Node.js Handbook

Flavio Copes

Updated July 31, 2026

Contents

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 originally published in 2018 and updated in 2025.

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.

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 http from 'node:http'

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

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/')
})
```

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

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

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

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

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

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

Those 2 objects are essential to handle the HTTP call.

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

The second is used to return data to the caller.

In this case with

```
res.statusCode = 200
```

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

We set the Content-Type header:

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

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

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

How to install Node.js

Download and install Node.js from the official website: https://nodejs.org

Choose the LTS (Long Term Support) version for stability or the latest version for newest features.

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 because 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.

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

Node.js includes a built-in watch mode (since Node.js 18) that automatically restarts your application when files change. This eliminates the need for external tools like `nodemon` for most development workflows.

To use watch mode, run your application with the `--watch` flag:

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

You can also watch specific files or directories:

```
node --watch --watch-path=./src app.js
```

Environment Variables

Environment variables are essential for configuring Node.js applications securely and flexibly across different environments.

Basic Environment Variable Usage

A simple way to use environment variables in Node.js is to access them from the `process.env` object. For example:

```
const port = process.env.PORT || 3000
console.log('Server will run on port:', port)
```

This code checks if the `PORT` environment variable is set. If it is, it uses that value; otherwise, it defaults to `3000`.

You can set environment variables when running your script:

```
PORT=4000 node app.js
```

This will print:

```
Server will run on port: 4000
```

If you don't set `PORT`, it will print:

```
Server will run on port: 3000
```

This approach is useful for configuring your app for different environments (development, production, etc.) without changing your code.

For development, use a `.env` file in your project root:

```
# .env file (never commit this to version control!)
NODE_ENV=development
PORT=3000
DATABASE_URL=postgresql://user:pass@localhost:5432/mydb
API_SECRET=your-secret-key-here
```

Load environment variables in your application:

```
// Load environment variables from .env file
import 'dotenv/config'

// Now you can access the variables
const config = {
  port: parseInt(process.env.PORT ?? '3000', 10),
  dbUrl: process.env.DATABASE_URL,
  apiSecret: process.env.API_SECRET,
  isDevelopment: process.env.NODE_ENV === 'development',
  isProduction: process.env.NODE_ENV === 'production'
}

export default config
```

Security Considerations

```
// Never log sensitive environment variables
console.log('Config:', {
  port: config.port,
  nodeEnv: config.nodeEnv,
  // Never do this: apiSecret: config.apiSecret
})

// Use environment-specific defaults
const logLevel = process.env.NODE_ENV === 'production' ? 'warn' : 'debug'
const corsOrigin = process.env.NODE_ENV === 'production'
  ? process.env.ALLOWED_ORIGINS?.split(',')
```

```
: ['http://localhost:3000']
```

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

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. The best way to do so is by using the `minimist` library, which helps dealing with arguments:

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

args.name // joe
```

Modern Package Management

Package Installation Best Practices

```
# Install dependencies (creates package-lock.json)
npm install minimist

# For development dependencies
npm install -D nodemon typescript

# Production installs (uses package-lock.json exactly)
npm ci

# Global installations (avoid when possible)
npm install -g nodemon
```

Security and Maintenance

```
# Check for vulnerabilities
npm audit

# Fix vulnerabilities automatically
npm audit fix

# Check for outdated packages
npm outdated

# Update packages (be careful with major versions)
npm update
```

Package.json Best Practices

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "type": "module",
  "main": "index.js",
  "scripts": {
```

```
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "node --test",
    "build": "tsc"
  },
  "dependencies": {
    "minimist": "^1.2.8"
  },
  "devDependencies": {
    "nodemon": "^3.0.0"
  },
  "engines": {
    "node": ">=18.0.0"
  }
}
```

Install the required `minimist` package using `npm`:

```
npm install minimist
```

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

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

Output to the command line

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.

Accept input from the command line

Node.js provides several ways to accept input from the command line, making your applications interactive and dynamic.

Using the readline module

The simplest way to accept input from the command line is to use the `readline` module, which provides an interface for reading data from a readable stream (such as the `process.stdin` stream).

