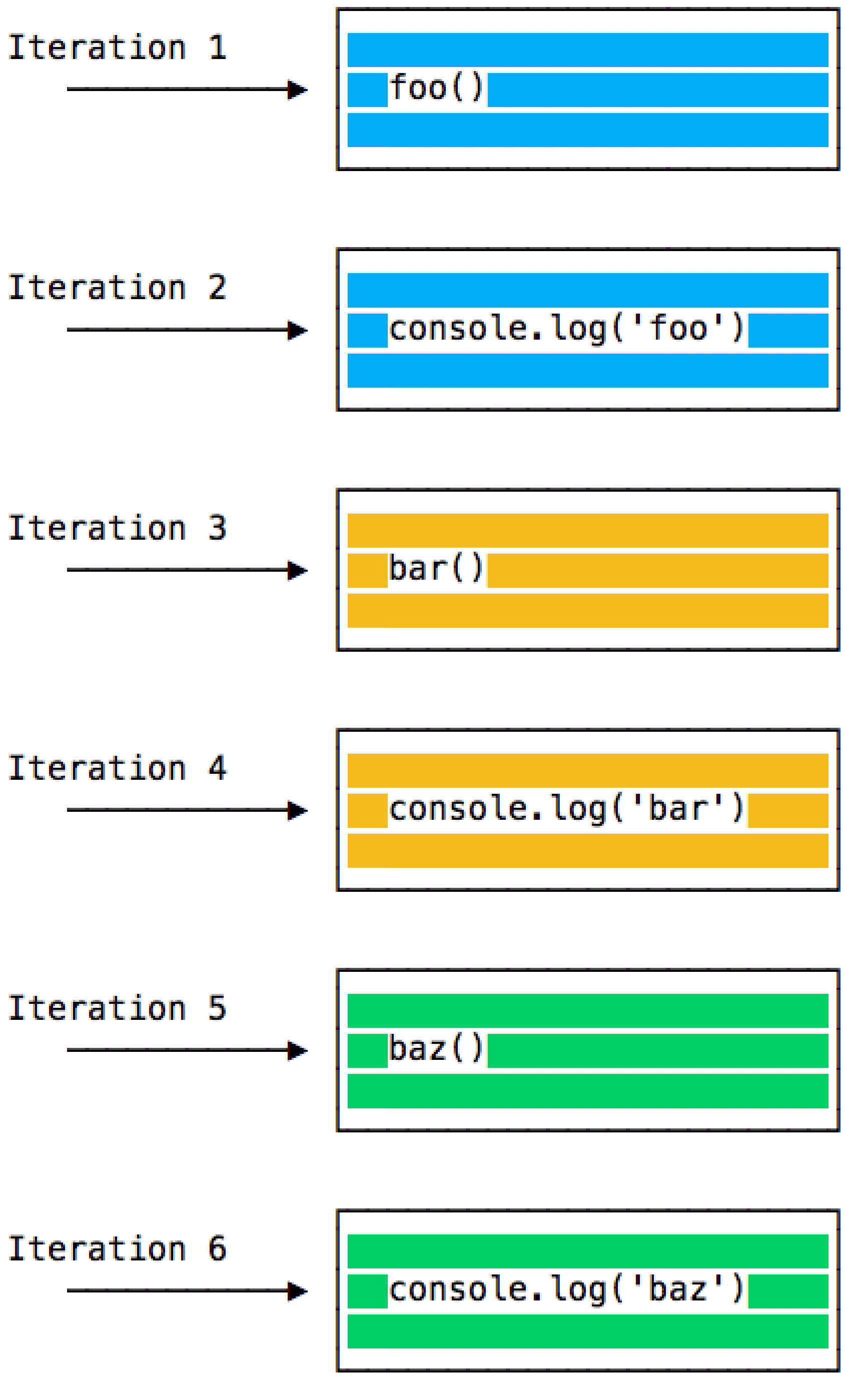
as expected.

When this code runs, first `foo()` is called. Inside `foo()` we first call `bar()`, then we call `baz()`.

At this point the call stack looks like this:



The event loop on every iteration looks if there's something in the call stack, and executes it:



until the call stack is empty.

Queuing function execution

The above example looks normal, there's nothing special about it: JavaScript finds things to execute, runs them in order.

Let's see how to defer a function until the stack is clear.

The use case of `setTimeout(() => {}, 0)` is to call a function, but execute it once every other function in the code has executed.

Take this example:

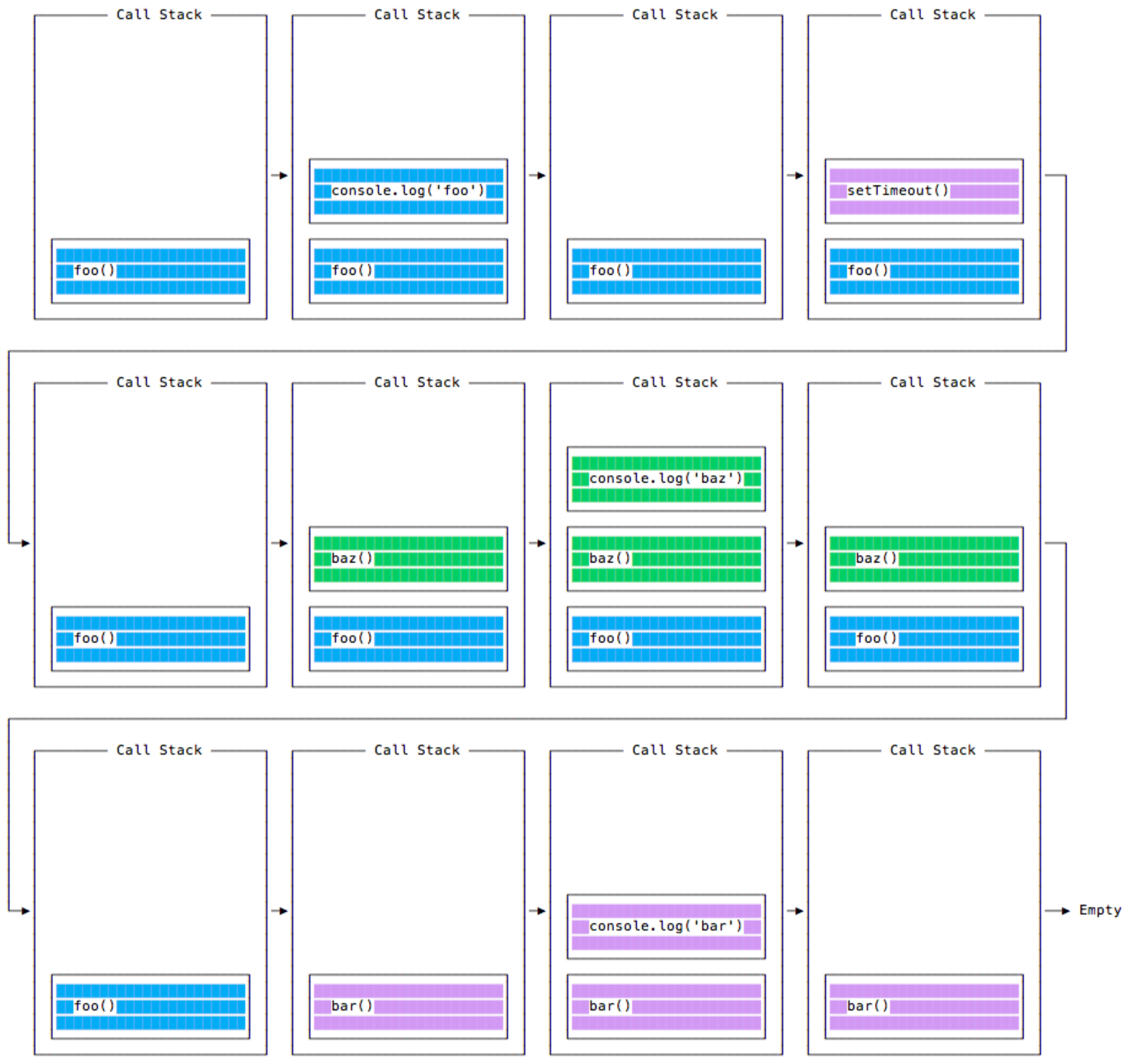
```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  setTimeout(bar, 0)
  baz()
}

foo()
```

This code prints, maybe surprisingly:

