THE

HANDBOOK

Flavio Copes

Table of Contents

The Node.js Handbook

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

I find this approach gives a well-rounded overview. This book does not try to cover everything under the sun related to Node. If you think some specific topic should be included, tell me.

You can reach me on Twitter @flaviocopes.

I hope the contents of this book will help you achieve what you want: **learn the basics Node.js**.

This book is written by Flavio. I **publish web development tutorials** every day on my website flaviocopes.com.

Enjoy!

Introduction to Node

This post is a getting started guide to Node.js, the server-side JavaScript runtime environment. Node.js is built on top of the Google Chrome V8 JavaScript engine, and it's mainly used to create web servers - but it's not limited to that

Node.js is a **runtime environment for JavaScript** that runs on the **server**.

Node.js is open source, cross-platform, and since its introduction in 2009, it got hugely popular and now plays a significant role in the web development scene. If GitHub stars are one popularity indication factor, having 58000+ stars means being very popular.

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. Node.js is able to leverage the work of the engineers that made (and will continue to make) the Chrome JavaScript runtime blazing fast, and this allows Node.js to benefit from the huge performance improvements and the Just-In-Time compilation that V8 performs. Thanks to this, JavaScript code running in Node.js can become very performant.

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

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

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

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to run the server-side code and frontend-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 with flags.

Node.js has a huge 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.

A brief history of Node

A look back on the history of Node.js from 2009 to today

Believe it or not, Node.js is just 9 years old.

In comparison, JavaScript is 23 years old and the web as we know it (after the introduction of Mosaic) is 25 years old.

9 years is such a little amount of time for a technology, but Node.js seems to have been around forever.

I've had the pleasure to work with Node since the early days when it was just 2 years old, and despite the little information available, you could already feel it was a huge thing.

In this post, I want to draw the big picture of Node 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*, which could create dynamic pages using server-side JavaScript. So the idea of server-side JavaScript was not introduced by Node.js, but it's old just like JavaScript - but at the time it was not successful.

One key factor that led to the rise of Node.js was timing. JavaScript since a few years was starting being considered a serious language, thanks for the "Web 2.0" applications that showed the world what a modern experience on the web could be like (think Google Maps or GMail).

The JavaScript engines performance bar raised considerably thanks to the browser competition battle, which is still going strong. Development teams behind each major browser work hard every day to give us better performance, which is a huge win for JavaScript as a platform. V8, the engine that Node.js uses under the hood, is one of those and in particular it's the Chrome JS engine.

But of course, Node.js is not popular just because of pure luck or timing. It introduced much innovative thinking on how to program in JavaScript on the server.

2009

- Node.js is born
- The first form of npm is created

2010

- Express is born
- Socket.io is born

2011

- npm hits 1.0
- Big companies start adopting Node: LinkedIn, Uber Hapi is born

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
- npm introduces private modules
- 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
- Work on Deno started to move server-side JS into the next decade with modern JavaScript support

2020

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

How to install Node

How you can install Node.js on your system: a package manager, the official website installer or nvm

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/creationix/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?

If you are just starting out with JavaScript, how much deeply do you need to know the language?

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

How writing JavaScript application in Node.js differs from programming for the Web inside the browser

Both the browser and Node 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.

As a frontend developer who extensively uses Javascript, Node apps brings with it, 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 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 uses the CommonJS module system, while in the browser we are starting to see the ES Modules standard being implemented.

In practice, this means that for the time being you use `require()` in Node and `import` in the browser.

How to exit from a Node.js program

Learn how to terminate a Node.js app in the best possible way

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 I 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 is 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 will later end, Node will return that exit code.

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

Many times with Node 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'))
```

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*.

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 signals that tells a process to immediately terminate, and would ideally act like `process.exit()`.

`SIGTERM` is the signals 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

Learn how to read and make use of environment variables in a Node.js program

Environment variables are especially useful because we can avoid typing API keys and other sensible data in the code and have it pushed by mistake to GitHub.

And modern deployment platforms like Vercel and Netlify (and others) have ways to let us add environment variables through their interfaces.

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

Here is an example that accesses the NODE_ENV environment variable, which is set to `development` by default.

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

```
process.env.NODE_ENV // "development"
```

Setting it to "production" before the script runs will tell Node that this is a production environment.

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

Here we set 2 variables for API_KEY and API_SECRET

```
API_KEY=123123 API_SECRET=456456 node app.js
```

We can get them in Node.js by running

```
process.env.API_KEY // "123123"
process.env.API_SECRET // "456456"
```

You can write the environment variables in a `.env` file (which you should add to `.gitignore` to avoid pushing to GitHub), then

```
npm install dotenv
```

and at the beginning of your main Node file, add

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

In this way you can avoid listing the environment variables in the command line before the `node` command, and those variables will be picked up automatically.

> Note that some tools, like Next.js for example, make environment variables defined in `.env` automatically available without the need to use `dotenv`.

Node hosting options

A Node.js application can be hosted in a lot of places, depending on your needs. This is a list of all the various options you have at your disposal

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
- Zero configuration deployments
  - Glitch
  - Codepen
- Serverless
- PAAS
  - Zeit Now
  - Nanobox
  - Heroku
  - Microsoft Azure
  - Google Cloud Platform
- Virtual Private Server
- Bare metal

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)

Use the Node REPL

REPL stands for Read-Evaluate-Print-Loop, and it's a great way to explore the Node features in a quick way

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

```
node script.js
```

If we omit the filename, we use it in REPL mode:

```
node
```

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 ".

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()`.

We can now enter a new line of JavaScript.

Use the tab to autocomplete

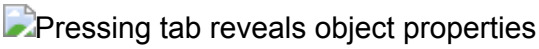
The cool thing about the REPL is that it's interactive.

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:


Pressing tab reveals object properties

Explore global objects

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


Globals

The _ special variable

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

Dot commands

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

- `.help`: shows the dot commands help
- `.editor`: enables editor more, 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`: exists 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.

Pass arguments from the command line

How to accept arguments in a Node.js program passed 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 flavio
```

or

```
node app.js name=flavio
```

This changes how you will retrieve this value in the Node code.

The way you retrieve it is using the `process` object built into Node.

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

The first argument 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 flavio
```

you can access it using

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

In this case:

```
node app.js name=flavio
```

`args[0]` is `name=flavio`, 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'] //flavio
```

Output to the command line

How to print to the command line console using Node, from the basic console.log to more complex scenarios

Basic output using the console module

Node 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 will print both.

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

For example:

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

- `%s` format a variable as a string
- `%d` or `%i` format a variable as an integer

- `%f` format a variable as a floating point number
- `%0` used to print an object representation

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 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)
})
```

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 I try this in the Node 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 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`, 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 I 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

How to make a Node.js CLI program interactive using the built-in readline Node module

How to make a Node.js CLI program interactive?

Node since version 7 provides the `readline` module to perform exactly this: get input from a readable stream such as the `process.stdin` stream, which during the execution of a Node program is the terminal input, one line at a time.

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

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

This piece of code asks the username, and once the text is entered and the user presses enter, we send a greeting.

The `question()` method shows the first parameter (a question) and waits for the user input. It calls the callback function once enter is pressed.

In this callback function, we close the readline interface.

`readline` offers several other methods, and I'll let you check them out on the package documentation I linked above.

If you need to require a password, it's best to not echo it back, but instead showing a `*` symbol.

The simplest way is to use the `readline-sync` package which is very similar in terms of the API and handles this out of the box.

A more complete and abstract solution is provided by the Inquirer.js package.

You can install it using `npm install inquirer`, and then you can replicate the above code like this:

```
const inquirer = require('inquirer')

var questions = [{
  type: 'input',
  name: 'name',
  message: "What's your name?",
}]

inquirer.prompt(questions).then(answers => {
  console.log(`Hi ${answers['name']}!`)
})
```

Inquirer.js lets you do many things like asking multiple choices, having radio buttons, confirmations, and more.

It's worth knowing all the alternatives, especially the built-in ones provided by Node, but if you plan to take CLI input to the next level, Inquirer.js is an optimal choice.

Expose functionality from a Node file using exports

How to use the module.exports API to expose data to other files in your application, or to other applications as well

Node 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*:

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

module.exports = car

//..in the other file

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('./items')
items.car
```

or

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

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

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

npm

A quick guide to npm, the powerful package manager key to the success of 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.

