Node.js Course

Flavio Copes

Updated August 3, 2026

Contents

Learn to run JavaScript outside the browser, organize modules and packages, work with files, handle asynchronous events, and build HTTP servers.

What you'll learn: Build a small Node.js command-line program and an HTTP server using modules, files, asynchronous code, and environment configuration.

The Node.js runtime

Install Node, run scripts, use the REPL, and work with process.

The Node.js Guide

Node.js runs JavaScript on the server. A curated guide to the best Node.js tutorials on this site: core modules, files, HTTP, streams, and npm.

Node.js is a runtime that lets you run JavaScript outside the browser. Built on the V8 engine, it powers servers, command line tools, and build systems everywhere.

If you know JavaScript, you already know most of what you need. What changes is the environment: no DOM, but full access to files, the network, and the operating system.

I wrote a lot of Node.js tutorials on this site. This page is your map. Follow the path below if you're starting out, or jump to the topic you need.

Where to start

Follow this path in order:

1. How to install Node.js: get Node running on your machine
2. Differences between Node and the browser: same language, different world
3. How much JavaScript do you need to know to use Node?: check your prerequisites
4. How to use the Node.js REPL: experiment with Node interactively
5. Output to the command line using Node: your first tool, `console.log()`

6. Node, accept arguments from the command line: make your scripts configurable
7. Build an HTTP server: the classic first Node program

Core concepts

These are the ideas that make Node what it is. The event loop is the big one: understand it and everything else follows.

- The Node.js event loop
- Understanding process.nextTick()
- Understanding setImmediate()
- The Node event emitter
- Node buffers
- Node.js streams
- Error handling in Node.js

Working with files

Reading and writing files is the bread and butter of Node scripts.

- Reading files with Node
- Writing files with Node
- The Node fs module
- How to use the fs module with async/await
- Working with folders in Node
- Node file paths

HTTP and networking

Node was born to build network programs. These posts cover both serving and making requests.

- Making HTTP requests with Node
- Make an HTTP POST request using Node
- Get HTTP request body data using Node
- The Node http module
- HTTP requests in Node using Axios
- Using WebSockets with Node.js

npm and the ecosystem

Every Node project lives on npm. Learn the package manager and the files that drive it.

- An introduction to the npm package manager
- The package.json guide
- The package-lock.json file
- npm global or local packages
- npm dependencies and devDependencies
- Semantic versioning using npm
- The npx Node package runner

Go deeper

I collected all of this material in the free Node.js Handbook. It's the organized, start-to-finish version of everything above.

Every post on the topic is listed under the node tag.

How to install Node.js

Learn how to install Node.js with a version manager or the official installer, choose an LTS release, verify Node and npm, and switch project versions.

The best way to install Node.js for development is usually a version manager. It lets you install more than one Node release and switch between them for different projects.

If you only need one version, you can use an installer from the official Node.js download page.

For most production work, choose an active Long Term Support release rather than the Current release. The exact supported releases change over time, so check the Node.js release schedule.

Install Node.js with nvm

`nvm` is a popular version manager for macOS, Linux, and Windows Subsystem for Linux. It does not support native Windows shells.

Use the install command published in the official nvm repository. The command includes the current nvm release, so copy it from the repository instead of relying on an old version in a tutorial.

After installing nvm, open a new terminal and install the latest LTS release:

```
nvm install --lts
nvm use --lts
```

The first version installed becomes the default in new shells. You can also set it explicitly:

```
nvm alias default 'lts/*'
```

List the Node versions installed on your machine:

```
nvm ls
```

List releases available to install:

```
nvm ls-remote --lts
```

Pin the Node version for a project

Create an `.nvmrc` file in the project root:

```
lts/*
```

Then run:

```
nvm install
nvm use
```

A team can put a specific supported release line in `.nvmrc` when it needs reproducible local and CI environments. Update that file as part of normal dependency maintenance.

Use the official installer

The Node.js download page provides installers and prebuilt binaries.

This is simple, but switching between Node releases later is less convenient than with a version manager. Avoid installing Node from an unofficial download site.

Verify the installation

Node includes npm. Check both commands:

```
node --version
npm --version
```

You can now run a JavaScript file:

```
node app.js
```

If your shell says `node: command not found`, restart the terminal and check the setup instructions printed by your installer or version manager. Do not use `sudo` to work around npm permission errors; fix the Node installation or use a per-user version manager instead.

Differences between Node and the Browser

Understand the key differences between writing JavaScript for Node.js and the browser, from the missing DOM and filesystem access to CommonJS versus ES modules.

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

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

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

As a frontend developer who extensively uses Javascript, Node apps brings with it, a huge advantage - the comfort of programming everything, the frontend and the backend, in a single language.

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

What changes is the ecosystem.

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

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

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

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

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

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

Another difference is that Node uses the CommonJS module system, while in the browser we are starting to see the ES Modules standard being implemented.

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

How to use the Node.js REPL

Learn how to use the Node.js REPL, the Read-Evaluate-Print-Loop, to run JavaScript interactively with tab autocomplete, the variable, and dot commands.

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

```
node script.js
```

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

```
node
```

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

```
  node
>
```

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

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

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

Start simple and enter

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

The first value, `test`, is the output we told the console to print, then we get undefined which is the return value of running `console.log()`.

We can now enter a new line of JavaScript.

Use the tab to autocomplete

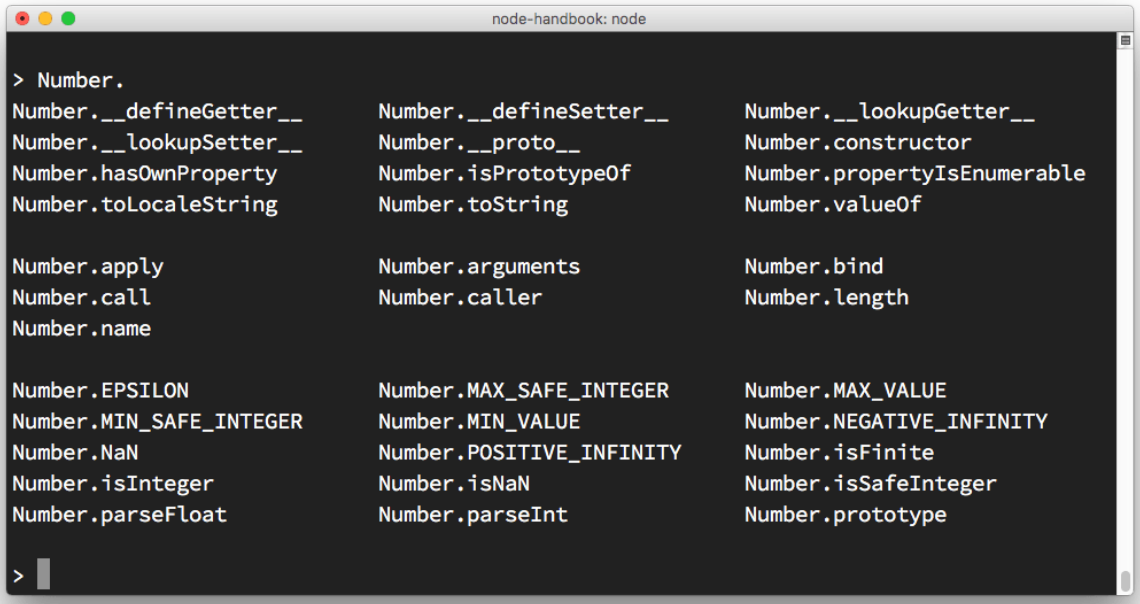
The cool thing about the REPL is that it's interactive.

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

Exploring JavaScript objects

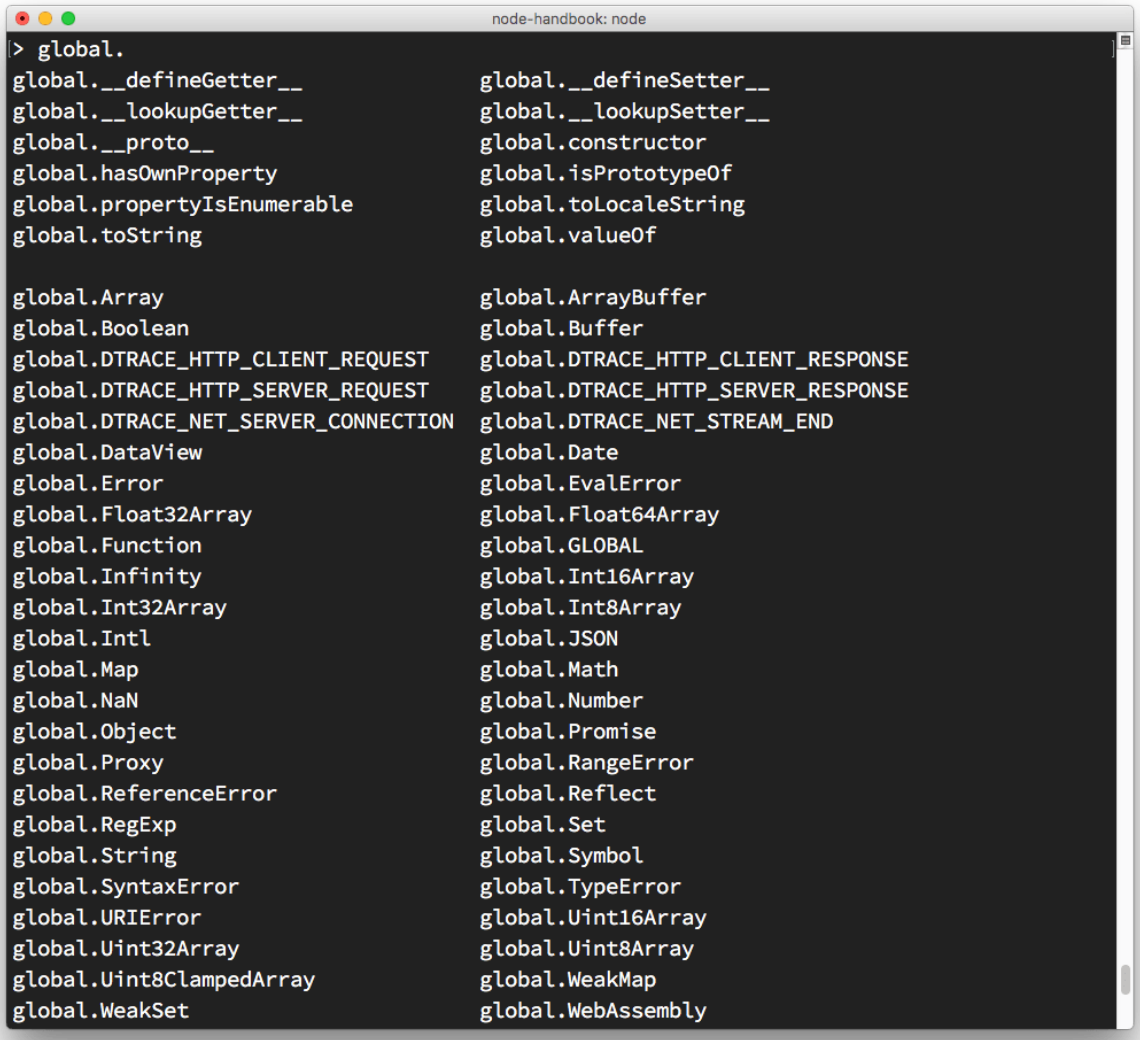
Try entering the name of a JavaScript class, like `Number`, add a dot and press `tab`.