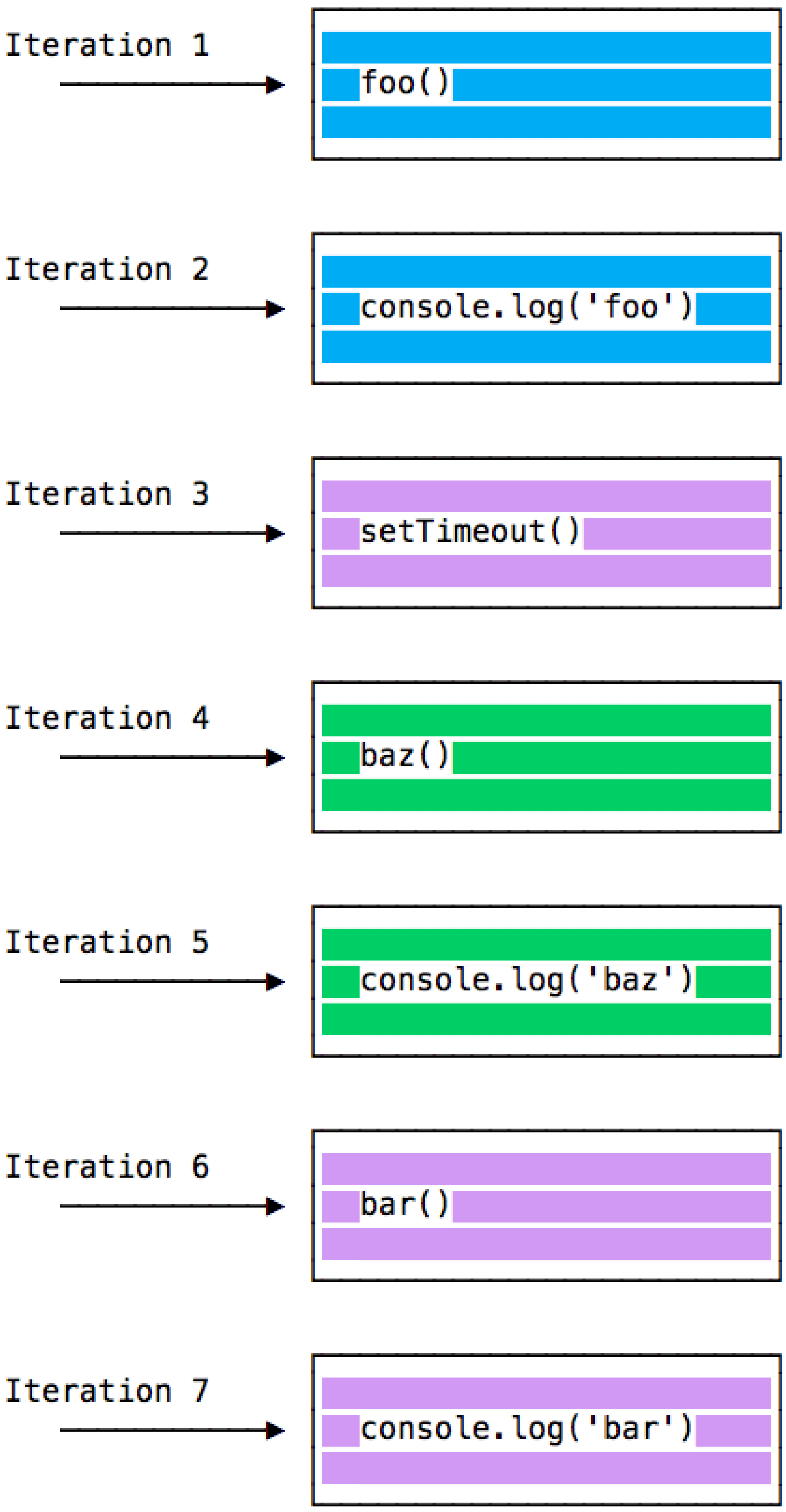
```
foo
baz
bar
```

When this code runs, first foo() is called. Inside foo() we first call setTimeout, passing `bar` as an argument, and we instruct it to run immediately as fast as it can, passing 0 as the timer. Then we call baz().

At this point the call stack looks like this:



Here is the execution order for all the functions in our program:



Why is this happening?

The Message Queue

When setTimeout() is called, the Browser or Node.js starts the timer. Once the timer expires, in this case immediately as we put 0 as the timeout, the callback function is put in the **Message Queue**.

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.

Promises that resolve before the current function ends will be executed right after the current function.

Similar to a rollercoaster ride at an amusement park: the message queue puts you at the back of the queue, behind all the other people, where you will have to wait for your turn, while the job queue is the fastpass ticket that lets you take another ride right after you finished the previous one.

Example:

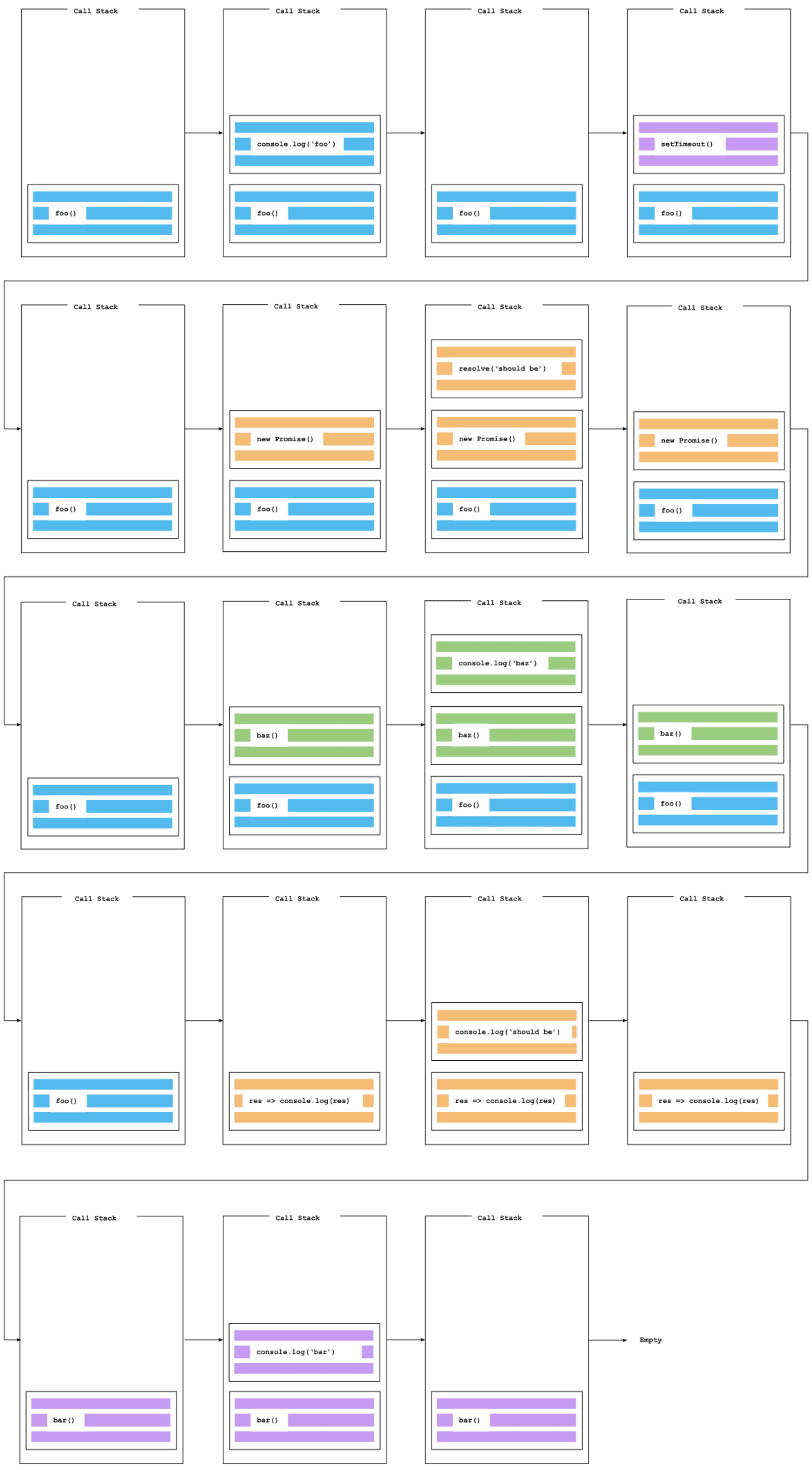
```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  setTimeout(bar, 0)
  new Promise((resolve, reject) =>
    resolve('should be right after baz, before bar')
  ).then((resolve) => console.log(resolve))
  baz()
}

foo()
```

This prints

```
foo
baz
should be right after baz, before bar
bar
```

That's a big difference between Promises (and Async/await, which is built on promises) and plain old asynchronous functions through `setTimeout()` or other platform APIs.

