EXPERT INSIGHT

How Not to Node.js

A handbook to avoid common Node.js pitfalls and build cleaner, faster, and production-ready apps



Lucas Santos

‹packt›

How Not to Node.js

A handbook to avoid common Node.js pitfalls and build cleaner, faster, and production-ready apps

Lucas Santos

‹packt›

How Not to Node.js

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Ashwin Nair
Portfolio Manager: Aaron Lazar
Project Manager: Ruvika Rao
Content Engineer: Nisha Cleetus
Technical Editor: Arjun Varma
Copy Editor: Nisha Cleetus
Indexer: Pratik Shirodkar
Proofreader: Nisha Cleetus
Production Designer: Shankar Kalbhor
Growth Lead: Priyadarshini Sharma

First published: October 2026

Production reference: 1151026

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83664-849-9
www.packtpub.com

To everyone who ever believed I would get this far. Thank you.

– Lucas Santos

I'd like to thank Rogério Munhoz, who started this book with me and helped shape its first chapters, all my friends and family for keeping me company in the many late-night writing sessions it took to finish it. And all the team that helped me review and make this book the best possible content for all the readers.

Contributors

About the author

Lucas Santos is a Brazilian engineer, an eight-time Microsoft MVP, a seven-time Google Developer Expert, and a former Docker Captain living in Stockholm, Sweden. He has been programming professionally since 2011 and has worked at big corporations such as Microsoft and Klarna, as well as at several startups. He is currently a staff engineer at Detectify. He is an avid content creator and has been writing about tech content on his blog at lsantos.dev for over 10 years. He has presented over 150 talks in several countries. He is a former member and core contributor of the Node.js core team and has contributed to multiple open-source projects. When he is not writing code, he is usually playing music, building guitar pedals, or 3D printing something he probably does not need.

About the reviewers

Krishan Kant Sharma is a Principal Software Engineer at Fidelity Investments in Dallas, Texas, with nine years of experience in platform engineering, cloud-native systems, developer tooling, and observability. He created FlagLint, an open-source feature-flag migration tool, and is active in the OpenTelemetry and OpenFeature communities. A technical reviewer, speaker, and GeeksforGeeks mentor, he has been recognized for mentoring, including a Times Square feature. He holds a BS in Electronics and an MS in Information Systems and is pursuing an MS in Machine Learning at Georgia Tech. Outside technology, he enjoys travel and is pursuing a private pilot certificate in a Piper PA-38 Tomahawk.

Emmanuel Demey, with 15+ years in software engineering, helps development teams design, build, and evolve robust, maintainable systems. As a freelance consultant and certified trainer across France and the UK, he works across modern frontend and backend stacks, including React, Angular, TypeScript, Node.js, and Spring Boot. His technical focus spans software architecture, Clean and Hexagonal Architecture, web accessibility (WCAG/RGAA), Elasticsearch and search relevance, semantic search and RAG, as well as Green IT and digital eco-design. A Google Developer Expert since 2018, he has trained 200+ developers, co-organizes DevLille, and hosts the Tech Ethic podcast.

Table of Contents

Preface **xiii**

Free benefits with your book ..... **xvii**

Chapter 1: Understanding the Concept of Patterns and Anti-patterns **1**

Technical requirements **2**
Understanding what Design Patterns and Anti-Patterns mean **2**
What is a Design Pattern? • 3
What is an Anti-pattern? • 9
The Impact of Patterns in Application Design **9**
Patterns save brainpower • 9
Circuit breaker • 10
Common Examples of Anti-Patterns **10**
Callback Hell • 11
Global scoping • 12
Summary **16**

Chapter 2: Spotting and Solving Anti-patterns in JavaScript **17**

Technical requirements **17**
Identifying Code Problems and Anti-patterns **18**
High coupling between modules • 18
Godlike Structures • 21
Hard To Read Code • 24
Not DRY • 27
Too many parameters • 34
(Not-so) Smart functions • 36
Magic values • 39
Strategies for resolving Anti-patterns **43**
The code should speak for itself • 43
Ask Simpler Questions • 44

Make it Shorter • 45
Summary 45

Chapter 3: Staying Away from Smelly Code 47

Technical requirements..... 48
Understanding Code Smells 48
Common JavaScript Code Smells..... 50
Implicit type coercion • 52
Strings and Numbers • 52
Using numbers the right way • 53
Booleans and Objects • 56
Prototype Pollution • 60
False can be true • 61
Keeping track of the Spec 62
TypeScript and typing..... 64
TypeScript does not replace JavaScript at runtime • 65
Summary 66

Chapter 4: Dealing with the Most Common Anti-patterns in Node.js 69

Technical requirements..... 70
Working with the most common problem in Node.js..... 70
The use of blocking I/O • 70
Diving into blocking I/O • 72
Password Encryption • 75
Optimizing requests • 77
Understanding why data structures matter 85
Maps and Objects • 89
Objects and prototype pollution • 92
Arrays and Sets • 93
Working with large datasets, with Sets • 95
Choosing the tool for the job • 100
Objects versus Maps • 100
Sets versus Arrays • 101
Summary 102

Chapter 5: Understanding Promises and Callbacks 103

Technical requirements ... **104**
Taming the asynchronous code ... **104**
A tale of callbacks and promises • 104
Callbacks or promises? • 106
Understanding the callback anti-patterns ... **108**
The pyramid of doom • 108
Taming Zalgo • 111
Multiple execution • 115
Promise combinators • 116
Hanging operation • 117
Loss of context • 119
Understanding the promise anti-patterns ... **122**
The pyramid of doom (again) • 123
Dangling Promises • 124
Uncaught errors • 124
Memory leaks • 125
Understanding Async/Await • 126
Wrong promisification • 127
The never-caught expression • 128
Promisification of events • 129
Repromise • 130
Summary ... **133**
References ... **133**

Chapter 6: Logging Done Right 135

Technical requirements ... **136**
console.log is not a logger ... **136**
What console.log is really doing • 136
The pipe problem • 137
Mind the performance • 138
The debug package is not a production logger ... **139**
Picking a production logger ... **140**
Enters Pino • 141

pino-pretty is a development tool • 144
Adjust your log levels 145
Getting levels wrong • 146
Structured Logging 147
What to Log and what not to log 148
Correlation IDs • 149
What to never log • 150
Too much, too little • 151
Where logs should go 152
Always Log the full error 153
Observability 155
OpenTelemetry and the future of Node.js logging • 155
The anti-pattern checklist 156
Summary 157

Chapter 7: Taming Your Environment 159

Technical requirements 160
How process.env Works 160
The five failure modes • 161
Node's Native --env-file Flag 163
What --env-file Does Not Do • 164
Programmatic Loading • 165
process.env Everywhere 166
The validated singleton pattern • 167
Gotchas with environment variables and Zod • 170
TypeScript declare global • 171
Let's talk about dotenv 172
Let's talk about dotenvx • 172
Cascading .env Files 174
Enter config • 176
Putting It Together 178
The Anti-pattern Checklist 179
Summary 180

Chapter 8: What is Synchronous and What is Asynchronous 181

Technical requirements ... **181**
Understanding the Node.js event loop ... **181**
Blocking vs. non-blocking • 182
Event loop phases • 183
Exploring microtasks and macrotasks ... **187**
Microtasks: process.nextTick() and Promises • 189
Macrotasks: timers and I/O callbacks • 190
Mixed interactions • 192
Understanding why you should not block the event loop ... **195**
Cooperative Scheduling (chunking) • 196
Work delegation (Worker Threads) • 201
Learning not to block the event loop ... **203**
Summary ... **205**

Chapter 9: Memory Leaks and Where They Come From 207

Technical requirements ... **208**
How Node.js manages memory ... **208**
All you need is reachability • 209
Reading memory usage • 209
What a memory leak actually is ... **210**
Where memory leaks come from ... **212**
Out-of-control module-level states • 212
Closures and variable leaks • 213
Caches with no bound • 214
Event listeners need to stop listening at some point • 215
Timers that are never cleared • 216
Promises and streams • 217
Promises that never settle • 217
Unbounded fan-out • 218
Streams without backpressure • 219
Finding a leak ... **219**
Step one: confirm it is actually a leak • 220
Step two: capture the heap and compare • 221

Capturing snapshots without a browser • 222
Other tooling • 223
How to not leak in the first place 223
Bound everything that grows • 223
WeakRef and FinalizationRegistry, with care • 224
Keep scopes... Scoped • 225
The anti-pattern checklist 226
Summary 226

Chapter 10: Unlock the Code Bundle and the PDF Version 229

Unlock this book's free benefits in three easy steps 230

Other Books You May Enjoy 234

Index 237

Preface

Node.js makes it easy to get a server running. A few lines of code, a `Node index.js`, and you're handling real traffic. That low barrier to entry is one of the best things about the runtime, and it's also how most of the bugs in this book get written. The easy path works on your machine, passes the tests, gets approved in code review, and then breaks in production: under load, after a few days of uptime, or the first time someone forgets to set an environment variable.

This book is about those bugs. It won't teach you Node.js from scratch. Instead, each chapter takes one area where applications tend to go wrong, explains how that part of the runtime works under the hood, and then walks through the anti-patterns I keep finding in real codebases, along with what to do instead. Most of the fixes are small. The hard part is knowing why they're needed, and that's what most of the pages are spent on.

I've been writing Node.js professionally for over a decade and contributing to Node.js itself for the last few years. A lot of what's in here comes from incidents I had to debug, pull requests I reviewed, and questions I answered more times than I can count. My hope is that, after reading it, you'll catch these problems in review instead of at 3 a.m.

Who this book is for

This book is for JavaScript developers who already build Node.js applications and want to stop shipping the bugs that only show up in production. You should be comfortable with modern JavaScript, `async/await`, and the basics of building a Node.js service, such as starting an HTTP server, reading files, and installing packages with npm. You don't need to know how the event loop, V8, or libuv work internally; each chapter explains the parts you need as it goes.

If you lead a team or review a lot of code, you'll get extra mileage out of the anti-pattern checklists in the chapters. They're written to be used during code review.

What this book covers

Chapter 1, *Understanding the Concept of Patterns and Anti-patterns*, introduces design patterns and anti-patterns, explaining how they affect software development and using practical JavaScript examples to demonstrate how problematic code can be improved.

Chapter 2, *Spotting and Solving Anti-patterns in JavaScript*, explores how to identify common JavaScript anti-patterns, including tightly coupled modules and overly complex structures, and introduces practical strategies for resolving them.

Chapter 3, *Staying Away from Smelly Code*, examines common JavaScript code smells, including implicit type coercion, and explains how to identify and address issues that affect code readability and maintainability.

Chapter 4, *Dealing with the Most Common Anti-patterns in Node.js*, focuses on performance-related anti-patterns in Node.js, particularly blocking I/O and inefficient data structures, and demonstrates how to address them.

Chapter 5, *Understanding Promises and Callbacks*, explains the differences between callbacks and promises, explores common asynchronous programming anti-patterns, and demonstrates how to avoid them.

Chapter 6, *Logging Done Right*, examines common logging mistakes in Node.js and introduces production logging practices, including structured logging, appropriate log levels, and secure handling of sensitive information.

Chapter 7, *Taming Your Environment*, explores environment variable management in Node.js, highlighting the limitations of `process.env` and explaining how to load, validate, and manage application configuration reliably.

Chapter 8, *What is Synchronous and What is Asynchronous*, explains how the Node.js event loop handles synchronous and asynchronous operations, exploring blocking behavior, event loop phases, and their impact on application performance.

Chapter 9, *Memory Leaks and Where They Come From*, explores memory management and garbage collection in Node.js, explaining how memory leaks occur and how to identify, diagnose, and prevent them.

To get the most out of this book

All the examples in this book are written in plain JavaScript and run on Node.js 22 or later. They were tested on Node.js 24, which is the version I recommend you install. Some chapters use third-party packages, such as Pino and Zod; each chapter lists the packages it needs and how to install them.

Software covered in the book	Operating system requirements
Node.js 22 or later	Windows, macOS, or Linux
npm 10 or later	Windows, macOS, or Linux

> **Tip**
>
> If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository. Doing so will help you avoid any potential errors related to copying and pasting and will also help you learn.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to `packtpub.com`
2. Click your profile picture and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code:



2. Search for this book
3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

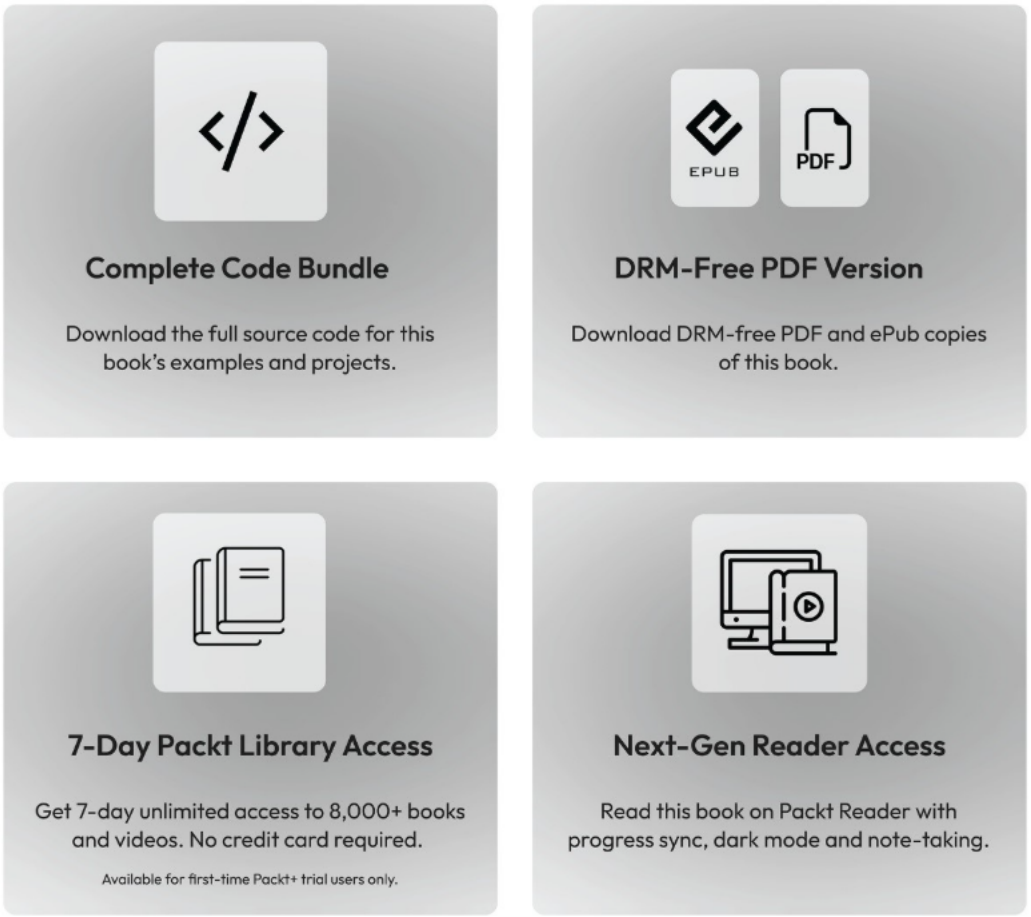
Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Share your thoughts

Once you've read *How Not to Node.js*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



https://packt.link/r/1836648499

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Understanding the Concept of Patterns and Anti-patterns

If you're a developer, you have probably heard about the infamous *Design Patterns*, but very few of us have heard about their archfoes and opposites, the *Anti-Patterns*. In this chapter, we will dive into what is a design pattern, what makes a design pattern be called a design pattern and understand the concepts of what it means to be a pattern or anti-pattern.

More important than understanding just what patterns are, we will also understand how they impact your application in a real scenario, and why they are called patterns after all, but we won't just be talking in concepts, we will also see some examples of how they look in real code and how we can turn anti-patterns into more manageable code.

In this chapter we are going to cover the following topics:

- What are Patterns and Anti-Patterns in the context of software engineering
- How those patterns impact your application in the real world
- What are some common patterns and anti-patterns we see when working with JavaScript

It's not only extremely important for you to understand what we will be covering in this book, but also, as a developer, it's important to understand and spot those patterns so you know how an application is built and can save precious time debugging the code. This is what you will be able to do by the end of this chapter.

Technical requirements

For this chapter, you don't have to know a lot of previous content, since we won't be diving into the internals or anything that relates to complicated computer science concepts. The idea here is that you already have a coding background, as we won't go deeper into the language itself, or its concepts. The focus of this chapter is solely to level your knowledge in the basic building block we'll be talking about: Anti-patterns. So, it's important that you already know basic concepts of coding like:

- Scope
- Variables
- Context
- Callbacks, Promises, and Asynchronous programming

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

We encourage you to download and run the examples, so you understand how they work.

It's also a prerequisite to have Node (22 or later) installed and ready.

> **Note**
>
> **Download the code bundle and the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Understanding what Design Patterns and Anti-Patterns mean

In 1977, three architects, *Christopher Alexander*, *Sara Ishikawa*, and *Murray Silverstein* published a book called *A Pattern Language: Towns, Buildings, Construction* about the concept of a *language* that would describe how you can design urban environments. The unit of this language was called a *pattern*. In this context, a pattern is a description of common architectural standards, like, for example, how high windows should be, how the stairs should be positioned, how tall a room should be, and so on.

Then, in 1987, *Kent Beck* and *Ward Cunningham* started experimenting with the idea of applying common patterns to programming. A few years later, in 1994, the idea of patterns was picked up by *Erich Gamma*, *John Vlissides*, *Ralph Johnson*, and *Richard Helm* and published into *Design Patterns: Elements of Reusable Object-Oriented Software* which is probably one of the most famous and impactful books in the software engineering industry. Due to its lengthy name the book is simply called *The book by the gang of four* or *The GoF book*.

In this book, they applied the same concepts that Kent and Ward were applying to programming but in a more documented way. Through several experiments, observations, and applications, they presented a list of the more common problems faced by the software industry and a series of *patterns* that could be applied to the application design to solve them. Hence the name: **design patterns**.

What is a Design Pattern?

A **design pattern** is simply a well-tested, documented solution to a common problem in software design, for example, a common problem in software design is a race condition. When two clients (by clients, we mean anything that could perform an action, like a function, a class, or anything else) try to modify a resource at the same time. This is particularly common in files.