```
import readline from 'node:readline'

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

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

Using ES Modules in Node.js

To use ES modules (import/export) in Node.js, you have two options:

1. **Add `"type": "module"` to your package.json:**

```
{
  "name": "my-app",
  "type": "module"
}
```

2. **Use `.mjs` file extension** for individual files

With ES modules enabled, you get:

- Top-level `await` support
- `import.meta.url` for current file URL
- Strict mode by default
- Better tree-shaking and optimization

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

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 can get file details using the async/await pattern:

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

try {
  const stats = await fs.stat('/Users/joe/test.txt')
```

```
  // we have access to the file stats in `stats`
} catch (err) {
  console.error(err)
}
```

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 `fs.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.log(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.

Working with paths

Key path module methods:

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

// Extract path information
path.dirname(filePath)   // '/users/joe'
path.basename(filePath)  // 'notes.txt'
path.extname(filePath)   // '.txt'

// Build paths
path.join('/', 'users', 'joe', 'notes.txt') // '/users/joe/notes.txt'
path.resolve('notes.txt') // '/current/working/dir/notes.txt'

// Clean up paths
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

Use `fs.readFile()` with async/await:

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

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

This method reads the full file content into 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

Use `fs.writeFile()` with async/await:

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

const content = 'Some content!'

try {
  await fs.writeFile('/Users/joe/test.txt', content)
} catch (err) {
  console.error(err)
}
```

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

You can modify the default by specifying a flag:

```
fs.writeFile('/Users/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):

```
import fs from 'node:fs'

const content = 'Some content!'

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

Here is a promise-based `fs.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.log(err)
  }
}
example()
```

Using streams

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

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

Working with folders

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

Check if a folder exists

Use `fs.access()` (and its promise-based counterpart from `fs/promises`) 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 the promise-based version from `fs/promises` to create a new folder.

```
import fs from 'node:fs'

const folderName = '/Users/joe/test'

try {
```

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

Read the content of a directory

Use `fs.readdir()` or `fs.readdirSync()` or the promise-based version from `fs/promises` 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 the promise-based version from `fs/promises` 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)
}
```

The promise-based version from `fs/promises` is:

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

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

Remove a folder

Use `fs.rm()` to remove a folder and its contents.

Removing a folder that has content can be done with the `{ recursive: true }` option:

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

try {
  await fs.rm(dir, { recursive: true, force: true })
  console.log(`${dir} is deleted!`)
} catch (err) {
  console.error(err)
}
```

For synchronous operation, you can use:

```
import fs from 'node:fs'

try {
  fs.rmSync(dir, { recursive: true, force: true })
  console.log(`${dir} is deleted!`)
} catch (err) {
  console.error(err)
```

```
}
```

The Node fs module

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

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

```
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.rm()`: remove a file or folder (replaces deprecated `fs.rmdir()`)
- `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.

> Promise-based APIs are now stable and recommended in current Node.js versions (v18+ LTS)

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

```
// Example: Read a file and change its content and read
// it again using modern async/await with proper error handling.
import fs from 'node:fs/promises'

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 (error) {
  console.error('File operation failed:', error.message)
}
```

Error Handling Best Practices

When working with file operations or any asynchronous code in Node.js, proper error handling is crucial:

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

// Always use try/catch with async/await
async function safeFileOperations() {
  try {
    const data = await fs.readFile('config.json', 'utf8')
    const config = JSON.parse(data)
    return config
  } catch (error) {
    // Specific error handling
    if (error.code === 'ENOENT') {
      console.error('File not found')
      return null
    }
    if (error instanceof SyntaxError) {
      console.error('Invalid JSON format')
      return null
    }
    // Re-throw unexpected errors
    throw error
  }
}

// Handle errors at the call site
try {
  const config = await safeFileOperations()
  if (config) {
    console.log('Configuration loaded:', config)
  }
} catch (error) {
  console.error('Unexpected error:', error.message)
  process.exit(1)
}
```

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:

`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()
```

EventEmitter Methods

Key EventEmitter methods for practical use:

- `emitter.on(event, callback)` - Adds listener
- `emitter.once(event, callback)` - Adds one-time listener
- `emitter.removeListener(event, callback)` - Removes specific listener
- `emitter.removeAllListeners(event)` - Removes all listeners for event

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.

Properties

The HTTP module provides access to common HTTP methods and status codes through `http.METHODS` and `http.STATUS_CODES`. These are primarily used for framework development.

`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.

Key Methods

- `http.createServer(callback)` - Creates HTTP server
- `http.request(options, callback)` - Makes HTTP request
- `http.get(url, callback)` - Convenience method for GET requests

The HTTP module provides server and client functionality through its main classes: Server, ServerResponse, IncomingMessage, and ClientRequest.