Finally, here's what the call stack looks like for the example above:


Call Stack
foo()
console.log('foo')
setTimeout()
new Promise()
resolve('should be')
baz()
console.log('baz')
res => console.log(res)
console.log('should be')
bar()
console.log('bar')
Empty

Understanding process.nextTick()

As you try to understand the Node.js event loop, one important part of it is `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:

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

The event loop is busy processing the current function code.

When this operation ends, the JS engine runs all the functions passed to `nextTick` calls during that operation.

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.

Calling `setTimeout(() => {}, 0)` will execute the function at the end of next tick, much later than when using `nextTick()` which prioritizes the call and executes it just before the beginning of the next tick.

Use `nextTick()` when you want to make sure that in the next event loop iteration that code is already executed.

Understanding setImmediate()

When you want to execute some piece of code asynchronously, but as soon as possible, one option is to use the `setImmediate()` function provided by Node.js:

```
setImmediate(() => {
  // run something
})
```

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()` and `Promise.then()` ?

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.

A `process.nextTick` callback is added to `process.nextTick queue` . A `Promise.then()` callback is added to `promises microtask queue` . A `setTimeout` , `setImmediate` callback is added to `macrotask queue` .

Event loop executes tasks in `process.nextTick queue` first, and then executes `promises microtask queue` , and then executes `macrotask queue` .

Here is an example to show the order between `setImmediate()` , `process.nextTick()` and `Promise.then()` :

```
const baz = () => console.log('baz')
const foo = () => console.log('foo')
const zoo = () => console.log('zoo')
const start = () => {
  console.log('start')
  setImmediate(baz)
  new Promise((resolve, reject) => {
    resolve('bar')
  }).then((resolve) => {
    console.log(resolve)
    process.nextTick(zoo)
  })
  process.nextTick(foo)
}
start()

// start foo bar zoo baz
```

This code will first call `start()`, then call `foo()` in `process.nextTick queue`. After that, it will handle `promises microtask queue`, which prints `bar` and adds `zoo()` in `process.nextTick queue` at the same time. Then it will call `zoo()` which has just been added. In the end, the `baz()` in `macrotask queue` is called.

Discover JavaScript Timers

`setTimeout()`

When writing JavaScript code, you might want to delay the execution of a function.

This is the job of `setTimeout`. You specify a callback function to execute later, and a value expressing how later you want it to run, in milliseconds:

```
setTimeout(() => {
  // runs after 2 seconds
}, 2000)

setTimeout(() => {
  // runs after 50 milliseconds
}, 50)
```

This syntax defines a new function. You can call whatever other function you want in there, or you can pass an existing function name, and a set of parameters:

```
const myFunction = (firstParam, secondParam) => {
  // do something
}

// runs after 2 seconds
setTimeout(myFunction, 2000, firstParam, secondParam)
```

`setTimeout` returns the timer id. This is generally not used, but you can store this id, and clear it if you want to delete this scheduled function execution:

```
const id = setTimeout(() => {
  // should run after 2 seconds
}, 2000)

// I changed my mind
clearTimeout(id)
```

Zero delay

If you specify the timeout delay to `0`, the callback function will be executed as soon as possible, but after the current function execution:

```
setTimeout(() => {
  console.log('after ')
}, 0)

console.log(' before ')
```

This code will print

```
before
after
```

This is especially useful to avoid blocking the CPU on intensive tasks and let other functions be executed while performing a heavy calculation, by queuing functions in the scheduler.

> Some browsers (IE and Edge) implement a `setImmediate()` method that does this same exact functionality, but it's not standard and unavailable on other browsers. But it's a standard function in Node.js.

`setInterval()`

`setInterval` is a function 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):

```
setInterval(() => {
  // runs every 2 seconds
}, 2000)
```

The function above runs every 2 seconds unless you tell it to stop, using `clearInterval`, passing it the interval id that `setInterval` returned:

```
const id = setInterval(() => {
  // runs every 2 seconds
}, 2000)

clearInterval(id)
```

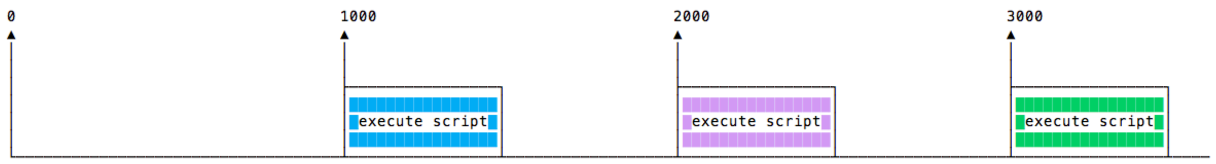
It's common to call `clearInterval` inside the setInterval callback function, to let it auto-determine if it should run again or stop. For example this code runs something unless App.somethingIWait has the value `arrived` :

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

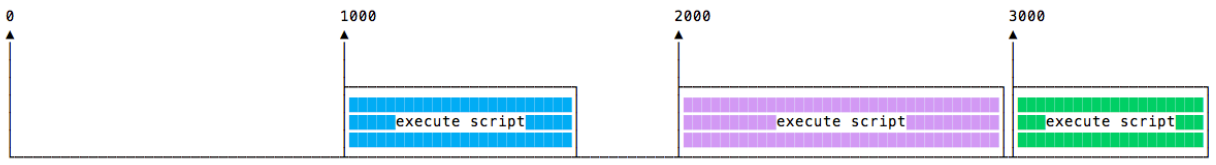
Recursive setTimeout

`setInterval` starts a function every n milliseconds, without any consideration about when a function finished its execution.

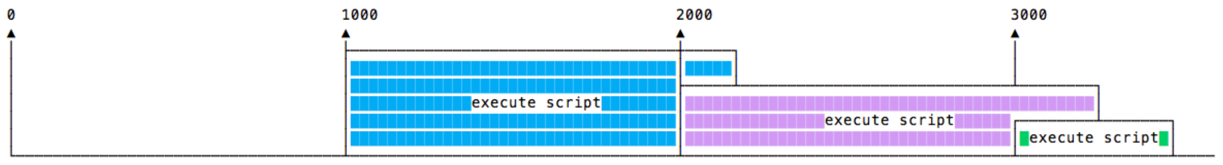
If a function always takes the same amount of time, it's all fine:



Maybe the function takes different execution times, depending on network conditions for example:



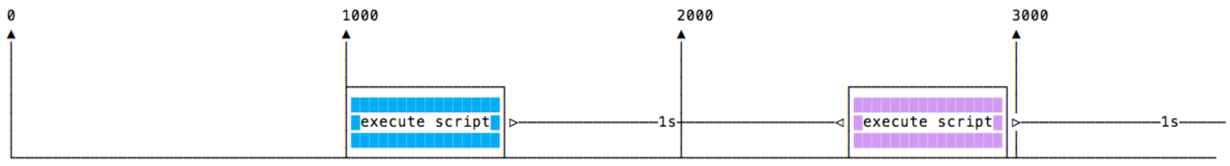
And maybe one long execution overlaps the next one:



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)
```

to achieve this scenario:



`setTimeout` and `setInterval` are available in Node.js, through the Timers module.

Node.js also provides `setImmediate()`, which is equivalent to using `setTimeout(() => {}, 0)`, mostly used to work with the Node.js Event Loop.

The Node Event emitter

If you worked with JavaScript in the browser, you know how much of the interaction of the user is handled through events: mouse clicks, keyboard button presses, reacting to mouse movements, and so on.

On the backend side, Node.js offers us the option to build a similar system using the `events` module.

This module, in particular, offers the `EventEmitter` class, which we'll use to handle our events.

You initialize that using

```
const EventEmitter = require('events')

const eventEmitter = new EventEmitter()
```

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

For example, let's create a `start` event, and as a matter of providing a sample, we react to that by just logging to the console:

```
eventEmitter.on('start', () => {
  console.log('started')
})
```

When we run

```
eventEmitter.emit('start')
```

the event handler function is triggered, and we get the console log.

You can pass arguments to the event handler by passing them as additional arguments to `emit()`:

```
eventEmitter.on('start', (number) => {
  console.log(`started ${number}`)
})

eventEmitter.emit('start', 23)
```

Multiple arguments:

```
eventEmitter.on('start', (start, end) => {
  console.log(`started from ${start} to ${end}`)
})

eventEmitter.emit('start', 1, 100)
```

The EventEmitter object also exposes several other methods to interact with events, like

- `once()` : add a one-time listener
- `removeListener()` / `off()` : remove an event listener from an event
- `removeAllListeners()` : remove all listeners for an event

You can read all their details on the events module page at https://nodejs.org/api/events.html

Working with file descriptors in Node

Before you're able to interact with a file that sits in your filesystem, you must get a file descriptor.

A file descriptor is a reference to an open file, a number (fd) returned by opening the file using the `open()` method offered by the `fs` module. This number (`fd`) uniquely identifies an open file in operating system:

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

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

Notice the `r` we used as the second parameter to the `fs.open()` call.

That flag means we open the file for reading.

Other flags you'll commonly use are:

- `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.

You can also open the file by using the `fs.openSync` method, which returns the file descriptor, instead of providing it in a callback:

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

try {
  const fd = fs.openSync('/Users/joe/test.txt', 'r')
} catch (err) {
  console.error(err)
}
```

Once you get the file descriptor, in whatever way you choose, you can perform all the operations that require it, like calling `fs.close()` and many other operations that interact with the filesystem.

