NODE.js HANDBOOK

FLAVIO COPES

Preface

This book aims to be an introduction to Node.js, the most popular server-side JavaScript runtime.

If you're unfamiliar with JavaScript, before reading this book I highly recommend reading the JavaScript Handbook and the TypeScript Handbook.

This book was updated in 2025 for Node.js v22 LTS, the current Long Term Support version.

Legal

Introduction to Node.js

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

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

As of 2025, Node.js v22 is the current LTS (Long Term Support) version, offering stable and production-ready features including native support for ES modules, built-in test runner, Web Crypto API, and enhanced performance.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent

JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

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

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

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

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

An example Node.js application

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

```
import { createServer } from 'node:http'

const hostname = '127.0.0.1'
const port = 3000

const server = 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.mjs` file (or `server.js` with `"type": "module"` in your package.json) and run `node server.mjs` in your terminal.

This code first imports the Node.js `http` module using ES modules syntax, which is now the recommended approach in modern Node.js. Note the use of the `node:` prefix which is the recommended way to import Node.js built-in modules.

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')
```

How to install Node.js

Node.js can be installed in different ways.

Let me show 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. For production use, Node.js v22 LTS is recommended as it provides stability and will be supported until April 2027.

On the site they have packages for Windows, Linux, and 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 you to install Node.js very easily, by running this command in the CLI:

```
brew install node
```

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

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

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

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

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

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

Differences between Node and the Browser

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

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

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

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

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

What changes is the ecosystem.

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

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

Another big difference is that in Node.js you control the environment. Unless you are building an open source application that anyone can deploy anywhere, you know which version of Node.js you will run the application on. Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very convenient.

This means that you can write all the modern ES2015+ JavaScript that your Node version supports, including features like async/await, optional chaining, nullish coalescing, private class fields, and more.

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 or other transpilers to transform your code for older browsers, but in Node.js v22, you can use all modern JavaScript features natively without any transpilation.

Another difference is that Node originally used the CommonJS module system (`require` / `module.exports`), while browsers use ES Modules (`import` / `export`). Since Node.js v12, ES Modules are fully supported and are now the recommended module system.

In Node.js v22, ES Modules are stable and performant. You can use them by:

- Using `.mjs` file extensions
- Setting `"type": "module"` in your `package.json`
- Using the `--input-type=module` flag

While CommonJS (`require`) is still supported for backward compatibility, ES Modules are the future of JavaScript and should be used for new projects.

Run Node.js scripts from the command line

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

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

```
node app.js
```

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

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

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

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

// your code
```

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

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

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

Restart the application automatically

Built-in Watch Mode (Recommended)

Starting from Node.js v18, you can use the built-in `--watch` flag to automatically restart your application when files change:

```
node --watch app.js
```

This is the recommended approach as it doesn't require any external dependencies.

Using nodemon (Legacy)

For older Node.js versions or more advanced features, you can use `nodemon`:

```
npm i -D nodemon
npx nodemon app.js
```

However, the built-in `--watch` flag is sufficient for most use cases in Node.js v22.

How to read environment variables

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

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

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

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

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

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

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

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

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

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

In your js file

```
// Using ES modules
import 'dotenv/config'
// Or with CommonJS: require('dotenv').config()

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

Built-in .env File Support

Starting from Node.js v20.6.0, you can load .env files natively without any external packages:

```
node --env-file=.env app.js
```

This built-in feature eliminates the need for the dotenv package in most cases. You can also load multiple .env files:

```
node --env-file=.env --env-file=.env.local app.js
```

For older Node.js versions or more advanced features, you can still use the dotenv package with `node -r dotenv/config index.js`.

Getting arguments from the command line

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

```
node app.js
```

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

For example:

```
node app.js joe
```

or

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

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

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

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

The first element is the full path of the `node` command.

The second element is the full path of the file being executed.

All the additional arguments are present from the third position going forward.

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

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

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

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

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

```
node app.js joe
```

you can access it using

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

In this case:

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

`args[0]` is `name=joe`, and you need to parse it.

Using Built-in parseArgs (Recommended)

Starting from Node.js v18.3.0, you can use the built-in `parseArgs` utility:

```
import { parseArgs } from 'node:util'

const { values } = parseArgs({
  args: process.argv.slice(2),
  options: {
    name: {
      type: 'string',
    },
  },
})

console.log(values.name) // joe
```

Run it with:

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

Using Third-Party Libraries

For more complex argument parsing needs, you can use libraries like `minimist` or `yargs`, but the built-in `parseArgs` is sufficient for most use cases.

Output to the command line

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

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

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

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

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

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

and Node.js will print both.

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

For example:

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

- `%s` format a variable as a string
- `%d` format a variable as a number
- `%i` format a variable as its integer part only
- `%o` format a variable as an object

Example:

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

Accept input from the command line

Node.js provides several ways to accept input from users in command-line applications.

The simplest way is using the `readline` module, which provides an interface for reading data from a readable stream (like `process.stdin`) one line at a time.

Here's how to ask a question and wait for user input

```
import { createInterface } from 'node:readline'

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

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

For more complex scenarios, you can use the promise-based version:

```
import { createInterface } from 'node:readline/promises'
import { stdin as input, stdout as output } from 'node:process'

const readline = createInterface({ input, output })

const answer = await readline.question('What is your name? ')
console.log(`Hello ${answer}!`)

readline.close()
```

Node.js Module System

Node.js has a built-in module system. A Node.js file can import functionality exposed by other Node.js files.

When you want to import something you use:

```
import library from './library.js'
```

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.

In modern Node.js (v12+), we use ES Modules with `export` and `import` statements. This is now the recommended approach for new projects.

You can export values in several ways:

Default Export - Export a single value as the default:

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

export default car
```

```
// index.js
import car from './car.js'
```

Named Exports - Export multiple values:

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

export const truck = {
  brand: 'Chevy',
  model: 'Silverado',
}
```

```
// index.js
import { car, truck } from './car.js'
```

Mixed Exports - Combine default and named exports:

```
// car.js
const defaultCar = {
  brand: 'Tesla',
  model: 'Model S',
}

export const fordCar = {
  brand: 'Ford',
  model: 'Fiesta',
}

export default defaultCar
```

```
// app.js
import tesla, { fordCar } from './car.js'
```

```
console.log(tesla, fordCar)
```

CommonJS vs ES Modules

While Node.js still supports CommonJS (`require` / `module.exports`) for backward compatibility, ES Modules are the standard for JavaScript and should be used for new projects. To use ES Modules:

1. Add `"type": "module"` to your `package.json`
2. Use `.mjs` file extension
3. Or use the `--input-type=module` flag

ES Modules provide better static analysis, tree shaking, and are the future of JavaScript modules.

Top-level Await

ES modules in Node.js support top-level await, allowing you to use `await` at the module's top level without wrapping it in an async function:

```
// config.js
// You can now use await at the top level
const response = await fetch('https://api.example.com/config')
const config = await response.json()

export default config
```

```
// app.js
import config from './config.js'

// The config is already resolved when imported
console.log(config)
```

This is particularly useful for:

- Loading configuration from external sources
- Establishing database connections
- Reading files at module initialization
- Any asynchronous setup that needs to complete before the module is used

Note that modules using top-level await will delay the loading of importing modules until the awaited promises are resolved.

The Node Event emitter

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

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

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

You initialize that using

```
import { EventEmitter } from 'node:events'

const eventEmitter = new EventEmitter()
```

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

- `emit` is used to trigger an event
- `on` is used to add a callback function that's going to be executed when the event is triggered

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

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

When we run

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

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

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

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

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

Multiple arguments:

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

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

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

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

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

Working with file descriptors in Node

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

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

```
import fs from 'node:fs'

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

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

That flag means we open the file for reading.

Other flags you'll commonly use are:

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

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

```
import fs from 'node:fs'

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

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

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

The `fs/promises` module provides promise-based versions of all fs methods. This is the recommended approach for modern Node.js applications as it works seamlessly with async/await.

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

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

File information

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

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

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

```
import fs from 'node:fs'
```

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

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

```
import fs from 'node:fs'

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

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

A lot, including:

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

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

```
import fs from 'node:fs'

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

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

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

```
import fs from 'node:fs/promises'

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

File paths

Every file in the system has a path.

On Linux and macOS, a path might look like:

`/users/joe/file.txt`

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

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

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

You include this module in your files using

```
import path from 'node:path'
```

and you can start using its methods.

Getting information out of a path

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

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

Example:

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

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

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

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

Working with paths

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

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

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

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

In this case Node.js will simply append `/joe.txt` to the current working directory. If you specify a second parameter folder, `resolve` will use the first as a base for the second:

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

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

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

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

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

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

Reading files

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

```
import fs from 'node:fs'

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

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

```
import fs from 'node:fs'

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

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

```
import fs from 'node:fs/promises'

async function example() {
  try {
    const data = await fs.readFile('/Users/joe/test.txt', { encoding:
'utf8' })
    console.log(data)
  } catch (err) {
    console.error(err)
  }
}
example()
```

All three of `fs.readFile()`, `fs.readFileSync()` and `fsPromises.readFile()` read the full content of the file in memory before returning the data.

This means that big files are going to have a major impact on your memory consumption and speed of execution of the program.

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

Writing files

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

Example:

```
import fs from 'node:fs'

const content = 'Some content!'

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

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

```
import fs from 'node:fs'

const content = 'Some content!'

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

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

```
import fs from 'node:fs/promises'

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

```
  } catch (err) {
    console.error(err)
  }
}
example()
```

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

You can modify the default by specifying a flag:

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

The flags you'll likely use are

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

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

Append to a file

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

```
const content = 'Some content!'

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

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

```
import fs from 'node:fs/promises'

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

```
  } catch (err) {
    console.error(err)
  }
}
example()
```

Using streams

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

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

Working with folders

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

Check if a folder exists

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

Create a new folder

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

```
import fs from 'node:fs'

const folderName = '/Users/joe/test'

try {
  // Check if folder exists using fs.accessSync (fs.existsSync is
deprecated)
  try {
    fs.accessSync(folderName)
  } catch {
    fs.mkdirSync(folderName)
  }
} catch (err) {
  console.error(err)
}
```

Read the content of a directory

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

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

```
import fs from 'node:fs'

const folderPath = '/Users/joe'

fs.readdirSync(folderPath)
```

You can get the full path:

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

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

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

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

Rename a folder

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

```
import fs from 'node:fs'

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

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

```
import fs from 'node:fs'

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

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

```
import fs from 'node:fs/promises'

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

Remove a folder

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

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

```
import fs from 'node:fs'

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

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

> NOTE: In Node.js v22, use `fs.rm` or `fs.rmSync` with the `{ recursive: true }` option to delete folders that have content:

```
import fs from 'node:fs'

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

```
  }

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

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

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

Install it using

```
npm install fs-extra
```

and use it like this:

```
import fs from 'fs-extra'

const folder = '/Users/joe'

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

It can also be used with promises:

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

or with async/await:

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

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

The Node fs module

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

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

```
import fs from 'node:fs'
```

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

- `fs.access()` : check if the file exists and Node.js can access it with its permissions
- `fs.appendFile()` : append data to a file. If the file does not exist, it's created
- `fs.chmod()` : change the permissions of a file specified by the filename passed. Related: `fs.lchmod()` , `fs.fchmod()`
- `fs.chown()` : change the owner and group of a file specified by the filename passed. Related: `fs.fchown()` , `fs.lchown()`
- `fs.close()` : close a file descriptor
- `fs.copyFile()` : copies a file
- `fs.createReadStream()` : create a readable file stream
- `fs.createWriteStream()` : create a writable file stream
- `fs.link()` : create a new hard link to a file
- `fs.mkdir()` : create a new folder
- `fs.mkdtemp()` : create a temporary directory
- `fs.open()` : opens the file and returns a file descriptor to allow file manipulation
- `fs.readdir()` : read the contents of a directory
- `fs.readFile()` : read the content of a file. Related: `fs.read()`
- `fs.readlink()` : read the value of a symbolic link
- `fs.realpath()` : resolve relative file path pointers (`.` , `..`) to the full path
- `fs.rename()` : rename a file or folder
- `fs.rmdir()` : remove a folder
- `fs.stat()` : returns the status of the file identified by the filename passed. Related: `fs.fstat()` , `fs.lstat()`
- `fs.symlink()` : create a new symbolic link to a file
- `fs.truncate()` : truncate to the specified length the file identified by the filename passed. Related: `fs.ftruncate()`
- `fs.unlink()` : remove a file or a symbolic link

- `fs.unwatchFile()`: stop watching for changes on a file
- `fs.utimes()`: change the timestamp of the file identified by the filename passed. Related: `fs.futimes()`
- `fs.watchFile()`: start watching for changes on a file. Related: `fs.watch()`
- `fs.writeFile()`: write data to a file. Related: `fs.write()`

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

For example:

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

This makes a huge difference in your application flow.