In the following file called `race-condition.js` we have a race condition where two operations try to write to the same file at the same time asynchronously. Every time this script is run, you might end up with the second phrase being written before the first:

```
import { writeFile, unlink } from 'node:fs/promises'
import { dirname, join, resolve } from 'node:path'
import { fileURLToPath } from 'node:url'

async function raceCondition() {
    const filename = join(resolve(dirname(fileURLToPath(import.meta.url)), './
race.txt'))
    await unlink(filename)

    writeFile(filename, 'Written from first promise\n', { flag: 'a' })
    writeFile(filename, 'Written from second promise\n', { flag: 'a' })
}

raceCondition()
 .then(() => console.log('Race condition completed'))
 .catch(console.error)
```

> **Note**
>
> The prior code can be found in the `race-condition.js` file within this chapter's GitHub folder. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface*.

This code block tries to create a **race condition**, a state where two different threads try to access or modify the same resource (for example, a file) at the same time. We first select the directory where our file will be read or created, which is the local path of this same script. If the file exists, we remove it with `unlink` so we always have a clean slate to start the script. Then we start two async operations of `writeFile`. Each one of them will create a `promise` object which won't be tied to the current function (since we didn't use an `await` keyword). So, they will run in background at their own pace, can you imagine what will happen?

If you thought that the file will be accessed at the same time, you are correct! Each of the promises will open a file descriptor to that specific resource and will try to write in it, however, it won't follow the order we defined because they both want to access the same resource at the same time.

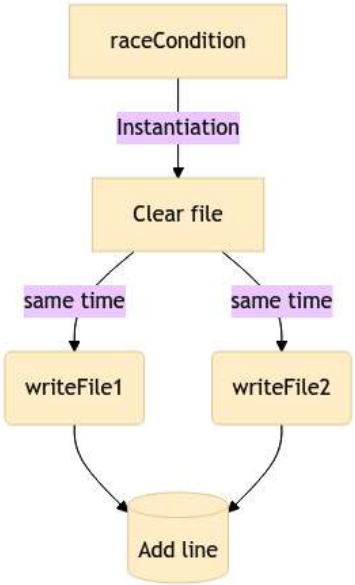


Figure 1.1: Two handlers trying to write to the same file

If you run this command using (on Unix-based systems) `watch -n1 node race-condition.js` and keep the file open in a second window, you will see that the order of the phrases varies from time to time.

This is one example of a race condition, another example could be when we try to read from a file and it's already open for writing in another place, so we won't get the most recent changes to that file.

A common way to solve this problem is to create what's called a **mutex**, a structure that can act as a semaphore to control access to the file.

There are different implementations of mutexes out in the ecosystem, in user libraries like async-mutex, but also in native implementations with structures like Atomics and SharedArrayBuffers, however, these implementations require the code to be running in different threads.

One way to do this is to centralize the access to the files in a class that's responsible for managing when a file is being written or not, blocking other threads from accessing the file while it's being used. This is what the following code does:

```
import { setTimeout } from 'node:timers/promises'
class FileWriter {
    #isWriting = false // this is our lock
    // There can only be one instance, a singleton pattern
    static instance = null

    constructor() {
        // If there's no instance, create one
        if (!FileWriter.instance) {
            FileWriter.instance = this
        }
        // Otherwise return the existent one
        return FileWriter.instance
    }
```

```
    async writeFile(filename, data) {
        // Check for the lock state
        if (this.#isWriting) {
            console.log('Already writing. Waiting for the current write to
complete.')
            await setTimeout(250)
            return this.writeFile(filename, data)
        }

        // If the lock is open, close it
        this.#isWriting = true
        try {
            console.log(`Writing to ${filename}`)
           return await writeFile(filename, data, { flag: 'a' })
           }
            finally {
            this.#isWriting = false
             }
    }
}
```

The `FileWriter` class has one single responsibility: to take all the operations that would access a particular file and control the access to it so no two operations have it open at the same time, even if they're async.

There can only ever be one instance of this semaphore at a time, otherwise it would be useless (can you imagine the same street with two semaphores?). We do this by creating a static property that will store the current instance of this class and, on its constructor, we check if this instance has been initialized, if it has, we just return it, if not, we create a new instance and set it on the property.

The real magic happens with the private property `isWriting`, a flag that will flip to `true` every time we call the `writeFile` function. If the flag is `false`, we proceed to write to the file, but if it's `true`, we will log a message to the screen and wait 250 milliseconds, then call the same function again with the same parameters.

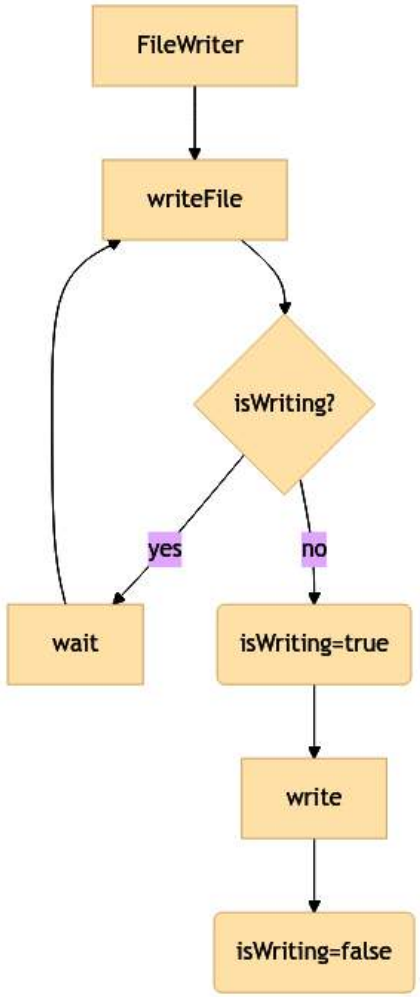


Figure 1.2: The Singleton-mutex pattern

Up until now, this is just a class, we need to instantiate it and run our code. This can be done if we add the following lines just below it:

```
const writer = new FileWriter()
async function raceCondition() {
    const filename = join(resolve(dirname
        (fileURLToPath(import.meta.url)), './race.txt'))
    await rm(filename, { force: true })
```

```
    writer.writeFile(filename, 'Written from first promise\n')
    writer.writeFile(filename, 'Written from second promise\n')
}

raceCondition()
.then(() => console.log('Race condition completed'))
.catch(console.error)
```

Note

This code can be found in the `race-condition-singleton.js` file within this chapter's folder in the GitHub project. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface*.

It's the same code we had before, but now, instead of just calling the function from the `node:fs/promises` package, we will use an instance of our `FileWriter` class to do so.

This code will always print the correct output, with the first line appearing before the second in the file. Additionally, it will print an output even when the file is busy.

In addition to the Mutex, this pattern is called a **Singleton**.

The Singleton is just one of a myriad of proven techniques that, since the 1980s, have been applied to software development to create solutions to several of the problems you will see as a programmer.

However, it's also important to notice that, while design patterns are a good practice, they bring their own problems to the table. So, overusing them can be just as harmful as anti-patterns. Singletons are one of these patterns.

If overused (or misused) it can prove to be a liability. Since singletons create a single instance of the class in memory, testing this type of code is especially difficult because you can't simply change its behavior on-the-go, for example, with module mocking, singletons need to be recreated with the desired behavior, which can lead to longer testing times.

Another downside – however, rarely seen – is that, in distributed systems, singletons will behave as a single entity. Note that this only applies within a single process: child processes do not share memory, so each one gets its own singleton. Any changes or misbehavior with the singleton will propagate to all the code that's using it.

This outlines an important aspect of application design: *There's no free lunch*. Every decision you take in programming will come with an evil counterpart, it's up to you as the architect to weigh if the counterpart is acceptable or not.

What is an Anti-pattern?

We've discussed design patterns, but this book focuses on their opposite—anti-patterns. As you might expect, an anti-pattern is the reverse of a design pattern.

While design patterns are proven practices that solve recurring problems effectively, anti-patterns are practices that tend to create more problems than they solve. Anti-patterns often arise from poor design choices, misunderstood requirements, or the temptation to take shortcuts. Developers might initially follow an anti-pattern because it appears to work, but it typically leads to technical debt, maintenance issues, or scalability problems later.

Unlike design patterns, which are refined solutions developed and validated over time by the broader developer community, anti-patterns often reflect local, ad-hoc approaches that seem helpful in the short term but result in complexity, inefficiency, and difficulty in maintaining or extending code. This is why we focus on design patterns first—it's essential to know what to do, but it's just as important to recognize what not to do and how to identify these bad practices in your code.

In the following chapters, we will explore common anti-patterns in more detail. A few notable examples include **God Objects**, where a single object takes on too many responsibilities; **callback hell**, where deeply nested callbacks make the code hard to follow; and **spaghetti code**, where code becomes so tangled and unstructured that it's nearly impossible to understand or debug.

Now that we understood what patterns and anti-patterns are, and even saw an example of how they look in real code, let's understand *why* it's so important to know them.

The Impact of Patterns in Application Design

As a software developer, you will often be solving problems that someone else already had to solve before. In that context, design patterns essentially allow you to make use of the knowledge of those people to make your applications more robust and easier to maintain, saving you and your team a huge amount of time and brainpower.

Patterns save brainpower

Speaking of brainpower, another one of the main advantages of using design patterns is that you and your team can communicate more effectively. Imagine the following scenario: you work for an e-commerce company; whenever a user completes a purchase, they should receive an email informing them of the successful transaction. To do that, you have a queue with one job per email that has to be sent, and a few worker instances that fetch and execute the jobs. However, the email service your company uses is constantly running into downtime, and changing providers is financially out of the question.

Now the big problem is every time the email service goes down, your workers start to go crazy on resource consumption because they keep retrying the same jobs over and over, as well as new jobs that arrive.

To solve that, you can use key-value pair storage to store a value that tells workers if the queue is *paused* or not. Whenever too many jobs fail consecutively, that value is set to *true*, and workers check that value before they try to process a job (if it's true, it fails immediately) without wasting resources on an API call that is expected to fail. From time to time, a lambda function can be spawned to make an attempt at calling the API and, if the call is successful, set the value to false and allow job processing to continue.

Circuit breaker

Everything you just read can be simplified to two simple words: *Circuit Breaker*, which is the name of a pattern that fits exactly into that description and provides a way to *back off* from trying something you know will fail to save resources and time.

Design patterns give you and your team a universal language that allows for efficient and clear communication. After all, it's way easier to propose a solution when the whole team knows what you're talking about without you having to explain it step by step.

Patterns such as the Circuit Breaker pattern have been tested throughout the years and are known solutions for common problems. Using them can improve code maintainability and legibility, while also reducing the brainpower required to communicate with your team. They aim to ensure you steer clear of known pitfalls and can focus on your business logic, instead of wasting time trying to solve problems that others have already solved before.

But enough about the concepts, we understood what patterns are, why they are important, and got some examples on what to do with them, but this book is about anti-patterns, so let's see some examples of what *not* to do!

Common Examples of Anti-Patterns

We already saw a few examples of patterns being used in the code, but how do we identify anti-patterns? Well, it's not that easy. While design patterns are well-documented, well-tested, and well-known, anti-patterns are the complete opposite. Some look like bad code—these are the easiest to spot, like the **Callback Hell**, others are not so easy to see, like **memory leaks**, **scope leaks**, and many others.

Some anti-patterns are like that buzzing mosquito in your bedroom at night, you can't really see it so easily, but you can hear it buzzing in your ear while you sleep, and you need to make a conscious effort to make it stop! It's the same with code; you don't always see an anti-pattern right away, but it will present itself, usually in situations where you need to perform either

maintenance or create a new feature, and, just like the mosquito, if you don't try to kill it, it's going to keep buzzing all night long...

Let's go through some of the most common anti-patterns in Node.js so we can warm up for what's coming next. But this is not all! We will cover some of them loosely here, but we will get into more detail in the upcoming chapters.

Callback Hell

The **Callback Hell**, or **Pyramid of Doom**, is possibly the most common anti-pattern not only in JavaScript but in every language that uses asynchronous operations through callbacks. Simply put, it's when you nest too many callbacks in a single call. You'll see this mostly when there are steps to be taken in order, usually through multiple API calls or multiple timeouts, for example:

```
function stepOne(callback) {
  setTimeout(() => {
    console.log("Step One completed");
    callback(null, "Data from Step One");
  }, 1000);
}

function stepTwo(dataFromStepOne, callback) {
  setTimeout(() => {
    console.log("Step Two completed");
    callback(null, "Data from Step Two");
  }, 1000);
}
```

In the prior code we are creating two different functions that will be basically tied to each other, the second function needs data from the first function. This way we can showcase this anti-pattern. But why keep it simple? Let's add another one!

```
function stepThree(dataFromStepTwo, callback) {
  setTimeout(() => {
    console.log("Step Three completed");
    callback(null, "Data from Step Three");
  }, 1000);
}
// Now we execute them
stepOne((err, resultOne) => {
  if (err) {
```

```
    console.error("Error in Step One");
    return;
  }
  stepTwo(resultOne, (err, resultTwo) => {
    if (err) {
      console.error("Error in Step Two");
      return;
    }
    stepThree(resultTwo, (err, resultThree) => {
      if (err) {
        console.error("Error in Step Three");
        return;
      }
      console.log("All steps completed: " + resultThree);
    });
  });
});
```

We will discuss the solution to this problem in *Chapter 5* but, for now, let's focus on what this code means to your application.

As you can see, when every timeout ends, we need to start a new callback checking for errors in the previous step and continue to do so in the future callbacks. This can go on forever with the next callback being one level deeper than the previous. The effects of this anti-pattern are usually not seen in performance, but in developer experience; It's *extremely* hard to debug and understand what's happening in there.

This is a clear example of an anti-pattern: **the code is hard to read and understand**.

Global scoping

Another very common example of an anti-pattern that's easy to see is the use of `var`. The `var` keyword has been around since the beginning of JavaScript, but, since ES6, its usage should be thoroughly avoided. The reason is simple: it can easily access global scopes; this is one of the worst possible anti-patterns ever.

In the original definition, `var` works in two ways:

- If the variable is declared within a function, its scope is bound to that function, which means the variable will be accessible through the entire function scope
- If declared outside of a function, its scope is bound to the global scope, which means everything inside the program will have access to that value (in browser scripts; in Node.js modules, a top-level var is scoped to the module)

Let's understand this with a simple example. A function that declares a variable inside an if scope and tries to access it outside of the scope:

```
function example() {
  if (true) {
    var message = "This is a message.";
  }
  console.log(message);
}
```

In normal conditions, you would expect that the `message` variable does not exist outside of the if scope. When the runtime compiles the code, it will perform an action that's called **hoisting**, this essentially means that the entire block where the variable is being defined will be hoisted to the beginning of the function. The actual code that's being executed is this:

```
function example() {
  var message;
  if (true) {
    message = "This is a message.";
  }
  console.log(message);
}
```

This is a simple example, but using `var` can be even worse than that. For example, what would the following code print?

```
for (var i = 0; i < 3; i++) {
    setTimeout(function () {
        console.log("var loop:", i);
    }, 1000);
}
```

The expectation we have is to receive an output like:

```
var loop: 0
var loop: 1
var loop: 2
```

But this is not what happens. Because var is function-scoped, the loop has only one i, and all three callbacks share it. When they run after the loop has finished, i is already 3, so the output will be:

```
var loop: 3
var loop: 3
var loop: 3
```

This, of course, is only happening because we are delaying the printing of the value. In normal conditions, it would work as expected, but it's a good example to show what high-level scopes are about, and the problems they can bring. This delay could be caused by an API call, or any other long-running process.

Both examples can be solved using the most recent standards of `let` and `const`. They behave very differently than `var`:

- Both `let` and `const` are block-scoped. They are hoisted to the top of their block, but you cannot access them before their declaration (the temporal dead zone), so you get an error instead of undefined.
- Const does not allow reassignments, so once it's been defined, it cannot be redefined, however, objects can be mutated.

If we were to solve the previous examples, we would apply `let` and `const` instead of var. For the first example, we need to create a function-scoped variable, since defining it inside the if-block would scope it only to that block:

```
function example() {
    let message = ''
    if (true) {
      message = "This is a message.";
    }
    console.log(message);
}
```

If we don't define `message` outside the block, the variable won't exist, and this will return us an error. For the second example, we can simply replace `var` with `let`, since `const` won't allow the loop to reassign the variable:

```
for (let i = 0; i < 3; i++) {
    setTimeout(function () {
        console.log("var loop:", i);
```

```
    }, 1000);
}
```

Now the output of this function will be correct, because at every iteration, the reference passed to the callback will be a different one containing the value of `i` at that point. Essentially, every new iteration will create a new variable, which happens when you use let (or const in for...of and for...in loops).

But this is not it. We have yet another example where `var` falls short:

```
function example() {
    var x = 10;
    if (true) {
        var x = 20;
        console.log("Inside if block:", x);
    }
    console.log("Outside if block:", x);
}
```

In this example, both consoles will print 20, because even though we first defined the variable `x` outside of the scope, we are redefining it inside the `if` block, since `var` is function-scoped, the var `x` inside the if block is the same variable, so you are replacing its value from `10` to `20`.

Again, the solution here is simple, using `const` would allow you to define two variables with the same name, but in different scopes, which doesn't cause a conflict. So, instead of the previous code, you can do it like this:

```
function example() {
    const x = 10;
    if (true) {
        const x = 20;
        console.log("Inside if block:", x);
    }
    console.log("Outside if block:", x);
}
```

And now both variables will be correctly printed.

These are just some examples of some bad patterns and smelly codes we will go through. We have been talking about them very broadly for now, but in the upcoming chapters we will start to dig deeper into what they mean and the specifics of each one.

Summary

In this chapter, we began by defining what a pattern is and what an anti-pattern is, providing examples to help distinguish between good patterns and bad ones. We also learned that, while design patterns are easy to spot, anti-patterns are pesky little mosquitoes that lurk in your codebase, sometimes they're big and buzzy, but most of the time they're hiding there just waiting for the perfect moment to buzz in your ear. To exemplify that we went through two of the most common anti-patterns in JavaScript. The first one is callback hell, where the nesting of multiple callbacks stacks up until the code is very hard to read, and this is the first sign that this is an anti-pattern.

In our second common anti-pattern, we played around with the scopes, showing how using the keyword `var` can be damaging to your code because it causes the code to behave in unexpected ways. We saw that all of this can be easily solved by using `let` and `const` keywords, but these are just a few examples of the anti-patterns you might encounter in JavaScript. In the next chapter, we'll explore how to identify common anti-patterns and the strategies you can use to resolve them.

2

Spotting and Solving Anti-patterns in JavaScript

Up until now, you learned what a pattern is and its opposite, the *anti-pattern*. You understood the differences between each of them and even saw a few of those pesky anti-patterns. However, you still do not know *how* to spot these smelly pieces of code, or how to solve the most common anti-patterns that have been around for years.