You can also open the file by using the promise-based `fsPromises.open` method offered by the `fs/promises` module.

The `fs/promises` module is available starting only from Node.js v14. Before v14, after v10, you can use `require('fs').promises` instead. Before v10, after v8, you can use `util.promisify` to convert `fs` methods into promise-based methods.

```
const fs = require('fs/promises')
// Or const fs = require('fs').promises before v14.
async function example() {
  let filehandle
  try {
    filehandle = await fs.open('/Users/joe/test.txt', 'r')
    console.log(filehandle.fd)
    console.log(await filehandle.readFile({ encoding: 'utf8' }))
  } finally {
    await filehandle.close()
  }
}
example()
```

Here is an example of `util.promisify` :

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

async function example() {
  const open = util.promisify(fs.open)
  const fd = await open('/Users/joe/test.txt', 'r')
}
example()
```

To see more details about the `fs/promises` module, please check fs/promises API.

Node file stats

Every file comes with a set of details that we can inspect using Node.js.

In particular, using the `stat()` method provided by the `fs` module.

You call it passing a file path, and once Node.js gets the file details it will call the callback function you pass, with 2 parameters: an error message, and the file stats:

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

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

Node.js also provides a sync method, which blocks the thread until the file stats are ready:

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

try {
  const stats = fs.statSync('/Users/joe/test.txt')
} catch (err) {
  console.error(err)
}
```

The file information is included in the stats variable. What kind of information can we extract using the stats?

A lot, including:

- 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` .

There are other advanced methods, but the bulk of what you'll use in your day-to-day programming is this.

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

fs.stat('/Users/joe/test.txt', (err, stats) => {
  if (err) {
    console.error(err)
    return
  }

  stats.isFile() // true
  stats.isDirectory() // false
  stats.isSymbolicLink() // false
  stats.size // 1024000 //= 1MB
})
```

You can also use promise-based `fsPromises.stat()` method offered by the `fs/promises` module if you like:

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

async function example() {
  try {
    const stats = await fs.stat('/Users/joe/test.txt')
    stats.isFile() // true
    stats.isDirectory() // false
    stats.isSymbolicLink() // false
    stats.size // 1024000 //= 1MB
  } catch (err) {
    console.log(err)
  }
}
example()
```

Node File Paths

Every file in the system has a path.

On Linux and macOS, a path might look like:

`/users/joe/file.txt`

while Windows computers are different, and have a structure such as:

`C:\users\joe\file.txt`

You need to pay attention when using paths in your applications, as this difference must be taken into account.

You include this module in your files using

```
const path = require('path')
```

and you can start using its methods.

Getting information out of a path

Given a path, you can extract information out of it using those methods:

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

Example:

```
const notes = '/users/joe/notes.txt'

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

You can get the file name without the extension by specifying a second argument to `basename` :

```
path.basename(notes, path.extname(notes)) // notes
```

Working with paths

You can join two or more parts of a path by using `path.join()` :

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

You can get the absolute path calculation of a relative path using `path.resolve()`:

```
path.resolve('joe.txt') // '/Users/joe/joe.txt' if run from my home folde
```

In this case Node.js will 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:

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

If the first parameter starts with a slash, that means it's an absolute path:

```
path.resolve('/etc', 'joe.txt') // '/etc/joe.txt'
```

`path.normalize()` is another useful function, that will try and calculate the actual path, when it contains relative specifiers like `.` or `..`, or double slashes:

```
path.normalize('/users/joe/..//test.txt') // '/users/test.txt'
```

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

Reading files with Node

The simplest way to read a file in Node.js is to use the `fs.readFile()` method, passing it the file path, encoding and a callback function that will be called with the file data (and the error):

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

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

Alternatively, you can use the synchronous version `fs.readFileSync()` :

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

try {
  const data = fs.readFileSync('/Users/joe/test.txt', 'utf8')
  console.log(data)
} catch (err) {
  console.error(err)
}
```

You can also use the promise-based `fsPromises.readFile()` method offered by the `fs/promises` module:

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

async function example() {
  try {
    const data = await fs.readFile('/Users/joe/test.txt', { encoding: 'ut
    console.log(data)
  } catch (err) {
    console.log(err)
  }
}
example()
```

All three of `fs.readFile()` , `fs.readFileSync()` and `fsPromises.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.

Writing files with Node

The easiest way to write to files in Node.js is to use the `fs.writeFile()` API.

Example:

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

const content = 'Some content!'

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

Alternatively, you can use the synchronous version `fs.writeFileSync()`:

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

const content = 'Some content!'

try {
  fs.writeFileSync('/Users/joe/test.txt', content)
  // file written successfully
} catch (err) {
  console.error(err)
}
```

You can also use the promise-based `fsPromises.writeFile()` method offered by the `fs/promises` module:

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

async function example() {
  try {
    const content = 'Some content!'
    await fs.writeFile('/Users/joe/test.txt', content)
  } catch (err) {
    console.log(err)
  }
}
example()
```

By default, this API will **replace the contents of the file** if it does already exist.

You can modify the default by specifying a flag:

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

The flags you'll likely use are

- `r+` open the file for reading and writing
- `w+` open the file for reading and writing, positioning the stream at the beginning of the file. The file is created if it does not exist
- `a` open the file for writing, positioning the stream at the end of the file. The file is created if it does not exist
- `a+` open the file for reading and writing, positioning the stream at the end of the file. The file is created if it does not exist

(you can find more flags at https://nodejs.org/api/fs.html#fs_file_system_flags)

Append to a file

A handy method to append content to the end of a file is `fs.appendFile()` (and its `fs.appendFileSync()` counterpart):

```
const content = 'Some content!'

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

Here is a `fsPromises.appendFile()` example:

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

async function example() {
  try {
    const content = 'Some content!'
    await fs.appendFile('/Users/joe/test.txt', content)
  } catch (err) {
    console.log(err)
  }
}
example()
```

Using streams

All those methods write the full content to the file before returning the control back to your program (in the async version, this means executing the callback)

In this case, a better option is to write the file content using streams.

Working with folders

The Node.js `fs` core module provides many handy methods you can use to work with folders.

Check if a folder exists

Use `fs.access()` (and its promise-based `fsPromises.access()` counterpart) to check if the folder exists and Node.js can access it with its permissions.

Create a new folder

Use `fs.mkdir()` or `fs.mkdirSync()` or `fsPromises.mkdir()` to create a new folder.

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

const folderName = '/Users/joe/test'

try {
  if (!fs.existsSync(folderName)) {
    fs.mkdirSync(folderName)
  }
} catch (err) {
  console.error(err)
}
```

Read the content of a directory

Use `fs.readdir()` or `fs.readdirSync()` or `fsPromises.readdir()` to read the contents of a directory.

This piece of code reads the content of a folder, both files and subfolders, and returns their relative path:

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

const folderPath = '/Users/joe'

fs.readdirSync(folderPath)
```

You can get the full path:

```
fs.readdirSync(folderPath).map((fileName) => {
  return path.join(folderPath, fileName)
})
```

You can also filter the results to only return the files, and exclude the folders:

```
const isFile = (fileName) => {
  return fs.lstatSync(fileName).isFile()
}

fs.readdirSync(folderPath)
  .map((fileName) => {
    return path.join(folderPath, fileName)
  })
  .filter(isFile)
```

Rename a folder

Use `fs.rename()` or `fs.renameSync()` or `fsPromises.rename()` to rename folder. The first parameter is the current path, the second the new path:

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

fs.rename('/Users/joe', '/Users/roger', (err) => {
  if (err) {
    console.error(err)
  }
  // done
})
```

`fs.renameSync()` is the synchronous version:

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

try {
  fs.renameSync('/Users/joe', '/Users/roger')
} catch (err) {
  console.error(err)
}
```

`fsPromises.rename()` is the promise-based version:

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

async function example() {
  try {
    await fs.rename('/Users/joe', '/Users/roger')
  } catch (err) {
    console.log(err)
  }
}
example()
```

Remove a folder

Use `fs.rmdir()` or `fs.rmdirSync()` or `fsPromises.rmdir()` to remove a folder.