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

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

```
import fs from 'node: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:

```
import fs from 'node:fs'

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

```
}
```

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

You should use the promise-based API provided by `fs/promises` module with async/await for cleaner, more readable code:

```
// Example: Read a file and change its content and read
// it again using callback-based API.
import fs from 'node:fs'

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

For multiple operations, async/await provides much cleaner code than nested callbacks:

```
// Example: Read a file and change its content and read
// it again using promise-based API.
import fs from 'node:fs/promises'
```

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

The Node path module

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

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

```
import path from 'node:path'
```

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

These are the `path` methods:

`path.basename()`

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

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

`path.dirname()`

Return the directory part of a path:

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

path.extname()

Return the extension part of a path

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

path.format()

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

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

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

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

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

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

path.isAbsolute()

Returns true if it's an absolute path

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

`path.join()`

Joins two or more parts of a path:

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

`path.normalize()`

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

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

`path.parse()`

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

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

Example:

```
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:

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

`path.resolve()`

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

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

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

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

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

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

The Node os module

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

```
import os from 'node:os'
```

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

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

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

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

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

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

41.1. `os.arch()`

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

41.2. `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,
    },
  },
]
```

```
*/
```

41.3. `os.freemem()`

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

41.4. `os.homedir()`

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

Example:

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

41.5. `os.hostname()`

Return the host name.

41.6. `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]
```

41.7. `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 } ] }
```

41.8. `os.platform()`

Return the platform that Node.js was compiled for:

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

41.9. `os.release()`

Returns a string that identifies the operating system release number

41.10. `os.tmpdir()`

Returns the path to the assigned temp folder.

41.11. `os.totalmem()`

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

41.12. `os.type()`

Identifies the operating system:

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

41.13. `os.uptime()`

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

41.14. `os.userInfo()`

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

The Node events module

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

```
import { EventEmitter } from 'node:events'

const door = new EventEmitter()
```

The event listener has these in-built events:

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

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

42.1. `emitter.addListener()`

Alias for `emitter.on()`.

`emitter.emit()`

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

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

`emitter.eventNames()`

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

```
door.eventNames()
```

`emitter.getMaxListeners()`

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

```
door.getMaxListeners()
```

42.5. `emitter.listenerCount()`

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

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

42.6. `emitter.listeners()`

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

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

42.7. `emitter.off()`

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

42.8. `emitter.on()`

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

Usage:

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

42.9. `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.

```
import { EventEmitter } from 'node:events'

const ee = new EventEmitter()

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

`emitter.prependListener()`

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

`emitter.prependOnceListener()`

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

`emitter.removeAllListeners()`

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

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

`emitter.removeListener()`

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

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

42.14. `emitter.setMaxListeners()`

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

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

The Node http module

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

It can be included using

```
import http from 'node:http'
```

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

43.1. Properties

43.1.1. `http.METHODS`

This property lists all the HTTP methods supported:

```
> 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' ]
```

43.1.2. `http.STATUS_CODES`

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

```
> 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' }
```

43.1.3. `http.globalAgent`

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

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

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

43.2. Methods

43.2.1. `http.createServer()`

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

Usage:

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

43.2.2. `http.request()`

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

43.2.3. `http.get()`

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

43.3. Classes

The HTTP module provides 5 classes:

- `http.Agent`
- `http.ClientRequest`
- `http.Server`

- `http.ServerResponse`
- `http.IncomingMessage`

43.3.1. `http.Agent`

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

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

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

43.3.2. `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.

43.3.3. `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

43.3.4. `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'
```

43.3.5. `http.IncomingMessage`

An `http.IncomingMessage` object is created by:

- `http.Server` when listening to the `request` event
- `http.ClientRequest` when listening to the `response` event

It can be used to access the response:

- status using its `statusCode` and `statusMessage` methods
- headers using its `headers` method or `rawHeaders`
- HTTP method using its `method` method

- HTTP version using the `httpVersion` method
- URL using the `url` method
- underlying socket using the `socket` method

The data is accessed using streams, since `http.IncomingMessage` implements the Readable Stream interface.

Node.js Streams

44.1. What are streams

Streams are one of the fundamental concepts that power Node.js applications.

They are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

Streams are not a concept unique to Node.js. They were introduced in the Unix operating system decades ago, and programs can interact with each other passing streams through the pipe operator (`|`).

For example, in the traditional way, when you tell the program to read a file, the file is read into memory, from start to finish, and then you process it.

Using streams you read it piece by piece, processing its content without keeping it all in memory.

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

44.2. Why streams

Streams basically provide two major advantages over using other data handling methods:

- **Memory efficiency**: you don't need to load large amounts of data in memory before you are able to process it
- **Time efficiency**: it takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available

44.3. An example of a stream

A typical example is reading files from a disk.

Using the Node.js `fs` module, you can read a file, and serve it over HTTP when a new connection is established to your HTTP server:

```
import http from 'node:http'
import fs from 'node:fs'
import { fileURLToPath } from 'node:url'
import { dirname, join } from 'node:path'

// Note: In ES modules, __dirname is not available. We need to create it:
const __dirname = dirname(fileURLToPath(import.meta.url))

const server = http.createServer(function (req, res) {
  fs.readFile(join(__dirname, 'data.txt'), (err, data) => {
    if (err) {
      res.statusCode = 500
      res.end('Error loading file')
      return
    }
    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:

```
import http from 'node:http'
import fs from 'node:fs'
import { fileURLToPath } from 'node:url'
import { dirname, join } from 'node:path'

const __dirname = dirname(fileURLToPath(import.meta.url))

const server = http.createServer((req, res) => {
  const stream = fs.createReadStream(join(__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.

44.4. 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)
```

44.5. Streams-powered Node.js APIs

Due to their advantages, many Node.js core modules provide native stream handling capabilities, most notably:

- `process.stdin` returns a stream connected to stdin
- `process.stdout` returns a stream connected to stdout
- `process.stderr` returns a stream connected to stderr
- `fs.createReadStream()` creates a readable stream to a file
- `fs.createWriteStream()` creates a writable stream to a file
- `net.connect()` initiates a stream-based connection
- `http.request()` returns an instance of the http.ClientRequest class, which is a writable stream
- `zlib.createGzip()` compress data using gzip (a compression algorithm) into a stream
- `zlib.createGunzip()` decompress a gzip stream.
- `zlib.createDeflate()` compress data using deflate (a compression algorithm) into a stream
- `zlib.createInflate()` decompress a deflate stream