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 is an alternative to npm. Make sure you check it out as well.

Installation

`npm` is installed when you install Node.js. Head to https://nodejs.org and install Node, if you haven't installed it already on your system.

How to use npm

`npm` manages downloads of dependencies of your project.

Installing all dependencies

If a project has a `packages.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>
```

Often you'll see more flags added to this command:

- `--save` installs and adds the entry to the `package.json` file *dependencies* (default as of npm 5)
- `--save-dev` installs and adds the entry to the `package.json` file *devDependencies*

The difference is mainly that devDependencies are usually development tools, like a testing library, while `dependencies` are bundled with the app in production.

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

How to find out where npm installs the packages

> Read the npm guide if you are starting out with npm, it's going to go in a lot of the basic details of it.

When you install a package using `npm` (or yarn), 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.

I for example use `nvm` and my packages location was shown as `/Users/flavio/.nvm/versions/node/v8.9.0/lib/node_modules`.

How to use or execute a package installed using npm

How to include and use in your code a package installed in your node_modules folder

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


The node_modules folder content

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

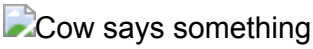

The binary files

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
```

and npx will find the package location.



The package.json file

The package.json file is a key element in lots of app codebases based on the Node.js ecosystem.

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 of the package it installed.

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 I extracted this 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.dev.conf.js",
    "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:

- `name` sets the application/package name
- `version` indicates the current version
- `description` is a brief description of the app/package
- `main` set 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 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. I 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 the https://www.npmjs.com/, other 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": "Flavio Copes <your@email.com> (https://flaviocopes.com)"
}
```

Can also be used with this format:

```
{
  "author": {
    "name": "Flavio Copes",
    "email": "your@email.com",
    "url": "https://flaviocopes.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": [
    "Flavio Copes <your@email.com> (https://flaviocopes.com)"
  ]
}
```

Can also be used with this format:

```
{
  "contributors": [
    {
      "name": "Flavio Copes",
      "email": "your@email.com",
      "url": "https://flaviocopes.com"
    }
  ]
}
```

bugs

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

Example:

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

homepage

Sets the package homepage

Example:

```
{
  "homepage": "https://flaviocopes.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:flaviocopes/testing",
```

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

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

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

You can explicitly set the version control system:

```
"repository": {
  "type": "git",
  "url": "https://github.com/flaviocopes/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.dev.conf.js",
  "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 --dev <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 you 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:

- `~`: 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 update patch and minor releases: `0.13.1`, `0.14.0` and so on.
- `*`: if you write `*`, that means you accept all updates, including major version upgrades.
- `>`: 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 to the one you specify

There are other rules, too:

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

and you can combine most of the above 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

The package-lock.json file is automatically generated when installing node packages. Learn what it's about

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 the 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 update patch and minor releases: `0.13.1`, `0.14.0` and so on.
- 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 install`.

This concept is not new, and other programming languages package managers (like Composer in PHP) use 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.tgz",
      "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-2.1.1.tgz",
      "integrity": "sha512-nOqH59deCq9SRHlxq1Aw85Jnt4w6KvLKqWVik6oA9ZklXLNIOlqg4F2yrT1MVa
      "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-4.0.0.tgz",
      "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.tgz",
      "integrity": "sha1-u0P/VZim6wXYm1n80SnJgzE2Br8="
    },
    "wordwrap": {
      "version": "0.0.3",
      "resolved": "https://registry.npmjs.org/wordwrap/-/wordwrap-0.0.3.tgz",
      "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

How to find out which version of a particular package you have installed in your app

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

```
npm list
```

Example:

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

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

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

This also works for dependencies of packages you installed:

```
❯ npm list minimist
/Users/flavio/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
```

How to install an older version of an npm package

Learn how to install an older version of an npm package, something that might be useful to solve a compatibility problem

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 version 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' ]
```

How to update all the Node dependencies to their latest version

How do you update all the npm dependencies store in the package.json file, to their latest version available?

When you install a package using `npm install <packagename>`, the latest available version of the package is downloaded and put in the `node_modules` folder, and a corresponding entry is added to the `package.json` and `package-lock.json` files that are present in your current folder.

npm calculates the dependencies and installs the latest available version of those as well.

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

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

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

and this is an extract of `package-lock.json`, where I removed the nested dependencies 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+BTTDkMAjufp+0F9eLjzRnOH
      "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 rule for updates is `^1.3.1`, which for the npm versioning rules means that npm can update to patch and minor releases: `1.3.2`, `1.4.0` and so on.

But not for major version changes that break compatibility, which means, in this example, `2.0` and higher.

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.

`package.json` remains unchanged.

To discover new releases of the packages, you run `npm outdated`.

Here's the list of a few outdated packages in one repository I didn't update for quite a while:

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

To update to a new major version all the packages, install the `npm-check-updates` package globally:

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

then run it:

```
ncu -u
```

this will upgrade all the version hints in the `package.json` file, to `dependencies` and `devDependencies`, so npm can install the new major version.

You are now ready to run the update:

```
npm update
```

If you just downloaded the project without the `node_modules` dependencies and you want to install the shiny new versions first, just run

```
npm install
```

Semantic versioning rules

Semantic Versioning is a convention used to provide a meaning to versions

If there's one great thing in Node.js packages, is that 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 those symbols:

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

Let's see those rules in detail:

- `^`: if you write `^0.13.0` when running `npm update` it can update to patch and minor releases: `0.13.1`, `0.14.0` and so on.

- `~` : 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 to 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

How to uninstall an npm Node package, locally or globally

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

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

from the project root folder (the folder that contains the node_modules folder).

This operation will also remove the reference in the `package.json` file.

If the package was a development dependency, listed in the devDependencies of the `package.json` file, you must use the `-D` / `--save-dev` flag to remove it from the file:

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

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 the folder where you currently are does not matter.

Global or local packages

When is a package best installed globally? Why?

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, they are both required in the same way:

```
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`
- `create-react-app`
- `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 is a package a dependency, and when is it a dev dependency?

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: before 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.

npx

npx is a very cool way to run Node code, and provides many useful features

In this post, I want to introduce a very powerful command that's been available in **npm** starting version 5.2, released in July 2017: **npx**.

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

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

Easily run local commands

Node 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 `npm`, 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 |
                ||     ||
```

Now, this 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 having it installed locally:

```
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 create my-vue-app`
- creating a new React app using `create-react-app`: `npx create-react-app my-react-app`

and many more.

Once downloaded, the downloaded code will be wiped.

Run some code using a different Node version

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

```
npx node@6 -v #v6.14.3
npx node@8 -v #v8.11.3
```

This helps to avoid tools like `nvm` or the other Node 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 event loop

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

Introduction

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

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

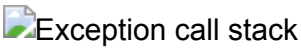
The call stack

The call stack is a LIFO queue (Last In, First Out).