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



Explore global objects

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



The _ special variable

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

Dot commands

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

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

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

For example if you start typing an iteration like this:

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

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

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

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

Run a Node.js script

Create a JavaScript file, execute it with Node, read its output, and interpret a non-zero exit status.

Create `hello.js`:

```
const name = process.argv[2]

if (!name) {
  console.error('Usage: node hello.js <name>')
  process.exitCode = 1
} else {
  console.log(`Hello, ${name}`)
}
```

Run it from a terminal:

```
node hello.js Ada
```

The shell starts a new Node.js **process**. Node reads the entry file, decides which module system it uses, evaluates its top-level code, and keeps the process alive while referenced timers, servers, sockets, or other work remain. This script has no pending work, so it exits after printing.

`process.argv` contains the Node executable, the entry-script path, and then the arguments supplied by the user. Here the first user argument is at index `2`.

Paths passed by your code are often resolved from the terminal's current working directory, which can differ from the directory containing the script. Print both when diagnosing a path mistake:

```
console.log({ cwd: process.cwd(), script: process.argv[1] })
```

Successful programs conventionally exit with status `0`; a non-zero status tells a shell, CI job, or another process that the command failed. `console.error()` writes the explanation to standard error. Setting `process.exitCode = 1` lets pending output and normal cleanup finish before Node exits.

Avoid calling `process.exit()` for ordinary validation failures because it can terminate before asynchronous output is flushed. Syntax errors, missing modules, and uncaught exceptions already make Node exit non-zero.

Your action is to run the script once with a name and once without one. Inspect both output streams and confirm that the second command returns a non-zero status in your shell.

Node, accept arguments from the command line

Learn how to accept command line arguments in a Node.js program using process.argv, the array that holds every value passed after the node command itself.

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

```
node app.js
```

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

For example:

```
node app.js flavio
```

or

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

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

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

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

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

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

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

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

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

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

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

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

```
node app.js flavio
```

you can access it using

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

In this case:

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

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

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

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

Output to the command line using Node

Learn how to output to the command line in Node.js with the console module, from console.log and format specifiers to count(), trace(), time(), and colors.

Basic output using the console module

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

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

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

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

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

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

and Node will print both.

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

For example:

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

- `%s` format a variable as a string
- `%d` or `%i` format a variable as an integer
- `%f` format a variable as a floating point number
- `%O` used to print an object representation

Example:

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

Clear the console

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

Counting elements

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

Take this code:

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

```
)
console.count(
  'The value of y is ' + y + ' and has been checked .. how many times?'
)
```

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

You can just count apples and oranges:

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

Print the stack trace

There might be cases where it's useful to print the call stack trace of a function, maybe to answer the question *how did you reach that part of the code?*

You can do so using `console.trace()`:

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

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

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

Calculate the time spent

You can easily calculate how much time a function takes to run, using `time()` and `timeEnd()`

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

```
}
measureDoingSomething()
```

stdout and stderr

As we saw console.log is great for printing messages in the Console. This is what's called the standard output, or `stdout`.

`console.error` prints to the `stderr` stream.

It will not appear in the console, but it will appear in the error log.

Color the output

You can color the output of your text in the console by using escape sequences. An escape sequence is a set of characters that identifies a color.

Example:

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

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

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

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

```
const chalk = require('chalk')
console.log(chalk.yellow('hi!'))
```

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

Check the project link I posted above for more usage examples.

Create a progress bar

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

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

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

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

Accept input from the command line in Node

Learn how to accept input from the command line in Node.js and make CLI programs interactive using the built-in readline module and the Inquirer.js package.

How to make a Node.js CLI program interactive?

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

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

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

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

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

In this callback function, we close the readline interface.

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

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

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

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

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

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

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

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

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

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

Check your understanding: the Node.js runtime

Check what Node provides, how it differs from a browser, running files, the REPL, arguments, V8, concurrency, and process status.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Modules and npm

Organize code and manage project packages and versions.

The Node Core Modules

A reference to the Node.js core modules that ship with the platform, like fs, http, crypto, path, events, and stream, and what each one lets you do.

Node.js has a set of core modules that are part of the platform and come with the Node.js installation.

We have a lot of them:

Name	Description
assert	provides a set of assertion functions useful for testing
buffer	provides the ability to handle buffers containing binary data
child_process	provides the ability to spawn child processes
console	provides a simple debugging console
cluster	allows to split a Node.js process into multiple workers to take advantage of multi-core systems
crypto	provides cryptographic functionality
dgram	provides an implementation of UDP Datagram sockets
dns	provides name resolution and DNS lookups
events	provides an API for managing events
fs	provides an API for interacting with the file system
http	provides an HTTP client/server implementation
http2	provides an HTTP/2 client/server implementation
https	provides an HTTPS client/server implementation
net	provides an asynchronous network API
os	provides operating system-related utility methods and properties
path	provides utilities for working with file and directory paths
perf_hooks	to enable the collection of performance metrics
process	provides information about, and control over, the current Node.js process
querystring	provides utilities for parsing and formatting URL query strings
readline	provides an interface for reading data from a Readable stream
repl	provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalo
stream	an abstract interface for working with streaming data
string_decoder	provides an API for decoding Buffer objects into strings

Name	Description
timers	provide functions to schedule functions to be called at some future period of time
tls	provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (
tty	provides functionality used to perform I/O operations in a text terminal
url	provides utilities for URL resolution and parsing
util	supports the needs of Node.js internal APIs, useful for application and module developers as v
v8	exposes APIs that are specific to the version of V8 built into the Node.js binary
vm	enables compiling and running code within V8 Virtual Machine contexts
wasi	provides an implementation of the WebAssembly System Interface specification
worker	enables the use of threads that execute JavaScript in parallel
zlib	provides compression functionality

Check out my detailed tutorials on

- The Node.js `events` module
- The Node.js `fs` module
- The Node.js `http` module
- The Node.js `os` module
- The Node.js `path` module

CommonJS modules

Use require and module.exports in projects that use Node's original module system.

CommonJS loads a module with `require()` and exposes values through `module.exports`.

```
// math.cjs
function add(a, b) {
  return a + b
}

module.exports = { add }

// app.cjs
const { add } = require("./math.cjs")
console.log(add(2, 3))
```

Node wraps each CommonJS file in a function. That gives the module its own top-level scope and the local values `require`, `module`, `exports`, `__filename`, and `__dirname`. Variables declared in `math.cjs` do not become globals.

Relative requests such as `./math.cjs` are resolved from the requiring file, not from the terminal's current directory. Package requests such as `require('fastify')` are resolved through `node_modules`. The `node:` prefix makes a built-in explicit:

```
const { readFile } = require('node:fs')
```

The value returned by `require()` is `module.exports`. The `exports` variable starts as a shortcut to that same object, so this works:

```
exports.add = add
```

Reassigning the shortcut does not work:

```
exports = { add } // require() still receives the old module.exports
```

Assign a replacement value to `module.exports` instead.

CommonJS modules are cached by resolved filename after the first load. Requiring the same file again normally returns the same exported object without rerunning its top-level code. This is useful for shared state but makes import-time side effects harder to test and reason about. Export a function when work should happen for every call.

Use `.cjs` for an explicit CommonJS file. A `.js` file is also CommonJS when the nearest `package.json` declares `"type": "commonjs"`; being explicit avoids tools guessing.

Your action is to add a top-level `console.log()` to `math.cjs`, require it twice, and explain why the message appears once. Then reproduce and fix the `exports = { add }` mistake.

Expose functionality from a Node file using exports

Learn how to expose functionality from a Node.js file using module.exports and exports, and understand the difference between the two approaches.

Node has a built-in module system.

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

When you want to import something you use

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

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

In this file, functionality must be exposed before it can be imported by other files.

Any other object or variable defined in the file by default is private and not exposed to the outer world.

This is what the `module.exports` API offered by the `module` system allows us to do.

When you assign an object or a function as a new `exports` property, that is the thing that's being exposed, and as such, it can be imported in other parts of your app, or in other apps as well.

You can do so in 2 ways.

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

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

module.exports = car

//..in the other file

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

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

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

exports.car = car
```

or directly

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

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

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

or

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

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

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

ES modules in Node.js

Use import and export with explicit module configuration and Node built-in module specifiers.

Set `"type": "module"` in `package.json`, or use the `.mjs` extension.