These topics are why, in this chapter, you are going to tackle these issues with the following main topics:

- How to identify code smells before they become bigger problems
- Common anti-patterns in JavaScript and how to solve them
- Common strategies for resolving anti-patterns in Node.js

The chapter starts with how to identify the pesky code smells before they turn into huge problems, guiding you bit by bit on how to take an iterative approach to solving them. Then, the chapter moves on to some other common anti-patterns in JavaScript to show how these ideas can be applied. Finally, it all comes together in several strategies for resolving anti-patterns in Node.js. You can dive into it!

Technical requirements

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Identifying Code Problems and Anti-patterns

Code smells and anti-patterns are not born as huge problems. Instead, they usually start with small, innocent-looking details and then evolve into huge snowballs. These details are why it is so important to learn how to spot these kinds of problems as soon as they get introduced into the codebase, whether in a code review session or before you commit your changes.

Whether you are writing your own code or reviewing someone else's work, it is important to be on the lookout for some signs that might indicate an anti-pattern is starting to brew. The next paragraphs investigate a few of those issues.

Be aware that some of those issues are not limited to JavaScript; they can happen to essentially any language, and, therefore, the solutions can also apply to a variety of other tools.

High coupling between modules

Whenever two modules of an application need to be invoked to complete a certain task, it is inevitable that they share information about the task itself. However, each module should take care of its own domain and should not know about the inner workings of other modules.

You can look at an example of a user service that is responsible for creating a user and calling an email service to send a welcome mail to them. In the following code, you see that the user service knows about the different email providers available to the email service and performs some logic to know what method to call when using each provider.

This example uses TypeScript because it is important for you to understand the implications of encapsulation.

```
// userService.ts
 import { EmailService } from './emailService';

export class UserService {
  constructor(private emailService: EmailService) {}
     createUser(username: string, email: string) {
    if (email.endsWith('@gmail.com')) {
      this.emailService.sendViaGmail(email, 'Welcome!');
    } else if (email.endsWith('@outlook.com')) {
      this.emailService.sendViaOutlook(email, 'Welcome!');     } else {
      throw new Error('Unsupported email provider');
    }
    console.log(`User ${username} created.`);
```

```
  }
}
```

The following code defines the email service and exposes one method for each provider:

```
// emailService.ts
 export class EmailService {
  sendViaGmail(email: string, message: string) {
    console.log(`Sending email via Gmail to ${email}: ${message}`);
  }

  sendViaOutlook(email: string, message: string) {
   console.log(`Sending email via Outlook to ${email}: ${message}`);
  }
}
```

Now if you want to add a new provider, you must change both the user and email services. Furthermore, if any of the providers require different parameters or if any of the methods need to have a different signature, that change will also affect the user service.

All this means both services are too tightly coupled. The user service knows and depends on the inner workings of the email service and even takes over some of the responsibilities since it must decide which email provider to call.

Furthermore, if you need to use the email service anywhere else, you will have to choose a provider again, leading you to code that is even harder to maintain.

The following code snippets show one way of solving that problem. By bringing the logic of choosing a provider to the email service itself, you can expose a single method, allowing much simpler and generic usage of its capabilities.

```
// userService.ts

import { EmailService } from './emailService';

export class UserService {
  constructor(private emailService: EmailService) {}

  createUser(username: string, email: string) {
    this.emailService.sendEmail(email, 'Welcome!');
    console.log(`User ${username} created.`);
  }
```

```
}
// emailService.ts

export class EmailService {
  sendEmail(email: string, message: string) {
    if (email.endsWith('@gmail.com')) {
      this.sendViaGmail(email, message);
    } else if (email.endsWith('@outlook.com')) {
      this.sendViaOutlook(email, message);
    } else {
      throw new Error('Unsupported email provider');
    }
  }

  private sendViaGmail(email: string, message: string) {
    // Gmail-specific logic here
    console.log(`Sending email via Gmail to ${email}: ${message}`);
  }

  private sendViaOutlook(email: string, message: string) {
    // Outlook-specific logic here
    console.log(`Sending email via Outlook to ${email}: ${message}`);
  }
}
```

Now, any other module that needs to send an email can do so without the need to know which provider will be used, or how many providers are available. This design allows developers to add any number of providers without the need to touch any other part of the code. It also allows any other module to send emails without needing to worry about which provider to use.

Finally, if the logic for choosing a provider changes, or if the parameters needed to call a provider change, the email service is the only one that needs to change. This separation of concerns is one of the key pillars of object-oriented Programming, from which patterns originally came, so the way to deal with them is to apply one of the **SOLID** principles, in this case, the **S** for single responsibility.

Whether it is a class, a function, or a method, each component should have a single purpose. In this case, the email service needs to know everything about emails, while the user service needs to know everything about the user; one thing cannot leak into the other.

Godlike Structures

Another example of coupling but in a way that most people would eventually overlook is the so-called **God Object**. This type of structure will eventually pop up in your code because, of all the anti-patterns, this is the one that sprouts most naturally from your minds. Think about that time when you were coding something, and the ideas just started to flow, you jotted them down in code and it worked! However, then you realize that all the code you wrote is in a single huge method. This mistake is why the God Object is dangerous, it is too easy to make this mistake.

A God Object is a single class, module, or structure in general that is responsible for too many things at once. You might initially think it's a good idea because it is fewer places to look for code, or because all the contents of a domain are encapsulated in a single place, but most of the time, this is not what is happening.

You can look at a simple example from a class. Imagine your system has a user manager and you can list the users, fetch the data from external APIs, and so on... You might think of doing a single `UserManager` class with everything included, right?

```
class UserManager {
 constructor() {
  this.users = [];
 }

 createUser(name) {
  const user = { id: Date.now(), name };
  this.users.push(user);
  return user;
 }

 fetchFromAPI() {
  // Also handles API requests
  return fetch('https://api.example.com/users');
 }

 saveToDatabase(user) {
  // Also directly interacts with the database
  database.insert(user);
 }
}
```

At first glance, everything looks convenient, right? You can do it all just by importing the `UserManager` class, but then you suddenly have to implement some changes in the code to allow the user to be inserted differently depending on the type of object it is, you would have to go through your code looking what types of user you have and how you are saving that. Or even worse, you need to test this code, how do you mock the database? Should you inject this?

The problem you have is that everything you do in your application will eventually touch this file because this file touches too many parts. This problem is a bad practice because not only you get constrained by customizing the code too much, you also might end up having conflicts with other teammates because everyone will also be touching this file at some point.

A better option would be to split this into multiple parts. Try this exercise: think about how you would split this up in a way that each part can be reused and does not need to know more than it should from other parts of the code. Then, continue reading this chapter.

If you thought about dividing the user into logical domains, this is probably the best course of action. Instead of treating the entire entity as one single big object, you can split the user structure into different parts. For example, this is one possible split:

- User becomes a single entity, and it is split from everything else
- Everything related to an external API, such as the fetch call, will go to another service
- Database calls go into a user repository that is directly responsible for dealing with data

So, you could split your code this way, starting with a new user entity:

```
class User {
    #data = {
        id: crypto.randomUUID(),
        name: null,
        age: null
    }

    constructor(props) {
        this.#data = { ...this.#data, ...props }
    }
}
```

In the preceding code, you are creating a new class that will represent your user, all the user methods should live in there, so any business logic or entity-specific information will be contained in this class and propagated to whatever place this entity is used. In the constructor, you are receiving a single object with all the properties that the user has, in this case, it is just

three, but it could be dozens, using objects will allow you to keep your constructor signature shorter, plus JavaScript works way better with objects than single parameters.

To avoid having to assign all the properties manually, you are putting everything into a private property called `#data`, this makes it easy for future typing with TypeScript, but you will miss getters and setters, which you can solve easily by which you can solve easily by setting those properties dynamically:

```
class User {
    #data = {
        id: crypto.randomUUID(),
        name: null,
        age: null
    }

    constructor(props) {
        this.#data = { ...this.#data, ...props }
        for (let key in this.#data) {
            Object.defineProperty(this, key, {
                get: () => this.#data[key],
                set: (value) => this.#data[key] = value
            })
        }
    }
}
```

This approach will allow you to call user`.name` or `user.id` when needed and set these properties automatically as well. Of course, you can also set the data property to public and access that data directly, then you will not be needing the dynamic creation.

Next up you can split the API and database parts into two other entities:

```
class APIService {
  fetchUsers() {
    return fetch('https://api.example.com/users');
  }
}

class UserRepository {
  save(user) {
    // Save to the database
```

```
    database.insert(user);
  }
}
```

In the repository, you can even go one step further and allow the user to pass on the database adapter they want to use, this way the repository is fully separated apart from the rest of the code. However, how does all this come together? In another entity, of course!

```
class UserService{
  constructor(userRepository) {
    this.userRepository = userRepository;
  }

  createUser(userData) {
    const user = new User(userData);
    this.userRepository.save(user);
    return user;
  }
}
```

This structure seems more complex, but this is the first lesson of patterns: more entities do not necessarily mean that the code is more complex. In fact, splitting the entities into multiple responsibilities like this will allow you to finely control which entity does what and write code only where it is needed, without the problem of having everything cramped up in a single place.

God objects are probably easy to spot, because they will start to feel uncomfortable, and they are possibly one of the most common ways to spot anti-patterns in code.

Hard To Read Code

When you talk about code there is always a difficulty in estimating what is good code versus what is bad code. In general, good code is the one that is easy to maintain and debug, but this can only happen if it is readable enough so others can also maintain it.

You must remember that 99% of the time, the person who will be looking at your code is not you, but other developers who must maintain it, the same way you have maintained several other pieces of code before. Readable code is essential for maintaining, debugging, and scaling any application. Readability matters because unreadable code often leads to wasted time, errors, and, ultimately, a reduction in productivity for you and others.

Unreadable code also creates what I call a **"code black hole,"** a place where the code is so dense that improvement is impossible, so nothing is removed, only added because other people are too afraid to break something that they eventually prefer to leave the application as it is rather than improving it.

The first tip is to avoid being "too clever" with the code. And by being "too clever" I mean that you know you're an amazing developer and you know all the quirks of the language, but the goal is to make the code readable for others as well, so instead of this:

```
const userName = user?.name || 'Unknown';
const userCity = user?.address?.city || 'Unknown City';
const userPostalCode = user?.address?.postalCode | 0
    ? user.address.postalCode
    : 'N/A';
const userScore = Math.max(user?.score ?? 0, 0);

console.log(
  `User ${userName} from ${userCity}, ` +
  `Postal Code: ${userPostalCode} has a score of ${userScore}`
);
```

This code is basically a bunch of logical expressions, they all get a property of the user object after several tests, these tests are not all widely known, for example, the ~ operator bitwise inverts the property, so when the `score` property is lower than 0, it will go back to 0 so the score cannot be negative. Some of those are not needed, such as the bitwise or operator `|`, and this can be greatly simplified to:

```
const { name, address, score } = user ?? {}
const userData = {
    name: name ?? 'Unknown',
    city: address?.city ?? 'Unknown city',
    postalCode: address?.postalCode ?? 'N/A',
    score: Math.max(user?.score ?? 0, 0)
}
console.log(
```

```
  `User ${userData.name} from ${userData.city}, ` +
  `Postal Code: ${userData.postalCode} has a score of ${userData.score}`
);
```

This makes the code much more readable but still preserves the main idea, it does not use APIs or operators who are not widely known and it is easier for other people to read and maintain it.

However, the problem is not the operators themselves but the difficulty of reading the code that uses them. Bitwise operators are sometimes needed for performance, and sometimes you cannot avoid less commonly used operators. To give you another perspective, you can have perfectly simple operators and still have a code that's hard to read:

```
const userScores = (await getUsers())
  .filter(({ isActive, age }) => isActive && age >= 18)
  .map(({ scores }) => scores.reduce((sum, score) => sum + score, 0))
  .sort((a, b) => b - a)
  .slice(0, 10)
  .reduce((acc, score, index) => ({ ...acc, [`Top_${index + 1}`]: score }), {});
```

In this case, you are just using normal array operators, but those operators nest. For example, you have a `reduce` within a `map`, all this makes the code more complex to read. Then, you need to think as you did in the previous exercise, sometimes splitting things up is the better approach and *not always more code means it is worse*. For example, the previous code can be split as follows:

```
const users = await getUsers();

// Filter to get active users aged 18 and above
const activeAdultUsers = users.filter(({ isActive, age }) => isActive && age >=
18);

// Map to calculate each user's total score

const userTotalScores = activeAdultUsers.map(({ scores }) =>
  scores.reduce((sum, score) => sum + score, 0)
);

// Sort scores in descending order
const sortedScores = userTotalScores.sort((a, b) => b - a);

// Get the top 10 scores
```

```
const topScores = sortedScores.slice(0, 10);

// Transform top scores into an object with labeled keys
const topScoresObject = topScores.reduce((acc, score, index) => {
  acc[`Top_${index + 1}`] = score;
  return acc;
}, {});
```

Way better to read, isn't it? This code leads you to another two things you need to understand about easily readable code:

- You are naming variables consistently across your namespace. All of them are descriptive and use the same `camelCase` pattern.
- You are commenting on the code saying what it does.

Naming is probably one of the most difficult things in computer science. Choosing names for variables, functions, and other structures can seem to be superfluous but it has a significant cognitive load for everyone who reads the code if you do not stick to conventions such as always using the same casing schema, and meaningful variable naming, the code will be way harder to read, and you will need some context switching. Try to follow these simple steps:

- Always use the same casing, if you started with camelCase, use it everywhere. The only exception to this rule is when you must perform external calls which require a different casing. In this case, you can either perform the conversion at the site or use the same casing as the external resource.
- Provide meaningful names to variables. Avoid things such as `processUser`, and `consumeData` because these names are simply too broad. Prefer more descriptive names even if they are longer. If your naming does not fit your method, this is a clear indication that your method is doing too many things.
- Avoid naming variables with single letters or abbreviations. You are past the time when memory size was an issue, besides that, variable name sizes will not have any considerable impact on the performance of your code, so do not be economical with them.
- After all these points, the biggest and the most problematic one is yet to come!

Not DRY

Don't Repeat Yourself (DRY) means *Do not Repeat Yourself*, and this is most likely the worst possible code smell in all the codebases. It is also the easiest way to spot an anti-pattern. The rule of thumb is:

if it's repeated, it is an anti-pattern.

Almost every time you have duplicate code, you will have an increase in the risk that this code will generate a bug, moreover, duplicated code generates a branch that you need to keep in mind when changing the original behavior, for example, if you have the same code duplicated in three different places and the original logic changes, you will then need to change this in three different places. This might be ok for the person who knows this is a duplication, but it's not ok for someone who does not know the codebase, you should always remember the rules you learned in the previous chapter: the code you write is not just your own, it is meant to everybody.

One of the most common places you find duplicated code in JavaScript is in validation functions and utility code, and this generally implies that code duplication will make your app run slower. For example, an Express application uses different validation middleware multiple times, leading to the same authorization validation being applied repeatedly, which reduces application performance because it computes the same result twice.

How to spot these in the code (besides the obvious look into repeated code)? DRY violations usually appear in a few forms:

- **Validations:** Usually, validations are done in place when it is the first time, but they tend to repeat across the code highly often, for example, checking whether the route body is correct can be done multiple times within the route handlers, but this functionality can be more useful if extracted to middlewares.
- **Patterns:** Regular expressions and other patterns used to check, extract, or validate a string also tend to appear in multiple places because different developers often need to validate the same pattern against different strings. For example, email and phone validations can appear multiple times in multiple places in the code and are often better extracted into a utility function or middleware.
- **Preprocessing:** Transforming an input before using it is especially common in applications that handle data from multiple sources, such as a payment method that accepts multiple credit-card issuers, each with a different payload. These preprocessing functions tend to have common steps between them, like removing useless fields, trimming whitespace and other applications, which will most likely be repeated in the code.
- **Sanitization:** A special case of the previous, sanitizations often occur when you are ingesting data that needs to be in a certain format, this is mostly common when you are dealing with external data that comes into the system, for example, an optical character recognition (OCR) reading that needs to have its data extracted but it needs to be stripped of all the whitespace and tab characters. These types of functions will

eventually be performed in different places, but they will often have different requirements, so it is important to know when to duplicate or not. There are few ways you can deal with duplicated code, all of them will eventually deduplicate the code, but this can be done either extracting the entire function to a single place, for example, look at this express route:

```
import Express from 'express'
import { z } from 'zod'

const app = Express()
const userSchema = z.object({
    name: z.string(),
    age: z.number().positive(),
    address: z.object({
        street: z.string(),
        number: z.string(),
        other: z.string().optional(),
        postcode: z.string()
    }).optional()
})

app.post('/users', async (req, res) => {
    const result = await userSchema.safeParseAsync(req.body)
    if (!result.success) {
        return res.status(422).json(result.error)
    }

    const addedUser = await ourUserService.addUser(result.data)
    return res.status(201).json(addedUser)
})
```

This route is a common route to add a user to an application. It is simplified, but the idea is that you will receive the request, get the body and validate it against a schema, then you will use that schema as your input data. Can you spot the problem in this code? No? What about now:

```
import Express from 'express'
import { z } from 'zod'

const app = Express()
```

```
const userSchema = z.object({
    name: z.string(),
    age: z.number().positive(),
    address: z.object({
        street: z.string(),
        number: z.string(),
        other: z.string().optional(),
        postcode: z.string()
    }).optional()
})

app.post('/users', async (req, res) => {
    const result = await userSchema.safeParseAsync(req.body)
    if (!result.success) {
        return res.status(422).json(result.error)
    }

    const addedUser = await ourUserService.addUser(result.data)
    return res.status(201).json(addedUser)
})

const accountSchema = z.object({
    userId: z.string().uuid(),
    name: z.string(),
    products: z.array(z.string().uuid())
})

app.post('/accounts', async (req, res) => {
    const result = await accountSchema.safeParseAsync(req.body)
    if (!result.success) {
        return res.status(422).json(result.error)
    }

    if (!(await ourUserService.findById(result.data.userId))) {
        return res.status(404).json({
            message: 'No user found'
        })
    }

    const addedAccount = await ourAccountService.addAccount(result.data)
```

```
    return res.status(201).json(addedAccount)
})
```

Again, this is a simplified version of a functioning route, but the problem is that every route that needs a body will have it validated against your schema validator, in this case it is Zod, so you need to take this repeated code out of the equation, otherwise, if you have to change your response code, for example, we will have to go through all our routes to make it happen and there's a high chance you will forget something.