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 to 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:

 Exception call stack

A simple event loop explanation

Let's pick an example:

> I use `foo`, `bar` and `baz` as *random names*. Enter any kind of name to replace them.

```
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:


Call stack first example

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


Execution order first example

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:


Call stack second example

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


Execution order second example

Why is this happening?

The Message Queue

When setTimeout() is called, the Browser or Node.js start 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.

I find nice the analogy of 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.

Conclusion

This article introduced you to the basic building blocks of the Node.js Event Loop.

This is an essential part of any program written in Node.js, and I hope that some of the concepts explained here will be useful to you in the future.

nextTick

The Node.js process.nextTick function interacts with the event loop in a special way

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.

setImmediate

The Node.js setImmediate function interacts with the event loop in a special way

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()` ?

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.

Timers

When writing JavaScript code, you might want to delay the execution of a function. Learn how to use setTimeout and setInterval to schedule functions in the future

`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 ')
```

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)
```

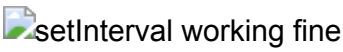
It's common to call `clearInterval` inside the setInterval callback function, to let it auto-determine if it should run again or stop. For example this code runs something unless App.somethingIWait has the value `arrived`:

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

Recursive setTimeout

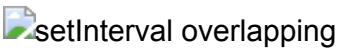
`setInterval` starts a function every n milliseconds, without any consideration about when a function finished its execution.

If a function takes always 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:



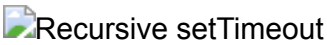
To avoid this, you can schedule a recursive setTimeout to be called when the callback function finishes:

```
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.

Callbacks

JavaScript is synchronous by default, and is single threaded. This means that code cannot create new threads and run in parallel. Find out what asynchronous code means and how it looks like

Asynchronicity in Programming Languages

Computers are asynchronous by design.

Asynchronous means that things can happen independently of the main program flow.

In the current consumer computers, every program runs for a specific time slot, and then it stops its execution to let another program continue its execution. This thing runs in a cycle so fast that's impossible to notice, and we think our computers run many programs simultaneously, but this is an illusion (except on multiprocessor machines).

Programs internally use *interrupts*, a signal that's emitted to the processor to gain the attention of the system.

I won't go into the internals of this, but just keep in mind that it's normal for programs to be asynchronous, and halt their execution until they need attention, and the computer can execute other things in the meantime. When a program is waiting for a response from the network, it cannot halt the processor until the request finishes.

Normally, programming languages are synchronous, and some provide a way to manage asynchronicity, in the language or through libraries. C, Java, C#, PHP, Go, Ruby, Swift, Python, they are all synchronous by default. Some of them handle async by using threads, spawning a new process.

JavaScript

JavaScript is **synchronous** by default and is single threaded. This means that code cannot create new threads and run in parallel.

Lines of code are executed in series, one after another, for example:

```
const a = 1
const b = 2
const c = a * b
console.log(c)
doSomething()
```

But JavaScript was born inside the browser, its main job, in the beginning, was to respond to user actions, like `onClick`, `onMouseOver`, `onChange`, `onSubmit` and so on. How could it do this with a synchronous programming model?

The answer was in its environment. The **browser** provides a way to do it by providing a set of APIs that can handle this kind of functionality.

More recently, Node.js introduced a non-blocking I/O environment to extend this concept to file access, network calls and so on.

Callbacks

You can't know when a user is going to click a button, so what you do is, you **define an event handler for the click event**. This event handler accepts a function, which will be called when the event is triggered:

```
document.getElementById('button').addEventListener('click', () => {
  //item clicked
})
```

This is the so-called **callback**.

A callback is a simple function that's passed as a value to another function, and will only be executed when the event happens. We can do this because JavaScript has first-class functions, which can be assigned to variables and passed around to other functions (called **higher-order functions**)

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

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

Callbacks are used everywhere, not just in DOM events.

One common example is by using timers:

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

XHR requests also accept a callback, in this example by assigning a function to a property that will be called when a particular event occurs (in this case, the state of the request changes):

```
const xhr = new XMLHttpRequest()
xhr.onreadystatechange = () => {
  if (xhr.readyState === 4) {
    xhr.status === 200 ? console.log(xhr.responseText) : console.error('error')
  }
}
xhr.open('GET', 'https://yoursite.com')
xhr.send()
```

Handling errors in callbacks

How do you handle errors with callbacks? One very common strategy is to use what Node.js adopted: the first parameter in any callback function is the error object: **error-first callbacks**

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

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

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

The problem with callbacks

Callbacks are great for simple cases!

However every callback adds a level of nesting, and when you have lots of callbacks, the code starts to be complicated very quickly:

```
window.addEventListener('load', () => {
  document.getElementById('button').addEventListener('click', () => {
    setTimeout(() => {
      items.forEach(item => {
        //your code here
      })
    }, 2000)
  })
})
```

This is just a simple 4-levels code, but I've seen much more levels of nesting and it's not fun.

How do we solve this?

Alternatives to callbacks

Starting with ES6, JavaScript introduced several features that help us with asynchronous code that do not involve using callbacks:

- Promises (ES2015)
- Async/Await (ES2017)

Promises

Promises are one way to deal with asynchronous code in JavaScript, without writing too many callbacks in your code.

Introduction to promises

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

Promises are one way to deal with asynchronous code, without writing too many callbacks in your code.

Although they've been around for years, they were standardized and introduced in ES2015, and now they have been superseded in ES2017 by async functions.

Async functions use the promises API as their building block, so understanding them is fundamental even if in newer code you'll likely use async functions instead of promises.

How promises work, in brief

Once a promise has been called, it will start in **pending state**. This means that the caller function continues the execution, while it waits for the promise to do its own processing, and give the caller function some feedback.

At this point, the caller function waits for it to either return the promise in a **resolved state**, or in a **rejected state**, but *the function continues its execution while the promise does it work*.

Which JS API use promises?

In addition to your own code and library code, promises are used by standard modern Web APIs such as:

- the Battery API
- the Fetch API
- Service Workers

It's unlikely that in modern JavaScript you'll find yourself *not* using promises, so let's start diving right into them.

Creating a promise

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

```
let done = true

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

As you can see the promise checks the `done` global variable, and if that's true, we return a resolved promise, otherwise a rejected promise.

Using `resolve` and `reject` we can communicate back a value, in the above case we just return a string, but it could be an object as well.

Consuming a promise

In the last section, we introduced how a promise is created.

Now let's see how the promise can be *consumed* or used.

```
const isItDoneYet = new Promise()
//...

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

Running `checkIfItsDone()` will execute the `isItDoneYet()` promise and will wait for it to resolve, using the `then` callback, and if there is an error, it will handle it in the `catch` callback.

Chaining promises

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

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

The Fetch API is a promise-based mechanism, and calling `fetch()` is equivalent to defining our own promise using `new Promise()`.

Example of chaining promises

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

const json = response => response.json()

fetch('/todos.json')
  .then(status)
  .then(json)
  .then(data => {
    console.log('Request succeeded with JSON response', data)
  })
  .catch(error => {
    console.log('Request failed', error)
  })
```

In this example, we call `fetch()` to get a list of TODO items from the `todos.json` file found in the domain root, and we create a chain of promises.

Running `fetch()` returns a response, which has many properties, and within those we reference:

- `status`, a numeric value representing the HTTP status code
- `statusText`, a status message, which is `OK` if the request succeeded

`response` also has a `json()` method, which returns a promise that will resolve with the content of the body processed and transformed into **JSON**.

So given those premises, this is what happens: the first promise in the chain is a function that we defined, called `status()`, that checks the response status and if it's not a success response (between 200 and 299), it rejects the promise.

This operation will cause the promise chain to skip all the chained promises listed and will skip directly to the `catch()` statement at the bottom, logging the `Request failed` text along with the error message.

If that succeeds instead, it calls the `json()` function we defined. Since the previous promise, when successful, returned the `response` object, we get it as an input to the second promise.

In this case, we return the data JSON processed, so the third promise receives the JSON directly:

```
.then((data) => {
  console.log('Request succeeded with JSON response', data)
})
```

and we log it to the console.

Handling errors

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

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

```
new Promise((resolve, reject) => {
  throw new Error('Error')
}).catch(err => {
  console.error(err)
})