44.6. 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

44.7. How to create a readable stream

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

First create a stream object:

```
import Stream from 'node:stream'

const readableStream = new Stream.Readable()
```

then implement `_read`:

```
readableStream._read = () => {}
```

You can also implement `_read` using the `read` option:

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

Now that the stream is initialized, we can send data to it:

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

44.8. 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:

```
import Stream from 'node: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)
```

44.9. How to get data from a readable stream

How do we read data from a readable stream? Using a writable stream:

```
import Stream from 'node:stream'

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

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

readableStream.pipe(writableStream)

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

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

You can also consume a readable stream directly, using the `readable` event:

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

44.10. How to send data to a writable stream

Using the stream `write()` method:

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

44.11. Signaling a writable stream that you ended writing

Use the `end()` method:

```
import Stream from 'node:stream'

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

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

readableStream.pipe(writableStream)

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

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

```
readableStream.destroy()
```

In the above example, `end()` is called within a listener to the `close` event on the readable stream to ensure it is not called before all write events have passed through the pipe, as doing so would cause an `error` event to be emitted.
Calling `destroy()` on the readable stream causes the `close` event to be emitted.
The listener to the `close` event on the writable stream demonstrates the completion of the process as it is emitted after the call to `end()`.

44.12. How to create a transform stream

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

First create a transform stream object:

```
import { Transform } from 'node:stream'

const transformStream = new Transform()
```

then implement `_transform`:

```
transformStream._transform = (chunk, encoding, callback) => {
  transformStream.push(chunk.toString().toUpperCase())
  callback()
}
```

Pipe readable stream:

```
process.stdin.pipe(transformStream).pipe(process.stdout)
```

Web Crypto API

Node.js provides the Web Crypto API, offering cryptographic operations compatible with browsers. This API provides a standard way to perform cryptographic operations like hashing, encryption, and key generation:

```
import { webcrypto } from 'node:crypto'
```

```
// Hashing data with SHA-256
async function hashData(data) {
  const encoder = new TextEncoder()
  const dataBuffer = encoder.encode(data)
  const hashBuffer = await webcrypto.subtle.digest('SHA-256', dataBuffer)
  return Array.from(new Uint8Array(hashBuffer))
    .map(b => b.toString(16).padStart(2, '0'))
    .join('')
}

const hash = await hashData('Hello World')
console.log(hash) //
a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e

// Generating cryptographic keys
async function generateKeyPair() {
  const keyPair = await webcrypto.subtle.generateKey(
    {
      name: 'RSA-OAEP',
      modulusLength: 2048,
      publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
      hash: 'SHA-256',
    },
    true,
    ['encrypt', 'decrypt']
  )
  return keyPair
}
```

The Web Crypto API is particularly useful when:

- Building isomorphic applications that run in both Node.js and browsers
- Needing cryptographic operations that follow web standards
- Working with secure random number generation
- Implementing encryption/decryption workflows

For Node.js-specific crypto operations, you can still use the traditional `crypto` module, but the Web Crypto API provides better cross-platform compatibility.

The difference between development and production

You can have different configurations for production and development environments.

Node.js assumes it's always running in a development environment.
You can signal Node.js that you are running in production by setting the `NODE_ENV=production` environment variable.

This is usually done by executing the command

```
export NODE_ENV=production
```

in the shell, but it's better to put it in your shell configuration file (e.g. `.bash_profile` with the Bash shell) because otherwise the setting does not persist in case of a system restart.

You can also apply the environment variable by prepending it to your application initialization command:

```
NODE_ENV=production node app.js
```

This environment variable is a convention that is widely used in external libraries as well.

Setting the environment to `production` generally ensures that

- logging is kept to a minimum, essential level
- more caching levels take place to optimize performance

For example Pug, the templating library used by Express, compiles in debug mode if `NODE_ENV` is not set to `production`. Express views are compiled in every request in development mode, while in production they are cached. There are many more examples.

You can use conditional statements to execute code in different environments:

```
if (process.env.NODE_ENV === 'development') {
  // ...
}
if (process.env.NODE_ENV === 'production') {
  // ...
}
if (['production', 'staging'].indexOf(process.env.NODE_ENV) >= 0) {
  // ...
}
```

For example, in an Express app, you can use this to set different error handlers per environment:

```
if (process.env.NODE_ENV === 'development') {
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }))
}
```

```
if (process.env.NODE_ENV === 'production') {
  app.use(express.errorHandler())
}
```

Error handling

Errors in Node.js are handled through exceptions.

Creating exceptions

An exception is created using the `throw` keyword:

```
throw value
```

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

Usually in client-side code `value` can be any JavaScript value including a string, a number or an object.

In Node.js, we don't throw strings, we just throw Error objects.

Error objects

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

```
throw new Error('Ran out of coffee')
```

or

```
class NotEnoughCoffeeError extends Error {
  // ...
}
throw new NotEnoughCoffeeError()
```

Handling exceptions

An exception handler is a `try` / `catch` statement.

Any exception raised in the lines of code included in the `try` block is handled in the corresponding `catch` block:

```
try {
  // lines of code
} catch (e) {}
```

`e` in this example is the exception value.

You can add multiple handlers, that can catch different kinds of errors.

Catching uncaught exceptions

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

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

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

You don't need to import the `process` core module for this, as it's automatically injected.