The best option here is to extract the entire functionality to a middleware:

```
export const zodValidationMiddleware = (schema) => {
    return async (req, res, next) => {
        const validation = await schema.safeParseAsync(req.body)
        if (validation.success) {
            res.locals = {
                ...(res.locals ?? {}),
                validated: {
                    body: validation.data
                }
            }
            return next()
        }

        return res.status(422).json({
            code: 'VALIDATION_ERROR',
            message: validation.error
        })
    }
}
```

Then replace your routes with:

```
import { zodValidationMiddleware } from './middlewares/zodValidation.js'
import Express from 'express'
import { z } from 'zod'

const app = Express()
const userSchema = z.object({
    name: z.string(),
    age: z.number().positive(),
```

```
    address: z.object({
        street: z.string(),
        number: z.string(),
        other: z.string().optional(),
        postcode: z.string()
    }).optional()
})

app.post('/users', zodValidationMiddleware(userSchema), async (req, res) => {
    const { body } = res.locals.validated
    const addedUser = await ourUserService.addUser(body)
    return res.status(201).json(addedUser)
})

const accountSchema = z.object({
    userId: z.string().uuid(),
    name: z.string(),
    products: z.array(z.string().uuid())
})

app.post('/accounts', zodValidationMiddleware(accountSchema), async (req, res) =>
{
    const { body } = res.locals.validated
    if (!(await ourUserService.findById(body.userId))) {
        return res.status(404).json({
            message: 'No user found'
        })
    }

    const addedAccount = await ourAccountService.addAccount(body)
    return res.status(201).json(addedAccount)
})
```

As you can see, it's easier to maintain and easier to understand what is happening if you have the common functions in one single place. Another example is when you have different functions that contain one or more duplicated steps, in this case it is easier to extract the common code to another function but keep the original functions as they are. For example:

```
function processInputFile (file) {
    const sanitizedFile = file.replaceAll(/[\s,\t]/g, '').toLowerCase()
    return doAction(sanitizedFile)
```

```
}

function processAnotherInputFile (file) {
    const sanitizedFile = file.replaceAll(/[\s,\t]/g, '').toLowerCase()
    return doAnotherAction(sanitizedFile)
}
```

The same sanitization step is done in both functions; however, the rest of the function body is different, in this case it is better to extract the common step than trying to figure out a way to make both of them generic enough to receive the same input:

```
function trimSpaces (str) {
    return str.replaceAll(/[\s,\t]/g, '').toLowerCase()
}

function processInputFile (file) {
    return doAction(trimSpaces(file))
}

function processAnotherInputFile (file) {
    return doAnotherAction(trimSpaces(file))
}
```

This way the code is much more readable, and it keeps the function calls in place in all the references, so you do not have to keep looking for changes in the code.

Fixing DRY violations seems simple, but there's a catch! Is it always worth fixing **DRY** violations?

While **DRY** is a valuable core principle of software architecture, its application can sometimes lead to convoluted and complex code, where the functions you write become super generic and end up losing clarity. Remember that, the more generic, the more complex it is to read the code. This can be especially problematic in code meant to handle business logic that contains domain-specific knowledge, where too much abstraction will reduce the readability and render your utility functions almost impossible to maintain.

The main goal you must keep in mind is to reduce unnecessary repetition, but not compromise the code's clarity or intent. You have always to remember that this is a principle that *guides* you to something, it does not state you *must* do something.

The goal is to remove unnecessary duplication, but if this duplication sacrifices the readability of the code in a way that renders the code impossible to understand then there is no point in

fixing the duplication because the effort to maintain a code that is unreadable is far greater than the effort to maintain duplicated code.

Too many parameters

Using JavaScript has its perks, but you also need to understand how the language itself works, for example, one of the biggest complaints about JavaScript from developers is that its structures use a positional parameter system, which means that the parameters to any functions will be passed on as an array, and the orders of those parameters will be inferred from the position of this array, you can see an example:

```
function createUser (name, birthDate, email) {
    return {
        name,
        birthDate: birthDate.toISOString(),
        email
    }
}
```

This simple function receives three parameters and just returns them in a new object for the sake of simplicity. So, you would call it something like this:

```
createUser('Lucas', new Date(), 'hello@lsantos.dev')
```

Now, what happens if you shift the order of the parameters and put it like this:

```
function createUser (birthDate, name, email) {
    return {
        name,
        birthDate: birthDate.toISOString(),
        email
    }
}
```

The answer is that you will have an error because `birthDate and name are now reversed` and JavaScript will receive the name string instead of the Date object in it. This parameter-order change happens to all functions in JavaScript, essentially, writing the above is equivalent to writing:

```
function createUser (...args) {
    return {
```

```
        name: args[1],
        birthDate: args[0].toISOString(),
        email: args[2]
    }
}
```

This type of structure is fine for smaller functions, like the one you just saw, but when you have tens of parameters, it can be very prone to error. Consider the following example:

```
function createUser (name, birthDate, email, isActive, addressStreet,
addressNumber, addressTown, id) {
    // Do something
}
```

You can probably see where I'm going with this. When you need to call these types of functions, besides that the call will only contain the values, which are confusing, if you ever need to remove or change any parameters, you will also have to change all the other parameters because their order might change. To remedy that, you can use a parameter object:

```
function createUser (userDetails) {
  // Do Something
}

console.log(createUser({
  name: 'Lucas',
  birthDate: new Date()
}))
```

This way the parameters will be named, and you will be able to identify them all by their key, so it does not matter if the order changes, or if you remove any parameters because the object itself will be the source of truth. There is just one catch with this method, because JavaScript does not have types, figuring out what is inside the object is a hassle not just for you but for everyone using the application. One solution to this problem is to use TypeScript (which I heavily encourage), however, if you do not want to use it, you can take advantage of JSDoc (or TSDoc) comments:

```
/**
* @param {{ name: string, birthDate: Date, email: string}} userDetails The user's
information
* @returns {yourReturnType}
```

```
*/
function createUser (userDetails) {
  // Do Something
}
```

Many code editors, like Visual Studio Code, will automatically pick these up and show a nice description of the function and the object within it:

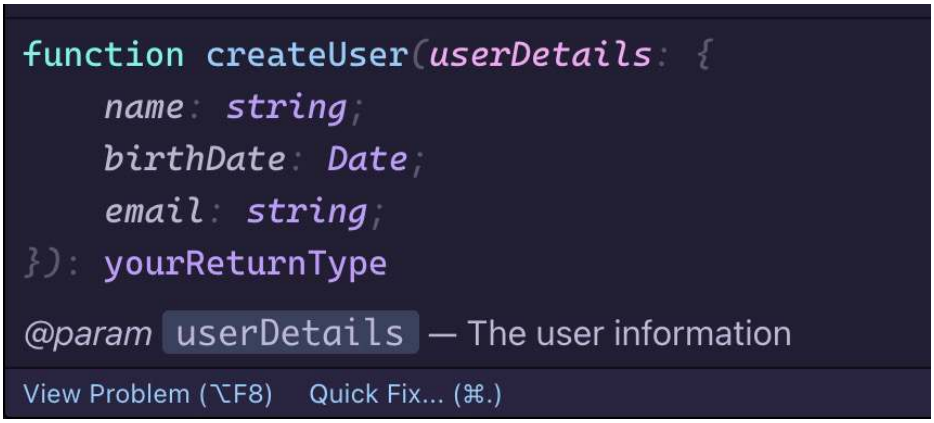


Figure 2.1: VSCode hint on the function

Adding these types of comments is not only a good code practice for this case, but it also allows you to communicate clearly to your teammates and any other developers in the future what this function is supposed to do, you don't need to add details to trivial functions, but parameters are always a good option and, in this particular case, a much-needed one.

You can also enforce the use of fewer parameters by applying ESLint rules such as max-params (`https://eslint.org/docs/latest/rules/max-params`) to make sure your functions will not go over a desired amount of params. A good base number is 4, but this is a personal taste more than a tip.

(Not-so) Smart functions

One particular anti-pattern that is often overlooked are the so-called "Smart Functions." These functions try to be smart by identifying the types of parameters being passed and behaving accordingly, this can also be called an "overload," for example:

```
import {readFile, readFileSync} from 'fs'

function syncOrAsyncRead (path, callback) {
  if (typeof callback === 'function') {
```

```
    return readFile(path, callback)
  }
  return readFileSync(path,'utf8')
}
```

This function behaves differently depending on the number and type of parameter you pass into it, if you give it a callback, the function will behave asynchronously, if not, it will behave synchronously. If you type this using TypeScript, this is what you would get:

```
import { readFile, readFileSync } from 'fs';

function syncOrAsyncRead(
    path: string,
    callback?: (
        err: NodeJS.ErrnoException | null,
        data: Buffer
    ) => void
): Buffer | void {
    if (typeof callback === 'function') {
        return readFile(path, callback);
    }

    return readFileSync(path);
}
```

You can see that the return value is unclear because the function can return either a buffer or nothing, so you need to check it at runtime. In TypeScript you could also create an overload:

```
function syncOrAsyncRead(path: string): Buffer;

function syncOrAsyncRead(
    path: string,
    callback: (err: Error | null, data: Buffer | null) => void
): void;

function syncOrAsyncRead(
    path: string,
    callback?: (err: Error | null, data: Buffer | null) => void
): Buffer | void {
    if (typeof callback === 'function') {
        readFile(path, callback);
```

```
        return;
    }

    return readFileSync(path);
}
```

This makes the understanding of the function a bit easier, but the fact is that many of JavaScript's functions and APIs follow this same pattern. In some cases, like this one, where the actual logic of the function does not change, just the way it is executed, this is fine, the problem happens when the logic inside the function changes based on the type of parameters passed in, like this example:

```
function smartFn (param1, param2) {
  if (!param2) {
    switch (typeof param1) {
      case 'string':
        return param1.replace('\n', '-')
      case 'number':
        return Math.round(param1)
      case 'object':
        if (!Array.isArray(param1)) {
          return JSON.stringify(Object.entries(param1))
        }
        return param1.map((item) => smartFn(item));
    }
  }
  return param2(param1)
}
```

This function is one complex function – I did overcomplicate it a bit for show too – but the main point is that there are so many branches in this function that it is nearly impossible to understand what would come out of it at any given time unless you look at the source code, which is a serious problem when you are dealing with large codebases or working in a team, because you are never sure of what the function does.

TypeScript does mitigate this issue a bit, however, unless you use a large number of overloads, it is very difficult to type these sorts of functions because their inner workings are not the same given different parameters, moreover, accepting a wide range of parameter types, making the

function "smart" seems interesting and nice on a first look but when you dig in, the function is not smart at all:

- It breaks the single responsibility principle by (probably) doing multiple things at a time
- From the previous, it creates multiple code branches that may or might not return the desired values, sometimes a function will not return a value at all
- It is very difficult (or impossible) to know what the function is doing unless you look at the source
- Having such complex functions add uncertainty to your code, and computation must be deterministic to be easy to deal with
- The key for removing and solving this anti-pattern is simple: avoid creating smart functions. The more deterministic and simpler a function is, the better it is maintained over time. As discussed earlier, there are some ways where smart functions can be useful:
- Running synchronously or async based on the callback parameter, even though, it is better to always default to asynchronous running when possible
- Receiving lists or individual parameters, for example, receiving a list of objects or a single object, this type of smart function could be useful because it is smart enough to identify that it should sanitize its own input, but not so smart that it should know what to do with its variations. Most of the times these functions will include a call to themselves, as `smartFn` did with the array before
- Following Design Patterns such as the "Strategy" or "Command" patterns which require the function to react to their parameters in some cases
- The rule of thumb is: keep it simple, avoid doing too much. *Being too smart is often not that good*

Magic values

The last anti-pattern from this session is something that everyone is prone to use at any given time: **magic values**. A magic value is a constant that is hardcoded into your code, it can be a string, a number, a Boolean, or any other primitive. They are generally found in early stage proofs of concept (POCs) or minimum viable products (MVPs) that were once supposed to be discarded but weren't or, in some more serious cases, in production deployments that were not well reviewed, for example:

```
function calculateDiscount(price) {
  return price * 0.9; // Magic number 0.9 (10% discount)
```

```
}

function connectToServer() {
  const url = 'http://example.com'; // Magic string
  return fetch(url);
}
```

In both of these functions you have "magic" primitives. Developers put them in those places usually because they think this value will not repeat itself, so it is fine to use it where it is because "it will never change." This is the greatest mistake of them all.

When you assume something won't change, you better be right, because if it does, you just unleashed an entire refactoring upon your code. For example, imagine that you calculate discounts differently for different categories of products, regions, countries and stores. Each calculation has its own formula and can be used in many places, but there is always a base discount that is applied to all of them. If you simply put that number in there and one day that value changes, things will get very messy because you will have to go through all the implementations and change that value.

Moreover, if there was not any comment in the function, like this:

```
function calculateDiscount(price) {
  return price * 0.9;
}
```

Would you be able to figure out this was a 10% discount? Writing code does not mean writing in code. Your code should be readable and understandable by everyone. To mitigate such problems is simple: *don't use magic constants*. Or, if you want to specify even better: *do not use magic constants more than once*.

These constants should always be moved to either a **constants object** or an **environment variable** depending on the case. Moving to constant objects have the advantage that the values will be well documented and always exist in runtime, reducing the chance of errors:

```
const APP_URLS = {
  CARD_TRANSACTIONS_API: 'http://example.com'
}
function connectToTransactionsAPI() {
  return fetch(APP_URLS.CARD_TRANSACTIONS_API);
}
```

On the flip side, values in *POJOs (Plain Old JavaScript Objects)* cannot be sensitive as they will be shown in plain text in the code, and they will also require a system restart to change, because they are compiled in runtime. To mitigate the first problem, you can use **environment variables (envs)**, which will allow you to only pass the value from the environment, so the actual primitive never appears in the code:

```
const crypto = require('crypto');

function decrypt(text) {
    const algorithm = 'aes-256-cbc';
    const key = Buffer.from(process.env.SECRET_KEY, 'hex');
    const iv = Buffer.from(text.slice(0, 32), 'hex');
    const encryptedText = Buffer.from(text.slice(32), 'hex');

    const decipher = crypto.createDecipheriv(algorithm, key, iv);
    let decrypted = decipher.update(encryptedText);
    decrypted = Buffer.concat([decrypted, decipher.final()]);

    return decrypted.toString();
}
```

In the preceding code you decrypt an AES-256-encrypted text with a key provided through the environment variable named `SECRET_KEY`, this is pretty straightforward when the variable name is exactly what you will be using, but how about when the name changes the meaning of the code? You can take your discount code from before, how would it look like if you used an env called `DISCOUNT_AMOUNT`? You cannot just set this variable to `0.9` because this is not the amount of discount, but the value after the discount, let's say you want to give a 10% discount, then the value this variable would have would be `0.1`, this requires some change in logic:

```
function calculateDiscount(price) {
  return price * (1 - Number(process.env.DISCOUNT_AMOUNT));
}
```

See how the function changed to adapt to the environment variable name. In this case, you would pass `0.1` instead of `0.9` so the name makes sense and, because all envs are strings, you need to cast it to a number. Environment management is covered in *Chapter 7*, but you can stick to this for now.

The downside of environment variables is that they belong to the environment, which means you need to reconfigure the entire application in each environment you run it in, be it locally,

staging, production, or any other. Plus, they also require a full restart of the application so the `process.env` object can be populated. You fixed one issue, but the other one is still open, how do you change running configurations without restarting the app?

The answer to this is that it is not possible to do so using only runtime memory such as envs or objects because once created, the memory cannot be replaced, unless you read from an external input, so the answer lies in I/O and reading external files. You could have a configuration file that is read every time a new env needed to be used, this would mitigate the problems completely because both the file would not be open to the public (and, even if it was, you could encrypt it), and the new values would be loaded onto the memory dynamically.

All seems good, but the problem you need to consider is that I/O is extremely slow when compared to a RAM memory readout, so it is not practical to replace all the environment calls for I/O reads. It is ok to do so if you seldom read an environment, or if the app is critical enough that you are willing to sacrifice performance for availability, but one thing is certain: **nothing will come for free**. This phrase will appear often during this book, in which you must remember the diamond of application choice:

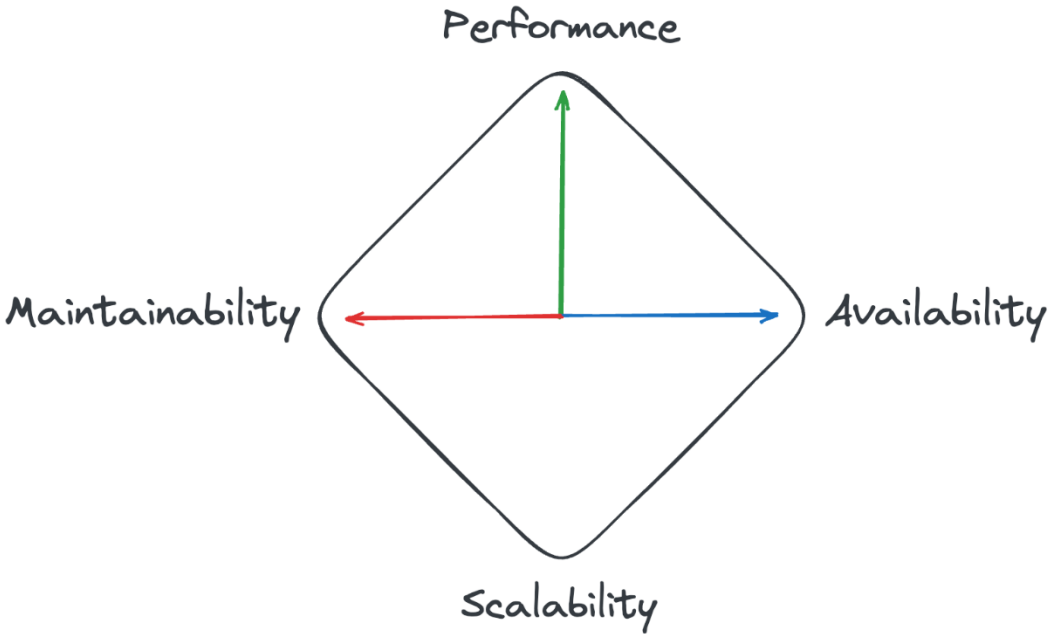


Figure 2.2: The diamond of choice

You have four main metrics in your application: performance, availability, scalability, and maintainability. You can never satisfy them all at once, you can choose three, but to do so, you

will have to sacrifice one. This principle is true for almost all patterns and anti-patterns out there.