// or

new Promise((resolve, reject) => {
  reject('Error')
}).catch(err => {
  console.error(err)
})
```

Cascading errors

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

```
new Promise((resolve, reject) => {
  throw new Error('Error')
})
  .catch(err => {
    throw new Error('Error')
  })
  .catch(err => {
    console.error(err)
  })
```

Orchestrating promises

`Promise.all()`

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

Example:

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

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

The ES2015 destructuring assignment syntax allows you to also do

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

You are not limited to using `fetch` of course, **any promise is good to go**.

`Promise.race()`

`Promise.race()` runs as soon as one of the promises you pass to it resolves, and it runs the attached callback just once with the result of the first promise resolved.

Example:

```
const promiseOne = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'one')
})
const promiseTwo = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'two')
})

Promise.race([promiseOne, promiseTwo]).then(result => {
  console.log(result) // 'two'
})
```

Common errors

Uncaught TypeError: undefined is not a promise

If you get the `Uncaught TypeError: undefined is not a promise` error in the console, make sure you use `new Promise()` instead of just `Promise()`

async/await

Discover the modern approach to asynchronous functions in JavaScript. JavaScript evolved in a very short time from callbacks to Promises, and since ES2017 asynchronous JavaScript is even simpler with the async/await syntax

Introduction

JavaScript evolved in a very short time from callbacks to promises (ES2015), and since ES2017 asynchronous JavaScript is even simpler with the async/await syntax.

Async functions are a combination of promises and generators, and basically, they are a higher level abstraction over promises. Let me repeat: **async/await is built on promises**.

Why were async/await introduced?

They reduce the boilerplate around promises, and the "don't break the chain" limitation of chaining promises.

When Promises were introduced in ES2015, they were meant to solve a problem with asynchronous code, and they did, but over the 2 years that separated ES2015 and ES2017, it was clear that *promises could not be the final solution*.

Promises were introduced to solve the famous *callback hell* problem, but they introduced complexity on their own, and syntax complexity.

They were good primitives around which a better syntax could be exposed to developers, so when the time was right we got **async functions**.

They make the code look like it's synchronous, but it's asynchronous and non-blocking behind the scenes.

How it works

An async function returns a promise, like in this example:

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

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

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

A quick example

This is a simple example of async/await used to run a function asynchronously:

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

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

console.log('Before')
doSomething()
console.log('After')
```

The above code will print the following to the browser console:

```
Before
After
I did something //after 3s
```

Promise all the things

Prepending the `async` keyword to any function means that the function will return a promise.

Even if it's not doing so explicitly, it will internally make it return a promise.

This is why this code is valid:

```
const aFunction = async () => {
  return 'test'
}

aFunction().then(alert) // This will alert 'test'
```

and it's the same as:

```
const aFunction = async () => {
  return Promise.resolve('test')
}

aFunction().then(alert) // This will alert 'test'
```

The code is much simpler to read

As you can see in the example above, our code looks very simple. Compare it to code using plain promises, with chaining and callback functions.

And this is a very simple example, the major benefits will arise when the code is much more complex.

For example here's how you would get a JSON resource, and parse it, using promises:

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

getFirstUserData()
```

And here is the same functionality provided using await/async:

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

getFirstUserData()
```

Multiple async functions in series

Async functions can be chained very easily, and the syntax is much more readable than with plain promises:

```
const promiseToDoSomething = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('I did something'), 10000)
  })
}

const watchOverSomeoneDoingSomething = async () => {
  const something = await promiseToDoSomething()
  return something + ' and I watched'
}

const watchOverSomeoneWatchingSomeoneDoingSomething = async () => {
  const something = await watchOverSomeoneDoingSomething()
  return something + ' and I watched as well'
}

watchOverSomeoneWatchingSomeoneDoingSomething().then(res => {
  console.log(res)
})
```

Will print:

```
I did something and I watched and I watched as well
```

Easier debugging

Debugging promises is hard because the debugger will not step over asynchronous code.

Async/await makes this very easy because to the compiler it's just like synchronous code.

The Node Event Emitter

How to work with custom events in Node

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 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 an EventEmitter object using this syntax:

```
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

Emit and listen for events

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.

`addListener()` is an alias for `on()`, in case you see that used.

Passing arguments to the event

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)
```

Listen for an event just once

The EventEmitter object also exposes the `once()` method, which you can use to create a one-time event listener.

Once that event is fired, the listener stops listening.

Example:

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

eventEmitter.emit('start')
eventEmitter.emit('start') //not going to fire
```

Removing an event listener

Once you create an event listener, you can remove it using the `removeListener()` method.

To do so, we must first have a reference to the callback function of `on`.

In this example:

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

Extract the callback:

```
const callback = () => {
  console.log('started')
}

eventEmitter.on('start', callback)
```

So that later you can call

```
eventEmitter.removeListener('start', callback)
```

You can also remove all listeners at once on an event, using:

```
eventEmitter.removeAllListeners('start')
```

Getting the events registered

The `eventNames()` method, called on an EventEmitter object instance, returns an array of strings that represent the events registered on the current EventListener:

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

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

eventEmitter.eventNames() // [ 'start' ]
```

`listenerCount()` returns the count of listeners of the event passed as parameter:

```
eventEmitter.listenerCount('start') //1
```

Adding more listeners before/after other ones

If you have multiple listeners, the order of them might be important.

An EventEmitter object instance offers some methods to work with order.

`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.

HTTP

A detailed description of how the HTTP protocol, and the Web, work

HTTP (*Hyper Text Transfer Protocol*) is one of the application protocols of TCP/IP, the suite of protocols that powers the Internet.

Let me fix that: it's not *one* of the protocols, it's the most successful and popular one, by all means.

HTTP is what makes the World Wide Web work, giving browsers a language to communicate to remote servers that host web pages.

HTTP was first standardized in 1991, as a result of the work that Tim Berners-Lee did at CERN, the European Center of Nuclear Research, since 1989.

The goal was to allow researchers to easily exchange and interlink their papers. It was meant as a way for the scientific community to work better.

Back then the internet main applications basically consisted in FTP (the File Transfer Protocol), Email and Usenet (newsgroups, today almost abandoned).

In 1993 Mosaic, the first graphical web browser, was released, and things skyrocketed from there.

The Web became the killer app of the Internet.

Over time the Web and the ecosystem around it have dramatically evolved, but the basics still remain. One example of evolution: HTTP now powers, in addition to web pages, REST APIs, one common way to programmatically access a service over the Internet.

HTTP got a minor revision in 1997 with HTTP/1.1, and in 2015 its successor, HTTP/2, was standardized and it's now being implemented by the major Web Servers used across the globe.

The HTTP protocol is considered insecure, just like any other protocol (SMTP, FTP..) not served over an encrypted connection. This is why there is a big push nowadays towards using HTTPS, which is HTTP served over TLS.

That said, the building blocks of HTTP/2 and HTTPS have their roots in HTTP, and in this article I'll introduce how HTTP works.

HTML documents

HTTP is the way **web browsers** like Chrome, Firefox, Edge and many others (also called *clients* from here on) communicate with **web servers**.

The name Hyper Text Transfer Protocol derives from the need of transferring not just files, like in FTP - the "File Transfer Protocol", but hypertexts, which would be written using HTML, and then represented graphically by the browser with a nice presentation and interactive links.

Links were the driving force that drove adoption, along with the ease of creation of new web pages.

HTTP is what transfer those hypertext files (and as we'll see also images and other file types) over the network.

Hyperlinks

Inside a web browser, a document can point to another document using links.

A link is composed by a first part that determines the protocol and the server address, either through a domain name or an IP.

This part is not unique to HTTP, of course.

Then there's the document part. Anything appended to the address part represents the document path.

For example, this document address is `https://flaviocopes.com/http/`:

- `https` is the protocol.
- `flaviocopes.com` is the domain name that points to my server
- `/http/` is the document URL relative to the server root path.