Exceptions with promises

Using promises you can chain different operations, and handle errors at the end:

```
doSomething1()
  .then(doSomething2)
  .then(doSomething3)
  .catch((err) => console.error(err))
```

How do you know where the error occurred? You don't really know, but you can handle errors in each of the functions you call (`doSomethingX`), and inside the error handler throw a new error, that's going to call the outside `catch` handler:

```
const doSomething1 = () => {
  // ...
  try {
```

```
    // ...
  } catch (err) {
    // ... handle it locally
    throw new Error(err.message)
  }
  // ...
}
```

To be able to handle errors locally without handling them in the function we call, we can break the chain. You can create a function in each `then()` and process the exception:

```
doSomething1()
  .then(() => {
    return doSomething2().catch((err) => {
      // handle error
      throw err // break the chain!
    })
  })
  .then(() => {
    return doSomething3().catch((err) => {
      // handle error
      throw err // break the chain!
    })
  })
  .catch((err) => console.error(err))
```

Error handling with async/await

Using async/await, you still need to catch errors, and you do it this way:

```
async function someFunction() {
  try {
    await someOtherFunction()
  } catch (err) {
    console.error(err.message)
  }
}
```

Testing in Node.js

Node.js includes a built-in test runner, eliminating the need for external testing frameworks for many use cases:

```
import { test, describe, it } from 'node:test'
import assert from 'node:assert'

describe('Math operations', () => {
  it('should add numbers correctly', () => {
    assert.strictEqual(2 + 2, 4)
  })

  test('async operations', async () => {
    const result = await Promise.resolve(42)
    assert.strictEqual(result, 42)
  })
})
```

Run tests with: `node --test`

The built-in test runner supports:

- Synchronous and asynchronous tests
- Test suites with `describe` and `it`
- Before/after hooks
- Test coverage with `--test-coverage`
- Watch mode with `--test --watch`

For more complex testing needs, you can still use external frameworks like Jest or Mocha, but the built-in test runner handles most common scenarios.

Build an HTTP Server

Here is a sample Hello World HTTP web server:

```
import http from 'node:http'

const port = process.env.PORT || 3000

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/html')
  res.end('<h1>Hello, World!</h1>')
})

server.listen(port, () => {
```

```
  console.log(`Server running at port ${port}`)
})
```

Let's analyze it briefly. We include the `http` module.

We use the module to create an HTTP server.

The server is set to listen on the specified port, `3000`. When the server is ready, the `listen` callback function is called.

The callback function we pass is the one that's going to be executed upon every request that comes in. Whenever a new request is received, the `request` event is called, providing two objects: a request (an `http.IncomingMessage` object) and a response (an `http.ServerResponse` object).

`request` provides the request details. Through it, we access the request headers and request data.

`response` is used to populate the data we're going to return to the client.

In this case with

```
res.statusCode = 200
```

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

We also set the Content-Type header:

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

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

```
res.end('<h1>Hello, World!</h1>')
```

Making HTTP requests

Perform a GET Request

There are many ways to perform an HTTP GET request in Node.js, depending on the abstraction level you want to use.

The simplest way to perform an HTTP request using Node.js is to use the native `fetch` API (available since Node.js v18):

```
try {
  const response = await fetch('https://example.com/todos')
  const data = await response.json()
  console.log(`statusCode: ${response.status}`)
  console.log(data)
} catch (error) {
  console.error(error)
}
```

Alternatively, you can use the Node.js standard `https` module, although it's more verbose:

```
import https from 'node:https'

const options = {
  hostname: 'example.com',
  port: 443,
  path: '/todos',
  method: 'GET',
}

const req = https.request(options, (res) => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', (d) => {
    process.stdout.write(d)
  })
})

req.on('error', (error) => {
  console.error(error)
})

req.end()
```

48.2. Perform a POST Request

Similar to making an HTTP GET request, you can use the native `fetch` API to perform a POST request:

```
try {
  const response = await fetch('https://whatever.com/todos', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      todo: 'Buy the milk',
    }),
  })
  const data = await response.json()
  console.log(`statusCode: ${response.status}`)
  console.log(data)
} catch (error) {
  console.error(error)
}
```

Or alternatively, use Node.js standard modules:

```
import https from 'node:https'

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

const options = {
  hostname: 'whatever.com',
  port: 443,
  path: '/todos',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': data.length,
  },
}

const req = https.request(options, (res) => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', (d) => {
    process.stdout.write(d)
  })
})
```

```
req.on('error', (error) => {
  console.error(error)
})

req.write(data)
req.end()
```

48.3. PUT and DELETE

PUT and DELETE requests use the same POST request format - you just need to change the `options.method` value to the appropriate method.

49. Get HTTP request body data

Here is how you can extract the data that was sent as JSON in the request body.

If you are using Express, that's quite simple: use the `express.json()` middleware which is available in Express v4.16.0 onwards.

For example, to get the body of this request:

```
fetch('https://whatever.com/todos', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    todo: 'Buy the milk',
  }),
})
```

This is the matching server-side code:

```
import express from 'express'

const app = express()

app.use(
  express.urlencoded({
    extended: true,
  })
)

app.use(express.json())
```

```
app.post('/todos', (req, res) => {
  console.log(req.body.todo)
})
```

If you're not using Express and you want to do this in vanilla Node.js, you need to do a bit more work, of course, as Express abstracts a lot of this for you.

The key thing to understand is that when you initialize the HTTP server using `http.createServer()`, the callback is called when the server got all the HTTP headers, but not the request body.

The `request` object passed in the connection callback is a stream.

So, we must listen for the body content to be processed, and it's processed in chunks.

We first get the data by listening to the stream `data` events, and when the data ends, the stream `end` event is called, once:

```
const server = http.createServer((req, res) => {
  // we can access HTTP headers
  req.on('data', (chunk) => {
    console.log(`Data chunk available: ${chunk}`)
  })
  req.on('end', () => {
    // end of data
  })
})
```

So to access the data, assuming we expect to receive a string, we must concatenate the chunks into a string when listening to the stream `data`, and when the stream `end`, we parse the string to JSON:

```
const server = http.createServer((req, res) => {
  let data = ''
  req.on('data', (chunk) => {
    data += chunk
  })
  req.on('end', () => {
    console.log(JSON.parse(data).todo) // 'Buy the milk'
    res.end()
  })
```

```
})
```

The `for await .. of` syntax simplifies async iteration and makes the code more linear and readable:

```
const server = http.createServer(async (req, res) => {
  const buffers = []

  for await (const chunk of req) {
    buffers.push(chunk)
  }

  const data = Buffer.concat(buffers).toString()

  console.log(JSON.parse(data).todo) // 'Buy the milk'
  res.end()
})
```

The Node.js Event Loop

Introduction

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

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

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

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

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

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

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

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

Blocking the event loop

Any JavaScript code that takes too long to return back control to the event loop will block the execution of any JavaScript code in the page, even block the UI thread, and the user cannot click around, scroll the page, and so on.

Almost all the I/O primitives in JavaScript are non-blocking. Network requests, filesystem operations, and so on. Being blocking is the exception, and this is why JavaScript is based so much on callbacks, and more recently on promises and async/await.

The call stack

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

The event loop continuously checks the **call stack** to see if there's any function that needs to run.