Strategies for resolving Anti-patterns

Spotting anti-patterns is not an easy task, solving them is also something that will not come easy in most of the cases because each anti-pattern has a specific problem, and these problems need to be dealt with in different ways. However, there are some techniques that you can follow to make your code have fewer anti-patterns.

The code should speak for itself

As much as you will love to think that your code is clear, sometimes this is not what you get. Over time, code tends to get more and more complex, so it is imperative that you avoid at all costs creating code that is difficult to read from the start.

This will sound weird, but sometimes writing less is not optimal because you will eventually rely on techniques that will reduce the number of characters onscreen but will greatly complicate the reading of the code, this is especially true in fluent interfaces such as map, filter, and reduce where you can chain the result of one function into another.

The key takeaway is to make the code clear enough to read like a story. You should read it as you would ordinary prose; this can be achieved by:

- Naming your variables, parameters, and functions with meaningful names
- Reducing the complexity of chained calls
- Reducing the number of nesting levels in functions, or enforcing it with an ESLint rule such as max-depth (`https://eslint.org/docs/latest/rules/max-depth`)
- Avoiding the use of recursive functions
- Using named parameters and/or objects whenever possible

These bullet points alone will prevent multiple anti-patterns from even appearing in your code. If you still cannot read, or if the code is inherently complex, for example, a collision detection algorithm, a physics engine, and so on. Then you might want to leverage the comments to explain what you are doing, do not be afraid of them, comments are there to make sure the hardest parts of the code are understood by everyone, if you think something will be missing or some context will be lost, comment it step by step

Ask Simpler Questions

Still on the topic of being too complex, another problem that often comes up in code is that you ask complicated questions, for example:

```
if (
  !user.status.includes('expired') &&
  user.active &&
  ['locked', 'disabled'].includes(account.status) &&
  !account.banned &&
  new Date(user.lastActive).getTime() < new
Date(new Date().setMonth(new Date().getMonth() - 3)).getTime()
) {
  // do something
}
```

Do you see how you are asking too many questions in one go? When you do this, your conditionals become too complex to understand and will often be overlooked. You can replace this all by check variables:

```
const threeMonthsAgo = new Date(new Date().setMonth(new Date().getMonth() - 3))
const isUserActive = !user.status.includes('expired') && user.active;
const isAccountRestricted = ['locked', 'disabled'].includes(account.status) && !
account.banned;
const isLastActiveOverThreeMonths = new Date(user.lastActive).getTime() <
threeMonthsAgo.getTime();

if (isUserActive && isAccountRestricted && isLastActiveOverThreeMonths) {
  // do something
}
```

See how the checks are now split in bits that are easy to read? You can even simplify it further by giving meaning to a `today` variable:

```
const today = new Date()
const threeMonthsAgo = new Date(new Date().setMonth(today.getMonth() - 3))
const isUserActive = !user.status.includes('expired') && user.active;
const isAccountRestricted = ['locked', 'disabled'].includes(account.status) && !
account.banned;
const isLastActiveOverThreeMonths = new Date(user.lastActive).getTime() <
threeMonthsAgo.getTime();
```

```
if (isUserActive && isAccountRestricted && isLastActiveOverThreeMonths) {
  // do something
}
```

Using this new variable increases the code size, but makes it easier to read, and the whole question is now clearer and can be easily maintainable.

Make it Shorter

Finally, a quick but useful tip: **keep your functions short.**

Sometimes you want to include as much information in one place as you can to avoid having to make multiple calls to other functions, however this branches from the previous tip, when you are doing too much in one single place, your questions get too complicated, in this case your whole speech gets too complex.

Set your editor to warn you on lines that are longer than 120 characters and functions or methods that go over 20 or 25 lines (discounting the comments). Ideally, what you are trying to do is to keep the entire code in one viewport, the code of a function should be entirely visible within your editor view with no scrolls (vertical or horizontal), having to scroll while reading a function is a cognitive change that will make you have a hard time understanding it. In order to do this, you can use the following ESLint rules:

- max-len: `https://eslint.org/docs/latest/rules/max-len`
- complexity (complexity limits the number of branches in a function, which tends to keep functions small): `https://eslint.org/docs/latest/rules/complexity`

This rule helps be silly, but keeping tabs on the sizes of your functions will greatly help you understand when your code is becoming too difficult to read or too long to understand.

Summary

This chapter was a lot! However, you reached the end of this chapter! You saw some of the most common anti-patterns in Node.js, how to spot and identify these patterns and code smells in the code, and finally some tips on how to fix them. In general, spotting anti-patterns boils down to code that is becoming increasingly hard to read, while solving them generally involves making them simpler and splitting the problem into several parts!

You first saw how to identify code problems in your application, initially with high coupling between different modules inside the same application, this can usually be resolved by extracting the business logic from one module into another in a way you can reuse those modules throughout the application.

However, you need to be aware not to put too many things in one place to avoid the creation of the God Objects, these objects do too many things at once, touch several parts of the application, and are generally required by many other modules. This can also be solved by splitting a single object in multiple ones, but not only from the logic point of view, but also understanding what they mean to your code, for example, do not try to include an API call and a database operation in the same place, they belong in different domains.

Finally, you understood that code that is hard to read is often the cause of many anti-patterns, as well as repeating yourself. These patterns are the simplest ones to get away because the solution lies in simplification!

After this, you looked at some common anti-patterns around functions.

Some other common anti-patterns revolve around functions, namely them having too many parameters and being too smart. Functions should be lean and simple, if you have too many parameters, group them all in an object, and if your function tries to assume too much from the user's will, do not! Functions should be told what to do, with minor intelligence, ideally none, this makes them easier to debug, more predictable and easier to test!

Finally, avoid the use of magic values, those values that appear once in code and have no apparent meaning. Make all your constants mean something in a variable, remember that your code is being written so other humans can also understand it!

Now, let's learn some more ways to stay away from smelly code!

3

Staying Away from Smelly Code

Now that you are familiar with identifying and dealing with anti-patterns, it is time to take a closer look at code smells – the subtle, sometimes overlooked symptoms that can signal trouble within your codebase. While they might not break your code outright, code smells can lead to serious issues if left unchecked, slowly eroding performance, readability, and maintainability.

Some of these code smells can be avoided just by having the knowledge of how the language works internally. In this chapter, we will break down:

- What exactly makes code *smelly* and why it is worth understanding these characteristics
- Common JavaScript code smells that sneak up in everyday coding, and how to handle them without creating more complexity
- How to track JavaScript's evolving specs to keep your code future-proof while mastering best practices for type conversion to prevent sneaky bugs caused by JavaScript's loose typing
- Together, we will dive into practical techniques to keep your code clean, organized, and resilient. So, get started on the journey to staying away from smelly code!

Note

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Understanding Code Smells

If you have been working in the field for some time, you have probably heard of code smells at some point. At first sight, they look like innocent implementation details and can even go unnoticed by less experienced eyes. They usually consist of simple things, such as methods with names that are too long, classes that change too often, and functions that take too many params, some of the things that we talked about in the last chapter.

In essence, code smells are problems that are small enough to not break your code, but big enough to make experienced developers twist their nose. Often, code smells are not even the biggest problem but are indicative of some underlying misconception or bad design, and those tend to grow and multiply, making a nightmare out of the poor lives of maintainers.

Now, in the long run, what tends to happen to code smells is that they grow more noticeable and smellier, starting to draw attention and be a bigger and bigger pain, just like ignoring a small leak in the roof. By the time the team realizes action is needed, it is often too late, resulting in a hefty technical debt spread across the codebase. Untangling these issues becomes a costly, morale-draining process that eats away at productivity.

As a quick reminder from what you have seen in the last chapter, you can look at a function that creates a user in the database. The function takes in all the user data, but each piece of data is a different parameter. Because every user is created as a guest by default, the last parameter, which is the role, is optional:

```
export function createUser(name, email, password,
  phoneNumber, role = 'guest') {
  return db.users.create({
    id: crypto.randomUUID(),
    name,
    email,
    password: hashPassword(password),
    phoneNumber,
    role
  })
}
```

Passing each piece of user data as a separate parameter might seem harmless initially, but it quickly becomes a headache as parameters change or grow. For instance, imagine you need to add a new property, the user's home address, to their account. If you add a new parameter as the last one, you need to make it optional, because `role` is already optional, and a required parameter cannot come after an optional one.

Note

A quick note about the code

These examples use `crypto.randomUUID()` that is a Node.js constructor, so this code will only execute in Node.js environments.

To make the address required, you would have to put it before the role parameter, which means the function signature changes, and so every place where it's called with a `role` as the last parameter must be changed too.

```
export function createUser(name, email, password,
    phoneNumber, address, role = 'guest') {
  return db.users.create({
    id: crypto.randomUUID(),
    name,
    email,
    password: hashPassword(password),
    phoneNumber,
    address,
    role
  })
}
```

At first glance, passing user data as function parameters might not seem like a big problem. In the long run, however, it can cause significant maintenance problems. That long-term cost is exactly what makes the pattern a code smell. A cleaner, more maintainable approach would be to refactor the function to take in an object, like so:

```
export function createUser({
  name,
  email,
  password,
  phoneNumber,
  role = 'guest'
```

```
}) {
  return db.users.create({
    id: crypto.randomUUID(),
    name,
    email,
    password: hashPassword(password),
    phoneNumber,
    role
  })
}
```

By shifting to an object parameter, you make the function more adaptable to changes, reduce the likelihood of errors, and keep your code clean, ultimately avoiding that *smelly* pitfall. This way, if you need to add more information to the user account, you can just add a new property to that object. This not only simplifies how you call the function but also removes the need to worry about parameter order, making your code cleaner and easier to maintain.

Knowing how to spot code smells quickly is key to preventing smelly problems. In the next section, you will dive into some of the most common code smells in JavaScript, learning how to spot and tackle them before they can grow into costly problems.

Common JavaScript Code Smells

To avoid repetition, you are going to look at another list of common JavaScript code smells. Keep in mind that code smells are different from misuse of design patterns, for example, when you have a temporary field in a class, or classes that do not do enough work to justify being a class. Things like that are bad pattern usage, and they play a huge role in making your code unreadable and unmaintainable.

You need to remember that code smells are smaller, often overlooked, problems with the code that usually will not play a huge role in making the code bad right away but, as they pile up over time, they start to become a pain, and they pile up fast because they are easy to miss.

JavaScript is notoriously known for handling types poorly. There are multiple images and memes mocking this behavior:

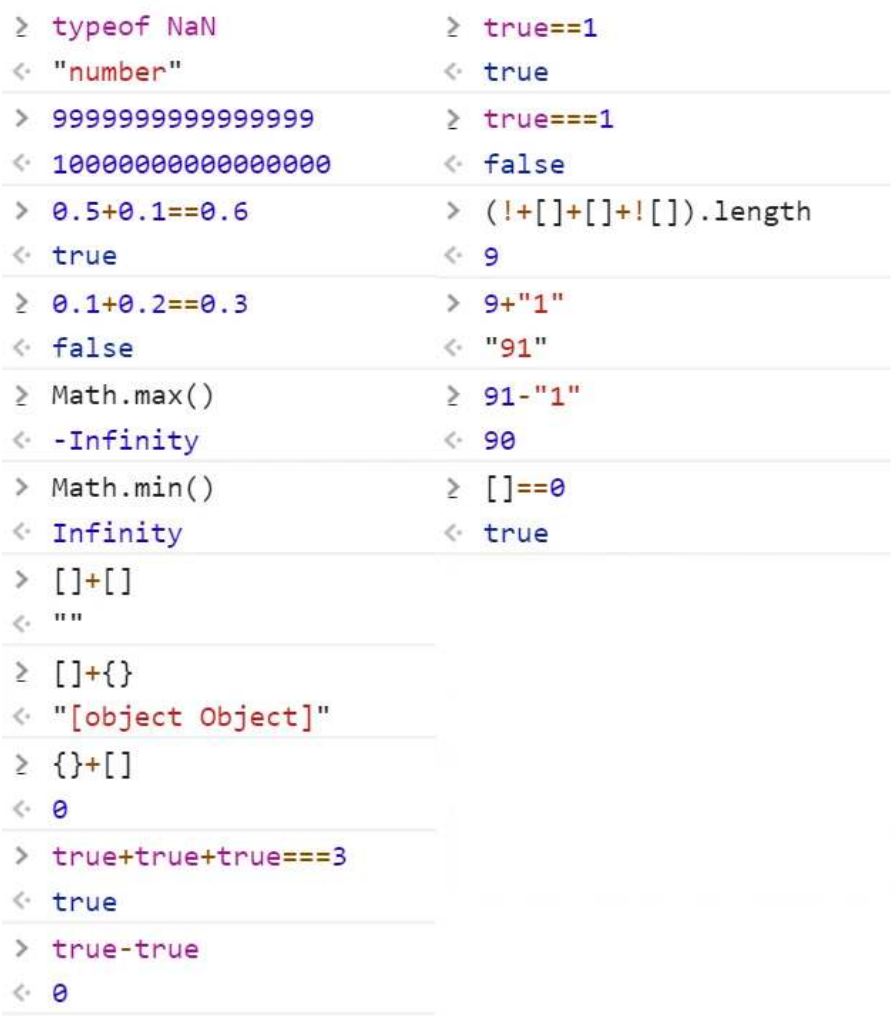


Figure 3.1: One of the multiple memes about JavaScript types

This is precisely the reason why type systems, such as TypeScript and flow were originally created, because the way JavaScript handled types in the past was too permissive, and it tried to be too smart, something we already talked about in the past.

While most (if not all) of these problems are now corrected (and we will not be looking at why these happen because it is outside the scope of this book), the spec needs to be compliant for previous versions as well (remember: **Do not break the web**). This is why most of the old, faulty implementations are still around, and they are the biggest source of code smells of all. Let's look at some of them.

Implicit type coercion

The root of the "*type problem*" is that JavaScript automatically converts values from one data type to another because it is not a strongly typed language, and all languages of this kind (PHP [PHP: Hypertext Preprocessor] and Perl are other examples) need to offer this type of feature otherwise coding in this language becomes a nightmare.

Type coercion is probably the most common dangling error that exists in JavaScript 'till this day. It's usually characterized by the wrong use of == over ===, but there are other places this also happens, for example, on arithmetic operations.

Strings and Numbers

Sometimes this automatic casting becomes a bit too smart, for example, what would be the output of this code:

```
console.log('5' + 3);
```

If you answered `"53"`, then you're right! What happens in this case is that the + sign is both a string concatenation as well as an arithmetic operator and, by default, when JavaScript is parsing the document, it will look on both sides of the operator. If any of the sides is a string, it will assume that the operator is a concatenation operator, so when it finds the string `'5'` it assumes that the operator you are using is to concatenate the next bit of string, so it will try to convert anything after or before the + sign to a string.

However, the same does not happen with the – operator just because it is not ambiguous and there is no other types besides numbers that use that operator, so if you write:

```
console.log('5' - 3);
```

The result is 2 because, when the engine parses the code, it finds the minus operator and treats the whole operation as a numeric one; no other types can use that operator. You can, in fact, write the same code using two strings and they will both be converted to numbers:

```
console.log('5' - '3');
```

The same applies to other numeric-only operators, such as * and / too. The key to avoid these types of problems is to make sure that numbers and strings are converted separately, and all the mathematical operations only receive numbers in any case.

Let's get to know your conversions.

Using numbers the right way

Most of the type conversion problems happen when you are trying to convert strings into numbers. There are well-known methods that will do that, for example, `parseInt` but this has deeper implications that you can see at first glance. And it can produce unexpected results.

For example, what happens when you try to convert `0.5` to an integer?

```
console.log(parseInt(0.5)) // 0
```

The output will be the expected, right? You will get a `0` back, because `0.5` is not an integer number. However, now try to convert `0.0000005`:

```
console.log(parseInt(0.0000005)) // 5
```

The resulting value should be `0`, but it's not, it is `5`... However, if you wrap the number in quotes, transforming it into a string, it now returns `0`!

```
console.log(parseInt('0.0000005')) // now it's 0
```

However, you might argue that this is meant to parse integers, right? You can change the number. If you use the number `1.000.000.000.000.000.000.000` (1 followed by 21 zeroes) you will have:

```
console.log(parseInt(1000000000000000000000)) // 1
```

To understand why it happens, and how this single problem can become a huge headache you need to first understand how JavaScript handles numbers, or better, how the `parseInt` function handles them.

If the argument is not a string, `parseInt` will first try to convert it into one. The function follows a series of rules (outlined in the Mozilla Developer Network documentation `https://lsantos.dev/nodebook-string`):

- If it's a string, no conversion is performed
- Undefined will return as a string `"undefined"`
- Null returns as a string `"null"`
- Booleans turn into their string representation, `"true"` , and `"false"`

- Numbers and BigInts are converted by calling the number's `.toString(10)` method (same as doing `2..toString(10)`)
- Symbols return a type error
- Objects follow the rule that we will discuss in the next section
- And the `toString(10)` function works in a simple way, it exists in all objects, but it's overridden in the number object, and it returns the string representation of that specific value following a specific radix (which you also specify as the second argument of `parseInt`), in this case, `10`

Note

What is a radix?

Radixes are the base on which a value will be converted, by default you use the decimal base, so the radix is 10. The decimal system means you can count from 0 to 9, then you repeat the first number to increase the count (10 would be 1 decimal unit + 0 unitary units). There are, however, several other bases, such as 2 that is binary, so you can only count using 0 and 1, and 16 that is hexadecimal, this means you can count from 0 to 9, then from 10 to 15 you use the letters A to F, for 16 you repeat the first number, so 16 in hexadecimal would be 10 (1 hexadecimal unit + 0 unitary units).

The string conversion is very straightforward:

- `0`, `-0`, and `Infinity` will return their string representations (just quote them)
- If the number is not an integer, it will be converted into the number's string representation. This will continue until the number becomes smaller than 10^{-6}. Once this happens, the number will be represented as a scientific notation. In this format, the integer parts will be separated by an e followed by their magnitude. For example:

```
0.5 remains as "0.5"0.0000005 is converted to "5e-7"
```

- If the number is a whole number with a magnitude less than 10^{21}, then it will be converted as is.
- If it's over that magnitude, the same rule for scientific notation will be applied.
- And this algorithm is what makes things complicated. Did you notice some familiar numbers? Like 21, or 7? If you guessed that's the reason why these examples don't work, you're right. When you pass `0.0000005` not as a string (because then it won't be

converted), but as a number, the function tries to convert it. The number becomes `5e-7` and `parseInt` will try to find the first non-number value in the string. Once it does, it will return all numbers up to that point, which means it will stop at `e` and return `5`.