The path can be nested: `https://flaviocopes.com/page/privacy/` and in this case the document URL is `/page/privacy`.

The web server is responsible for interpreting the request and, once analyzed, serving the correct response.

A request

What's in a request?

The first thing is **the URL**, which we've already seen before.

When we enter an address and press enter in our browser, under the hoods the server sends to the correct IP address a request like this:

```
GET /a-page
```

where /a-page is the URL you requested.

The second thing is **the HTTP method** (also called verb).

HTTP in the early days defined 3 of them:

- `GET`
- `POST`
- `HEAD`

and HTTP/1.1 introduced

- `PUT`
- `DELETE`
- `OPTIONS`
- `TRACE`

We'll see them in detail in a minute.

The third thing that composes a request is a set of **HTTP headers**.

Headers are a set of `key: value` pairs that are used to communicate to the server-specific information that is predefined, so the server can know what we mean.

I described them in detail in the HTTP request headers list.

Give that list a quick look. All of those headers are optional, except `Host`.

HTTP methods

`GET`

GET is the most used method here. It's the one that's used when you type an URL in the browser address bar, or when you click a link.

It asks the server to send the requested resource as a response.

`HEAD`

HEAD is just like GET, but tells the server to not send the response body back. Just the headers.

`POST`

The client uses the POST method to send data to the server. It's typically used in forms, for example, but also when interacting with a REST API.

`PUT`

The PUT method is intended to create a resource at that specific URL, with the parameters passed in the request body. Mainly used in REST APIs

`DELETE`

The DELETE method is called against an URL to request deletion of that resource. Mainly used in REST APIs

`OPTIONS`

When a server receives an OPTIONS request it should send back the list of HTTP methods allowed for that specific URL.

`TRACE`

Returns back to the client the request that has been received. Used for debugging or diagnostic purposes.

HTTP Client/Server communication

HTTP, as most of the protocols that belong to the TCP/IP suite, is a *stateless* protocol.

Servers have no idea what's the current state of the client. All they care about is that they get request and they need to fulfill them.

Any prior request is meaningless in this context, and this makes it possible for a web server to be very fast, as there's less to process, and also it gives it bandwidth to handle a lot of concurrent requests.

HTTP is also very lean, and communication is very fast in terms of overhead. This contrasts with the protocols that were the most used at the time HTTP was introduced: TCP and POP/SMTP, the mail protocols, which involve lots of handshaking and confirmations on the

receiving ends.

Graphical browsers abstract all this communication, but we'll illustrate it here for learning purposes.

A message is composed by a first line, which starts with the HTTP method, then contains the resource relative path, and the protocol version:

```
GET /a-page HTTP/1.1
```

After that, we need to add the HTTP request headers. As mentioned above, there are many headers, but the only mandatory one is `Host`:

```
GET /a-page HTTP/1.1
Host: flaviocopes.com
```

How can you test this? Using **telnet**. This is a command-line tool that lets us connect to any server and send it commands.

Open your terminal, and type `telnet flaviocopes.com 80`

This will open a terminal, that tells you

```
Trying 178.128.202.129...
Connected to flaviocopes.com.
Escape character is '^]'.
```

You are connected to the Netlify web server that powers my blog. You can now type:

```
GET /axios/ HTTP/1.1
Host: flaviocopes.com
```

and press enter on an empty line to fire the request.

The response will be:

```
HTTP/1.1 301 Moved Permanently
Cache-Control: public, max-age=0, must-revalidate
Content-Length: 46
Content-Type: text/plain
Date: Sun, 29 Jul 2018 14:07:07 GMT
Location: https://flaviocopes.com/axios/
Age: 0
Connection: keep-alive
Server: Netlify

Redirecting to https://flaviocopes.com/axios/
```

See, this is an HTTP response we got back from the server. It's a 301 Moved Permanently request. See the HTTP status codes list to know more about the status codes.

It basically tells us the resource has permanently moved to another location.

Why? Because we connected to port 80, which is the default for HTTP, but on my server I set up an automatic redirection to HTTPS.

The new location is specified in the `Location` HTTP response header.

There are other headers, all described in the HTTP response headers list.

In both the request and the response, an empty line separates the request header from the request body. The response body in this case contains the string

```
Redirecting to https://flaviocopes.com/axios/
```

which is 46 bytes long, as specified in the `Content-Length` header. It is shown in the browser when you open the page, while it automatically redirects you to the correct location.

In this case we're using telnet, the low-level tool that we can use to connect to any server, so we can't have any kind of automatic redirect.

Let's do this process again, now connecting to port 443, which is the default port of the HTTPS protocol. We can't use telnet because of the SSL handshake that must happen.

Let's keep things simple and use `curl`, another command-line tool. We cannot directly type the HTTP request, but we'll see the response:

```
curl -i https://flaviocopes.com/axios/
```

this is what we'll get in return:

```
HTTP/1.1 200 OK
Cache-Control: public, max-age=0, must-revalidate
Content-Type: text/html; charset=UTF-8
Date: Sun, 29 Jul 2018 14:20:45 GMT
Etag: "de3153d6eacef2299964de09db154b32-ssl"
Strict-Transport-Security: max-age=31536000
Age: 152
Content-Length: 9797
Connection: keep-alive
Server: Netlify

<!DOCTYPE html>
<html prefix="og: http://ogp.me/ns#" lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<title>HTTP requests using Axios</title>
....
```

I cut the response, but you can see that the HTML of the page is being returned now.

Other resources

An HTTP server will not just transfer HTML files, but typically it will also serve other files: CSS, JS, SVG, PNG, JPG, lots of different file types.

This depends on the configuration.

HTTP is perfectly capable of transferring those files as well, and the client will know about the file type, thus interpret them in the right way.

This is how the web works: when an HTML page is retrieved by the browser, it's interpreted and any other resource it needs to display property (CSS, JavaScript, images..) is retrieved through additional HTTP requests to the same server.

How HTTP Requests work

What happens when you type an URL in the browser, from start to finish

> This article describes how browsers perform page requests using the HTTP/1.1 protocol

If you ever did an interview, you might have been asked: "what happens when you type something into the Google search box and press enter".

It's one of the most popular questions you get asked. People just want to see if you can explain some rather basic concepts and if you have any clue how the internet actually works.

In this post, I'll analyze what happens when you type an URL in the address bar of your browser and press enter.

It's a very interesting topic to dissect in a blog post, as it touches many technologies I can dive into in separate posts.

This is tech that is very rarely changed, and powers one the most complex and wide ecosystems ever built by humans.

The HTTP protocol

First, I mention HTTPS in particular because things are different from an HTTPS connection.

I analyze URL requests only

Modern browsers have the capability of knowing if the thing you wrote in the address bar is an actual URL or a search term, and they will use the default search engine if it's not a valid URL.

I assume you type an actual URL.

When you enter the URL and press enter, the browser first builds the full URL.

If you just entered a domain, like `flaviocopes.com`, the browser by default will prepend `HTTP://` to it, defaulting to the HTTP protocol.

Things relate to macOS / Linux

Just FYI. Windows might do some things slightly differently.

DNS Lookup phase

The browser starts the DNS lookup to get the server IP address.

The domain name is a handy shortcut for us humans, but the internet is organized in such a way that computers can look up the exact location of a server through its IP address, which is a set of numbers like `222.324.3.1` (IPv4).

First, it checks the DNS local cache, to see if the domain has already been resolved recently.

Chrome has a handy DNS cache visualizer you can see at chrome://net-internals/#dns

If nothing is found there, the browser uses the DNS resolver, using the `gethostbyname` POSIX system call to retrieve the host information.

gethostbyname

`gethostbyname` first looks in the local hosts file, which on macOS or Linux is located in `/etc/hosts`, to see if the system provides the information locally.