Removing a folder that has content can be more complicated than you need. You can pass the option `{ recursive: true }` to recursively remove the contents.

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

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

  console.log(`${dir} is deleted!`)
})
```

> **NOTE:** In Node `v16.x` the option `recursive` is **deprecated** for `fs.rmdir` of callback API, instead use `fs.rm` to delete folders that have content in them:

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

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

  console.log(`${dir} is deleted!`)
})
```

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.

In this case the `remove()` method is what you want.

Install it using

```
npm install fs-extra
```

and use it like this:

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

const folder = '/Users/joe'

fs.remove(folder, (err) => {
  console.error(err)
})
```

It can also be used with promises:

```
fs.remove(folder)
  .then(() => {
    // done
  })
  .catch((err) => {
    console.error(err)
  })
```

or with async/await:

```
async function removeFolder(folder) {
  try {
    await fs.remove(folder)
    // done
  } catch (err) {
    console.error(err)
  }
}

const folder = '/Users/joe'
removeFolder(folder)
```

The Node fs module

The `fs` module provides a lot of very useful functionality to access and interact with the file system.

There is no need to install it. Being part of the Node.js core, it can be used by simply requiring it:

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

Once you do so, you have access to all its methods, which include:

- `fs.access()` : check if the file exists and Node.js can access it with its permissions
- `fs.appendFile()` : append data to a file. If the file does not exist, it's created
- `fs.chmod()` : change the permissions of a file specified by the filename passed. Related: `fs.lchmod()` , `fs.fchmod()`
- `fs.chown()` : change the owner and group of a file specified by the filename passed. Related: `fs.fchown()` , `fs.lchown()`
- `fs.close()` : close a file descriptor
- `fs.copyFile()` : copies a file

- `fs.createReadStream()` : create a readable file stream
- `fs.createWriteStream()` : create a writable file stream
- `fs.link()` : create a new hard link to a file
- `fs.mkdir()` : create a new folder
- `fs.mkdtemp()` : create a temporary directory
- `fs.open()` : opens the file and returns a file descriptor to allow file manipulation
- `fs.readdir()` : read the contents of a directory
- `fs.readFile()` : read the content of a file. Related: `fs.read()`
- `fs.readlink()` : read the value of a symbolic link
- `fs.realpath()` : resolve relative file path pointers (`.` , `..`) to the full path
- `fs.rename()` : rename a file or folder
- `fs.rmdir()` : remove a folder
- `fs.stat()` : returns the status of the file identified by the filename passed. Related: `fs.fstat()` , `fs.lstat()`
- `fs.symlink()` : create a new symbolic link to a file
- `fs.truncate()` : truncate to the specified length the file identified by the filename passed. Related: `fs.ftruncate()`
- `fs.unlink()` : remove a file or a symbolic link
- `fs.unwatchFile()` : stop watching for changes on a file
- `fs.utimes()` : change the timestamp of the file identified by the filename passed. Related: `fs.futimes()`
- `fs.watchFile()` : start watching for changes on a file. Related: `fs.watch()`
- `fs.writeFile()` : write data to a file. Related: `fs.write()`

One peculiar thing about the `fs` module is that all the methods are asynchronous by default, but they can also work synchronously by appending `Sync` .

For example:

- `fs.rename()`
- `fs.renameSync()`
- `fs.write()`

- `fs.writeSync()`

This makes a huge difference in your application flow.

> Node.js 10 includes experimental support for a promise based API

For example let's examine the `fs.rename()` method. The asynchronous API is used with a callback:

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

fs.rename('before.json', 'after.json', (err) => {
  if (err) {
    return console.error(err)
  }

  // done
})
```

A synchronous API can be used like this, with a try/catch block to handle errors:

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

try {
  fs.renameSync('before.json', 'after.json')
  // done
} catch (err) {
  console.error(err)
}
```

The key difference here is that the execution of your script will block in the second example, until the file operation succeeded.

You can use promise-based API provided by `fs/promises` module to avoid using callback-based API, which may cause callback hell. Here is an example:

```
// Example: Read a file and change its content and read
// it again using callback-based API.
const fs = require('fs')

const fileName = '/Users/joe/test.txt'
fs.readFile(fileName, 'utf8', (err, data) => {
  if (err) {
    console.log(err)
    return
  }
  console.log(data)
  const content = 'Some content!'
  fs.writeFile(fileName, content, (err2) => {
    if (err2) {
      console.log(err2)
      return
    }
    console.log('Wrote some content!')
    fs.readFile(fileName, 'utf8', (err3, data3) => {
      if (err3) {
        console.log(err3)
        return
      }
      console.log(data3)
    })
  })
})
```

The callback-based API may rises callback hell when there are too many nested callbacks. We can simply use promise-based API to avoid it:

```
// Example: Read a file and change its content and read
// it again using promise-based API.
const fs = require('fs/promises')

async function example() {
  const fileName = '/Users/joe/test.txt'
  try {
    const data = await fs.readFile(fileName, 'utf8')
    console.log(data)
    const content = 'Some content!'
    await fs.writeFile(fileName, content)
    console.log('Wrote some content!')
    const newData = await fs.readFile(fileName, 'utf8')
    console.log(newData)
  } catch (err) {
    console.log(err)
  }
}
example()
```

The Node path module

The `path` module provides a lot of very useful functionality to access and interact with the file system.

There is no need to install it. Being part of the Node.js core, it can be used by simply requiring it:

```
const path = require('path')
```

This module provides `path.sep` which provides the path segment separator (`\` on Windows, and `/` on Linux / macOS), and `path.delimiter` which provides the path delimiter (`;` on Windows, and `:` on Linux / macOS).

These are the `path` methods:

`path.basename()`

Return the last portion of a path. A second parameter can filter out the file extension:

```
require('path').basename('/test/something') // something
require('path').basename('/test/something.txt') // something.txt
require('path').basename('/test/something.txt', '.txt') // something
```

`path.dirname()`

Return the directory part of a path:

```
require('path').dirname('/test/something') // /test
require('path').dirname('/test/something/file.txt') // /test/something
```

`path.extname()`

Return the extension part of a path

```
require('path').extname('/test/something') // ''
require('path').extname('/test/something/file.txt') // '.txt'
```

`path.format()`

Returns a path string from an object, This is the opposite of `path.parse`
`path.format` accepts an object as argument with the following keys:

- `root` : the root
- `dir` : the folder path starting from the root
- `base` : the file name + extension
- `name` : the file name
- `ext` : the file extension

`root` is ignored if `dir` is provided
`ext` and `name` are ignored if `base` exists

```
// POSIX
require('path').format({ dir: '/Users/joe', base: 'test.txt' }) //  '/Use

require('path').format({ root: '/Users/joe', name: 'test', ext: '.txt' })

// WINDOWS
require('path').format({ dir: 'C:\\Users\\joe', base: 'test.txt' }) //  '
```

`path.isAbsolute()`

Returns true if it's an absolute path

```
require('path').isAbsolute('/test/something') // true
require('path').isAbsolute('./test/something') // false
```

`path.join()`

Joins two or more parts of a path:

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

`path.normalize()`

Tries to calculate the actual path when it contains relative specifiers like `.` or `..`, or double slashes:

```
require('path').normalize('/users/joe/..//test.txt') // '/users/test.txt'
```

`path.parse()`

Parses a path to an object with the segments that compose it:

- `root`: the root
- `dir`: the folder path starting from the root
- `base`: the file name + extension

- `name`: the file name
- `ext`: the file extension

Example:

```
require('path').parse('/users/test.txt')
```

results in

```
{
  root: '/',
  dir: '/users',
  base: 'test.txt',
  ext: '.txt',
  name: 'test'
}
```

`path.relative()`

Accepts 2 paths as arguments. Returns the relative path from the first path to the second, based on the current working directory.

Example:

```
require('path').relative('/Users/joe', '/Users/joe/test.txt') // 'test.tx
require('path').relative('/Users/joe', '/Users/joe/something/test.txt') //
```

`path.resolve()`

You can get the absolute path calculation of a relative path using `path.resolve()`:

```
require('path').resolve('joe.txt') // '/Users/joe/joe.txt' if run from my
```

By specifying a second parameter, `resolve` will use the first as a base for the second:

```
require('path').resolve('tmp', 'joe.txt') // '/Users/joe/tmp/joe.txt' if
```

If the first parameter starts with a slash, that means it's an absolute path:

```
require('path').resolve('/etc', 'joe.txt') // '/etc/joe.txt'
```

The Node os module

This module provides many functions that you can use to retrieve information from the underlying operating system and the computer the program runs on, and interact with it.