- The same happens with the whole number, but since the precision you can represent whole numbers is far greater than what you can do with floating-point numbers, it means any number of 10^21 or more (22 digits or more) will be written in scientific notation, and your huge number will be `1e+21`.
  This situation gets worse because, if you change the radix, it can understand some other characters, such as the e. For example, if you do:

```
console.log(parseInt(5e-7, 16)) // 94
```

- The result is 94, because, on base 16, the letter `"e"` is 14 (`5 * 16 + 14 = 80 + 14 = 94`). If you use base 36, all letters can be used and everything you pass in becomes a number.
- But... How is this a problem?
- Initially, you might think this cannot bring any harm. The problem is the same as the smart functions discussed in the last chapter. The function tries to be smarter and understand what you want but, in certain conditions, it doesn't behave as you expect. This means you will only catch an edge case when it happens, usually in production, and with a customer. To make matters worse, `parseFloat` does not have this problem because it understands scientific notation.
- The solution? Standardize it using the `Number` constructor:

```
console.log(Number(5e-7)) // 5e-7
console.log(Number(10000000000000000000)) // 1e+21
```

- The main difference here is that the constructor will parse the entire converted string first, before trying to find a number, and it's the same for integers and floats, it is easier to use, easier to read, and it is also a newer part of the spec.
- Along with all that, there's a performance take. `parseInt` can be around 30% to 80% slower than the constructor approach in large arrays exactly because of all these maneuvers they try to do.



Figure 3.2: Performance test results (source: https://lsantos.dev/nodebook-perf)

- The only good use case for `parseInt` is when you want to extract the number part out of a string, or roughly the integer part of a small float (up to 6 decimal places). In these cases, since it does a single pass in the string, you can gain up to 50% performance in the operations, but it's not the use it's meant for, which can become difficult to maintain unless you leave a comment to explain why you decided to use it.
- As a rule of thumb: **avoid using** `parseInt` **and** `parseFloat` **when possible, use Number**.

We talked about integers and floats; now we can jump into a whole another a problem: **Booleans**!

Booleans and Objects

Every computer language, no matter how small it is, will have Booleans. It's possibly the most primitive type that you can have in code, with some languages even working directly on all-Boolean operations, with no other types at all (see **Bitwise Cyclic Tag**, or **Bitwise Cyclic Tag**: `https://lsantos.dev/bct`)

In JavaScript, Boolean is handled quite well, or, at least, in some predictable way given that you know the rules it follows to parse Boolean structures, and these can be complicated to

understand in detail especially when you look at when to use == or === in comparisons, for example:

```
console.log(false == 0);         // true
console.log(false === 0);        // false
console.log('' == false);        // true
console.log('' === false);       // false
console.log(null == undefined);  // true
console.log(null === undefined); // false
```

Luckily, you are here to study why these things happen in runtime, so you do not commit the errors that cause these weird behaviors. However, as an initial rule of thumb, remember: **Never use loose comparisons (==), only strict comparisons (===).**

With that said, JavaScript has two concepts: **truthy** and **falsy** values. Truthy values are the ones that, when type-cast into Booleans, will represent the logic `true`. Falsy values are the opposite and will represent the logic `false`. To understand which of these values can be one or the other, you can build a falsy table. Everything in this table will be converted into `false`, all the rest is truthy:

Falsy Values	Description
`false`	Boolean false
`0`	Number zero
`-0`	Negative zero
`0n`	BigInt zero
`""`	Empty string
`null`	Null value
`undefined`	Undefined value
`NaN`	Not-a-Number

Table 3.1: Falsy values in JavaScript (JS)

You can test each of these by typing `Boolean(<the falsy value>)` in any Node.js read–eval–print loop (REPL) or JavaScript/TypeScript file of your choice. All other values, including any

type of objects, non-zero number, non-empty string, any arrays, and functions, are considered truthy, which is where things get very interesting. Let's look at our first code example from before:

```
console.log(false == 0);         // true
console.log(false === 0);        // false
console.log('' == false);        // true
console.log('' === false);       // false
console.log(null == undefined);  // true
console.log(null === undefined); // false
```

Can you see some inconsistencies? For example, why is `false == 0` true while `false === 0` is not? This comparison has to do with the == and === operators. When you use `==` with a Boolean, JavaScript first converts the Boolean to a number. So, in this case, false becomes `0`, and the operation `0 == 0` is executed; since both values are equal, the result is `true`.

However, when you use ===, you're telling JavaScript to match both **the value and the type of both sides of the operator**, this is essentially doing two operations:

- `0 == false` that results in `true`, as shown earlier, you can call this **1***
- `typeof 0 == typeof false` that results in `false`, because `typeof 0` is `'number'` and `typeof false` is `'boolean'` you can name this one **2***

If you perform `(1*) && (2*)` you will get `true && false` that is `false`. By knowing this rule alone you will avoid most of the problems with wrong type conversions, but you can dig deeper on this topic and look at how arrays and objects behave. Imagine this code:

```
console.log([] == false); // true
console.log([] === false); // false
console.log({} == false); // false
console.log({} === false); // false
```

All objects, independent of how they're composed, are truthy. All arrays (including empty arrays) are objects, which means that all arrays are also truthy, as well as functions. But something does not make sense here, how `[] == false` is `true`? If you convert `[]` to Boolean, should not you expect it to return `true`? That's when the knowledge of how the internals of the JavaScript specification is useful.

The MDN website lists a very thorough description on how the primitive coercion works on these pages:

- `https://lsantos.dev/nodebook-equality`
- `https://lsantos.dev/nodebook-primitives`

What happens is that, when comparing values with ==, JavaScript will try to make both types the same, it does that by calling three well-known methods in order:

1. `<someObject>.[Symbol.toPrimitive]('default')` with the 'default' hint, which expresses no preference for coercion. This function, if it exists, must return a primitive value. This is how you can override JavaScript's comparison engine
2. `<someObject>.valueOf()` and it must return a primitive value to be considered for comparison
3. `<someObject>.toString()` will always return the string representation of that object

So, the breakdown of that operation looks like this:

1. The engine will try to call `[][Symbol.toPrimitive]()` and this returns undefined because arrays have no such Symbol
2. Then, the engine will try to call `[].valueOf()`, which returns the object itself, which means it's not a primitive value, so it will be discarded
3. Last, the engine will call `[].toString()`, which returns an empty string
4. The comparison now is `"" == false`, since one of the operands is a Boolean, it will be converted to a number, this results in `"" == 0`
5. All strings are converted to numbers, since the string from *step 3* is empty, the result is also `0`, so now the comparison is `0 == 0`
6. Both operands are the same and equal, the resulting value is `true`

The process is done for every comparison in the list when the operator is ==. The === operator never converts its operands: if the types differ, it returns false immediately. This explanation alone should be enough to convince you not to use loose comparisons ever, but there's another quirk to it, **you can change the way JavaScript compares objects loosely**.

Which takes you to yet another common pitfall you should avoid.

Prototype Pollution

If you override the `Symbol.toPrimitive` function, you can return any primitive value from it. So, you can make your array always return true. First, you declare the array and do a sanity check to make sure all the comparisons are working the same:

```
const arr1 = []
const arr2 = []
console.log(arr1 == false) // true
console.log(arr2 == false) // true
```

Now, you can change your primitive conversion function to always return `1`, this means every time you compare this array, you will always get true in return:

```
arr1[Symbol.toPrimitive] = function () { return 1 }
```

Now, you can compare them again:

```
console.log(arr1 == false) // false
console.log(arr2 == false) // true
```

Overriding a built-in conversion on a single object like this is a bad idea, and while this is a niche example that you should never do, it is still an interesting way to show that comparisons should never be messed with; otherwise you will lose control of your code. You can make it even worse by changing the logic for all the arrays in JavaScript, which is called prototype pollution:

```
Array.prototype[Symbol.toPrimitive] = function () { return 1 }
```

The key takeaway here is to **never change the default behavior of the language**, be it through prototypes or any other way, if you must change a behavior for a whole class of objects, do not do it globally by overriding a prototype, but locally in your object.

However, there is one other pitfall you must study, falses can be trues...

False can be true

When you create a Boolean with the new keyword (or use any other wrapper constructor, such as new Number or new String), you will be creating an object:

```
const falseObj = new Boolean(false);
```

This is not a Boolean false, this is an object that represents the logical false. Since all objects will return true due to the reasons you already saw in the previous code, if you compare it by converting to a Boolean, you will always get a true:

```
if (falseObj) {
  console.log('This will execute.');
}
```

This truthy result happens because if (`falseObj`) converts its condition with ToBoolean, and ToBoolean returns true for every object. It does not call `toPrimitive`, `valueOf`, or `toString` at all, so the false value wrapped inside the object is never looked at.

You can see the difference by typing this code:

```
console.log(new Boolean(false)) // [Boolean: false]
console.log(false) // false
```

As a final piece of advice, remember this: **never create objects using primitive constructors, they will most likely not behave as you are expecting them to**. Always use the default function constructor, for example:

```
console.log(Boolean(false)) // false
```

All the above are considered as "hidden features of the language," but it's not like that... The behaviors you saw above (including the weird ones) are well documented in the JavaScript specification document, which is the next good practice you need to keep in mind.

Keeping track of the Spec

Keeping track of the specification's proposals and changes is essential for understanding the current pitfalls and what the community is doing to fix them.

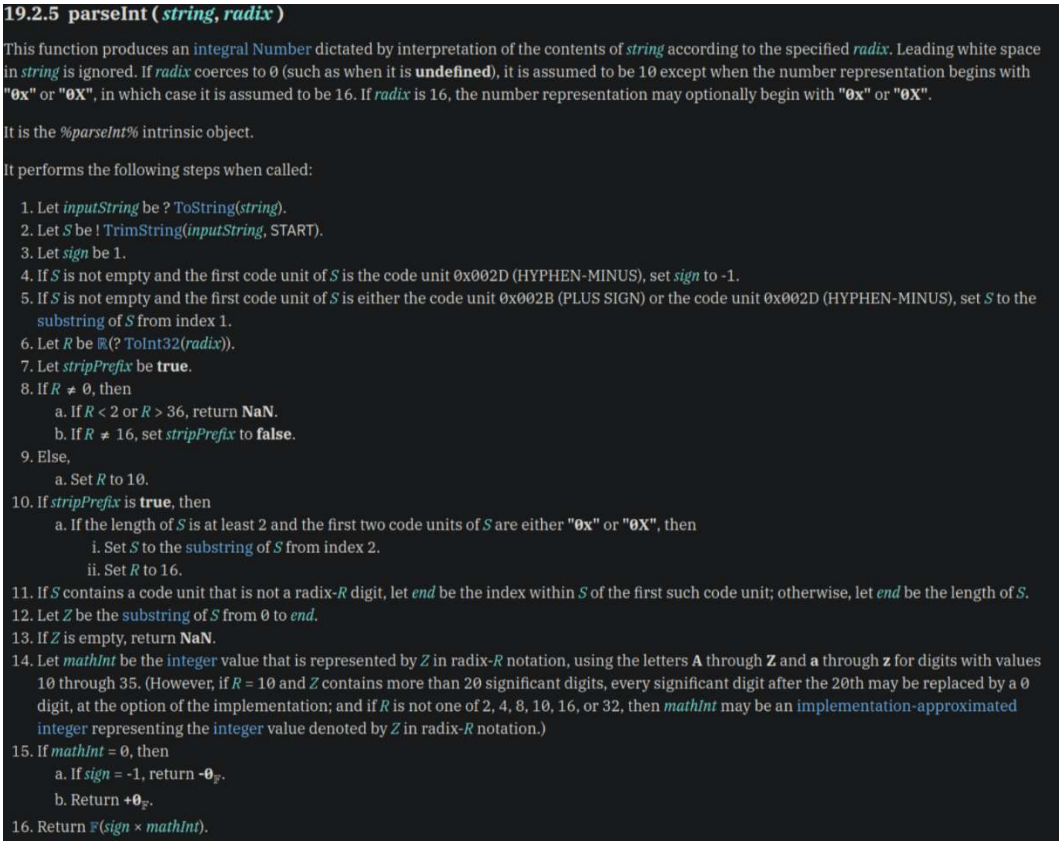
JavaScript is a constantly evolving language shaped by the collaborative efforts of **TC39** (short for **Technical Committee 39**) and governed by the **European computer manufacturers association (ECMA)-262** specification. Staying up to date with new language features and proposals is crucial for writing modern, efficient, and forward-compatible code.

TC39 is the committee responsible for evolving JavaScript, comprising representatives from major tech companies and individual contributors who propose, discuss, and refine new features for the language. The ECMA-262 specification serves as the official document detailing every aspect of JavaScript, including syntax, semantics, and APIs, and it's updated regularly to reflect changes.

To stay informed, you can explore TC39's staged process for new features, which moves proposals from *stage 0* (*ideas*) to *stage 4* (*finalized and included in the specification*). The TC39 GitHub repository at `https://github.com/tc39` is a central hub where you can track these proposals, read discussions, and follow updates. Pay special attention to the proposal's repository at `https://github.com/tc39/proposals`.

The most recent version of the ECMA-262 specification is available at `https://tc39.es/ecma262`, providing the definitive reference for JavaScript's current capabilities and how it's internals work. If you're interested in the rationale behind changes, the **GitHub issues** section on the TC39 repository is a great place to observe and, if you want to, participate in discussions about the language's development. You can even suggest proposals of your own.

As an example, you can look up the specification of the `parseInt` function you saw before, which is described in section *19.2.5* of the *2025 ECMAScript* specification (be aware of the year, since the sections can change over time):



19.2.5 parseInt (*string*, *radix*)

This function produces an integral Number dictated by interpretation of the contents of *string* according to the specified *radix*. Leading white space in *string* is ignored. If *radix* coerces to 0 (such as when it is **undefined**), it is assumed to be 10 except when the number representation begins with **"0x"** or **"0X"**, in which case it is assumed to be 16. If *radix* is 16, the number representation may optionally begin with **"0x"** or **"0X"**.

It is the *%parseInt%* intrinsic object.

It performs the following steps when called:

1. Let *inputString* be ? ToString(*string*).
2. Let *S* be ! TrimString(*inputString*, START).
3. Let *sign* be 1.
4. If *S* is not empty and the first code unit of *S* is the code unit 0x002D (HYPHEN-MINUS), set *sign* to -1.
5. If *S* is not empty and the first code unit of *S* is either the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS), set *S* to the substring of *S* from index 1.
6. Let *R* be ℝ(? ToInt32(*radix*)).
7. Let *stripPrefix* be **true**.
8. If *R* ≠ 0, then
    a. If *R* < 2 or *R* > 36, return **NaN**.
    b. If *R* ≠ 16, set *stripPrefix* to **false**.
9. Else,
    a. Set *R* to 10.
10. If *stripPrefix* is **true**, then
    a. If the length of *S* is at least 2 and the first two code units of *S* are either **"0x"** or **"0X"**, then
        i. Set *S* to the substring of *S* from index 2.
        ii. Set *R* to 16.
11. If *S* contains a code unit that is not a radix-*R* digit, let *end* be the index within *S* of the first such code unit; otherwise, let *end* be the length of *S*.
12. Let *Z* be the substring of *S* from 0 to *end*.
13. If *Z* is empty, return **NaN**.
14. Let *mathInt* be the integer value that is represented by *Z* in radix-*R* notation, using the letters **A** through **Z** and **a** through **z** for digits with values 10 through 35. (However, if *R* = 10 and *Z* contains more than 20 significant digits, every significant digit after the 20th may be replaced by a 0 digit, at the option of the implementation; and if *R* is not one of 2, 4, 8, 10, 16, or 32, then *mathInt* may be an implementation-approximated integer representing the integer value denoted by *Z* in radix-*R* notation.)
15. If *mathInt* = 0, then
    a. If *sign* = -1, return **-0**$_{\mathbb{F}}$.
    b. Return **+0**$_{\mathbb{F}}$.
16. Return 𝔽(*sign* × *mathInt*).

Figure 3.3: The parseInt Specification (https://lsantos.dev/nodebook-es-parseInt)

If you read carefully, you will see that *step 11* defines exactly the behavior of dropping everything from the first non-digit character onward(the e in our case). In *step 14* you see the boundaries for a number, which is 20 digits long, like you saw before. Along with that you can also check a few things we did not talk about, for example:

- If the number starts with `0x` then the radix is assumed to be `16`
- If the radix is greater than `36`, `parseInt` will always return `NaN`

Staying connected with the community is another excellent way to keep up. Many JavaScript developers, including TC39 members, share updates and insights on platforms like GitHub, X (formerly Twitter), and personal blogs. Tools, such as Babel and TypeScript often implement experimental features, providing a sandbox to explore and prepare for future changes.

Engaging with these resources not only sharpens your skills but also empowers you to provide feedback and adapt proactively to the language's evolution. Actively following JavaScript's

development ensures you're ready to leverage new features when they're adopted and allows you to understand why older features don't work as expected or even discover edge cases that would otherwise impact your application.

TypeScript and typing

Throughout this chapter you've seen how JavaScript's loose type system can be a source of subtle code smells.

TypeScript provides the type layer discussed here. TypeScript is a typed superset of JavaScript that adds a compile-time type layer on top of it. By annotating your code with types, you can catch a lot of code smells *before* they make it to production. You can revisit the `createUser` function from earlier in this chapter:

```
interface CreateUserInput {
  name: string;
  email: string;
  password: string;
  phoneNumber: string;
  role?: 'guest' | 'admin' | 'user';
}

export function createUser(input: CreateUserInput) {
  return db.users.create({
    id: crypto.randomUUID(),
    ...input,
    role: input.role ?? 'guest',
    password: hashPassword(input.password),
  })
}
```

With this definition, TypeScript will not let you call `createUser` with a number where a string is expected or pass an invalid role, such as `'superadmin'`. The compiler catches these mistakes at build time and prevents that from reaching production.

TypeScript also helps with many of the type coercion issues you saw. For example, if you enable `strict` mode in your `tsconfig.json` (which you should), this code will not compile:

```
function add(a: number, b: number): number {
  return a + b
}
```

```
add('5', 3) // Error: Argument of type 'string' is not assignable to parameter of
type 'number'
```

The accidental string concatenation that JavaScript would allow becomes a compile-time error. The same applies to comparisons: many TypeScript-aware linters and the compiler itself will warn you when comparing values of incompatible types, steering you away from the == versus === pitfalls we discussed.