If this does not give any information about the domain, the system makes a request to the DNS server.

The address of the DNS server is stored in the system preferences.

Those are 2 popular DNS servers:

- `8.8.8.8`: the Google public DNS server

- `1.1.1.1` : the CloudFlare DNS server

Most people use the DNS server provided by their internet provider.

The browser performs the DNS request using the UDP protocol.

TCP and UDP are two of the foundational protocols of computer networking. They sit at the same conceptual level, but TCP is connection-oriented, while UDP is a connectionless protocol, more lightweight, used to send messages with little overhead.

> How the UDP request is performed is not in the scope of this tutorial

The DNS server might have the domain IP in the cache. If not, it will ask the **root DNS server**. That's a system (composed of 13 actual servers, distributed across the planet) that drives the entire internet.

The DNS server does *not* know the address of each and every domain name on the planet.

What it knows is where the **top-level DNS resolvers** are.

A top-level domain is the domain extension: `.com` , `.it` , `.pizza` and so on.

Once the root DNS server receives the request, it forwards the request to that top-level domain (TLD) DNS server.

Say you are looking for `flaviocopes.com` . The root domain DNS server returns the IP of the .com TLD server.

Now our DNS resolver will cache the IP of that TLD server, so it does not have to ask the root DNS server again for it.

The TLD DNS server will have the IP addresses of the authoritative Name Servers for the domain we are looking for.

> How? When you buy a domain, the domain registrar sends the appropriate TDL the name servers. When you update the name servers (for example, when you change the hosting provider), this information will be automatically updated by your domain registrar.

Those are the DNS servers of the hosting provider. They are usually more than 1, to serve as backup.

For example:

- `ns1.dreamhost.com`
- `ns2.dreamhost.com`
- `ns3.dreamhost.com`

The DNS resolver starts with the first, and tries to ask the IP of the domain (with the subdomain, too) you are looking for.

That is the ultimate source of truth for the IP address.

Now that we have the IP address, we can go on in our journey.

TCP request handshaking

With the server IP address available, now the browser can initiate a TCP connection to that.

A TCP connection requires a bit of handshaking before it can be fully initialized and you can start sending data.

Once the connection is established, we can send the request

Sending the request

The request is a plain text document structured in a precise way determined by the communication protocol.

It's composed of 3 parts:

- the request line
- the request header
- the request body

The request line

The request line sets, on a single line:

- the HTTP method
- the resource location
- the protocol version

Example:

```
GET / HTTP/1.1
```

The request header

The request header is a set of `field: value` pairs that set certain values.

There are 2 mandatory fields, one of which is `Host`, and the other is `Connection`, while all the other fields are optional:

```
Host: flaviocopes.com
Connection: close
```

`Host` indicates the domain name which we want to target, while `Connection` is always set to `close` unless the connection must be kept open.

Some of the most used header fields are:

- `Origin`
- `Accept`
- `Accept-Encoding`
- `Cookie`
- `Cache-Control`
- `Dnt`

but many more exist.

The header part is terminated by a blank line.

The request body

The request body is optional, not used in GET requests but very much used in POST requests and sometimes in other verbs too, and it can contain data in JSON format.

Since we're now analyzing a GET request, the body is blank and we'll not look more into it.

The response

Once the request is sent, the server processes it and sends back a response.

The response starts with the status code and the status message. If the request is successful and returns a 200, it will start with:

```
200 OK
```

The request might return a different status code and message, like one of these:

```
404 Not Found
403 Forbidden
301 Moved Permanently
500 Internal Server Error
304 Not Modified
401 Unauthorized
```

The response then contains a list of HTTP headers and the response body (which, since we're making the request in the browser, is going to be HTML)

Parse the HTML

The browser now has received the HTML and starts to parse it, and will repeat the exact same process we did for all the resources required by the page:

- CSS files
- images
- the favicon
- JavaScript files
- ...

How browsers render the page then is out of the scope, but it's important to understand that the process I described is not just for the HTML pages, but for any item that's served over HTTP.

Build an HTTP server

How to build an HTTP server with Node.js

Here is the HTTP web server we used as the Node Hello World application in the Node.js introduction

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

const hostname = 'localhost'
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}/`)
})
```

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 hostname, `localhost`, on 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/plain')
```

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

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

Making HTTP requests

How to perform HTTP requests with Node.js using GET, POST, PUT and DELETE

> I use the term HTTP, but HTTPS is what should be used everywhere, therefore these examples use HTTPS instead of HTTP.

Perform a GET Request

```
const https = require('https')
const options = {
  hostname: 'flaviocopes.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

```
const https = require('https')

const data = JSON.stringify({
  todo: 'Buy the milk'
})

const options = {
  hostname: 'flaviocopes.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, and just change the `options.method` value.

Axios

Axios is a very convenient JavaScript library to perform HTTP requests in Node.js

Introduction

Axios is a very popular JavaScript library you can use to perform HTTP requests, that works in both Browser and Node.js platforms.

It supports all modern browsers, including support for IE8 and higher.

It is promise-based, and this lets us write async/await code to perform XHR requests very easily.

Using Axios has quite a few advantages over the native Fetch API:

- supports older browsers (Fetch needs a polyfill)
- has a way to abort a request
- has a way to set a response timeout
- has built-in CSRF protection
- supports upload progress
- performs automatic JSON data transformation
- works in Node.js

A video tutorial

Check out this video where I create an Express server that offers a POST endpoint, and I make an Axios request to it, to post data:

Installation

Axios can be installed using npm:

```
npm install axios
```

or yarn:

```
yarn add axios
```

or include it in your page using unpkg.com:

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

The Axios API

You can start an HTTP request from the `axios` object:

> I use `foo` and `bar` as *random names*. Enter any kind of name to replace them.

```
axios({
  url: 'https://dog.ceo/api/breeds/list/all',
  method: 'get',
  data: {
    foo: 'bar'
  }
})
```

but for convenience, you will generally use

- `axios.get()`
- `axios.post()`

(like in jQuery you would use `$.get()` and `$.post()` instead of `$.ajax()`)

Axios offers methods for all the HTTP verbs, which are less popular but still used:

- `axios.delete()`
- `axios.put()`
- `axios.patch()`
- `axios.options()`

and a method to get the HTTP headers of a request, discarding the body:

- `axios.head()`

GET requests

One convenient way to use Axios is to use the modern (ES2017) async/await syntax.

This Node.js example queries the Dog API to retrieve a list of all the dogs breeds, using `axios.get()`, and it counts them:

```
const axios = require('axios')

const getBreeds = async () => {
  try {
    return await axios.get('https://dog.ceo/api/breeds/list/all')
  } catch (error) {
    console.error(error)
  }
}

const countBreeds = async () => {
  const breeds = await getBreeds()

  if (breeds.data.message) {
    console.log(`Got ${Object.entries(breeds.data.message).length} breeds`)
  }
}

countBreeds()
```

If you don't want to use async/await you can use the Promises syntax:

```
const axios = require('axios')

const getBreeds = () => {
  try {
    return axios.get('https://dog.ceo/api/breeds/list/all')
  } catch (error) {
    console.error(error)
  }
}

const countBreeds = async () => {
  const breeds = getBreeds()
    .then(response => {
      if (response.data.message) {
        console.log(
          `Got ${Object.entries(response.data.message).length} breeds`
        )
      }
    })
    .catch(error => {
      console.log(error)
    })
}

countBreeds()
```

Add parameters to GET requests

A GET response can contain parameters in the URL, like this: `https://site.com/?foo=bar` .