```
import { readFile } from "node:fs/promises"

export async function loadConfig() {
  const text = await readFile(
    new URL('./config.json', import.meta.url),
    'utf8'
  )

  return JSON.parse(text)
}
```

ES modules use the JavaScript-standard `import` and `export` syntax. Static imports are resolved and linked before the importing module evaluates, which lets tools and Node understand the dependency graph without executing arbitrary `require()` calls first.

Use an explicit file extension for relative imports in Node:

```
import { loadConfig } from './config.js'
```

The module type applies to `.js` files in the package scope. `.mjs` is always ES module syntax, and `.cjs` is always CommonJS. Avoid switching a package's `type` without checking every `.js` entry point and tool configuration.

ES modules do not provide CommonJS globals such as `require`, `module`, or `__dirname`. Current

supported Node releases provide `import.meta.filename` and `import.meta.dirname` for file-based modules. A `URL` relative to `import.meta.url`, as in the example, is also a portable way to locate a resource beside the module.

The `node:` prefix clearly identifies a built-in module. Package imports such as `import fastify from 'fastify'` are resolved through package metadata and `node_modules`.

Dynamic `import()` returns a promise and is useful for conditional or late loading. Top-level `await` is allowed in ES modules, but it delays evaluation of modules that depend on that graph, so keep startup dependencies visible.

Node supports interoperability between CommonJS and ES modules, but default and named export behavior can surprise consumers. Prefer one module system inside a small project and test the actual public entry points when crossing the boundary.

Your action is to create two ES modules, import one with its `.js` extension, and load a text file beside the module. Run the entry point from a different working directory to prove the resource path does not depend on `process.cwd()`.

The package.json guide

Learn how package.json describes a Node.js project, including scripts, dependencies, module type, package entry points, engines, and semver version ranges.

The `package.json` file is the manifest for a Node.js project. It records project metadata, dependencies, scripts, module settings, and package entry points.

npm reads it when you install packages or run scripts. Node.js also reads fields such as `type`, `main`, `exports`, and `imports`.

The complete field reference lives in the official npm package.json documentation.

Create a package.json file

Run this command in a new project:

```
npm init
```

npm asks a few questions and writes the file.

Use `-y` to accept the default values:

```
npm init -y
```

Applications can use a small `package.json`:

```
{
  "name": "notes-app",
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "node --watch src/server.js",
    "start": "node src/server.js",
    "test": "node --test"
  },
  "devDependencies": {
    "eslint": "^10.6.0"
```

```
  }
}
```

A package published to npm needs a `name` and `version`. A private application does not need to pretend it is a public package.

The npm init guide explains the interactive and default setup modes.

package.json must be valid JSON

`package.json` uses JSON, not JavaScript.

Property names and string values need double quotes. Comments and trailing commas are not valid:

```
{
  "name": "notes-app",
  "private": true
}
```

name and version

The `name` identifies a package:

```
{
  "name": "string-tools"
}
```

Scoped package names start with an organization or username:

```
{
  "name": "@flaviocopes/string-tools"
}
```

The `version` identifies a release:

```
{
  "version": "2.1.0"
}
```

Published packages need both fields. Together they identify one exact package release.

description, keywords, and license

These fields help people understand a published package:

```
{
  "description": "Small string utilities for Node.js",
  "keywords": ["strings", "utilities"],
  "license": "MIT"
}
```

Use a valid SPDX license identifier such as `MIT`, `ISC`, or `Apache-2.0`.

author and contributors

The author can be a string:

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

Or an object:

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

Use `contributors` for additional people:

```
{
  "contributors": [
    {
      "name": "Roger"
    },
    {
      "name": "Syd"
    }
  ]
}
```

homepage, bugs, and repository

These fields connect a published package to its documentation and source code:

```
{
  "homepage": "https://github.com/flaviocopes/flaviocopes.com#readme",
  "bugs": {
    "url": "https://github.com/flaviocopes/flaviocopes.com/issues"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/flaviocopes/flaviocopes.com.git"
  }
}
```

If a package lives inside a monorepo, `repository.directory` can point to its subfolder.

private

Set `private` to `true` for an application or package that must not be published:

```
{
  "private": true
}
```

npm refuses to publish a package marked private. This protects against accidental publication.

scripts

Scripts give your project memorable commands:

```
{
  "scripts": {
    "dev": "node --watch src/server.js",
    "start": "node src/server.js",
    "test": "node --test",
    "lint": "eslint ."
  }
}
```

Run a custom script with:

```
npm run dev
```

`start` and `test` have shorter forms:

```
npm start
npm test
```

You can pass arguments to the underlying command after `--`:

```
npm test -- --test-name-pattern="user login"
```

The npm scripts documentation covers lifecycle scripts and the environment available to commands.

dependencies

`dependencies` lists packages required when the application runs.

This command adds a dependency:

```
npm install fastify
```

npm saves regular installs to `dependencies` by default and records the selected version range.

devDependencies

`devDependencies` contains tools needed to develop, test, or build the project:

```
{
  "devDependencies": {
    "eslint": "^10.6.0",
    "prettier": "3.9.5"
  }
}
```

Add one with:

```
npm install --save-dev eslint
```

A normal `npm install` installs both dependency groups. Production installs can omit development dependencies with `npm install --omit=dev`.

The official npm dependency guide explains when to use each field.

type

The `type` field tells Node.js how to interpret `.js` files:

```
{
  "type": "module"
}
```

With `"type": "module"`, `.js` files use ES Modules. With `"type": "commonjs"`, they use CommonJS.

The explicit `.mjs` and `.cjs` extensions always select ES Modules and CommonJS respectively.

Node recommends being explicit about `type`, especially for published packages. See the Node.js module system rules.

main and exports

`main` is the traditional package entry point:

```
{
  "main": "./dist/index.js"
}
```

`exports` is the modern way to define a package's public entry points:

```
{
  "exports": {
    ".": "./dist/index.js",
    "./format": "./dist/format.js"
  }
}
```

When `exports` is present, package users cannot import unlisted internal paths. This lets package authors define a clear public API.

Conditional exports can provide different files to `import` and `require`, but test that setup carefully. The Node.js package entry points guide explains the available patterns.

engines

The `engines` field documents supported runtime versions:

```
{
  "engines": {
    "node": ">=22"
  }
}
```

npm usually treats incompatible engines as a warning unless `engine-strict` is enabled. Do not rely on this field as the only runtime check.

files

For a published package, `files` controls what npm includes:

```
{
```

```
  "files": [
    "dist",
    "README.md"
  ]
}
```

Some files, including `package.json`, `README`, and the license, are always included. Test the final package before publishing:

```
npm pack --dry-run
```

workspaces

Workspaces let one root project manage several related packages:

```
{
  "private": true,
  "workspaces": [
    "packages/*"
  ]
}
```

npm links workspace packages during installation and can run commands in one or all workspaces. See the official npm workspaces guide.

Package version ranges

Dependencies use semantic version ranges:

```
{
  "dependencies": {
    "first-package": "2.3.1",
    "second-package": "~2.3.1",
    "third-package": "^2.3.1"
  }
}
```

The common rules are:

- `2.3.1` accepts only version `2.3.1`
- `~2.3.1` accepts patch updates below `2.4.0`
- `^2.3.1` accepts compatible updates below `3.0.0`
- `>=2.3.1 <3` defines an explicit range

Caret ranges behave differently before version `1.0.0`. For example, `^0.13.0` stays below `0.14.0`.

Use the npm semantic versioning guide when a range is not obvious.

`package-lock.json` records the exact resolved dependency tree. Commit it for applications so other developers and CI can reproduce the same install.

An introduction to the npm package manager

An introduction to npm, the standard package manager for Node.js: how to install dependencies, update packages, manage versions, and run scripts.

`npm` is the package manager that comes with Node.js. You use it to install packages, record project dependencies, update them, and run project scripts.

Most npm projects have a `package.json` file and a `package-lock.json` file. Installed packages usually live in the `node_modules` folder.

Install npm

The simplest way to get npm is to install Node.js. npm is included with the official Node.js installers.

You can check the installed versions with:

```
node --version
npm --version
```

Install all project dependencies

If a project has a `package.json` file, run:

```
npm install
```

npm reads the dependency lists, installs the packages, and updates `package-lock.json` when needed. The lockfile records the exact dependency tree so repeated installs are more predictable.

In automated environments, use `npm ci` when the project has a lockfile. It performs a clean install and fails if `package.json` and `package-lock.json` disagree.

The npm install documentation explains the available install modes.

Install a package

Install a package your application needs with:

```
npm install fastify
```

npm adds it to `dependencies` by default.

Development tools belong in `devDependencies`:

```
npm install --save-dev eslint
```

You can also use the shorter `-D` flag:

```
npm install -D eslint
```

Use `--save-exact` if you want npm to record one exact version instead of a version range.

Update packages

Run this to update installed packages within the version ranges allowed by `package.json`:

```
npm update
```

You can update one package:

```
npm update fastify
```

`npm update` does not normally widen the ranges in `package.json`. The official `npm update` guide documents how the command treats package and dependency constraints.

Use this command to see which direct dependencies have newer versions available:

```
npm outdated
```

Package versions

npm packages normally follow **semantic versioning**:

```
major.minor.patch
```

For version `4.2.1`:

- `4` is the major version
- `2` is the minor version
- `1` is the patch version

A caret range such as `^4.2.1` accepts compatible updates without moving to version 5. A tilde range such as `~4.2.1` stays within version 4.2.

Version zero has stricter caret behavior. For example, `^0.13.0` accepts versions below `0.14.0`, not every `0.x` minor release.

See the npm semantic versioning guide for the complete range rules.

Run project scripts

The `scripts` field in `package.json` gives a project short, repeatable commands:

```
{
  "scripts": {
    "dev": "node --watch server.js",
    "start": "node server.js",
    "test": "node --test"
  }
}
```

Run a script with:

```
npm run dev
```

`start` and `test` also have shortcuts:

```
npm start
npm test
```

npm adds executables from local dependencies to the script's `PATH`. This means a script can call locally installed tools without a global installation.

The npm scripts documentation describes lifecycle scripts and the environment available to commands.

npm dependencies and devDependencies

Learn the difference between npm dependencies and devDependencies, how the -D flag adds dev-only packages, and how --production skips them on deploy.

When you install an npm package using `npm install <package-name>`, npm records it as a **dependency** by default:

```
npm install fastify
```

Use `-D`, short for `--save-dev`, for a development dependency:

```
npm install -D eslint
```

The distinction is about when the installed application needs the package:

- `dependencies` are required when production code runs
- `devDependencies` are needed only to develop, test, lint, or build the project

A package imported by the running server belongs in `dependencies`. A test runner normally belongs in `devDependencies`. Build tools require judgment: if the deployment builds from source, the build stage needs them; a runtime image containing already-built output may not.

Both sections contain version ranges. The exact resolved tree belongs in `package-lock.json`, including development packages even when they are omitted from one install.

For a production-only install, current npm uses:

```
npm ci --omit=dev
```

The older `--production` option is still recognized, but `--omit=dev` states the intent directly. Omitted development packages remain represented in the lockfile; they are not physically installed in that `node_modules` tree.

Do not classify a dependency by package reputation. A compiler can be runtime-critical in one architecture and development-only in another. Start from the exact production command and ask whether it imports or executes the package.

Test the production artifact in a clean environment. A local machine can hide a mistake because a dev dependency or globally installed tool happens to be available.

Your action is to classify your project's direct packages from its real start and build commands. Run a clean `npm ci --omit=dev`, start the production artifact, and move any missing runtime package to the correct section.

Semantic Versioning using npm

Learn Semantic Versioning in npm: how the major.minor.patch numbers work and how the ^ and ~ range symbols control which versions npm update will install.

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

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

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

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

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

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

Why is that so important?

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

The rules use those symbols:

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

Let's see those rules in detail:

- `^`: if you write `^0.13.0` when running `npm update` it can update to patch and minor releases: `0.13.1`, `0.14.0` and so on.
- `~`: if you write `~0.13.0`, when running `npm update` it can update to patch releases: `0.13.1` is ok, but `0.14.0` is not.
- `>`: you accept any version higher than the one you specify
- `>=`: you accept any version equal to or higher than the one you specify
- `<=`: you accept any version equal or lower to the one you specify
- `<`: you accept any version lower to the one you specify
- `=`: you accept that exact version
- `-`: you accept a range of versions. Example: `2.1.0 - 2.6.2`
- `||`: you combine sets. Example: `< 2.1 || > 2.6`

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

There are other rules, too:

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

I also built a free semver advisor that tells you which version number to bump and explains what the `^` and `~` ranges allow.

Lock dependency versions

Understand why package-lock.json records a complete dependency tree and when npm ci is useful.

`package.json` declares acceptable version ranges. `package-lock.json` records the exact resolved dependency tree.

For example, a direct dependency range such as `^4.18.0` can accept newer compatible releases. That dependency has its own dependencies and ranges. Without a lockfile, an install next month can resolve a different complete tree even though your `package.json` did not change.

The lockfile records direct and transitive versions plus resolution and integrity information. Commit it for an application unless the project has a documented policy not to. Review lockfile changes together with the dependency command that caused them.

Use `npm install` while intentionally changing dependencies. It resolves a valid tree, updates `node_modules`, and updates the lockfile when needed.

Use a clean install in CI and deployment:

```
npm ci
```

`npm ci` requires an existing lockfile, removes the existing `node_modules` directory, installs the locked tree, and never rewrites `package.json` or the lockfile. If the manifests disagree, it fails instead of silently repairing them. That failure protects the evidence you reviewed.

Flags that affect dependency-tree shape must be consistent. If the lockfile was created with a setting such as `legacy-peer-deps`, commit the matching project configuration rather than relying on one developer to remember the flag.

A lockfile improves reproducibility; it does not make dependencies secure or make every operating system identical. Native and optional packages can still have platform conditions, and known vulnerabilities or compromised releases remain real risks. Pin the Node and npm versions used by automation when exact tooling behavior matters.

Your action is to run `npm ci` in a disposable project, change one dependency range in `package.json` without updating the lockfile, and confirm that the next clean install fails. Then use the normal dependency workflow to restore agreement.

Check your understanding: modules and npm

Check CommonJS, ES modules, package metadata, dependency types, semantic version ranges, lockfiles, local installs, and package runners.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Files and paths

Read, write, inspect, and locate files without blocking work.

The Node fs module

A guide to the Node.js fs module and its methods for working with the file system, from readFile and writeFile to rename, plus their blocking Sync versions.

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

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

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

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

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

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

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

For example:

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

This makes a huge difference in your application flow.

> Node 10 includes experimental support for a promise based API

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

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

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

  //done
```

```
})
```

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

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

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

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

Reading files with Node

Learn how to read files in Node.js with the fs module, using the asynchronous readFile() and synchronous readFileSync(), and why streams suit big files.

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

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

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

Run this and the output might surprise you. Instead of the file text you get the raw bytes:

```
<Buffer 48 65 6c 6c 6f>
```

That's because there is no default string encoding: if you don't specify one, Node hands you a **Buffer** object. Pass the encoding as the second parameter, before the callback, to get a string:

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

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

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

try {
  const data = fs.readFileSync('/Users/flavio/test.txt', 'utf8')
```

```
  console.log(data)
} catch (err) {
  console.error(err)
}
```

The trade-off is right in the name. `readFileSync()` blocks the entire process until the file is read. In a small command-line script that's perfectly fine, and the code is easier to follow. In a server, it freezes every other request while the disk works — use the asynchronous versions there.

There is also a promise-based API in `fs/promises`, which pairs nicely with `await`:

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

async function main() {
  const data = await fs.readFile('/Users/flavio/test.txt', 'utf8')
  console.log(data)
}

main()
```

When the file is missing

The most common failure is a wrong path. You get:

```
Error: ENOENT: no such file or directory, open '/Users/flavio/test.txt'
```

`ENOENT` means the file does not exist at that path. Check `err.code === 'ENOENT'` when a missing file is an expected case you want to handle gracefully, instead of treating every error the same way.

Big files

Both `fs.readFile()` and `fs.readFileSync()` read the full content of the file in memory before returning the data.

This means that big files are going to have a major impact on your memory consumption and speed of execution of the program. A 2 GB log file means 2 GB of RAM, just to look at it.

In this case, a better option is to read the file content using streams: `fs.createReadStream()` gives you the file in small chunks, so memory use stays flat no matter how large the file is.

How to use the Node.js fs module with async/await

Learn how to use the Node.js fs module with async/await through the promise-based node:fs/promises API, so you can await calls like fs.readdir() directly.

Node's filesystem module has a stable promise API in `node:fs/promises`:

```
import { readFile, readdir } from 'node:fs/promises'

const posts = await readdir('content')
const configText = await readFile('config.json', 'utf8')
const config = JSON.parse(configText)
```

The `node:` prefix makes it explicit that this is a Node built-in rather than a package from `node_modules`. `readFile()` returns a `Buffer` unless you request an encoding such as `'utf8'`.

An awaited filesystem failure rejects the promise. Catch errors where you can add useful context or make a specific recovery decision:

```
async function loadOptionalConfig(path) {
  try {
    return JSON.parse(await readFile(path, 'utf8'))
  } catch (error) {
    if (error.code === 'ENOENT') return {}
    throw new Error(`Cannot load ${path}`, { cause: error })
  }
}
```

Do not turn every error into an empty value. `ENOENT` means the path is missing; invalid JSON, permission errors, and I/O failures should remain visible.

Avoid checking with `access()` before reading. The file can change between the check and the read. Attempt the real operation and handle its result.

Promise-based filesystem work uses Node's worker pool, so it does not block the event-loop thread while waiting for ordinary file I/O. It is not automatically safe to launch overlapping writes to the same file. Those operations are not synchronized and can overwrite or corrupt the intended result.

Use `new URL('./config.json', import.meta.url)` when a resource belongs beside an ES module. Use a path relative to `process.cwd()` when the caller deliberately selects a file from the working directory.

Your action is to load one valid JSON file, one missing file, and one malformed file. Recover only from `ENOENT`, preserve the original error as `cause` for the others, and explain why a preflight existence check would still race.

Writing files with Node

Learn how to write files in Node.js with the fs module, using writeFile() and writeFileSync(), file flags, appendFile() to add content, and streams.

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

Example:

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

const content = 'Some content!'

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

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

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

```
const content = 'Some content!'

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

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

You can modify the default by specifying a flag:

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

The flags you'll likely use are

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

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

Append to a file

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

```
const content = 'Some content!'

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

Using streams

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

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

Working with folders in Node

Learn how to work with folders in Node.js using the fs module to check, create, read, rename, and remove directories, plus the fs-extra module.

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

Check if a folder exists

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

Create a new folder

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

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

const folderName = '/Users/flavio/test'

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

Read the content of a directory

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

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

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

const folderPath = '/Users/flavio'

fs.readdirSync(folderPath)
```

You can get the full path:

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

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

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

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

Rename a folder

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

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

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

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

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

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

Remove a folder

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

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

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

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

Install it using

```
npm install fs-extra
```

and use it like this:

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

const folder = '/Users/flavio'

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

It can also be used with promises:

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

or with async/await:

```
async function removeFolder(folder) {
```

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

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

Node File Paths

Learn how to work with file paths in Node.js using the path module: extract dirname, basename, and extname, and combine paths with join and resolve.

Every file in the system has a path.

On Linux and macOS, a path might look like:

`/users/flavio/file.txt`

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

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

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

You include this module in your files using

`const path = require('path')`

and you can start using its methods.

Getting information out of a path

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

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

Example:

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

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

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

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

Working with paths

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

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

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

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

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

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

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

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

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

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

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

Node file stats

Learn how to get the details of a file in Node.js using the fs module stat() method to check isFile(), isDirectory(), symbolic links, and file size.

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

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

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

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

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

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

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

A lot, including:

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

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

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

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

Choose a filesystem API style

Choose between synchronous, callback, and promise-based filesystem operations according to where the code runs.

Node exposes synchronous, callback, and promise-based filesystem APIs.

The three forms have different completion and error behavior:

```
import { readFileSync, readFile } from 'node:fs'
import { readFile as readFilePromise } from 'node:fs/promises'

const text = readFileSync('note.txt', 'utf8')

readFile('note.txt', 'utf8', (error, value) => {
  if (error) return console.error(error)
  console.log(value)
})

const value = await readFilePromise('note.txt', 'utf8')
```

The synchronous call blocks the event-loop thread until the filesystem operation finishes and throws an exception on failure. That can be acceptable for a tiny one-shot CLI or essential startup file before a server listens. It is dangerous inside a request handler because every client waits.

Callback and promise operations run asynchronously, normally using Node's worker pool for filesystem work. The callback form passes the error first. The promise form rejects and fits naturally with `await` and `try/catch`.

Use `node:fs/promises` for clarity in most application code. Callback APIs can use less allocation and may matter in a measured, extremely hot path. Do not choose them merely because they look "lower level."

Asynchronous does not mean synchronized. Two concurrent writes to the same file can race re-

gardless of whether callbacks or promises are used. Serialize related modifications or use storage designed for concurrent writers.

For very large files, use streams instead of reading the complete contents into memory.

Streams control memory by processing chunks and can respect backpressure between a fast reader and a slower destination. They do not automatically make CPU-heavy processing non-blocking.

Your action is to read the same missing file with all three API styles and capture how each reports the error. Then place the synchronous version inside a timer-driven demo and observe which scheduled callback is delayed.

Build a notes command-line program

Combine arguments, JSON parsing, paths, and promise-based filesystem calls in one small Node.js program.

Build a program with `add`, `list`, and `remove` commands.

Define the command contract before the storage code:

```
node notes.js add "Buy milk"
node notes.js list
node notes.js remove <id>
```

Parse only the arguments each command accepts. Missing text, an invalid ID, or an unknown command is a user error: print a short usage message to standard error and set a non-zero exit status.

Store notes in a JSON file beside the ES module, not wherever the user happened to launch it:

```
const dataUrl = new URL('./notes.json', import.meta.url)
```

Treat only a missing file as an empty collection:

```
async function readNotes() {
  try {
    return JSON.parse(await readFile(dataUrl, 'utf8'))
  } catch (error) {
    if (error.code === 'ENOENT') return []
    throw error
  }
}
```

Invalid JSON, permission errors, and I/O failures must remain visible. Silently replacing corrupt storage with `[]` could make the next write erase recoverable notes.

Write a complete new snapshot to a temporary file beside the destination, flush and close it as required by your durability needs, then rename it over `notes.json`. Renaming on the same filesystem prevents readers from observing a half-written JSON document. Clean up a leftover temporary file after a failed write.

Atomic replacement is not the same as concurrency control. Two CLI processes can both read the old list and then each rename a different update; the last rename wins. Either document that only one process may modify the file, add an appropriate lock, or use a database when concurrent writers are a requirement.

Keep the command flow explicit:

1. parse and validate the command
2. load and validate the stored array
3. calculate the new array without mutating partial state
4. persist it safely
5. print the result only after persistence succeeds

Use stable generated IDs rather than array indexes, because removing one note changes later indexes.

Return a non-zero exit status for an unknown command or invalid input.

Your action is to implement all three commands, then test a missing file, malformed JSON, invalid ID, failed write, and two simultaneous `add` commands. Document which concurrency guarantee your version actually provides.

Check your understanding: files and paths

Check asynchronous and synchronous file operations, encodings, overwrites, portable paths, module directories, existence checks, and folder listings.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Asynchronous code and events

The event loop, promises, emitters, streams, and failures.

The Node.js Event Loop

Understand the Node.js event loop and how its single thread, call stack, message queue, and ES6 job queue enable asynchronous, non-blocking I/O.

Introduction

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

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

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

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

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

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

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

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

Blocking the event loop

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

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

The call stack

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

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

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

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

A simple event loop explanation

Let's pick an example:

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

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

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

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

```
foo()
```

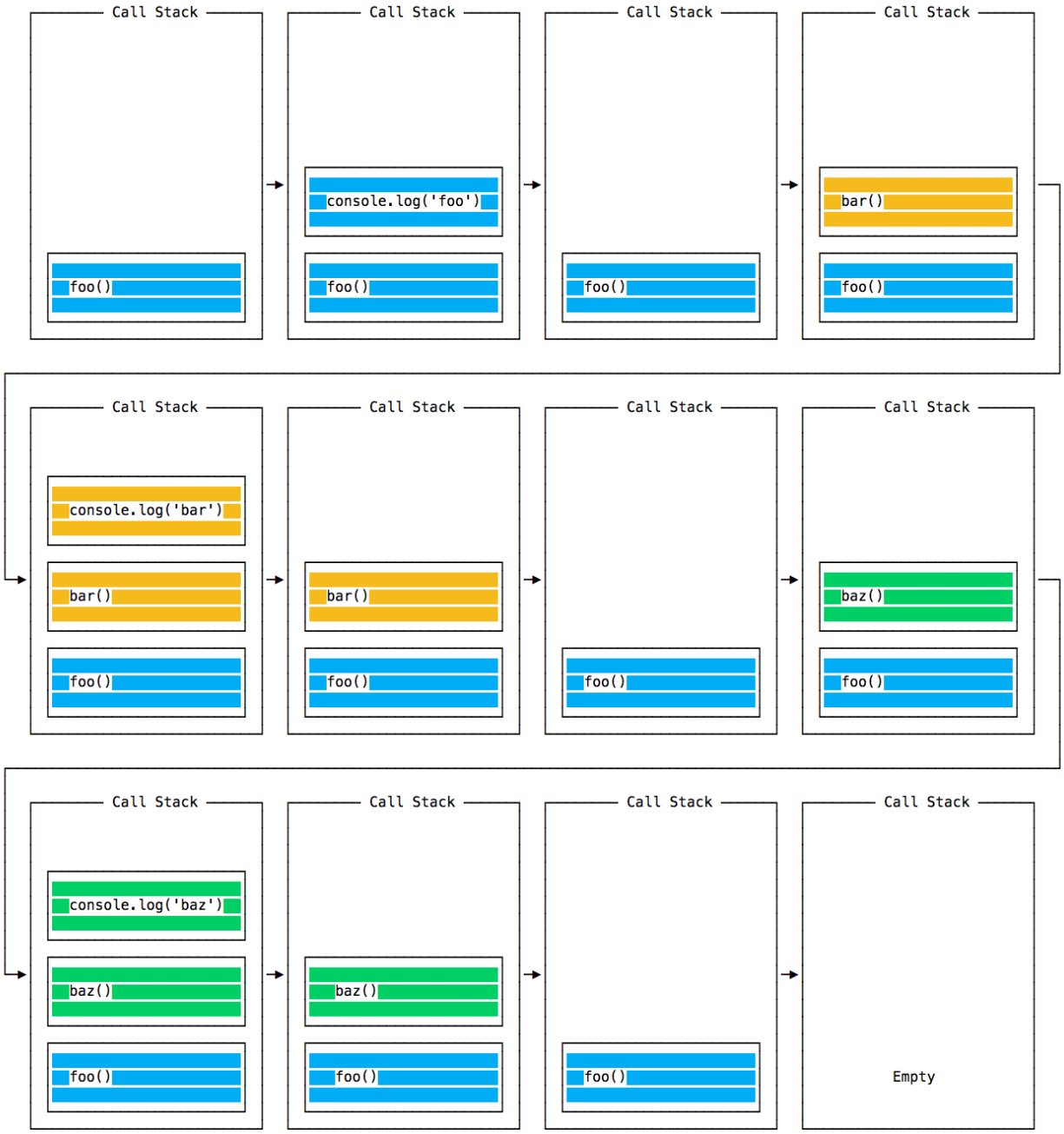
This code prints

```
foo
bar
baz
```

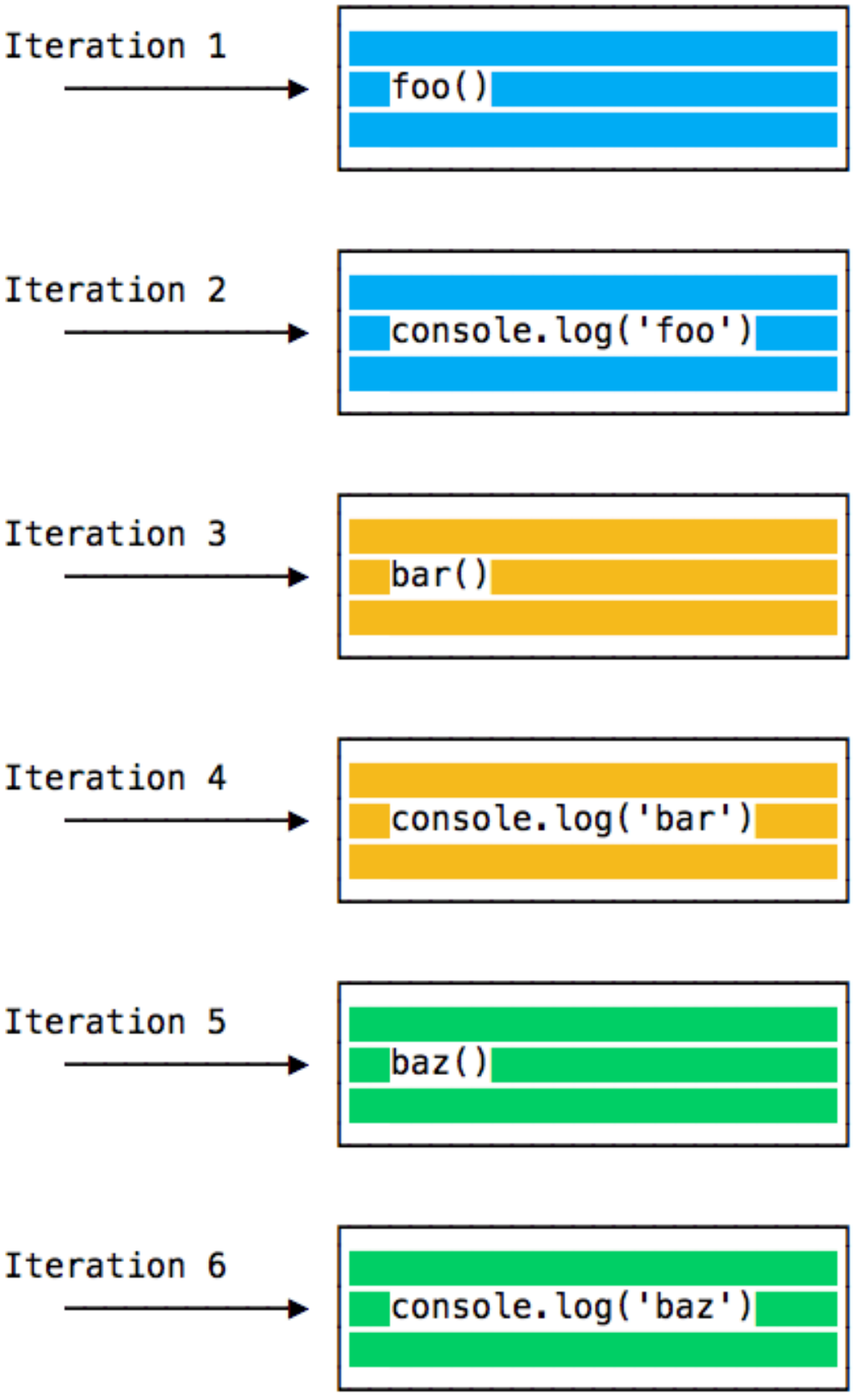
as expected.

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

At this point the call stack looks like this:



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()
```