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

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

A simple event loop explanation

Let's pick an example:

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

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

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

foo()
```

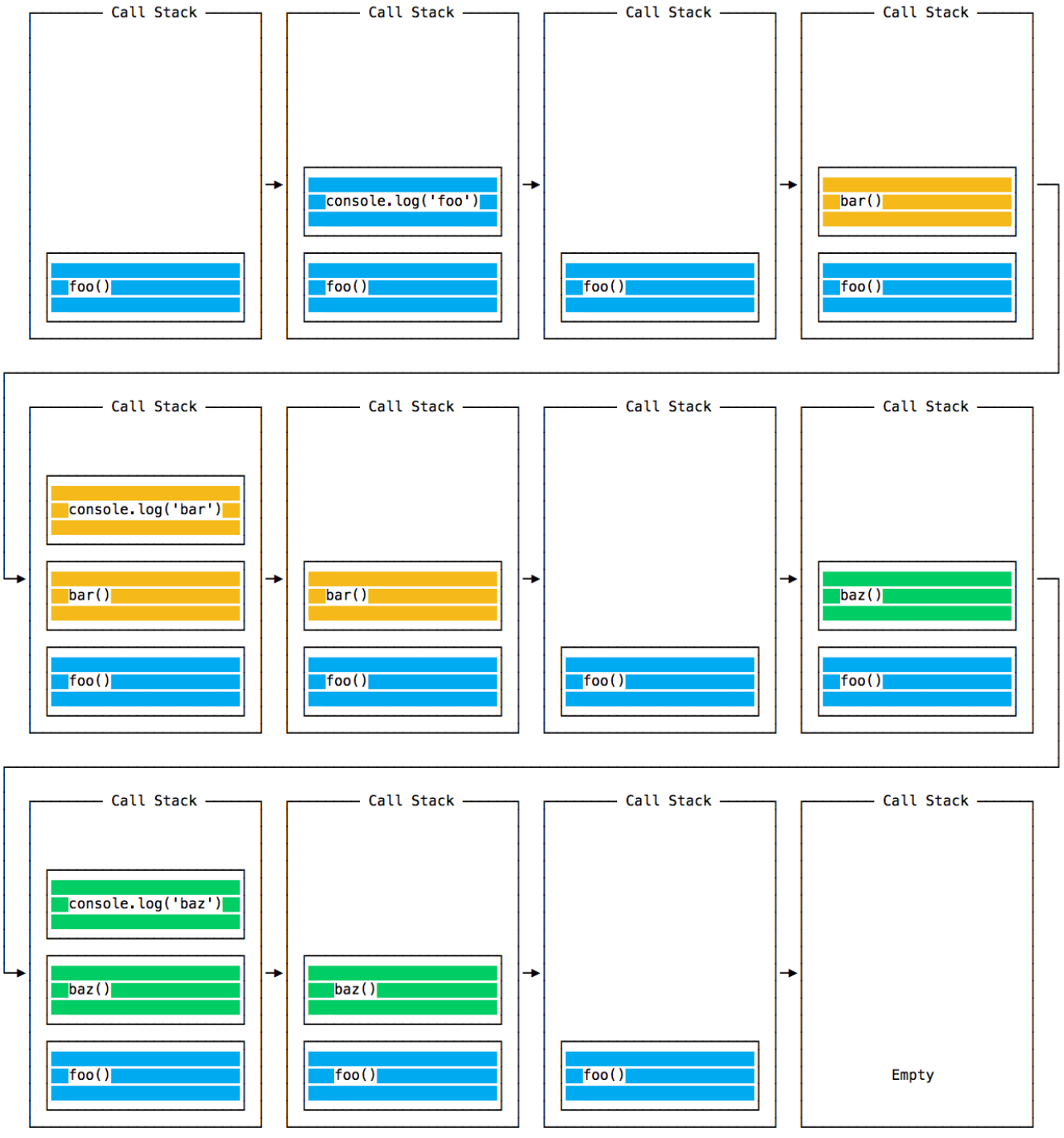
This code prints

```
foo
bar
baz
```

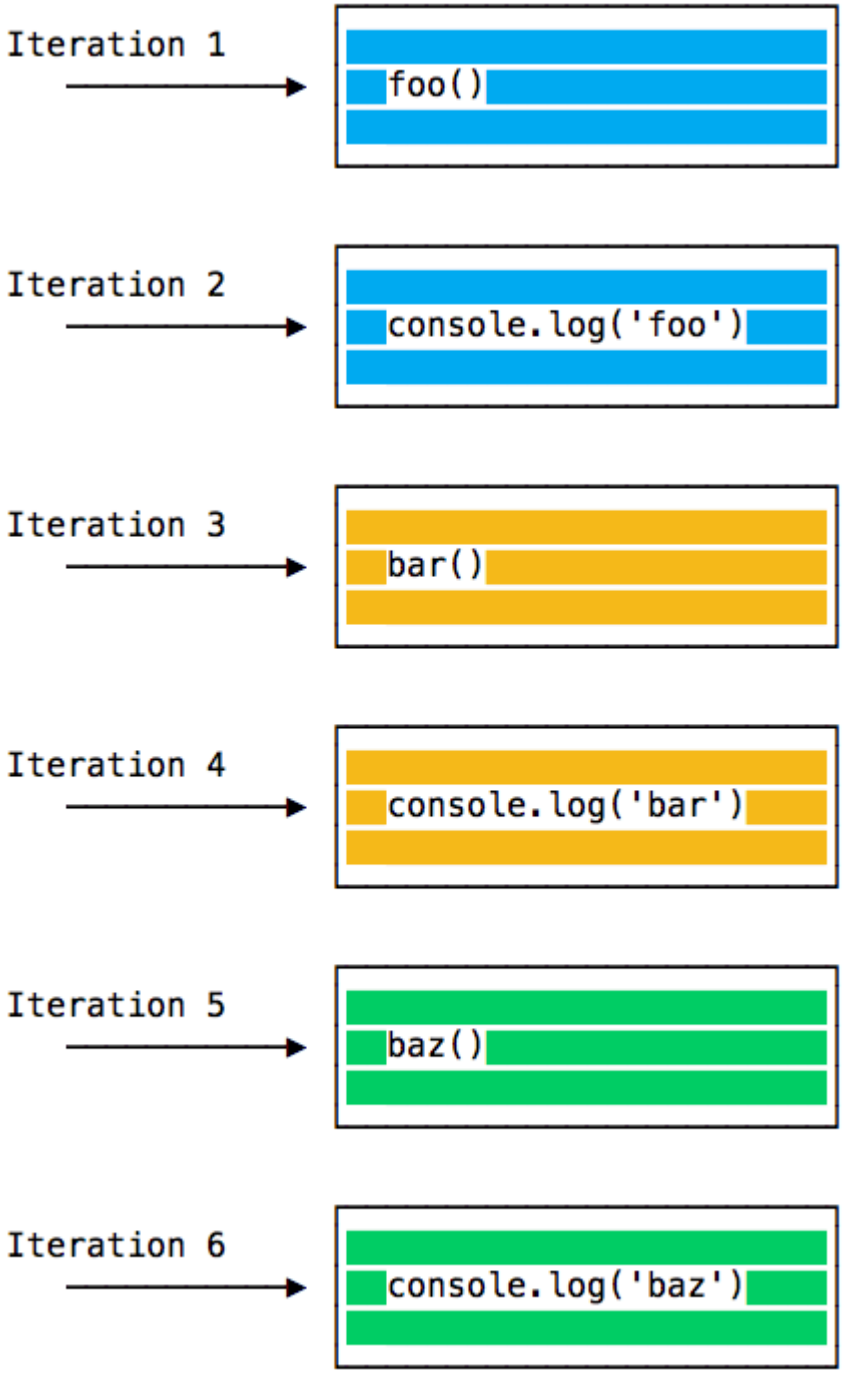
as expected.

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

At this point the call stack looks like this:



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



until the call stack is empty.

28.5. Queuing function execution

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

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

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

Take this example:

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

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

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

foo()
```