With Axios you can perform this by using that URL:

```
axios.get('https://site.com/?foo=bar')
```

or you can use a `params` property in the options:

```
axios.get('https://site.com/', {
  params: {
    foo: 'bar'
  }
})
```

POST Requests

Performing a POST request is just like doing a GET request, but instead of `axios.get` , you use `axios.post` :

```
axios.post('https://site.com/')
```

An object containing the POST parameters is the second argument:

```
axios.post('https://site.com/', {
  foo: 'bar'
})
```

Websockets

WebSockets are an alternative to HTTP communication in Web Applications. They offer a long lived, bidirectional communication channel between client and server.

WebSockets are an alternative to HTTP communication in Web Applications.

They offer a long lived, bidirectional communication channel between client and server.

Once established, the channel is kept open, offering a very fast connection with low latency and overhead.

Browser support for WebSockets

WebSockets are supported by all modern browsers.

 Browser support for WebSockets

How WebSockets differ from HTTP

HTTP is a very different protocol, and also a different way of communicate.

HTTP is a request/response protocol: the server returns some data when the client requests it.

With WebSockets:

- the **server can send a message to the client** without the client explicitly requesting something
- the client and the server can **talk to each other simultaneously**
- **very little data overhead** needs to be exchanged to send messages. This means a **low latency communication**.

WebSockets are great for **real-time** and **long-lived** communications.

HTTP is great for **occasional data exchange** and interactions initiated by the client.

HTTP is much simpler to implement, while WebSockets require a bit more overhead.

Secured WebSockets

Always use the secure, encrypted protocol for WebSockets, `wss://` .

`ws://` refers to the unsafe WebSockets version (the `http://` of WebSockets), and should be avoided for obvious reasons.

Create a new WebSockets connection

```
const url = 'wss://myserver.com/something'
const connection = new WebSocket(url)
```

`connection` is a WebSocket object.

When the connection is successfully established, the `open` event is fired.

Listen for it by assigning a callback function to the `onopen` property of the `connection` object:

```
connection.onopen = () => {
  //...
}
```

If there's any error, the `onerror` function callback is fired:

```
connection.onerror = error => {
  console.log(`WebSocket error: ${error}`)
}
```

Sending data to the server using WebSockets

Once the connection is open, you can send data to the server.

You can do so conveniently inside the `onopen` callback function:

```
connection.onopen = () => {
  connection.send('hey')
}
```

Receiving data from the server using WebSockets

Listen with a callback function on `onmessage`, which is called when the `message` event is received:

```
connection.onmessage = e => {
  console.log(e.data)
}
```

Implement a WebSockets server in Node.js

ws is a popular WebSockets library for Node.js.

We'll use it to build a WebSockets server. It can also be used to implement a client, and use WebSockets to communicate between two backend services.

Easily install it using

```
yarn init
yarn add ws
```

The code you need to write is very little:

```
const WebSocket = require('ws')

const wss = new WebSocket.Server({ port: 8080 })

wss.on('connection', ws => {
  ws.on('message', message => {
    console.log(`Received message => ${message}`)
  })
  ws.send('ho!')
})
```

This code creates a new server on port 8080 (the default port for WebSockets), and adds a callback function when a connection is established, sending `ho!` to the client, and logging the messages it receives.

See a live example on Glitch

Here is a live example of a WebSockets server: https://glitch.com/edit/#!/flavio-websockets-server-example

Here is a WebSockets client that interacts with the server: https://glitch.com/edit/#!/flavio-websockets-client-example

HTTPS, secure connections

The HTTPS protocol is an extension of HTTP, the Hyper Text Transfer Protocol, that provide secure communication

HTTP in insecure by design.

When you open your browser and ask a web server to send you a webpage, your data performs 2 trips: 1 from the browser to the web server, and 1 from the web server to the browser.

Then, depending on the content of the web page, you might have more connections required to get the CSS files, the JavaScript files, images, and so on.

During any of those connections, any network your data is going to cross can be **inspected** and **manipulated**.

The consequences can be serious: you might have all your network activity monitored and logged, by a 3rd party you are not even aware it exist, some networks might inject ads, and you might be subject to a man-in-the-middle attack, a security threat where the attacker can manipulate your data and even impersonate your computer over the network. It's very easy for someone to just listen to HTTP packets being transmitted over a public and unencrypted Wi-Fi network.

HTTPS aims to solve the problem at the root: the entire communication between your browser and the web server is encrypted.

Privacy and security are a major concern in today's internet. A few years ago, you could get away with just using an encrypted connection in login-protected pages, or during an e-commerce checkout. Also because of SSL certificates pricing and complications, most websites just used HTTP.

Today HTTPS is a requirement on any site. More than 50% of the whole Web uses it now. Google Chrome recently started marking HTTP sites as insecure, just to give you a valid reason to have HTTPS mandatory (and forced) on all your websites.

When using HTTP the default server port is 80, and on HTTPS it's 443. It does not need to be explicitly added if the server uses the default port, of course.

HTTPS is also sometimes called *HTTP over SSL*, or *HTTP over TLS*.

The difference between the two is simple: TLS is the successor of SSL.

When using HTTPS, the only thing that is not encrypted is the web server domain, and the server port.

Every other information, including the resource path, headers, cookies and query parameters are all encrypted.

I won't go in the details of analyzing how the TLS protocol works under the hoods, but you might think it's adding a good amount of **overhead**, and you would be right.

Any computation that's added to the processing of network resources causes overhead both on the client, the server, and to the transmitted packets size.

However HTTPS enables the use of the newest protocol **HTTP/2**, which has a huge advantage over HTTP/1.1: it way faster.

Why? There are many reasons, one is header compression, one is resource multiplexing. One is server push: the server can push more resources when one resource is requested. So, if the browser requests a page, it will also receive all the resources needed (images, CSS, JS).

Details aside, HTTP/2 is a huge improvement over HTTP/1.1 **and it requires HTTPS**. This means that HTTPS, despite having the encryption overhead, happens to be way faster than HTTP, if things are properly configured with a modern setup.

File descriptors

How to interact with file descriptors using 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 what's returned by opening the file using the `open()` method offered by the `fs` module:

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

fs.open('/Users/flavio/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
- `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 instead of providing the file descriptor object in a callback, it returns it:

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

try {
  const fd = fs.openSync('/Users/flavio/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.open()` and many other operations that interact with the filesystem.

File stats

How to get the details of a file using Node

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

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

You call it passing a file path, and once Node 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/flavio/test.txt', (err, stats) => {
  if (err) {
    console.error(err)
    return
  }
  //we have access to the file stats in `stats`
})
```

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

```
const fs = require('fs')
try {
  const stats = fs.stat('/Users/flavio/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/flavio/test.txt', (err, stats) => {
  if (err) {
    console.error(err)
    return
  }

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

File paths

How to interact with file paths and manipulate them in Node

Every file in the system has a path.

On Linux and macOS, a path might look like:

`/users/flavio/file.txt`

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

`C:\users\flavio\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/flavio/notes.txt'

path.dirname(notes) // /users/flavio
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 = 'flavio'
path.join('/', 'users', name, 'notes.txt') //'/users/flavio/notes.txt'
```

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

```
path.resolve('flavio.txt') //'/Users/flavio/flavio.txt' if run from my home folder
```

In this case Node will append `/flavio.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', 'flavio.txt')//'/Users/flavio/tmp/flavio.txt' if run from my home fol
```

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

```
path.resolve('/etc', 'flavio.txt')//'/etc/flavio.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/flavio/..//test.txt') ///users/test.txt
```

Both resolve and normalize will not check if the path exists. They just calculate a path based on the information they got.

Reading files

How to read files using Node and the `fs` module

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

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

fs.readFile('/Users/flavio/test.txt', (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/flavio/test.txt', 'utf8')
  console.log(data)
} catch (err) {
  console.error(err)
}
```

The default encoding is utf8, but you can specify a custom encoding using a a second parameter.

Both `fs.readFile()` and `fs.readFileSync()` 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

How to write files using 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/flavio/test.txt', content, (err) => {
  if (err) {
    console.error(err)
    return
  }
  //file written successfully
})
```

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

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

const content = 'Some content!'