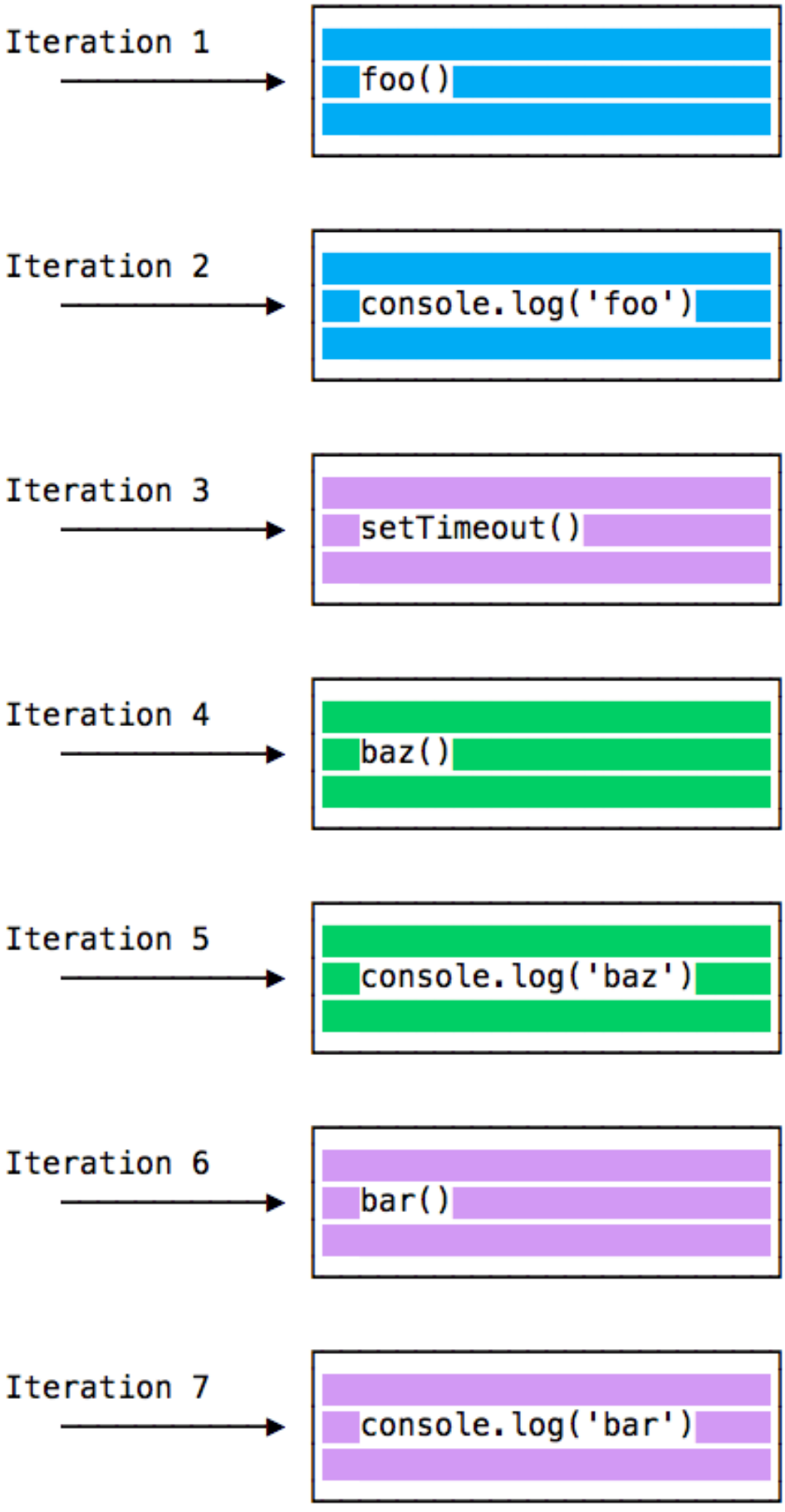
This code prints, maybe surprisingly:

```
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 start the timer. Once the timer expires, in this case immediately as we put 0 as the timeout, the callback function is put in the **Message Queue**.

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

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

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

ES6 Job Queue

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

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

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

Example:

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

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

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

foo()
```

This prints

```
foo
```

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

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

Conclusion

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

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

Understanding process.nextTick()

Understand how Node.js process.nextTick() works with the event loop, running your callback at the end of the current operation before the next tick starts.

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()

Understand how Node.js setImmediate() runs a callback in the next event loop iteration, and how it differs from setTimeout(0) and process.nextTick().

`setImmediate()` schedules a callback for the **check** phase of a future event-loop iteration:

```
setImmediate(() => {
  console.log('immediate')
})
```

It means “run after the current work yields and the loop reaches the check phase,” not “run immediately” and not “run after exactly zero milliseconds.”

Compare the three queues:

- `process.nextTick()` runs after the current JavaScript operation, before the event loop continues to another phase.
- `setTimeout(callback, 0)` schedules a timer after a minimum delay threshold. The operating system and other work can delay it.
- `setImmediate()` runs in the check phase, after the poll phase used for I/O callbacks.

From the top level of a script, the relative order of a zero-delay timeout and an immediate should not be treated as a contract:

```
setTimeout(() => console.log('timeout'), 0)
setImmediate(() => console.log('immediate'))
```

Inside an I/O callback, the immediate is scheduled for the check phase following that poll work and normally runs before a newly scheduled zero-delay timer:

```
import { readFile } from 'node:fs'

readFile(import.meta.filename, () => {
  setTimeout(() => console.log('timeout'), 0)
  setImmediate(() => console.log('immediate'))
})
```

`process.nextTick()` is not an ordinary event-loop phase. Recursively scheduling next-tick callbacks can starve timers and I/O because Node drains that queue before continuing. Use it for narrow API-ordering needs, not for breaking up large computations.

An `Immediate` normally keeps the process alive. Calling `.unref()` allows the process to exit if that callback is the only remaining work:

```
setImmediate(() => console.log('may run')).unref()
```

Ordering details can evolve with Node and its event-loop implementation. Write code around documented phase relationships, never a race observed once on your machine.

Your action is to run the top-level and I/O examples ten times on your Node version. Record the output, then add five recursive `process.nextTick()` calls and explain why an unbounded version would delay other work.

Use promises and await with Node.js callbacks

Learn how to use promises and async/await with Node.js callback-based functions by wrapping them with promisify from the util module to escape callback hell.

Most of the Node.js APIs were built in a time where promises weren't a thing yet, and they use a callback-based solution.

The typical Node.js API works like this:

```
doSomething(param, (err, result) => {

})
```

This also applies to libraries. One example is `node-redis`, and while working with it on a project, at some point I really had the need to remove all the callbacks, because I had too many levels of

callbacks nested into each other - a perfect "callback hell" scenario.

Also, sometimes it's absolutely necessary to avoid callbacks because you need to return from the function the result of a function call. If that's returned in a callback, the only way to get the result back would be to send it back with a function, and the callback party continues:

```
const myFunction = () => {
  doSomething(param, (err, result) => {
    return result //can't return this from `myFunction`
  })
}
```

```
const myFunction = callback => {
  doSomething(param, (err, result) => {
    callback(result) //no
  })
}

myFunction(result => {
  console.log(result)
})
```

There's an easy solution.

A solution provided by Node.js itself.

We can "promisify" any function that does not support promises (and as a consequence the async/await syntax) by importing `promisify` from the core Node.js `util` module:

```
const { promisify } = require('util')
```

Then we create new functions using it:

```
const ahget = promisify(client.hget).bind(client)
const asmembers = promisify(client.smembers).bind(client)
const ahkeys = promisify(client.hkeys).bind(client)
```

See how I added the `a` letter to mean *async*.

Now we can change this example "callback hell":

```
client.hget(`user:${req.session.userid}`, 'username', (err, currentUserName) => {
  client.smembers(`followers:${currentUserName}`, (err, followers) => {
    client.hkeys('users', (err, users) => {
      res.render('dashboard', {
        users: users.filter((user) => user !== currentUserName && followers.indexOf(user) ===
      })
    })
  })
})
```

into a much cleaner:

```
const currentUserName = await ahget(`user:${req.session.userid}`, 'username')
const followers = await asmembers(`followers:${currentUserName}`)
const users = await ahkeys('users')
```

```
res.render('dashboard', {
  users: users.filter((user) => user !== currentUserName && followers.indexOf(user) === -1)
})
```

This is optimal when using a function you don't have access to, like in this case where I use a 3rd party library.

Under the hood, promisify wraps the function in a promise, and returns it.

You can do this manually, too, returning a promise from a function, and then using it with async/await:

```
const handleLogin = (req, user) => {
  return new Promise((resolve, reject) => {
    req.login(user, (err) => {
      if (err) {
        return reject({
          error: true,
          message: err,
        })
      }
      return resolve({
        success: true,
      })
    })
  })
}

//...
const resultLogin = await handleLogin(req, user)
```

The Node Event emitter

Learn how to work with custom events in Node.js using the EventEmitter class from the events module, with on, emit, once, and removeListener methods.

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

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

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

You initialize an EventEmitter object using this syntax:

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

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

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

Emit and listen for events

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

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

When we run

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

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

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

Passing arguments to the event

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

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

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

Multiple arguments:

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

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

Listen for an event just once

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

Once that event is fired, the listener stops listening.

Example:

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

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

Removing an event listener

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

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

In this example:

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

Extract the callback:

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

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

So that later you can call

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

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

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

Getting the events registered

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

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

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

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

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

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

Adding more listeners before/after other ones

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

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

emitter.prependListener()

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

emitter.prependOnceListener()

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

Error handling in Node.js

Learn how to handle errors in Node.js using throw and Error objects, try/catch blocks, the uncaughtException event, promises, and async/await.

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 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 doSomething2().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)
  }
}
```

Node.js Streams

Learn how Node.js streams process data in chunks, handle backpressure, connect with pipeline, and create readable, writable, and transform streams.

Node.js streams process data over time instead of loading everything into memory first.

Files, HTTP messages, compression, standard input, and child-process output all use streams.

Streams are useful for two reasons:

- processing can start before all the data arrives
- memory use can stay bounded for large data

The important detail is **backpressure**. When a destination is slower than a source, streams signal the source to slow down.

The four stream types

Node.js has four main stream types:

- **Readable**: produces data
- **Writable**: consumes data
- **Duplex**: both readable and writable
- **Transform**: a duplex stream whose output is derived from its input

`fs.createReadStream()` returns a readable stream. `fs.createWriteStream()` returns a writable stream.

A TCP socket is duplex. A gzip compressor is a transform stream.

Copy a large file with pipeline

The simplest safe way to connect streams is `pipeline()`:

```
const { createReadStream, createWriteStream } = require('node:fs')
const { pipeline } = require('node:stream/promises')

async function copyFile() {
  await pipeline(
    createReadStream('video.mp4'),
    createWriteStream('video-copy.mp4'),
  )
}

copyFile().catch(console.error)
```

`pipeline()` forwards data, handles backpressure, propagates errors, and cleans up the connected streams.

It resolves when the pipeline finishes. It rejects when a stream fails.

The older `source.pipe(destination)` API is still useful, but errors are not automatically forwarded through a complete chain. Prefer `pipeline()` when you need completion and error handling.

Compress data through a transform

Add a transform stream between the source and destination:

```
const { createReadStream, createWriteStream } = require('node:fs')
const { pipeline } = require('node:stream/promises')
const { createGzip } = require('node:zlib')

async function compressFile() {
  await pipeline(
    createReadStream('archive.tar'),
    createGzip(),
    createWriteStream('archive.tar.gz'),
  )
}

compressFile().catch(console.error)
```

Each stream handles one job. The pipeline connects them.

Stream a file over HTTP

An HTTP response is a writable stream:

```
const { createReadStream } = require('node:fs')
const { createServer } = require('node:http')

const server = createServer((request, response) => {
  const file = createReadStream('guide.pdf')

  file.once('open', () => {
    response.writeHead(200, {
      'content-type': 'application/pdf',
    })

    file.pipe(response)
  })

  file.on('error', (error) => {
    if (!response.headersSent) {
      response.writeHead(404, {
        'content-type': 'text/plain',
      })
```

```
      response.end('File not found')
    } else {
      response.destroy(error)
    }

    console.error(error)
  })
})

server.listen(3000)
```

The response can begin as soon as the first file chunk is available.

The example waits for the file to open before sending a `200` response. Once response bytes have been sent, you cannot replace them with a clean error response.

Be careful when using `pipeline()` directly with an HTTP response. On an error it can destroy the socket before your code sends an error body.

Consume a readable stream

Readable streams are asynchronous iterables.

Use `for await...of` to process one chunk at a time:

```
const { createReadStream } = require('node:fs')

async function countBytes() {
  const stream = createReadStream('guide.pdf')
  let total = 0

  for await (const chunk of stream) {
    total += chunk.length
  }

  return total
}

countBytes().then(console.log)
```

The loop waits for the next chunk and rejects if the stream emits an error.

Do not mix consumption styles on one readable stream unless you understand its flowing state. Calling `pipe()`, listening for `data`, using `read()`, and async iteration are different ways to consume it.

Create a readable stream

Use `Readable.from()` for an iterable or async iterable:

```
const { Readable } = require('node:stream')

const stream = Readable.from([
  'Hello ',
```

```
  'from ',
  'a stream\n',
])

stream.pipe(process.stdout)
```

For a custom source, implement the `read()` method:

```
const { Readable } = require('node:stream')

let current = 1

const numbers = new Readable({
  read() {
    if (current > 3) {
      this.push(null)
      return
    }

    this.push(String(current))
    current += 1
  },
})
```

`push(null)` signals the end of the readable stream.

Do not call `push()` forever while ignoring its return value. Custom high-volume sources must respect the stream's demand and backpressure rules.

Create a writable stream

Implement `write()` in the constructor options:

```
const { Writable } = require('node:stream')

const logger = new Writable({
  write(chunk, encoding, callback) {
    console.log(chunk.toString())
    callback()
  },
})

logger.write('First message')
logger.end('Last message')
```

Call the callback when the chunk has been processed. Pass an error to the callback when writing fails.

`end()` can include one final chunk. It tells the stream that no more data will be written.

The writable emits `finish` after all data has been flushed. `close` means the underlying resource has closed. Those events do not mean the same thing.

Respect writable backpressure

`writable.write()` returns a boolean.

When it returns `false`, wait for `drain` before writing more:

```
const { once } = require('node:events')

async function writeChunks(writable, chunks) {
  for (const chunk of chunks) {
    if (!writable.write(chunk)) {
      await once(writable, 'drain')
    }
  }

  writable.end()
  await once(writable, 'finish')
}
```

Ignoring `false` lets data accumulate in memory and can eventually exhaust it.

`pipe()` and `pipeline()` manage this flow for you.

Create a transform stream

A transform receives chunks and produces new chunks:

```
const { Transform } = require('node:stream')

const uppercase = new Transform({
  transform(chunk, encoding, callback) {
    callback(null, chunk.toString().toUpperCase())
  },
})

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

Call the callback with an error as the first argument, or output as the second argument.

For a multi-stage production flow, use `pipeline()` instead of chaining bare `pipe()` calls.

Byte mode and object mode

Streams normally work with strings, buffers, and typed arrays.

Use object mode when each chunk is a JavaScript value:

```
const { Readable } = require('node:stream')

const users = Readable.from([
  { name: 'Flavio' },
  { name: 'Roger' },
])
```

`Readable.from()` uses object mode by default.

Object-mode buffer limits count objects. Byte-mode limits count bytes.

See the official Node.js documentation for streams, the promise-based `pipeline()`, and implementing stream classes.

Do not block the event loop

Recognize CPU-heavy or synchronous work that prevents a Node.js server from handling other clients.

JavaScript callbacks normally run on one event-loop thread. A long calculation or synchronous operation keeps that thread busy, so timers, requests, and completed I/O callbacks must wait.

Node can handle many connections with a small number of threads only when each callback finishes quickly. One request that performs a huge JSON parse, catastrophic regular expression, synchronous file read, or long loop can increase latency for every other client. If user input controls the expensive work, this can become a denial-of-service risk.

Asynchronous APIs separate waiting from JavaScript execution. Network I/O is coordinated by the event loop, while filesystem, crypto, and some other operations use a worker pool. That pool is finite too: flooding it with expensive tasks can delay unrelated work even though the event-loop thread remains free.

Make the delay visible with a heartbeat:

```
setInterval(() => {
  console.log('heartbeat', Date.now())
}, 100)

const end = Date.now() + 2000
while (Date.now() < end) {
  // Deliberately block the event loop for two seconds.
}
```

The missing heartbeat timestamps are evidence of event-loop delay, not slow networking. In a real service, measure tail request latency and event-loop delay with tools such as `monitorEventLoopDelay()` from `node:perf_hooks` before and after a change.

Choose the remedy based on the work:

- replace synchronous I/O in request paths with asynchronous APIs
- bound input size before parsing, matching, compressing, or hashing it
- split divisible JavaScript work into small batches that yield between them
- move sustained CPU-heavy JavaScript to worker threads or another process
- stream large data instead of buffering it all

Worker threads do not make an algorithm cheaper; they isolate CPU work so the main event loop can continue. They also add messaging and memory overhead. Yielding with `setImmediate()` can improve responsiveness for batchable work but still consumes the same main-thread CPU.

Your action is to run the heartbeat example, measure its largest gap, then replace the single loop with bounded batches that yield using `setImmediate()`. Compare responsiveness and total completion time.

Handle asynchronous failures

Catch expected promise failures near the operation and let unknown fatal states stop the process cleanly.

Wrap awaited operations in a useful error boundary and add context before reporting the failure.

Handle expected failures near the operation that understands them. A missing optional config file might have a valid fallback. A rejected database query might become an HTTP `503`. Preserve the original cause when adding context:

```
async function loadUser(id) {
  try {
    return await database.findUser(id)
  } catch (error) {
    throw new Error(`Cannot load user ${id}`, { cause: error })
  }
}
```

At the command entry point, catch the final rejected promise and mark failure:

```
async function main() {
  await runImport()
}

main().catch((error) => {
  console.error(error)
  process.exitCode = 1
})
```

This boundary is for reporting a failed command, not inventing a result. It also prevents a "floating" promise whose rejection has no owner.

Current Node behavior treats an unhandled rejection as an uncaught exception by default. An uncaught exception means the application may have partially changed state or violated an invariant. Do not install an `uncaughtException` listener and continue serving requests as though nothing happened.

If you need last-resort observation without changing Node's default crash behavior, `uncaughtExceptionMonitor` can synchronously record minimal diagnostic information. Avoid starting asynchronous recovery work that may never finish.

For a server, fatal handling and graceful operational shutdown are related but different. On a normal termination signal, drain work cleanly. After an unknown fatal error, perform only bounded, safe cleanup, stop accepting work, and let the process terminate. An external supervisor should restart it and apply backoff if failures repeat.

Do not call `process.exit()` immediately after `console.error()` unless abrupt termination is intentional; buffered output can be lost. Set `process.exitCode` or allow the uncaught error to terminate the process.

Your action is to create one expected rejected operation and one uncaught programmer error. Verify that the expected error gains context, both commands exit non-zero, and the programmer error does not leave a fake "healthy" server running.

Check your understanding: asynchronous code and events

Check event-loop ordering, callbacks, promisify, emitters, blocking work, uncaught failures, and nextTick versus setImmediate.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Servers and environment

Requests, responses, configuration, deployment, and shutdown.

How to read environment variables from Node.js

Learn how to read environment variables in Node.js through process.env, set custom ones on the command line, and load a .env file with dotenv.

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

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

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

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

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

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

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

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

Here we set 2 variables for API_KEY and API_SECRET

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

We can get them in Node.js by running

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

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

```
npm install dotenv
```

and at the beginning of your main Node file, add

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

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

If you ever need to convert a `.env` file to JSON or YAML (or back), I made a free env converter for that.

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

Load an environment file

Load local configuration with Node's built-in env-file flag while keeping secrets out of source control.

A current Node release can load a dotenv-style file without another package:

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

The file path is resolved from the current working directory. A missing file makes the command fail, which is useful when the application requires that configuration.

Example `.env`:

```
PORT=3000
API_BASE_URL=https://api.example.com
```

Node adds these values to `process.env`. Environment values are strings, so parse and validate them before use:

```
const port = Number.parseInt(process.env.PORT ?? '', 10)

if (!Number.isInteger(port) || port < 1 || port > 65535) {
  throw new Error('PORT must be an integer from 1 to 65535')
}
```

Do not let `Number(undefined)` silently become `NaN` far away from startup. Validate required configuration once, fail with the variable name, and never print secret values in an error.

An environment variable already set by the shell takes precedence over the value in the file:

```
PORT=4000 node --env-file=.env server.js
```

You can pass several `--env-file` flags. Later files override values from earlier files, while variables already present in the process environment still win. This supports a safe shared file plus a local override without custom merge code.

Current Node releases also provide `--env-file-if-exists` when an optional local file should not cause startup to fail. Use the strict flag for required production configuration so a missing file is visible.

Do not commit real secrets. Ignore `.env`, restrict its filesystem permissions, and provide `.env.example` containing variable names and harmless placeholders. Environment files reduce accidental source commits; they are not a full secret-management system.

Your action is to create a safe `.env` with `PORT=3000`, print the validated port, then override it from the shell. Try a missing and an invalid value and record each exit status without logging a secret.

Build an HTTP Server

Learn how to build a basic HTTP server in Node.js using the http module and createServer(), handling the request and response objects to send Hello World.

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

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

const hostname = 'localhost'
const port = 3000

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

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

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

We use the module to create an HTTP server.

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

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

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

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

In this case with

```
res.statusCode = 200
```

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

We also set the Content-Type header:

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

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

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

Serve an HTML page using Node.js

Learn how to serve an index.html page using Node.js with no dependencies, using the http module and piping fs.createReadStream() to the response.

You can serve one HTML page with Node's built-in HTTP and filesystem modules:

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

```
const indexPath = path.join(__dirname, 'index.html')

const server = http.createServer((req, res) => {
  const url = new URL(req.url, 'http://localhost')

  if (req.method !== 'GET' || url.pathname !== '/') {
    res.writeHead(404, { 'content-type': 'text/plain; charset=utf-8' })
    return res.end('Not found')
  }

  const file = fs.createReadStream(indexPath)

  file.on('open', () => {
    res.writeHead(200, { 'content-type': 'text/html; charset=utf-8' })
    file.pipe(res)
  })

  file.on('error', (error) => {
    console.error(error)

    if (!res.headersSent) {
      res.writeHead(500, { 'content-type': 'text/plain; charset=utf-8' })
      res.end('Cannot load page')
    } else {
      res.destroy(error)
    }
  })
})

server.listen(process.env.PORT || 3000, () => {
  const address = server.address()
  console.log(`Listening on http://localhost:${address.port}`)
})
```

`__dirname` locates the page beside this CommonJS module. A bare `'index.html'` path would be resolved from `process.cwd()`, so starting the server from another directory could break it.