Node already allows you run TypeScript code through the strip-types flag, but it also prevents you from using all that TypeScript has to offer.

TypeScript does not replace JavaScript at runtime

There is, however, an important distinction to keep in mind: **TypeScript types are erased at compile time**. When your code runs, it is plain JavaScript. This erasure means that every runtime behavior we covered here comes back to life.

Consider this example:

```
function processInput(value: number) {
  console.log(value + 1)
}

const userInput: any = '5'
processInput(userInput) // No compile error, prints "51" at runtime
```

Because `userInput` is typed as `any`, TypeScript's safety is bypassed. At runtime, JavaScript's coercion rules take over, and you get that same result. The same happens when data comes from external sources, such as API responses, user input, or database queries. TypeScript trusts that the types you declare match reality, but it cannot verify this at runtime. So, it depends on you to tell it the truth. It is up to you to know what types are important.

This distinction is why it is still important to understand the JavaScript behaviors we have discussed in this chapter even if you are writing TypeScript. TypeScript is a powerful tool for preventing code smells within the boundaries you control. But for data entering your application from the outside world, you still need runtime validation. Libraries like Zod can bridge this by defining schemas that validate at runtime and infer TypeScript types at the same time, giving you the best of both worlds.

TypeScript makes it much harder to *write* the kind of smelly code you've seen in this chapter, but it does not change how JavaScript *executes*. Use it as a first line of defense, not as a replacement for understanding the language underneath.

Summary

In this chapter, we explored the concept of **code smells**, those seemingly minor issues that, while not immediately breaking your code, can slowly degrade its quality. These subtle problems often act as warning signs of deeper issues in design or implementation. We discussed how ignoring them can lead to the accumulation of technical debt, resulting in costly fixes and a decline in productivity.

Understanding code smells is essential for writing maintainable JavaScript. This chapter explored common issues like overly complex function signatures, where passing too many parameters makes code harder to read and modify. A practical example was the refactoring of the `createUser` function, where replacing multiple parameters with an object improved clarity and adaptability. We also examined pitfalls, such as loose comparisons (== versus ===), improper type handling, and JavaScript's quirky type coercion. Notably, the chapter highlighted the risks of using parseInt with numbers in scientific notation and recommended using the number constructor for more predictable behavior.

To avoid subtle bugs caused by JavaScript's permissive type system, the chapter focused on truthy and falsy values and the importance of strict comparisons (===). We demonstrated how unexpected behaviors can arise, such as `[]` being coerced to false in loose comparisons, emphasizing the need to avoid modifying global prototypes. Additionally, we covered type conversion best practices, warning against the unintended consequences of using primitive constructors, such as `new Boolean(false)`, which always creates truthy objects. By applying these insights, you can write cleaner, more predictable code.

By implementing techniques, such as object parameterization, standardizing type handling, and avoiding loose comparisons, you can significantly improve your codebase's maintainability, readability, and resilience.

To make it easier to implement your code, here is a list of important questions to ask after you finish your implementation

- Are you using explicit comparisons with ===?
- Are you relying on implicit type coercion?
- Are you using too many positional parameters?
- Are you using older functions, such as parseInt instead of the primitive constructors?
- Are you modifying global prototypes?
- Are you sure that your booleans are being correctly converted?
- Can this code be better using TypeScript?

By asking yourself these questions you will cover most of the difficult patterns in the code, and they will help you model your thinking.

In the next chapter, you will expand from individual code smells to tackle common anti-patterns in Node.js.

4

Dealing with the Most Common Anti-patterns in Node.js

Node.js is a powerful platform for building scalable and high-performance applications, but with great power comes the potential for misuse. As developers, you often fall into patterns that, while seemingly convenient, can lead to hard-to-debug issues, degraded performance, or unmaintainable code. Recognizing these *anti-patterns* and addressing them is a critical step in writing reliable and efficient Node.js applications.

This chapter is designed to help you identify and tackle some of the most common anti-patterns encountered in Node.js development. You will learn how to detect problematic code and discover how to replace it with more meaningful and maintainable alternatives. Additionally, you will dive into essential best practices and coding standards that can elevate your Node.js projects, ensuring your applications are not just functional but also robust and easy to maintain.

By the end of this chapter, you will be equipped with practical strategies to recognize and fix anti-patterns in your codebase, implement tools that proactively guard against them, and adopt practices that align with industry standards.

In this chapter, we will cover the following main topics:

- The issue of blocking I/O, the main problem in Node.js
- Working with the correct data structures to be performant

Technical requirements

Note

For this chapter, you will be using Node.js in version `23.6.0, which` is the latest at the time of writing. However, odd Node.js versions are not considered stable or production-ready; they are what is called *intermediate versions*, a version between two even versions. Usually meant to implement and test new features to be tested by the community without touching **LTS** (**Long-term Support**) versions.

While most of the techniques shown here will be applicable to all Node versions, it's **strongly advisable** that, when using Node in production, you stick to LTS versions.

Some parts of the code in this chapter will be hidden to avoid repetition, however, all the related code can be found at the Packt repository. You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Also, in this chapter, you will make extensive use of benchmarking and performance tools that are native to Node. While they're reliable, it's unlikely that you will get the same results; they might differ depending on your hardware specifications and other external factors.

Working with the most common problem in Node.js

While Node shares all of the common anti-patterns with JavaScript – after all, they use the same language, Node is just another way of running JavaScript – it also has some quirks of its own.

Because of Node's server nature, its core functions have significant changes from what JavaScript can do in the browser, a simple example of that is the `node:fs` module, which allows you to access the local filesystem, something that can't be done in a common browser.

This is where your journey begins... With the most common problem Node has that browsers don't.

The use of blocking I/O

In the first and second chapters, we talked a lot about how asynchronous code is important to the JavaScript environment because the whole language interpreter runs in a single thread. However, both in the browser and in Node.js, you have a construct called **the event loop**.

They're implemented differently on different platforms, but they do essentially the same thing: run asynchronous code.

In a heavy simplification, the event loop (in Node.js) is essentially an infinite while loop that picks up tasks from a queue, offloads them to the system kernel when possible, and then returns the response to the main thread. The event loop is single threaded, however, because most kernels are multithreaded, they can handle the execution of these operations in the background and then tell Node.js so it can execute the necessary callbacks.

> **Note**
>
> We will not cover the event loop entirely here because we will talk about it in more detail in later chapters.

However, this delegated execution is not always automatic, for example, if you remember that in the previous chapters, when we talked about race conditions, we also made use of timers, such as `setTimeout`. These timers are delegated to the event loop and executed asynchronously and are an example of an asynchronous API that is automatically delegated and non-blocking. We call those **non-blocking I/O**, because they allow for the code in the main thread to continue to run while they are running in the background.

In the same way, you also have APIs that are *optionally asynchronous*, and these are the ones we call **blocking I/O operations**. They will essentially block and pause all the running code until they finish, even though they might not be related to anything else that's being run. This is the main anti-pattern you must avoid at all costs!

There are a few APIs in Node that have a blocking behavior. The most famous ones exist in the `fs` package, and you have already seen a few, for example, `readFileSync` and `writeFileSync`. However, filesystem functions are often easy to solve because they're very well-known to cause I/O issues, and they're very explicit when used in a synchronous way. In this chapter, you will focus on a different enemy, **cryptographic functions** that are not so straightforward when dealing with blocking I/O, making it difficult to debug when they cause performance problems.

Cryptographic functions are present in the Node.js crypto module. These functions are meant to be used when there is either a need to encrypt/decrypt, sign/validate, or hash any type of data. Despite being heavily optimized, in general, they all are pretty CPU-intensive, which means that, if you run any of them in the main thread, you will stop everything for a very long while.

You can look at a simple example. The crypto module has a function called `generateKeyPairSync`. This function generates a Rivest–Shamir–Adleman (RSA) key pair

synchronously and returns it as a `KeyObject`. RSA generation is highly CPU-intensive and can take a long time because it needs to derive very large primes. However, this is one of the functions that, just like the `fs` module, has an asynchronous alternative called `generateKeyPair`.

Diving into blocking I/O

Cryptographic functions are widely known to be CPU-intensive. To make this chapter even more interesting, we are not only going to show how to unblock the I/O operations, but you will also see how this has an actual impact on the performance of your applications.

You will use Node's native performance measurement tools from the `node:perf_hooks` module to measure the creation of 10,000 key pairs, both synchronously and asynchronously. You can write some code. First, you will need to set up the performance hooks:

```
import { generateKeyPair, generateKeyPairSync } from "node:crypto";
import { performance, PerformanceObserver } from "node:perf_hooks";

const perfObserver = new PerformanceObserver((items) => {
 for (const { name, startTime, duration } of items.getEntries()) {
 console.log(`${name} started at ${Math.floor(startTime)}ms and took $
{Math.floor(duration)}ms`);
 }
});

perfObserver.observe({
  entryTypes: ["measure"],
  buffer: true
});
```

Note

You can check the source code for this example in this book's GitHub repo. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface*.

Right now, you are not performing any actual measurements yet. You are just setting up the performance observer, the object that is responsible for looking at the executions of your code

and returning the execution time with high precision. Next, you can write your synchronous function:

```
performance.mark("start-sync");
for (let i = 0; i < 10000; i++) {
 generateKeyPairSync("rsa", {
 modulusLength: 1024,
 });
}
performance.mark("end-sync");
performance.measure("generateKeyPairSync", "start-sync", "end-sync");
```

The `performance` module works by placing *marks* on the executions and then measuring the differences between those marks, here, you are placing a mark at the start of the function and another mark at the end of it, you will then measure the time between those marks.

You are generating 10.000 RSA pairs with a modulus length of 1024 bits (a pretty low bit length), so it does not take too long (try to experiment with 2048 or 3072 and see the difference). After that, you will run the rest of the code with the asynchronous part:

```
performance.mark("start-async");
for (let i = 0; i < 10000; i++) {
 generateKeyPair("rsa", { modulusLength: 1024 },
 () => {});
}

performance.mark("end-async");
performance.measure("generateKeyPair", "start-async", "end-async");
```

The key difference here is the callback. The asynchronous function delegates the key generation to an external handler and registers the callback to be executed when it returns. You can save this file and call it `keypair-perf.mjs`. Then, you can run it with Node `keypair-perf.mjs` and, after a little while, you will have this output (note that the times will differ if you run it on your machine):

```
generateKeyPairSync started at 17ms and took 63072ms
generateKeyPair started at 63089ms and took 14ms
```

The synchronous function took about 63 seconds; the asynchronous one took just 14 milliseconds. This represents a decrease of more than *99.9% in time* (note that the asynchronous number only measures how long it takes to schedule the work, not to finish it),

but why is one so much faster than the other? You can solve this mystery by monitoring resource usage. If you comment all the asynchronous runs and leave only the synchronous ones, you can clearly see the reason:

> **Note**
>
> For the following picture, you can use btop (`https://github.com/aristocratos/btop`) or any system monitoring tool you're comfortable with, such as htop or the built-in Resource Monitor on MacOS and Task Manager on windows.

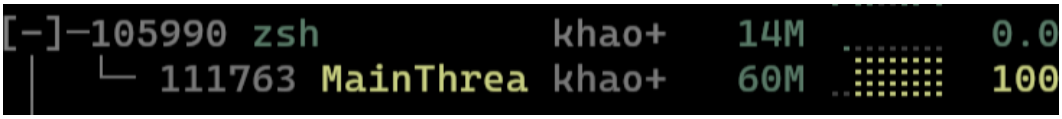


Figure 4.1: Synchronous code execution

Cryptographic functions use mainly the CPU, so when you run the key pair generation synchronously, it will run in the main thread, which is running in a single thread. You can see this by the 100% resource usage in the `MainThread` of the `zsh` shell, which is depicted by all the dots being stacked. This result means you are using a single core at a time, and this is even more visible if you look at the core breakdown:

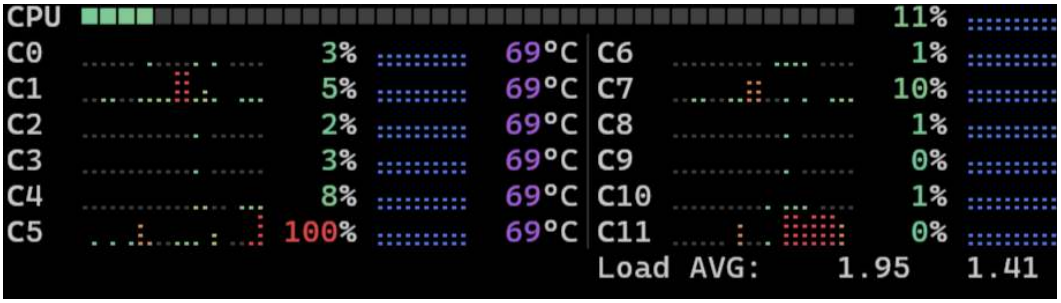


Figure 4.2: Synchronous code execution per CPU

Look at the core named `C5`, this is the one being used at the moment, but when the OS performs the scheduling of the processes' computing time, it will often cycle through other cores. This is what you see in `C11`, which was being used a short while before by some other process. This behavior is typical of a synchronous I/O blocking process looks like.

Asynchronous calls will often be instantaneous to start, but will take a while to finish, holding the main thread open even when nothing is executing. You can see this when you run the benchmark, and it stays open, even after the performance hook has printed out its output. The process remains open because the callbacks are still not finished, and the event loop is not out

of tasks to run. You can also clearly see this behavior if you do the same thing you just did for synchronous processes, but only for the asynchronous ones this time:

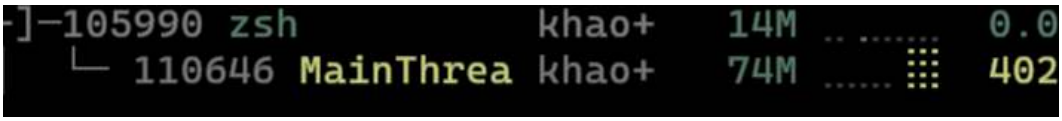


Figure 4.3: Asynchronous code execution

Look, now the process is taking 400% of the CPU (see the 402 on the right-hand side), but how is this possible? This happens because it's using 4 cores of my CPU at the same time, and this can also be seen in the core breakdown:

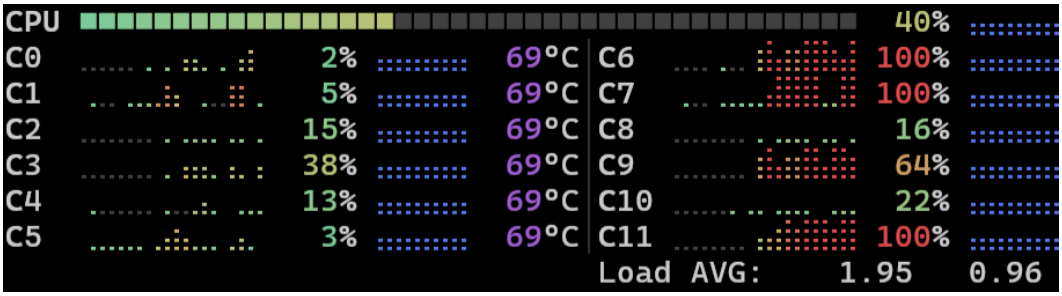


Figure 4.4: Asynchronous code execution per core

Cores `C6`, `C7`, `C9`, and `C11` are all being used at once, which means that, if you roughly divide the 10 thousand keys being generated, each core generates about 2500 keys, which makes the entire process four times faster.

> **Note**
>
> **Why 4 cores?**
>
> Internally, Node uses **libuv** for async operations. By default, the thread pool size that *libuv* allocates is defined by a variable called `UV_THREADPOOL_SIZE`, which gets an initial value of four if not set. Under bigger loads, increasing the pool size by setting the `UV_THREADPOOL_SIZE` environment variable can be the solution.

With the new understanding, you can explore how you can apply this knowledge to real-world scenarios.

Password Encryption

First and foremost, the main use case for cryptographic functions is to generate encrypted passwords or to encrypt data. Node has a native function that is made specifically to encrypt

user passwords and hash them both synchronously and asynchronously. This function is called `pbkdf2`, or **Password-Based Key Derivation Function** version 2 and, along with *BCrypt* and *Argon2*, it's one of the most widely used password hash libraries on the planet for production applications.

Now, you can write the same test you had before, but with the new function:

```
import { pbkdf2Sync, pbkdf2, randomBytes } from "node:crypto";
import { performance, PerformanceObserver } from "node:perf_hooks";

// The same code we had for the perfObserver here
// hidden for layout purposes

performance.mark("start-sync");
for (let i = 0; i < 10000; i++) {
 const password = randomBytes(64).toString("hex");
 pbkdf2Sync(password, randomBytes(8), 100000, 64, "sha512");
}
performance.mark("end-sync");
performance.measure("generateKeyPairSync", "start-sync", "end-sync");

performance.mark("start-async");

for (let i = 0; i < 10000; i++) {
 const password = randomBytes(64).toString("hex");
 pbkdf2(password, randomBytes(8), 100000, 64, "sha512", () => {});
}
performance.mark("end-async");

performance.measure("pbkdf2Sync", "start-sync", "end-sync");
performance.measure("pbkdf2", "start-async", "end-async");
```

Note

You can check the source code for this function in this book's GitHub repo. To access the repository link, follow the steps in the "*Download the example code files*" section in the *Preface*.

If you analyze the function, it generates an 8-byte salt with a 64-byte key hashed using SHA512 iterated 100 thousand times. As you can imagine, this is very hard on your poor CPU cores. When you run this file, you will see something like this:

```
pbkdf2Sync started at 16ms and took 480880ms
pbkdf2 started at 480897ms and took 43ms
```

It took about 8 minutes to run the entire code, while the asynchronous version finished in 43 ms (again, the asynchronous number only measures scheduling the work). This is how it looks in your resource manager:

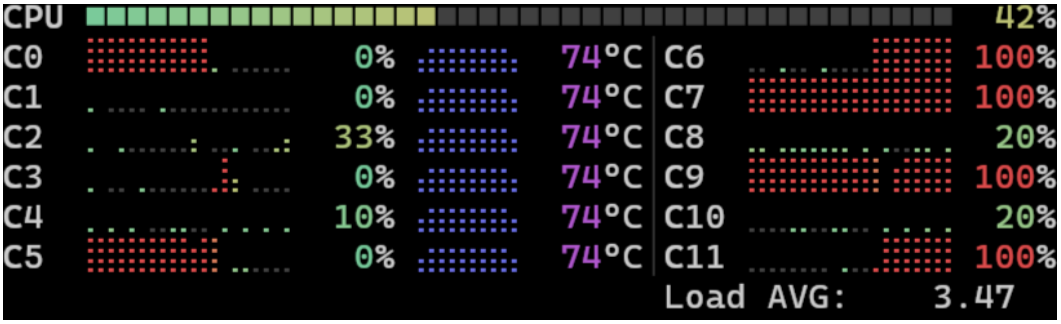


Figure 4.5: PBKDF executing in multiple cores

As you can see, it is still using the same four (`C6`, `C7`, `C9`, `C11`) cores, however, way more than the key generation function. In a real-world scenario, when you are either logging users in the application or generating user passwords, you should prefer to use the async variations of the cryptographic functions, because the synchronous ones will certainly hinder your performance a lot.