Node.js Streams

What are streams

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

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

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

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

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

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

Why streams

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

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

An example of a stream

A typical example is reading files from a disk.

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

```
import http from 'node:http'
import fs from 'node:fs/promises'
import { dirname } from 'node:path'
import { fileURLToPath } from 'node:url'
```

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

const server = http.createServer(async function (req, res) {
  try {
    const data = await fs.readFile(`${__dirname}/data.txt`)
    res.end(data)
  } catch (err) {
    res.statusCode = 500
    res.end('Error reading file')
  }
})
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 { dirname } from 'node:path'
import { fileURLToPath } from 'node:url'

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

const server = http.createServer((req, res) => {
  const stream = fs.createReadStream(`${__dirname}/data.txt`)
  stream.pipe(res)
})
server.listen(3000)
```

Instead of waiting until the file is fully read, we start streaming it to the HTTP client as soon as we have a chunk of data ready to be sent.

pipe()

The above example uses the line `stream.pipe(res)`: the `pipe()` method is called on the file stream.

What does this code do? It takes the source, and pipes it into a destination.

You call it on the source stream, so in this case, the file stream is piped to the HTTP response.

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

```
src.pipe(dest1).pipe(dest2)
```

This construct is the same as doing

```
src.pipe(dest1)
```

```
dest1.pipe(dest2)
```

Streams-powered Node.js APIs

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

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

Different types of streams

There are four classes of streams:

- `Readable`: a stream you can pipe from, but not pipe into (you can receive data, but not send data to it). When you push data into a readable stream, it is buffered, until a consumer starts to read the data.
- `Writable`: a stream you can pipe into, but not pipe from (you can send data, but not receive from it)
- `Duplex`: a stream you can both pipe into and pipe from, basically a combination of a Readable and Writable stream
- `Transform`: a Transform stream is similar to a Duplex, but the output is a transform of its input

Environment Configuration

Use `NODE_ENV` to differentiate between development and production:

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

Check environment in code:

```
if (process.env.NODE_ENV === 'production') {
  // production configuration
} else {
  // development configuration
}
```

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

Modern Async Pattern with async/await

Instead of promise chains, use async/await for cleaner, more readable asynchronous code:

```
// Modern approach with async/await
async function performOperations() {
  try {
    const result1 = await doSomething1()
    const result2 = await doSomething2(result1)
    const result3 = await doSomething3(result2)
    return result3
  } catch (error) {
    console.error('Operation failed:', error.message)
    throw error
  }
}

// Handle specific operations with individual error handling
async function performOperationsWithSpecificHandling() {
  try {
    const result1 = await doSomething1()

    let result2
    try {
      result2 = await doSomething2(result1)
    } catch (error) {
      console.error('Step 2 failed, using fallback')
      result2 = getDefaultValue()
    }

    const result3 = await doSomething3(result2)
    return result3
  } catch (error) {
    console.error('Critical operation failed:', error.message)
    throw error
  }
}
```

Error handling with async/await

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

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

Build an HTTP Server

Here is a sample Hello World HTTP web server:

```
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 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 18):

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

This uses the native Fetch API, no external libraries required.

A GET request is possible just using the Node.js standard modules, although it's more verbose than the option above:

```
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()
```

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',
    }),
  })
  console.log(`statusCode: ${response.status}`)
  const data = await response.json()
  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()
```

PUT and DELETE

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

Get HTTP request body data

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

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

For example, to get the body of this request:

```
import axios from 'axios'

axios.post('https://whatever.com/todos', {
  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 is available in all current Node.js versions. It simplifies the example above and makes it look more linear:

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

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

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

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

The Node.js Event Loop

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

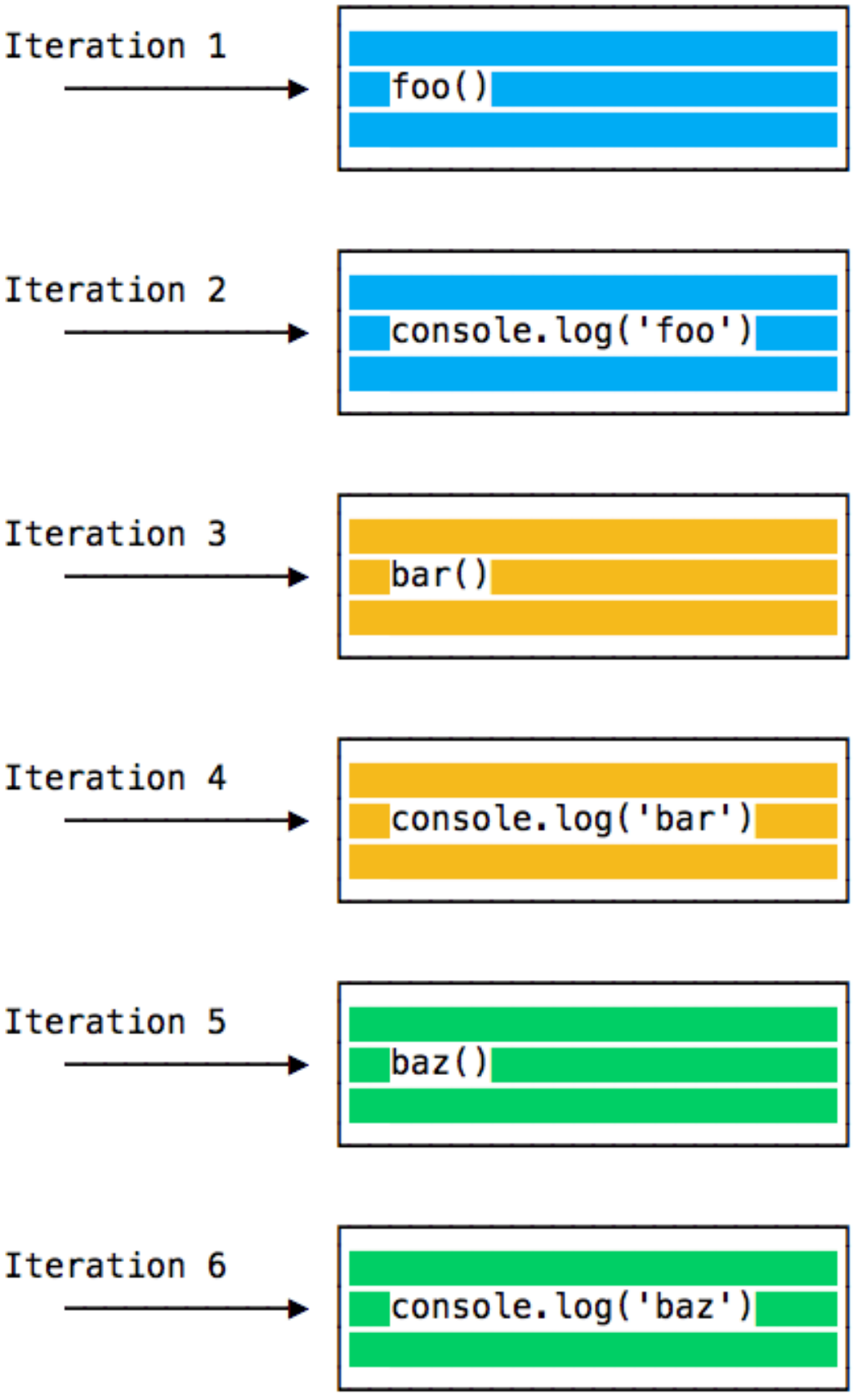
```
foo
bar
baz
```

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:


Iteration 1
foo()
Iteration 2
console.log('foo')
Iteration 3
bar()
Iteration 4
console.log('bar')
Iteration 5
baz()
Iteration 6
console.log('baz')

until the call stack is empty.

Queuing function execution

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

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

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

Take this example:

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

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

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

foo()
```

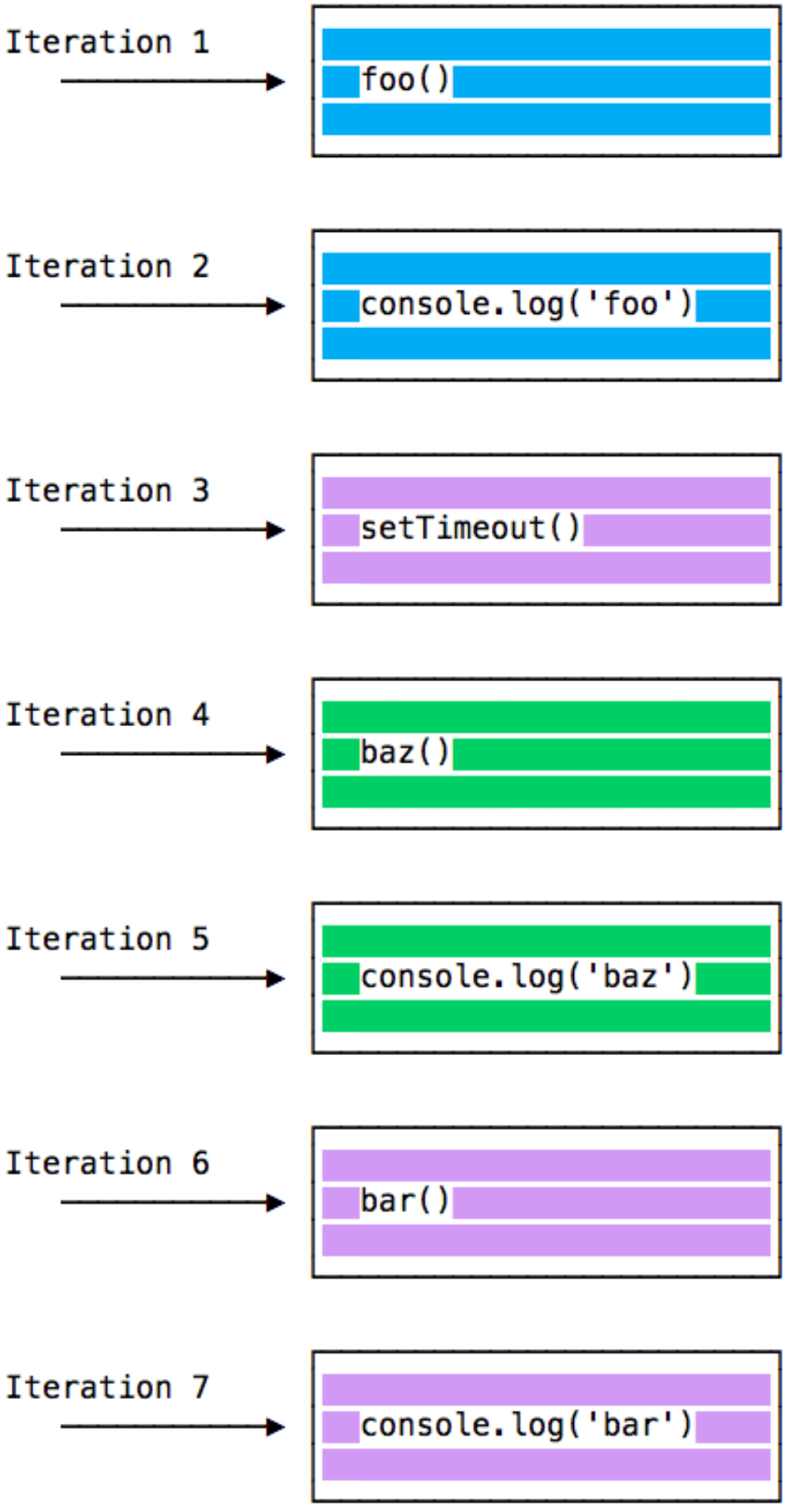
This code prints, maybe surprisingly:

```
foo
baz
bar
```

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

At this point the call stack looks like this:

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


Iteration 1
foo()
Iteration 2
console.log('foo')
Iteration 3
setTimeout()
Iteration 4
baz()
Iteration 5
console.log('baz')
Iteration 6
bar()
Iteration 7
console.log('bar')

Why is this happening?

The Message Queue

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

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

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

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

ES6 Job Queue

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

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

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

Example:

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

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

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

foo()
```

This prints

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

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

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


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

Understanding process.nextTick()

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

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

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

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

The event loop is busy processing the current function code.

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

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

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

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

Understanding setImmediate()

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

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

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

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

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

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

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

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

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

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

// start foo bar zoo baz
```

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

Discover JavaScript Timers

`setTimeout()`

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

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

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

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

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

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

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

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

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

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

Zero delay

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

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

console.log(' before ')
```

This code will print

```
before
after
```

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

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

`setInterval()`

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

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

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

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

```
clearInterval(id)
```

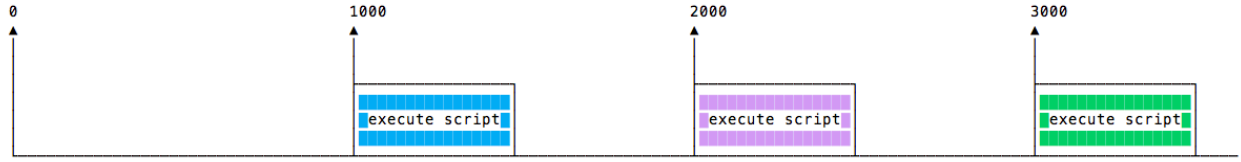
It's common to call `clearInterval` inside the setInterval callback function, to let it auto-determine if it should run again or stop. For example this code runs something unless App.somethingIWait has the value `arrived`:

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

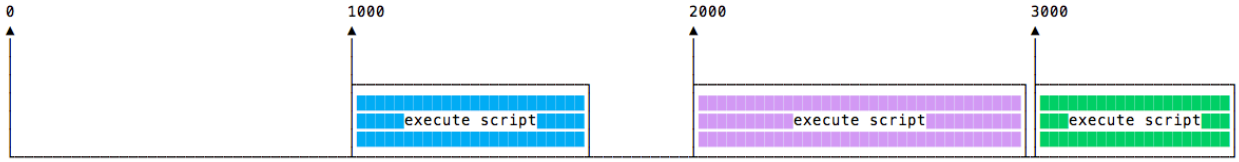
Recursive setTimeout

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

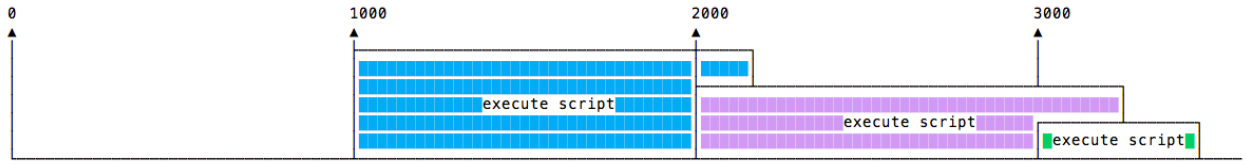
If a function always takes the same amount of time, it's all fine:



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



And maybe one long execution overlaps the next one:



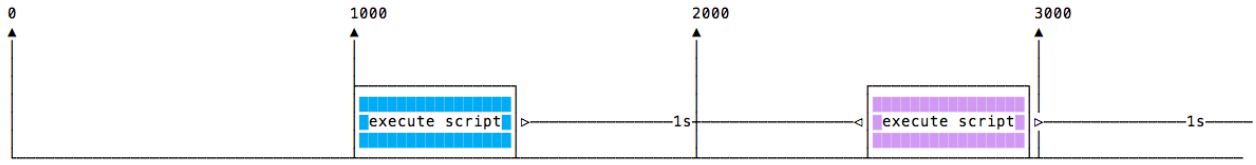
To avoid this, you can schedule a recursive setTimeout to be called when the callback function finishes:

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

  setTimeout(myFunction, 1000)
}

setTimeout(myFunction, 1000)
```

to achieve this scenario:



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

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

Modern Node.js Development Features

Watch Mode

Node.js includes a built-in watch mode (stable since Node.js 18) that automatically restarts your application when files change:

```
# Run with watch mode
node --watch app.js

# Watch specific files
node --watch --watch-path=./src app.js
```

This eliminates the need for external tools like `nodemon` for development.

Built-in Task Runner

Node.js 22+ includes a built-in task runner that can execute package.json scripts:

```
# Instead of npm run start
node --run start

# Instead of npm run test
node --run test
```

This provides faster script execution without requiring npm.

Testing with Node.js Built-in Test Runner

Node.js includes a built-in test runner (stable since Node.js 20) that eliminates the need for external testing frameworks like Jest or Mocha for basic testing needs.

Basic Test Structure

Create test files with `.test.js` or `.spec.js` extensions:

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

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

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

Running Tests

Run tests using the built-in test runner:

```
# Run all test files
node --test

# Run specific test files
node --test test/math.test.js

# Run with watch mode
node --test --watch
```

Test Organization

The test runner supports:

- `describe()` blocks for grouping tests
- `test()` or `it()` for individual test cases
- `before()`, `after()`, `beforeEach()`, `afterEach()` hooks
- Built-in assertion library with `node:assert`
- Async/await support throughout

This eliminates the need for external testing libraries for most Node.js applications.

```
// Example: Using environment variables safely

// Get the current environment (development or production)
const environment = process.env.NODE_ENV

// Set a debug flag based on the environment
const isDebug = environment !== 'production'

// Log only non-sensitive info
console.log('Running in environment:', environment)
console.log('Debug mode:', isDebug)
```