The stream avoids loading the complete file into memory before sending it. Waiting for its `open` event also avoids announcing `200 OK` before knowing the file can be opened. Stream errors after headers begin cannot be converted into a clean `500`; the response must be terminated.

This is intentionally not a general static-file server. It serves only `/`, so user-controlled URL paths never become filesystem paths. A naive `path.join(publicDir, url.pathname)` implementation can create traversal and encoding bugs unless it normalizes, confines, and validates the result. Production static servers also handle MIME types, caching, ranges, conditional requests, compression, and `HEAD`.

Your action is to run the server from its own directory and its parent directory, request `/` and `/missing`, then rename `index.html` and verify that the missing file returns `500` instead of an empty `200` or a crashed process.

Get HTTP request body data using Node

Learn how to get the data sent as JSON in an HTTP request body in Node.js, using body-parser with Express or the request stream data and end events.

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 `body-parser` Node module.

For example, to get the body of this request:

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

axios.post('/todos', {
  todo: 'Buy the milk',
})
```

This is the matching server-side code:

```
const bodyParser = require('body-parser')

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

app.use(bodyParser.json())

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

If you're not using Express and you want to do this in vanilla Node, 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 put it into an array:

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

Make an HTTP POST request using Node

Learn how to make an HTTP POST request in Node.js using the Axios library, the Request library, or the built-in https module with https.request().

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

Since Node 18, the `fetch` function is built in, no install required. Today this is the way I recommend:

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

console.log(`status: ${res.status}`)
console.log(await res.json())
```

Save this as `post.mjs` and run it with `node post.mjs` (the `.mjs` extension enables top-level `await`). A successful call logs `status: 200`, or `201` when the API creates a resource.

Note the two things `fetch` makes you do yourself: turn the object into a string with `JSON.stringify()`, and set the `Content-Type` header. Forget the header and many servers refuse to parse the body, so you get a confusing `400` back.

Before `fetch` was built in, the most popular approach was the Axios library:

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

axios
  .post('/todos', {
    todo: 'Buy the milk',
  })
  .then((res) => {
    console.log(`statusCode: ${res.status}`)
    console.log(res)
  })
  .catch((error) => {
    console.error(error)
```

```
})
```

Axios serializes the object and sets the JSON header for you. It's still a fine choice, especially in codebases that already use it.

You will also find older code using the Request library. It was hugely popular for years, but it is deprecated and unmaintained, so don't pick it for new projects:

```
const request = require('request')

request.post(
  '/todos',
  {
    json: {
      todo: 'Buy the milk',
    },
  },
  (error, res, body) => {
    if (error) {
      console.error(error)
      return
    }
    console.log(`statusCode: ${res.statusCode}`)
    console.log(body)
  }
)
```

Using the standard https module

A POST request is possible just using the Node standard modules, although it's more verbose than the preceding options:

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

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

const options = {
  hostname: 'yourwebsite.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()
```

Here you build the request piece by piece: `req.write(data)` queues the body, and nothing goes over the wire until you call `req.end()`. Forgetting `req.end()` is the classic mistake with this API — the program just hangs, waiting, with no error to point you at the cause.

To quickly test an endpoint like this from the terminal, I built a free route to curl tool that generates the curl (or fetch) command from a route description.

Route requests by method and URL

Read the incoming method and URL, choose a handler, and return an explicit status and content type for every path.

The request object provides the HTTP method, URL, and headers.

Parse `request.url` against a base URL so the pathname and query string are separate:

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

function sendJson(response, status, value, headers = {}) {
  response.writeHead(status, {
    'content-type': 'application/json; charset=utf-8',
    ...headers,
  })
  response.end(JSON.stringify(value))
}

const server = createServer((request, response) => {
  const url = new URL(request.url, 'http://localhost')

  if (url.pathname === '/notes' && request.method === 'GET') {
    return sendJson(response, 200, { notes: [] })
  }

  if (url.pathname === '/notes') {
    return sendJson(
      response,
      405,
      { error: 'Method not allowed' },
      { allow: 'GET, POST' }
```

```
    )
  }

  return sendJson(response, 404, { error: 'Not found' })
})
```

Match both method and pathname. A query such as `/notes?limit=10` still has pathname `/notes`; read `url.searchParams` separately and validate every value.

Use status codes to describe the failure precisely:

- `404 Not Found`: this pathname does not identify a route
- `405 Method Not Allowed`: the pathname exists, but not for this method
- `400 Bad Request`: the route exists, but the submitted syntax or values are invalid

A `405` response should include an `Allow` header listing supported methods. Return immediately after ending a response so later router code cannot write headers or a second body.

Set `Content-Type` on success and error responses. JSON should normally include `charset=utf-8`. Do not return an HTML stack trace to an API client.

Routing selects a handler; it does not establish authorization. A valid `/notes/42` match must still check whether the caller may read note 42.

Unexpected handler failures need a final error boundary. Log diagnostic context on the server and return a generic `500` only if headers have not already been sent. Once a streamed response has begun, the safe recovery can be to close that response instead.

Your action is to add `POST /notes`, validate one query parameter, and send requests covering `200`, `400`, `404`, and `405`. Verify each status, body content type, and `Allow` header.

Node, the difference between development and production

Learn how to set up different configurations for development and production in Node.js using the NODEENV environment variable, plus Express config hooks.

You can have different configurations for production and development environments.

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

This is usually done by executing the command

```
export NODE_ENV=production
```

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

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

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

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

Setting the environment to `production` generally ensures that

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

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

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

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

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

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

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

Where to host a Node.js app

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

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

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

One thing to keep an eye on when comparing hosts is bandwidth pricing: I built a free bandwidth calculator that estimates your monthly traffic and its cost on different providers.

Simplest option ever: local tunnel

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

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

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

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

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

Zero configuration deployments

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

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

I use it a lot for demo purposes.

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

Serverless

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

To very popular solutions are

- Serverless Framework
- Standard Library

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

PAAS

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

Zeit Now

> Zeit is now called Vercel

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

Nanobox Nanobox

Heroku Heroku is an amazing platform.

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

Microsoft Azure Azure is the Microsoft Cloud offering.

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

Google Cloud Platform Google Cloud is an amazing structure for your apps.

They have a good Node.js Documentation Section

Virtual Private Server

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

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

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

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

Bare metal

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

Shut a server down gracefully

Respond to termination signals by stopping new connections and allowing current work to finish before the process exits.

Production platforms send a termination signal before replacing or stopping a process.

On Unix-like platforms this is commonly `SIGTERM`; a terminal interruption is commonly `SIGINT`. Installing a signal listener replaces Node's default exit behavior, so your handler must eventually let the process finish or mark failure.

Use one idempotent shutdown function:

```
let shuttingDown = false
let ready = true

async function shutdown(signal) {
  if (shuttingDown) return
  shuttingDown = true

  console.log(`Received ${signal}; draining`)
  ready = false

  const deadline = setTimeout(() => {
    console.error('Shutdown deadline exceeded')
    server.closeAllConnections()
    process.exit(1)
  }, 10_000)

  try {
    await new Promise((resolve, reject) => {
      server.close((error) => error ? reject(error) : resolve())
    })
```

```
    await database.close()
    clearTimeout(deadline)
  } catch (error) {
    console.error('Graceful shutdown failed', error)
    process.exitCode = 1
    // Keep the deadline active so the process cannot hang forever.
  }
}

process.on('SIGTERM', () => void shutdown('SIGTERM'))
process.on('SIGINT', () => void shutdown('SIGINT'))
```

Mark readiness false first so the load balancer stops sending traffic. `server.close()` stops accepting new connections and waits for active requests; on current Node releases it also closes idle keep-alive connections before completing. Close database, queue, telemetry, and other clients only after work that needs them has drained.

The deadline prevents one stuck request from hanging deployment forever. `server.closeAllConnections()` is forceful and can cut active responses, so reserve it for the deadline rather than the normal path. The immediate `process.exit(1)` is also deliberately abrupt here: the graceful window has already failed. Older supported Node versions can differ in how idle HTTP connections are reaped; check the version your service runs.

Do not call `process.exit(0)` immediately after receiving the signal. That defeats draining and can truncate logs or writes. When the server and resources no longer keep the event loop alive, Node exits naturally with the selected status.

Keep health checks honest during shutdown so traffic moves elsewhere.

Your action is to start one slow request, send `SIGTERM`, and prove that a new request is rejected while the existing one completes. Then create a deliberately stuck request and confirm the deadline forces a non-zero shutdown.

Build a small JSON server

Combine routing, request bodies, status codes, environment variables, file storage, and graceful shutdown in one final project.

Build a small notes API using the built-in HTTP module.

Define a small contract:

```
GET  /notes  -> 200 with { "notes": [...] }
POST /notes  -> 201 with the stored note
```

Every other path returns `404`. A known path with another method returns `405` and `Allow: GET, POST`.

Read request bodies with a size limit. A client controls how much data it sends and how slowly it sends it. Reject an oversized body with `413`, malformed JSON with `400`, and a body whose `text` is missing or blank with `400`. Check the request content type before treating bytes as JSON. Configure header and request timeouts as well—a byte limit alone does not stop a client that sends a tiny body extremely slowly.

Do not send a success response before durable application state has been updated. Load and validate

`notes.json` before the server starts listening. Serialize modifications through one in-process queue so two POST handlers cannot both read the same snapshot and lose one update.

Persist each new snapshot safely:

1. serialize the complete validated array
2. write a unique temporary file in the same directory
3. close it and apply any durability flush your requirements need
4. rename it over the destination
5. send `201` only after the rename succeeds

Atomic rename prevents a partial JSON file, but it does not coordinate several Node processes. Use a single writer, a lock designed for the platform, or a database if multiple instances must update the same data.

Choose and validate the port at startup:

```
const port = Number.parseInt(process.env.PORT ?? '3000', 10)

if (!Number.isInteger(port) || port < 1 || port > 65535) {
  throw new Error('PORT must be an integer from 1 to 65535')
}
```

Keep internal errors out of API responses. Log the error and its cause on the server, then return a generic `500` if headers have not been sent.

On `SIGTERM`, mark readiness false, call `server.close()` to stop accepting connections, wait for active requests and the write queue, then close remaining resources. Add a final deadline so a stalled client cannot block deployment forever.

Your action is to build the server and automate tests for malformed and oversized JSON, an unknown path, a wrong method, two concurrent POST requests, a restart, a storage failure, and a termination signal during a slow request. For every case, assert the status and the state visible after restart.

Check your understanding: servers and environment

Check server creation, request and response objects, ports, environment values, env files, request bodies, deployment, and runtime modes.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/nodejs/.