```
const os = require('os')
```

There are a few useful properties that tell us some key things related to handling files:

`os.EOL` gives the line delimiter sequence. It's `\n` on Linux and macOS, and `\r\n` on Windows.

`os.constants.signals` tells us all the constants related to handling process signals, like SIGHUP, SIGKILL and so on.

`os.constants.errno` sets the constants for error reporting, like EADDRINUSE, EOVERFLOW and more.

You can read them all on https://nodejs.org/api/os.html#os_signal_constants.

Let's now see the main methods that `os` provides:

`os.arch()`

Return the string that identifies the underlying architecture, like `arm` , `x64` , `arm64` .

`os.cpus()`

Return information on the CPUs available on your system.

Example:

```
/*
[
  {
    model: 'Intel(R) Core(TM)2 Duo CPU     P8600  @ 2.40GHz',
    speed: 2400,
    times: {
      user: 281685380,
      nice: 0,
      sys: 187986530,
      idle: 685833750,
      irq: 0,
    },
  },
  {
    model: 'Intel(R) Core(TM)2 Duo CPU     P8600  @ 2.40GHz',
    speed: 2400,
    times: {
      user: 282348700,
      nice: 0,
      sys: 161800480,
      idle: 703509470,
      irq: 0,
    },
  },
]
*/
```

`os.freemem()`

Return the number of bytes that represent the free memory in the system.

`os.homedir()`

Return the path to the home directory of the current user.

Example:

```
'/Users/joe'
```

`os.hostname()`

Return the host name.

`os.loadavg()`

Return the calculation made by the operating system on the load average.

It only returns a meaningful value on Linux and macOS.

Example:

```
//[3.68798828125, 4.00244140625, 11.1181640625]
```

`os.networkInterfaces()`

Returns the details of the network interfaces available on your system.

Example:

```
{ lo0:
   [ { address: '127.0.0.1',
       netmask: '255.0.0.0',
       family: 'IPv4',
       mac: 'fe:82:00:00:00:00',
       internal: true },
     { address: '::1',
       netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
       family: 'IPv6',
       mac: 'fe:82:00:00:00:00',
       scopeid: 0,
       internal: true },
     { address: 'fe80::1',
       netmask: 'ffff:ffff:ffff:ffff::',
       family: 'IPv6',
       mac: 'fe:82:00:00:00:00',
       scopeid: 1,
       internal: true } ],
  en1:
   [ { address: 'fe82::9b:8282:d7e6:496e',
       netmask: 'ffff:ffff:ffff:ffff::',
       family: 'IPv6',
       mac: '06:00:00:02:0e:00',
       scopeid: 5,
       internal: false },
     { address: '192.168.1.38',
       netmask: '255.255.255.0',
       family: 'IPv4',
       mac: '06:00:00:02:0e:00',
       internal: false } ],
  utun0:
   [ { address: 'fe80::2513:72bc:f405:61d0',
       netmask: 'ffff:ffff:ffff:ffff::',
       family: 'IPv6',
       mac: 'fe:80:00:20:00:00',
       scopeid: 8,
       internal: false } ] }
```

`os.platform()`

Return the platform that Node.js was compiled for:

- `darwin`

- `freebsd`
- `linux`
- `openbsd`
- `win32`
- ...more

`os.release()`

Returns a string that identifies the operating system release number

`os.tmpdir()`

Returns the path to the assigned temp folder.

`os.totalmem()`

Returns the number of bytes that represent the total memory available in the system.

`os.type()`

Identifies the operating system:

- `Linux`
- `Darwin` on macOS
- `Windows_NT` on Windows

`os.uptime()`

Returns the number of seconds the computer has been running since it was last rebooted.

`os.userInfo()`

Returns an object that contains the current `username`, `uid`, `gid`, `shell`, and `homedir`

The Node events module

The `events` module provides us the EventEmitter class, which is key to working with events in Node.js.

```
const EventEmitter = require('events')

const door = new EventEmitter()
```

The event listener has these in-built events:

- `newListener` when a listener is added
- `removeListener` when a listener is removed

Here's a detailed description of the most useful methods:

`emitter.addListener()`

Alias for `emitter.on()`.

`emitter.emit()`

Emits an event. It synchronously calls every event listener in the order they were registered.

```
door.emit('slam') // emitting the event "slam"
```

`emitter.eventNames()`

Return an array of strings that represent the events registered on the current `EventEmitter` object:

```
door.eventNames()
```

`emitter.getMaxListeners()`

Get the maximum amount of listeners one can add to an `EventEmitter` object, which defaults to 10 but can be increased or lowered by using `setMaxListeners()`

```
door.getMaxListeners()
```

`emitter.listenerCount()`

Get the count of listeners of the event passed as parameter:

```
door.listenerCount('open')
```

`emitter.listeners()`

Gets an array of listeners of the event passed as parameter:

```
door.listeners('open')
```

`emitter.off()`

Alias for `emitter.removeListener()` added in Node.js 10

`emitter.on()`

Adds a callback function that's called when an event is emitted.

Usage:

```
door.on('open', () => {
  console.log('Door was opened')
})
```

`emitter.once()`

Adds a callback function that's called when an event is emitted for the first time after registering this. This callback is only going to be called once, never again.

```
const EventEmitter = require('events')

const ee = new EventEmitter()

ee.once('my-event', () => {
  // call callback function once
})
```

`emitter.prependListener()`

When you add a listener using `on` or `addListener`, it's added last in the queue of listeners, and called last. Using `prependListener` it's added, and called, before other listeners.

`emitter.prependOnceListener()`

When you add a listener using `once`, it's added last in the queue of listeners, and called last. Using `prependOnceListener` it's added, and called, before other listeners.

`emitter.removeAllListeners()`

Removes all listeners of an `EventEmitter` object listening to a specific event:

```
door.removeAllListeners('open')
```

`emitter.removeListener()`

Remove a specific listener. You can do this by saving the callback function to a variable, when added, so you can reference it later:

```
const doSomething = () => {}
door.on('open', doSomething)
door.removeListener('open', doSomething)
```

`emitter.setMaxListeners()`

Sets the maximum amount of listeners one can add to an `EventEmitter` object, which defaults to 10 but can be increased or lowered.

```
door.setMaxListeners(50)
```

The Node http module

The HTTP core module is a key module to Node.js networking.

It can be included using

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

The module provides some properties and methods, and some classes.

Properties

`http.METHODS`

This property lists all the HTTP methods supported:

```
> require('http').METHODS
[ 'ACL',
  'BIND',
  'CHECKOUT',
  'CONNECT',
  'COPY',
  'DELETE',
  'GET',
  'HEAD',
  'LINK',
  'LOCK',
  'M-SEARCH',
  'MERGE',
  'MKACTIVITY',
  'MKCALENDAR',
  'MKCOL',
  'MOVE',
  'NOTIFY',
  'OPTIONS',
  'PATCH',
  'POST',
  'PROPFIND',
  'PROPPATCH',
  'PURGE',
  'PUT',
  'REBIND',
  'REPORT',
  'SEARCH',
  'SUBSCRIBE',
  'TRACE',
  'UNBIND',
  'UNLINK',
  'UNLOCK',
  'UNSUBSCRIBE' ]
```

`http.STATUS_CODES`

This property lists all the HTTP status codes and their description:

```
> require('http').STATUS_CODES
{ '100': 'Continue',
  '101': 'Switching Protocols',
  '102': 'Processing',
  '200': 'OK',
  '201': 'Created',
  '202': 'Accepted',
  '203': 'Non-Authoritative Information',
  '204': 'No Content',
  '205': 'Reset Content',
  '206': 'Partial Content',
  '207': 'Multi-Status',
  '208': 'Already Reported',
  '226': 'IM Used',
  '300': 'Multiple Choices',
  '301': 'Moved Permanently',
  '302': 'Found',
  '303': 'See Other',
  '304': 'Not Modified',
  '305': 'Use Proxy',
  '307': 'Temporary Redirect',
  '308': 'Permanent Redirect',
  '400': 'Bad Request',
  '401': 'Unauthorized',
  '402': 'Payment Required',
  '403': 'Forbidden',
  '404': 'Not Found',
  '405': 'Method Not Allowed',
  '406': 'Not Acceptable',
  '407': 'Proxy Authentication Required',
  '408': 'Request Timeout',
  '409': 'Conflict',
  '410': 'Gone',
  '411': 'Length Required',
  '412': 'Precondition Failed',
  '413': 'Payload Too Large',
  '414': 'URI Too Long',
  '415': 'Unsupported Media Type',
  '416': 'Range Not Satisfiable',
  '417': 'Expectation Failed',
  '418': 'I\'m a teapot',
  '421': 'Misdirected Request',
  '422': 'Unprocessable Entity',
  '423': 'Locked',
  '424': 'Failed Dependency',
  '425': 'Unordered Collection',
  '426': 'Upgrade Required',
  '428': 'Precondition Required',
  '429': 'Too Many Requests',
  '431': 'Request Header Fields Too Large',
  '451': 'Unavailable For Legal Reasons',
  '500': 'Internal Server Error',
  '501': 'Not Implemented',
  '502': 'Bad Gateway',
  '503': 'Service Unavailable',
  '504': 'Gateway Timeout',
  '505': 'HTTP Version Not Supported',
  '506': 'Variant Also Negotiates',
  '507': 'Insufficient Storage',
  '508': 'Loop Detected',
  '509': 'Bandwidth Limit Exceeded',
  '510': 'Not Extended',
  '511': 'Network Authentication Required' }
```

`http.globalAgent`

Points to the global instance of the Agent object, which is an instance of the `http.Agent` class.

It's used to manage connections persistence and reuse for HTTP clients, and it's a key component of Node.js HTTP networking.

More in the `http.Agent` class description later on.

Methods

`http.createServer()`

Returns a new instance of the `http.Server` class.

Usage:

```
const server = http.createServer((req, res) => {
  // handle every single request with this callback
})
```

`http.request()`

Makes an HTTP request to a server, creating an instance of the `http.ClientRequest` class.

`http.get()`

Similar to `http.request()`, but automatically sets the HTTP method to GET, and calls `req.end()` automatically.

Classes

The HTTP module provides 5 classes:

- `http.Agent`
- `http.ClientRequest`
- `http.Server`
- `http.ServerResponse`
- `http.IncomingMessage`

`http.Agent`

Node.js creates a global instance of the `http.Agent` class to manage connections persistence and reuse for HTTP clients, a key component of Node.js HTTP networking.

This object makes sure that every request made to a server is queued and a single socket is reused.

It also maintains a pool of sockets. This is key for performance reasons.

`http.ClientRequest`

An `http.ClientRequest` object is created when `http.request()` or `http.get()` is called.

When a response is received, the `response` event is called with the response, with an `http.IncomingMessage` instance as argument.

The returned data of a response can be read in 2 ways:

- you can call the `response.read()` method
- in the `response` event handler you can setup an event listener for the `data` event, so you can listen for the data streamed into.

`http.Server`

This class is commonly instantiated and returned when creating a new server using `http.createServer()` .

Once you have a server object, you have access to its methods:

- `close()` stops the server from accepting new connections
- `listen()` starts the HTTP server and listens for connections

`http.ServerResponse`

Created by an `http.Server` and passed as the second parameter to the `request` event it fires.

Commonly known and used in code as `res` :

```
const server = http.createServer((req, res) => {
  // res is an http.ServerResponse object
})
```

The method you'll always call in the handler is `end()` , which closes the response, the message is complete and the server can send it to the client. It must be called on each response.

These methods are used to interact with HTTP headers:

- `getHeaderNames()` get the list of the names of the HTTP headers already set
- `getHeaders()` get a copy of the HTTP headers already set
- `setHeader('headername', value)` sets an HTTP header value
- `getHeader('headername')` gets an HTTP header already set

- `removeHeader('headername')` removes an HTTP header already set
- `hasHeader('headername')` return true if the response has that header set
- `headersSent()` return true if the headers have already been sent to the client

After processing the headers you can send them to the client by calling `response.writeHead()`, which accepts the statusCode as the first parameter, the optional status message, and the headers object.

To send data to the client in the response body, you use `write()`. It will send buffered data to the HTTP response stream.

If the headers were not sent yet using `response.writeHead()`, it will send the headers first, with the status code and message that's set in the request, which you can edit by setting the `statusCode` and `statusMessage` properties values:

```
response.statusCode = 500
response.statusMessage = 'Internal Server Error'
```

`http.IncomingMessage`

An `http.IncomingMessage` object is created by:

- `http.Server` when listening to the `request` event
- `http.ClientRequest` when listening to the `response` event

It can be used to access the response:

- status using its `statusCode` and `statusMessage` methods
- headers using its `headers` method or `rawHeaders`
- HTTP method using its `method` method
- HTTP version using the `httpVersion` method
- URL using the `url` method
- underlying socket using the `socket` method

The data is accessed using streams, since `http.IncomingMessage` implements the Readable Stream interface.

Node.js Streams

What are streams

Streams are one of the fundamental concepts that power Node.js applications.

They are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

Streams are not a concept unique to Node.js. They were introduced in the Unix operating system decades ago, and programs can interact with each other passing streams through the pipe operator (`|`).

For example, in the traditional way, when you tell the program to read a file, the file is read into memory, from start to finish, and then you process it.

Using streams you read it piece by piece, processing its content without keeping it all in memory.

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

Why streams

Streams basically provide two major advantages over using other data handling methods:

- **Memory efficiency**: you don't need to load large amounts of data in memory before you are able to process it
- **Time efficiency**: it takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available

An example of a stream

A typical example is reading files from a disk.

Using the Node.js `fs` module, you can read a file, and serve it over HTTP when a new connection is established to your HTTP server:

```
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)
```

`readFile()` reads the full contents of the file, and invokes the callback function when it's done.

`res.end(data)` in the callback will return the file contents to the HTTP client.

If the file is big, the operation will take quite a bit of time. Here is the same thing written using streams:

```
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)
```

Instead of waiting until the file is fully read, we start streaming it to the HTTP client as soon as we have a chunk of data ready to be sent.

pipe()

The above example uses the line `stream.pipe(res)`: the `pipe()` method is called on the file stream.

What does this code do? It takes the source, and pipes it into a destination.

You call it on the source stream, so in this case, the file stream is piped to the HTTP response.

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

```
src.pipe(dest1).pipe(dest2)
```

This construct is the same as doing

```
src.pipe(dest1)
dest1.pipe(dest2)
```

Streams-powered Node.js APIs

Due to their advantages, many Node.js core modules provide native stream handling capabilities, most notably:

- `process.stdin` returns a stream connected to stdin
- `process.stdout` returns a stream connected to stdout
- `process.stderr` returns a stream connected to stderr
- `fs.createReadStream()` creates a readable stream to a file
- `fs.createWriteStream()` creates a writable stream to a file
- `net.connect()` initiates a stream-based connection
- `http.request()` returns an instance of the http.ClientRequest class, which is a writable stream
- `zlib.createGzip()` compress data using gzip (a compression algorithm) into a stream
- `zlib.createGunzip()` decompress a gzip stream.
- `zlib.createDeflate()` compress data using deflate (a compression algorithm) into a stream
- `zlib.createInflate()` decompress a deflate stream

Different types of streams

There are four classes of streams:

- `Readable` : 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.
- `Writable` : a stream you can pipe into, but not pipe from (you can send data, but not receive from it)
- `Duplex` : a stream you can both pipe into and pipe from, basically a combination of a Readable and Writable stream
- `Transform` : a Transform stream is similar to a Duplex, but the output is a transform of its input

How to create a readable stream

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

First create a stream object:

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

const readableStream = new Stream.Readable()
```

then implement `_read` :

```
readableStream._read = () => {}
```

You can also implement `_read` using the `read` option:

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

Now that the stream is initialized, we can send data to it:

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

How to create a writable stream

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

First create a stream object:

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

const writableStream = new Stream.Writable()
```

then implement `_write`:

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

You can now pipe a readable stream in:

```
process.stdin.pipe(writableStream)
```

How to get data from a readable stream

How do we read data from a readable stream? Using a writable 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)

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

You can also consume a readable stream directly, using the `readable` event:

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

How to send data to a writable stream

Using the stream `write()` method:

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

Signaling a writable stream that you ended writing

Use the `end()` method:

```
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()
```

In the above example, `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()`.

How to create a transform stream

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

First create a transform stream object:

```
const { Transform } = require('stream')

const transformStream = new Transform()
```

then implement `_transform`:

```
transformStream._transform = (chunk, encoding, callback) => {
  transformStream.push(chunk.toString().toUpperCase())
  callback()
}
```

Pipe readable stream:

```
process.stdin.pipe(transformStream).pipe(process.stdout)
```

Node, the difference between development and production

You can have different configurations for production and development environments.

Node.js assumes it's always running in a development environment. You can signal Node.js that you are running in production by setting the `NODE_ENV=production` environment variable.

This is usually done by executing the command

```
export NODE_ENV=production
```

in the shell, but it's better to put it in your shell configuration file (e.g. `.bash_profile` with the Bash shell) because otherwise the setting does not persist in case of a system restart.

You can also apply the environment variable by prepending it to your application initialization command:

```
NODE_ENV=production node app.js
```

This environment variable is a convention that is widely used in external libraries as well.

Setting the environment to `production` generally ensures that

- logging is kept to a minimum, essential level
- more caching levels take place to optimize performance

For example Pug, the templating library used by Express, compiles in debug mode if `NODE_ENV` is not set to `production`. Express views are compiled in every request in development mode, while in production they are cached. There are many more examples.

You can use conditional statements to execute code in different environments:

```
if (process.env.NODE_ENV === 'development') {
  // ...
}
if (process.env.NODE_ENV === 'production') {
  // ...
}
if (['production', 'staging'].indexOf(process.env.NODE_ENV) >= 0) {
  // ...
}
```

For example, in an Express app, you can use this to set different error handlers per environment:

```
if (process.env.NODE_ENV === 'development') {
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }))
}

if (process.env.NODE_ENV === 'production') {
  app.use(express.errorHandler())
}
```

Error handling in Node.js

Errors in Node.js are handled through exceptions.

Creating exceptions

An exception is created using the `throw` keyword:

```
throw value
```

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.

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()
```

Handling exceptions

An exception handler is a `try` / `catch` statement.

Any exception raised in the lines of code included in the `try` block is handled in the corresponding `catch` block:

```
try {
  // lines of code
} catch (e) {}
```

`e` in this example is the exception value.

You can add multiple handlers, that can catch different kinds of errors.

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)
})
```

You don't need to import the `process` core module for this, as it's automatically injected.

Exceptions with promises

Using promises you can chain different operations, and handle errors at the end:

```
doSomething1()
  .then(doSomething2)
  .then(doSomething3)
  .catch((err) => console.error(err))
```

How do you know where the error occurred? You don't really know, but you can handle errors in each of the functions you call (`doSomethingX`), and inside the error handler throw a new error, that's going to call the outside `catch` handler:

```
const doSomething1 = () => {
  // ...
  try {
    // ...
  } catch (err) {
    // ... handle it locally
    throw new Error(err.message)
  }
  // ...
}
```

To be able to handle errors locally without handling them in the function we call, we can break the chain. You can create a function in each `then()` and process the exception:

```
doSomething1()
  .then(() => {
    return doSomething2().catch((err) => {
      // handle error
      throw err // break the chain!
    })
  })
  .then(() => {
    return doSomething3().catch((err) => {
      // handle error
      throw err // break the chain!
    })
  })
  .catch((err) => console.error(err))
```

Error handling with async/await

Using async/await, you still need to catch errors, and you do it this way:

```
async function someFunction() {
  try {
    await someOtherFunction()
  } catch (err) {
    console.error(err.message)
  }
}
```

Build an HTTP Server

Here is a sample Hello World HTTP web server:

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

const port = process.env.PORT || 3000

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/html')
  res.end('<h1>Hello, World!</h1>')
})

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

Let's analyze it briefly. We include the `http` module.

We use the module to create an HTTP server.

The server is set to listen on the specified port, `3000`. When the server is ready, the `listen` callback function is called.

The callback function we pass is the one that's going to be executed upon every request that comes in. Whenever a new request is received, the `request` event is called, providing two objects: a request (an `http.IncomingMessage` object) and a response (an `http.ServerResponse` object).

`request` provides the request details. Through it, we access the request headers and request data.

`response` is used to populate the data we're going to return to the client.

In this case with

```
res.statusCode = 200
```

we set the statusCode property to 200, to indicate a successful response.

We also set the Content-Type header:

```
res.setHeader('Content-Type', 'text/html')
```

and we end close the response, adding the content as an argument to `end()`:

```
res.end('<h1>Hello, World!</h1>')
```

Making HTTP requests with Node.js

Perform a GET Request

There are many ways to perform an HTTP GET request in Node.js, depending on the abstraction level you want to use.

The simplest way to perform an HTTP request using Node.js is to use the Axios library:

```
const axios = require('axios')

axios
  .get('https://example.com/todos')
  .then((res) => {
    console.log(`statusCode: ${res.status}`)
    console.log(res)
  })
  .catch((error) => {
    console.error(error)
  })
```

However, Axios requires the use of a 3rd party library.

A GET request is possible just using the Node.js standard modules, although it's more verbose than the option above:

```
const https = require('https')

const options = {
  hostname: 'example.com',
  port: 443,
  path: '/todos',
  method: 'GET',
}

const req = https.request(options, (res) => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', (d) => {
    process.stdout.write(d)
  })
})

req.on('error', (error) => {
  console.error(error)
})

req.end()
```

Perform a POST Request

Similar to making an HTTP GET request, you can use the Axios library to perform a POST request:

```
const axios = require('axios')

axios
  .post('https://whatever.com/todos', {
    todo: 'Buy the milk',
  })
  .then((res) => {
    console.log(`statusCode: ${res.status}`)
    console.log(res)
  })
  .catch((error) => {
    console.error(error)
  })
```

Or alternatively, use Node.js standard modules:

```
const https = require('https')

const data = JSON.stringify({
  todo: 'Buy the milk',
})

const options = {
  hostname: 'whatever.com',
  port: 443,
  path: '/todos',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': data.length,
  },
}

const req = https.request(options, (res) => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', (d) => {
    process.stdout.write(d)
  })
})

req.on('error', (error) => {
  console.error(error)
})

req.write(data)
req.end()
```

PUT and DELETE

PUT and DELETE requests use the same POST request format - you just need to change the `options.method` value to the appropriate method.

Get HTTP request body data

Here is how you can extract the data that was sent as JSON in the request body.

If you are using Express, that's quite simple: use the `express.json()` middleware which is available in Express v4.16.0 onwards.

For example, to get the body of this request:

```
const axios = require('axios')

axios.post('https://whatever.com/todos', {
  todo: 'Buy the milk',
})
```

This is the matching server-side code:

```
const express = require('express')

const app = express()

app.use(
  express.urlencoded({
    extended: true,
  })
)

app.use(express.json())

app.post('/todos', (req, res) => {
  console.log(req.body.todo)
})
```

If you're not using Express and you want to do this in vanilla Node.js, you need to do a bit more work, of course, as Express abstracts a lot of this for you.

The key thing to understand is that when you initialize the HTTP server using `http.createServer()`, the callback is called when the server got all the HTTP headers, but not the request body.

The `request` object passed in the connection callback is a stream.

So, we must listen for the body content to be processed, and it's processed in chunks.

We first get the data by listening to the stream `data` events, and when the data ends, the stream `end` event is called, once:

```
const server = http.createServer((req, res) => {
  // we can access HTTP headers
  req.on('data', (chunk) => {
    console.log(`Data chunk available: ${chunk}`)
  })
  req.on('end', () => {
    // end of data
  })
})
```

So to access the data, assuming we expect to receive a string, we must concatenate the chunks into a string when listening to the stream `data`, and when the stream `end`, we parse the string to JSON:

```
const server = http.createServer((req, res) => {
  let data = ''
  req.on('data', (chunk) => {
    data += chunk
  })
  req.on('end', () => {
    console.log(JSON.parse(data).todo) // 'Buy the milk'
    res.end()
  })
})
```

Starting from Node.js v10 a `for await .. of` syntax is available for use. It simplifies the example above and makes it look more linear:

```
const server = http.createServer(async (req, res) => {
  const buffers = []

  for await (const chunk of req) {
    buffers.push(chunk)
  }

  const data = Buffer.concat(buffers).toString()

  console.log(JSON.parse(data).todo) // 'Buy the milk'
  res.end()
})
```

Conclusion

Thanks a lot for reading this book.

For more, head over to The Valley Of Code.

Send any feedback, errata or opinions at hello@thevalleyofcode.com