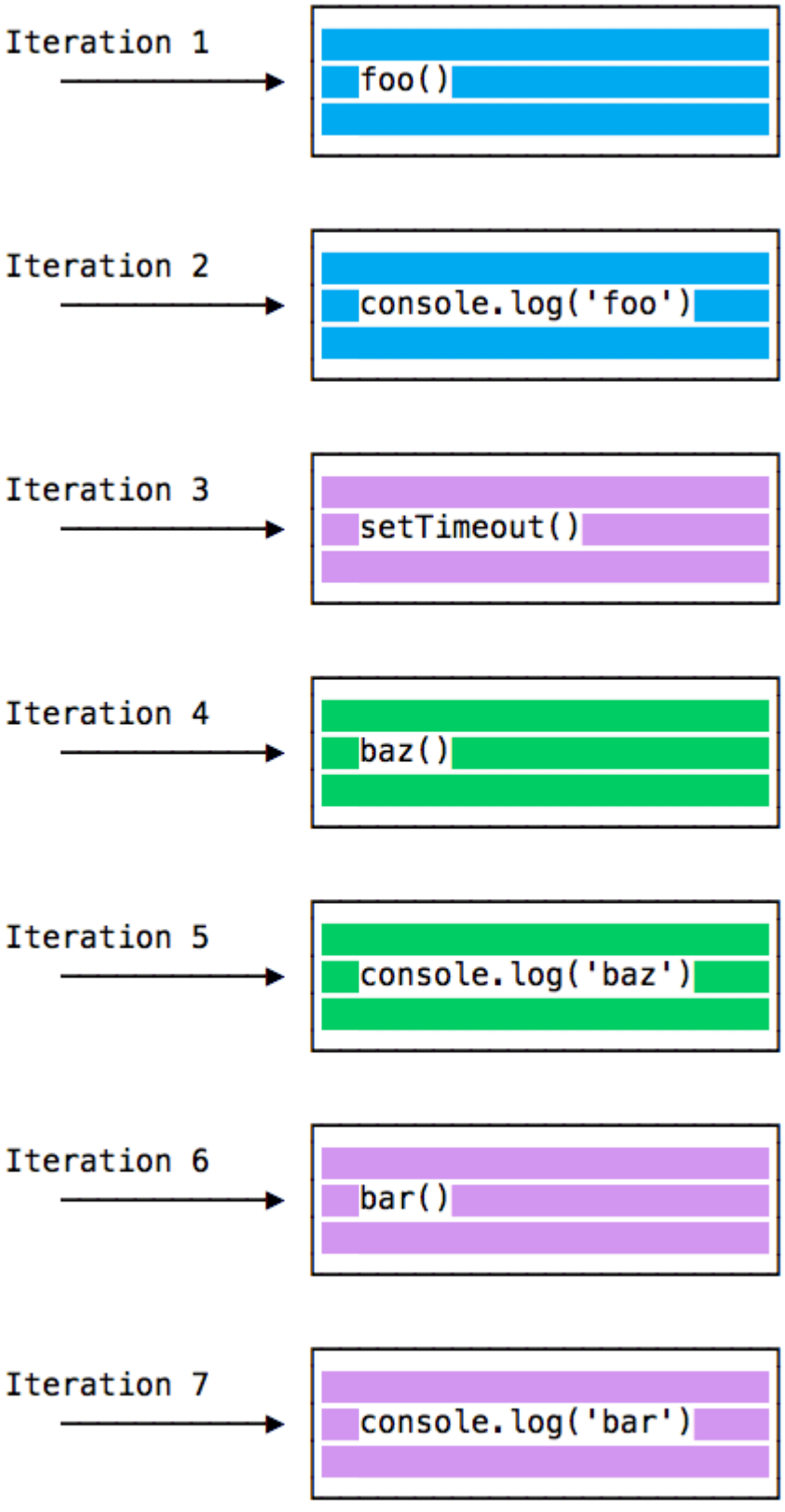
This code prints, maybe surprisingly:

```
foo
baz
bar
```

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

At this point the call stack looks like this:

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



Why is this happening?

28.6. The Message Queue

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

The Message Queue is also where user-initiated events like click or keyboard events, or fetch responses are queued before your code has the opportunity to react to them. Or also DOM events like `onload`.

The loop gives priority to the call stack, and it first processes everything it finds in the call stack, and once there's nothing in there, it goes to pick up things in the message queue.