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

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/flavio/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 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 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)
    return
  }
  //done!
})
```

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

How to interact with folders using Node

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()` to check if the folder exists and Node can access it with its permissions.

Create a new folder

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

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

const folderName = '/Users/flavio/test'

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

Read the content of a directory

Use `fs.readdir()` or `fs.readdirSync` 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 path = require('path')

const folderPath = '/Users/flavio'

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()` to rename folder. The first parameter is the current path, the second the new path:

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

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

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

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

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

Remove a folder

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

Removing a folder that has content can be more complicated than you need.

In this case I recommend installing the `fs-extra` module, which is very popular and well maintained, and it's a drop-in replacement of the `fs` module, providing 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/flavio'

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/flavio'
removeFolder(folder)
```

The fs module

The fs module of Node.js provides useful functions to interact with the file system

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 core, it can be used by 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 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()` : set the file mode
- `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 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.

The path module

The path module of Node.js provides useful functions to interact with file paths

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 core, it can be used by 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()`
- `path.dirname()`
- `path.extname()`
- `path.isAbsolute()`
- `path.join()`
- `path.normalize()`
- `path.parse()`
- `path.relative()`
- `path.resolve()`

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

path.normalize()

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

```
require('path').normalize('/users/flavio/..//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/flavio', '/Users/flavio/test.txt') //'test.txt'
require('path').relative('/Users/flavio', '/Users/flavio/something/test.txt') //'somethir
```

path.resolve()

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

```
path.resolve('flavio.txt') //'/Users/flavio/flavio.txt' if run from my home folder
```

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

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

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

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

The os module

The os module of Node.js provides useful functions to interact with underlying system

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.

> When I say Linux and macOS I mean POSIX platforms. For simplicity I exclude other less popular operating systems Node can run on.

`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()`
- `os.cpus()`
- `os.endianness()`
- `os.freemem()`
- `os.homedir()`
- `os.hostname()`
- `os.loadavg()`
- `os.networkInterfaces()`
- `os.platform()`
- `os.release()`
- `os.tmpdir()`
- `os.totalmem()`
- `os.type()`
- `os.uptime()`
- `os.userInfo()`

`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.endianness()`

Return `BE` or `LE` depending if Node was compiled with Big Endian or Little Endian.

`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/flavio'
```

`os.hostname()`

Return the hostname.

`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 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 information about the current user

The events module

The events module of Node.js provides the EventEmitter class

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

I published a full article on that, so here I will just describe the API without further examples on how to use it.

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

The event listener eats its own dog food and uses these 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()`
- `emitter.emit()`
- `emitter.eventNames()`
- `emitter.getMaxListeners()`
- `emitter.listenerCount()`
- `emitter.listeners()`
- `emitter.off()`
- `emitter.on()`
- `emitter.once()`
- `emitter.prependListener()`
- `emitter.prependOnceListener()`
- `emitter.removeAllListeners()`
- `emitter.removeListener()`
- `emitter.setMaxListeners()`

`emitter.addListener()`

Alias for `emitter.on()`.

`emitter.emit()`

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

`emitter.eventNames()`

Return an array of strings that represent the events registered on the current EventListener:

```
door.eventNames()
```

`emitter.getMaxListeners()`

Get the maximum amount of listeners one can add to an EventListener 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 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 event emitter 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 EventListener object, which defaults to 10 but can be increased or lowered.

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

The http module

The http module of Node.js provides useful functions and classes to build an HTTP server

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

- Properties
  - `http.METHODS`
  - `http.STATUS_CODES`
  - `http.globalAgent`
- Methods
  - `http.createServer()`
  - `http.request()`
  - `http.get()`
- Classes
  - `http.Agent`
  - `http.ClientRequest`
  - `http.Server`
  - `http.ServerResponse`
  - `http.IncomingMessage`

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 persistance and reuse for HTTP clients, and it's a key component of Node HTTP networking.

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

Methods

`http.createServer()`

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

Streams

Learn what streams are for, why are they so important, and how to use them.

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 build.

Why streams

Streams basically provide two major advantages 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 as soon as you have it, rather than waiting till the whole data payload is available to start

An example of a stream

A typical example is the one of reading files from a disk.

Using the Node `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 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

```
const Stream = require('stream')
const readableStream = new Stream.Readable()
```

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()
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()
const writableStream = new Stream.Writable()

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

readableStream.pipe(writableStream)

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

writableStream.end()
```

Working with MySQL

MySQL is one of the most popular relational databases in the world. Find out how to make it work with Node.js

MySQL is one of the most popular relational databases in the world.

The Node ecosystem of course has several different packages that allow you to interface with MySQL, store data, retrieve data, and so on.

We'll use `mysqljs/mysql`, a package that has over 12.000 GitHub stars and has been around for years.

Installing the Node mysql package

You install it using

```
npm install mysql
```

Initializing the connection to the database

You first include the package:

```
const mysql = require('mysql')
```

and you create a connection:

```
const options = {
  user: 'the_mysql_user_name',
  password: 'the_mysql_user_password',
  database: 'the_mysql_database_name'
}
const connection = mysql.createConnection(options)
```

You initiate a new connection by calling:

```
connection.connect(err => {
  if (err) {
    console.error('An error occurred while connecting to the DB')
    throw err
  }
})
```

The connection options

In the above example, the `options` object contained 3 options:

```
const options = {
  user: 'the_mysql_user_name',
  password: 'the_mysql_user_password',
  database: 'the_mysql_database_name'
}
```

There are many more you can use, including:

- `host`, the database hostname, defaults to `localhost`
- `port`, the MySQL server port number, defaults to 3306
- `socketPath`, used to specify a unix socket instead of host and port
- `debug`, by default disabled, can be used for debugging
- `trace`, by default enabled, prints stack traces when errors occur
- `ssl`, used to setup an SSL connection to the server (out of the scope of this tutorial)

Perform a SELECT query

Now you are ready to perform a SQL query on the database. The query once executed will invoke a callback function which contains an eventual error, the results and the fields.

```
connection.query('SELECT * FROM todos', (error, todos, fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }
  console.log(todos)
})
```

You can pass in values which will be automatically escaped:

```
const id = 223
connection.query('SELECT * FROM todos WHERE id = ?', [id], (error, todos, fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }
  console.log(todos)
})
```

To pass multiple values, just put more elements in the array you pass as the second parameter:

```
const id = 223
const author = 'Flavio'
connection.query('SELECT * FROM todos WHERE id = ? AND author = ?', [id, author], (error,
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }
  console.log(todos)
})
```

Perform an INSERT query

You can pass an object

```
const todo = {
  thing: 'Buy the milk'
  author: 'Flavio'
}
connection.query('INSERT INTO todos SET ?', todo, (error, results, fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }
})
```

If the table has a primary key with `auto_increment`, the value of that will be returned in the `results.insertId` value:

```
const todo = {
  thing: 'Buy the milk'
  author: 'Flavio'
}
connection.query('INSERT INTO todos SET ?', todo, (error, results, fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }}
  const id = results.resultId
  console.log(id)
)
```

Close the connection

When you need to terminate the connection to the database you can call the `end()` method:

```
connection.end()
```

This makes sure any pending query gets sent, and the connection is gracefully terminated.

Difference between development and production

Learn how to set up different configurations for production and development environments

You can have different configurations for production and development environments.

Node 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.

Express provides configuration hooks specific to the environment, which are automatically called based on the NODE_ENV variable value:

```
app.configure('development', () => {
  //...
})
app.configure('production', () => {
  //...
})
app.configure('production', 'staging', () => {
  //...
})
```

For example you can use this to set different error handlers for different mode:

```
app.configure('development', () => {
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
})

app.configure('production', () => {
  app.use(express.errorHandler())
})
```