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

28.7. ES6 Job Queue

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

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

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

Example:

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

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

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

foo()
```

This prints

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

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

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

Understanding process.nextTick()

As you try to understand the Node.js event loop, one important part of it is `process.nextTick()`.

Every time the event loop takes a full trip, we call it a tick.

When we pass a function to `process.nextTick()`, we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts:

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

The event loop is busy processing the current function code.

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

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

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

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

Understanding setImmediate()

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

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

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

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

A function passed to `process.nextTick()` is going to be executed on the current iteration of the event loop, after the current operation ends. This means it will always execute before

`setTimeout` and `setImmediate`.

A `setTimeout()` callback with a 0ms delay is very similar to `setImmediate()`. The execution order will depend on various factors, but they will be both run in the next iteration of the event loop.

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

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

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

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

// start foo bar zoo baz
```

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

Worker Threads

Worker threads provide a way to run JavaScript in parallel, allowing CPU-intensive operations without blocking the main event loop. Unlike child processes, worker threads

share memory with the main thread through SharedArrayBuffer.

```
import { Worker, isMainThread, parentPort, workerData } from
'node:worker_threads'
import { fileURLToPath } from 'node:url'

if (isMainThread) {
  // This code runs in the main thread
  console.log('Main thread starting')

  // Create a new worker
  const worker = new Worker(fileURLToPath(import.meta.url), {
    workerData: { num: 5 }
  })

  // Listen for messages from the worker
  worker.on('message', (result) => {
    console.log('Result from worker:', result)
  })

  worker.on('error', (err) => {
    console.error('Worker error:', err)
  })

  worker.on('exit', (code) => {
    if (code !== 0)
      console.error(`Worker stopped with exit code ${code}`)
  })
} else {
  // This code runs in the worker thread
  console.log('Worker thread starting')

  // Access data passed from main thread
  const result = workerData.num * 2

  // Send result back to main thread
  parentPort.postMessage(result)
}
```

When to Use Worker Threads

Worker threads are ideal for:

- CPU-intensive calculations (cryptography, compression, image processing)
- Large data processing tasks
- Parallel processing of independent tasks
- Background tasks that shouldn't block the main thread

Communication Between Threads

Workers communicate with the main thread through message passing:

```
// main.js
import { Worker } from 'node:worker_threads'

const worker = new Worker('./worker.js')

// Send data to worker
worker.postMessage({ cmd: 'START', data: [1, 2, 3, 4, 5] })

// Receive results
worker.on('message', (result) => {
  console.log('Processed:', result)
})
```

```
// worker.js
import { parentPort } from 'node:worker_threads'

parentPort.on('message', (task) => {
  if (task.cmd === 'START') {
    // Process the data
    const result = task.data.map(n => n * 2)

    // Send result back
    parentPort.postMessage(result)
  }
})
```

Worker threads are a powerful feature for improving Node.js application performance when dealing with CPU-bound operations.

Discover JavaScript Timers

`setTimeout()`

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

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

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

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

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

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

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

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

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

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

Zero delay

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

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

console.log(' before ')
```

This code will print

```
before
after
```

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

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

`setInterval()`

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

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

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

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

clearInterval(id)
```

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

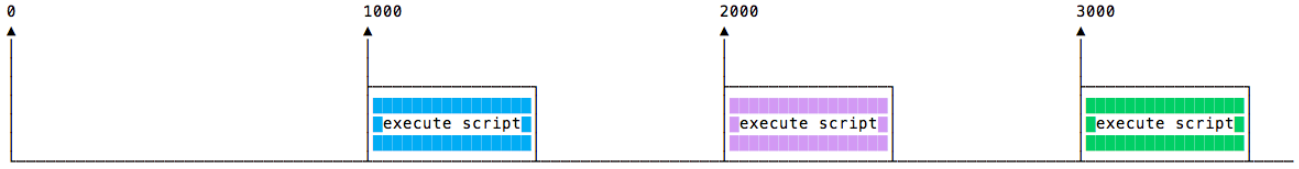
```
const interval = setInterval(() => {
  if (App.somethingIWait === 'arrived') {
    clearInterval(interval)
  }
  // otherwise do things
```

```
}, 100)
```

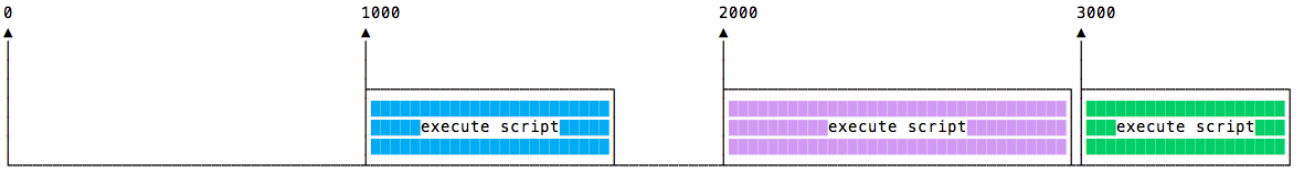
Recursive setTimeout

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

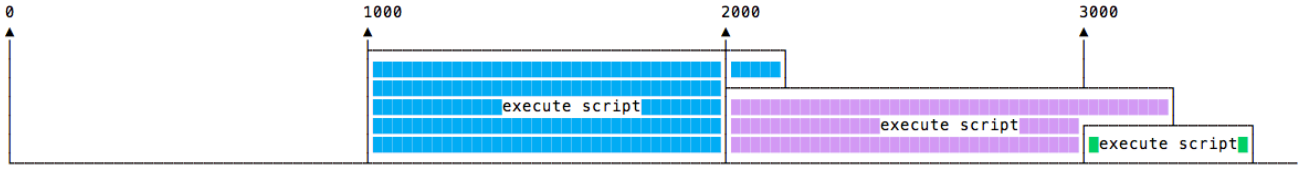
If a function always takes the same amount of time, it's all fine:



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



And maybe one long execution overlaps the next one:



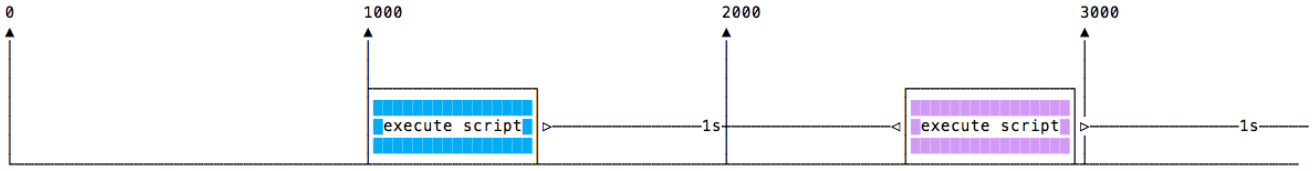
To avoid this, you can schedule a recursive setTimeout to be called when the callback function finishes:

```
const myFunction = () => {
  // do something

  setTimeout(myFunction, 1000)
}

setTimeout(myFunction, 1000)
```

to achieve this scenario:



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

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