However, this is not all, in some architectures, you might find yourself repeating these expensive calculations for every request with request signatures.

Optimizing requests

To illustrate that, you can look at a simple API endpoint that has a middleware that generates a simple signature for the request. The goal is to take the current date and use a generated private key to sign that data into a header called x-signature. Just like other crypto functions, the signature functions are also CPU-intensive.

Patterns like this are commonly used in zero-trust architectures, when you have a microservices architecture and one microservice needs to validate if the responses from others originate from the claimed sources. In these cases, they would validate the response against a well-known endpoint containing a JWKS public key.

> **Note**
>
> You can check the source code for the next example in this book's GitHub repo. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface*.

For this example, you will start by creating a new project using `npm init -y` in a fresh folder. Your goal is to build a small API with a single route that includes a signature header, where the signature is based on the current date.

You will implement two versions of this application: one poorly optimized and another with proper optimizations.

Next, you will install the necessary dependencies. To keep things simple, you will use only three packages:

- `express`: The most known Node.js web framework
- `node-jose`: Cryptographic library that deals with **JavaScript Object Signing and Encryption** (also known as **JOSE**). You will use it to sign your requests
- `autocannon`: Benchmarking framework to test your speed performance

You can install these dependencies by running:

```
npm install express node-jose autocannon
```

Alternatively, you can install autocannon as a development dependency with:

```
npm i -D autocannon
```

Next, you will update the package.json file by adding three scripts to the scripts section.

```
"scripts": {
 "start": "node src/server.mjs",
 "start:cached": "node src/serverCached.mjs",
 "benchmark": "autocannon -d 30 http://localhost:3000/signed"
}
```

Now, you can create a file in `src/server.mjs`, first you can add a simple API with a single route:

```
import Express from "express";

const app = Express();

app.get("/signed", (req, res) => {
 res.send("This is signed").end();
});

app.listen(3000, () => console.info("Server listening on port 3000"));
```

If you start the server with `npm start` and then hit `http://localhost:3000/signed` you will get `This is signed` back. You can benchmark just this by using `npm run benchmark`. This will give you a report that shows how many requests per second you can run based on your 30-second testing. In my case, I got this response:

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	10,863	10,863	12,847	13,127	12,781.34	394.66	10,856
Bytes/Sec	2.62 MB	2.62 MB	3.1 MB	3.16 MB	3.08 MB	95.3 kB	2.62 MB

```
Req/Bytes counts sampled once per second.
# of samples: 30

383k requests in 30.01s, 92.4 MB read
```

Figure 4.6: Testing a bare-bones endpoint

On average, the server handles around 12,000 requests per second, which is expected since you are running a super simple route with no logic, all locally.

> **Note**
>
> The benchmark results will vary depending on your computer's specs. Machines with more resources and computing power will handle more requests, but that's not the focus right now – just keep this number in mind.

You can improve your code and add your signing middleware.

First, you need to create a function that will sign the data. You can add some code to the beginning of the file. You need to create a new KeyStore object using `jose.JWK.createKeyStore()`, which is where you will generate and store your RSA keys:

```
import jose from "node-jose";
import { createSign } from "node:crypto";

console.log(`Creating key store...`);
const keyStore = jose.JWK.createKeyStore();

console.log(`Generating keys`);
await keyStore.generate("RSA", 2048, { alg: "RS512", use: "sig" });
console.log(
 `Keys generated: ${JSON.stringify(keyStore.toJSON(true), null, 2)}`,
);

const app = Express();
// The rest of the code
```

You generate this key store before anything else because it's immutable. You will use it for all requests, so there's no point in generating it repeatedly.

Next, you need to add the function that will sign the data:

```
// The code we just wrote

function signData(data) {
 const signingKey = keyStore.all({ use: "sig" })[0];
 if (!signingKey)
   throw new Error("Could not find signing key");

 const signer = createSign("RSA-SHA512");
 const signature = signer
   .update(Buffer.from(data))
   .sign(signingKey.toPEM(true), "hex");

 return { data, keyId: signingKey.kid, signature };
}

// The rest of the code
```

Here, you retrieve the first key that matches `use: 'sig'`, which means you want a key used exclusively for signing, not encryption.

Once you have this key, you use the native `createSign` function from `node:crypto`, specifying your algorithm. You use RSA for signing and SHA512 for hashing.

The actual signing happens in the `.sign()` call, where the data passed to `signer.update` is signed using a PEM-encoded private key, and the output is a hexadecimal signature.

In the end, you return the signed data, the key ID used for signing, and the actual signature.

With all that created you can then create a new middleware just below the function:

```
// ...Signing function

const signingMiddleware = (_req, res, next) => {
 const signed = signData(Date.now().toString());
 res.setHeader(
 "X-Signature", `data=${signed.data.toString()};kid=${signed.keyId};sha512=$
{signed.signature}`,
 );
 next();
};

// The rest of the code...
```

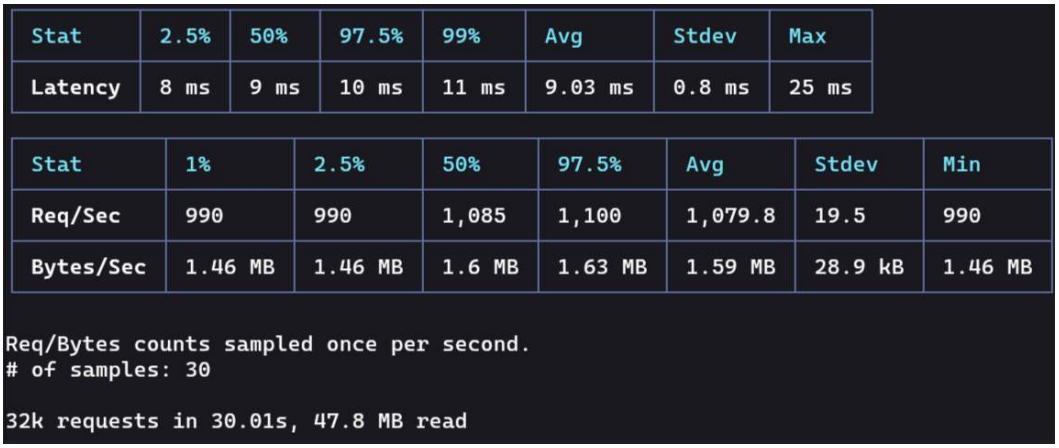
The middleware does just one thing: it signs the current millisecond count for the time and adds it to a header. You can then update your app call to use your middleware:

```
app.use(signingMiddleware);
app.get("/signed", (req, res) => {
 res.json({
   message: "This is a signed request",
   signature: res.getHeader("X-Signature"),
 });
});
```

For debugging purposes, you will also log the header in the response. You can run the server again; you should now get a different response when hitting the same endpoint. Something like this:

```
{
 "message": "This is a signed request",
 "signature": "data=<numbers>;kid=<key ID>;sha512=<very long signature>"
}
```

Now, you can run autocannon with `npm run benchmark`. The results are very interesting:



Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	8 ms	9 ms	10 ms	11 ms	9.03 ms	0.8 ms	25 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	990	990	1,085	1,100	1,079.8	19.5	990
Bytes/Sec	1.46 MB	1.46 MB	1.6 MB	1.63 MB	1.59 MB	28.9 kB	1.46 MB

```
Req/Bytes counts sampled once per second.
# of samples: 30

32k requests in 30.01s, 47.8 MB read
```

Figure 4.7: Benchmark with I/O blocking

From 12,000 requests per second down to just 1,000... Also, notice that the benchmark made 383,000 requests the first time versus just 32,000 now. This represents a 92% reduction in performance... How can you make this better?

The first thing that might come to mind is to wrap it all in an asynchronous wrapper, such as `setImmediate`, or possibly `Promise.resolve`, or even add an `async` annotation to the function. However, none of those will solve the problem. The issue is the blocking nature of the signing function. A nature you can't change unless you rewrite it from scratch using only asynchronous operators.

So, you have to work your way around it.

> **Note**
>
> You will learn more about this in the next chapters, but running any function in an asynchronous wrapper does **not** make this function asynchronous (remember the *sync-async* anti-pattern).

In this case, the solution is simple: execute this function as few times as possible. And you can do this by adding a cache to your code.

Let's copy the `server.mjs` file into a new file called `serverCached.mjs` and make some optimizations. First, you will create a global variable called `cachedSignature` and set it to null before instantiating your app:

```
// The rest of the code...
let cachedSignature = null;
const app = Express();
// The rest of the code...
```

Next, we'll modify our middleware so that instead of generating a new signature for every request, it updates the signature only once every 10 seconds. When the middleware runs and finds no cached signature, it will generate one, store it in `cachedSignature`, and start a timer using `setInterval` to refresh the signature every 10 seconds. Here is the update:

```
const signingMiddleware = (_req, res, next) => {
 if (!cachedSignature) {
   console.info("Signature is not cached");
   cachedSignature = signData(Date.now().toString());
   setInterval(
     () => (cachedSignature =signData(Date.now().toString())),
 10000);
 }

 res.setHeader(
      "X-Signature",          `data=${cachedSignature.data.toString()};kid=$
{cachedSignature.keyId};sha512=${cachedSignature.signature}`);
 next();
};
```

Note

Since you're dealing with signed data, the actual content isn't the key concern – we just need to verify that the data comes from the expected signer. This allows us to reuse the same signed data multiple times, since clients can validate it independently. However, this can be a different case for encryption or other crypto functions.

Now, with this small change, you can run `npm run start:cached` and then `npm run benchmark` to check the results.

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	10,119	10,119	12,239	12,463	12,194.14	411.34	10,114
Bytes/Sec	14.9 MB	14.9 MB	18.1 MB	18.4 MB	18 MB	609 kB	14.9 MB

```
Req/Bytes counts sampled once per second.
# of samples: 30

366k requests in 30.01s, 540 MB read
```

Figure 4.8: Optimized I/O blocking function

You are back on track! The performance is slightly lower than the initial barebones endpoint, but that's expected since you are now generating a key pair and signing each request.

The key takeaway here is that you improved performance not by making the function "asynchronous," but by minimizing how often it runs. When dealing with blocking functions that lack an async alternative (such as `pbkdf2` had), the best optimization is often about *when* the function runs, rather than *how* it runs.

Note

When Async doesn't help

There will be cases where a task is truly CPU-bound; this means that none of the async alternatives we mentioned, or any caching will help. In these cases, the option is to offload the work to another thread, background job queues, or even offload to external services. This is what I would call an "ethical hack," because it does not solve the problem, but it will take advantage of multiple processes that can truly use multi-cores to parallelize work.

That said, optimizing execution frequency isn't the only way to boost performance. Just as important as understanding async functions is choosing the right data structures for the job.

Understanding why data structures matter

I/O blocking may be the most common problem in the Node ecosystem, but it is far from the only one. The other problem is not precisely with the language itself or its internal workings, but rather how you, as developers, use its tools, especially the **data structures**.

There are several well-known data structures in the programming world, most of them are known and implemented in all languages, some of them are only available in specific ecosystems because of the capabilities they need, however, the misuse of data structures is one of the leading causes of performance or memory issues in most applications.

You can look into how you can prevent that by knowing your data structures and when to use them.

There are roughly two types of data structures. The ones focused on optimizing speed to the detriment of memory, and vice versa. In native JavaScript you already have plenty of effective data structures out of the box, Node adds some more implementations on top of those, which you will see in the following sections. Others are completely native, such as objects, maps, sets, and arrays. Which is awesome! Because all the performance improvements should work the same regardless of your JavaScript (JS) runtime.

In this section, you will explore the data structures that focus on speed and performance. When we talk about this, we are usually talking about either:

- Objects and Arrays
- Maps and Sets
- WeakMaps and WeakSets

And, even within these data structures you still have some patterns, for example if you had a function like this:

```
function getLogFunction(level) {
 if (level === "info") return infoLogFn;
 else if (level === "warn") return warnLogFn;
 else if (level === "error") return errorLogFn;
 else if (level === "fatal") return fatalLogFn;
 else return debugLogFn;
}
```

It seems quite... Cumbersome, does it not? What if you transform this into a more *modern* approach:

```
function getLogFunction(level) {
 switch (level) {
 case "warn":
   return warnLogFn;
 case "info":
   return infoLogFn;
 case "error":
   return errorLogFn;
 case "fatal":
   return fatalLogFn;
 default:
   return debugLogFn;
 }
}
```

This would work too, and it's a bit more readable. However, what if I told you, it's not optimized? How so? Aren't `if` and `switch` the lowest-level conditional operators? Well... not quite that. There's a faster approach known as **object lookups**.

You can recreate the same function but, instead of an if clause, you will use an object in which each key is the level you want to get, then you will just return the value of that key:

```
const lookup = {
 info: infoLogFn,
 warn: warnLogFn,
 error: errorLogFn,
 fatal: fatalLogFn,
};

function getLogFunction(level) {
 return lookup[level] ?? debugLogFn;
}
```

This approach is exactly twice as fast as the `switch` option, and about three times faster than the `if` option, according to this benchmark:

```
ifCheck took 0.0343ms
switchCase took 0.0232ms
strLookup took 0.0116ms
```

If you run the function multiple times, the differences will narrow, but the order will remain. The `if` is the slowest one, while the switch follows second by a close margin, and the object lookup comes faster:

```
ifCheck took 0.2045ms
switchCase took 0.2024ms
strLookup took 0.1515ms
```

However, these implementations and benchmarks might vary due to engine optimizations. For example, in browsers, using the `if` check has often been faster, whereas *less common* methods for value lookups tend to be slower.

Note

Notes about the benchmark

To save some pages and spare you from re-reading all the same code you have already written before, I'll skip the code writing of this benchmark here. But you can check it out *in this book's GitHub repo. To access the repository link, follow the steps in the "Download the example code files" section in the Preface.*

But why does this happen? It all comes down to how JavaScript engines implement objects using *hidden classes*. You will dive deeper into engine internals in the final chapters, but for a simplified overview: the V8 engine (which powers Node) treats an object as a class, and accessing keys is just a matter of moving a pointer through an offset within that class.

You can make this function even faster by replacing descriptive strings with numbers:

```
const lookup = {
 1: infoLogFn,
 2: warnLogFn,
 3: errorLogFn,
 4: fatalLogFn,
};
```

```
function getLogFunction(level) {
 return lookup[level] ?? debugLogFn;
}
```

As you can imagine, you would need to make some changes to your level variable since it could only accept numbers. The benchmark results are slightly better for them:

```
ifCheck took 0.2045ms
switchCase took 0.2024ms
strLookup took 0.1515ms
numberLookup took 0.1316ms
```

This result shows that the key length makes a difference in object lookups. You could also use another lookup data structure to make it work with larger strings, *a second lookup object*, for example:

```
const levels = { info: 1, warn: 2, error: 3, fatal: 4 };
const lookup = { 1: infoLogFn, 2: warnLogFn, 3: errorLogFn, 4: fatalLogFn };

function getLogFunction(level) {
  return lookup[levels[level]] ?? debugLogFn;
}
```

As expected, this is slower than looking up with only the numbers:

```
ifCheck took 0.2077ms
switchCase took 0.1813ms
strLookup took 0.1450ms
numberLookup took 0.1187ms
numberStrLookup took 0.2166ms
```

This brings you to the first drawback of using objects as lookup structures: *an object can only accept strings or symbols as keys*. Any other value will convert automatically to a string using the `toString()` method.

Now, it's time to present a new data structure that will save you: the Map. You can discover the maps.

Maps and Objects

Maps are relatively recent additions to JavaScript, but their potential is vast. While objects are better suited for static key-value storage and smaller datasets, Maps are optimized for scenarios that require frequent additions and deletions, especially when dealing with large datasets.

Beyond that, Maps allow any value as a key and any other value as the corresponding entry, making them highly versatile for building dictionaries that span multiple use cases. Maps are particularly useful when you need to iterate over the dataset or perform quick lookups.

For example, let's examine how Maps handles large datasets. To do this, you can create a million keys in both an Object and a Map. But first, you can start your performance counter:

```
import { performance, PerformanceObserver } from 'node:perf_hooks'

const perfObserver = new PerformanceObserver((items) => {
     for (const { name, startTime, duration } of items.getEntries()) {
         console.log(`${name} started at ${(startTime.toFixed(4))}ms and took $
{(duration.toFixed(4))}ms`)
     }
 })
 perfObserver.observe({ entryTypes: ["measure", "function"], buffer: true })
```

Then, you can initialize both your map and your object:

```
let myObject = {}
 let myMap = new Map()
 const TOTAL = 1_000_000

const objectSetup = performance.timerify(function createObject(size = TOTAL) {
     for (let i = 0; i < size; i++) {
         myObject[i.toString()] = i
     }
 })

const mapSetup = performance.timerify(function createMap(size = TOTAL) {
     for (let i = 0; i < size; i++) {
         myMap.set(i.toString(), i)
     }
 })
```

```
objectSetup()
 mapSetup()
```

Now, here's an interesting result: when running this benchmark, initializing an object is twice as fast as initializing a Map. However, this is the only case where objects outperform Maps:

```
createObject started at 12.8238ms and took 30.5704ms
createMap started at 43.5291ms and took 77.7110ms
```

Next, you can measure the performance of core operations for both Maps and Objects:

- Retrieving a random key
- Adding a value
- Checking for the existence of a key
- Iterating over all values
- Deleting a single key
- Clearing all keys

The full result of your benchmark is the following:

```
get random key from object started at 127.2490ms and took 0.0601ms
get random key from map started at 127.3145ms and took 0.0599ms

add random key to object started at 127.5055ms and took 0.0602ms
add random key to map started at 127.5697ms and took 0.0035ms

check random key of object started at 127.5856ms and took 0.0587ms
check random key of map started at 127.6471ms and took 0.0021ms

delete random key of object started at 422.3607ms and took 0.0032ms
delete random key of map started at 422.3645ms and took 0.0023ms
```

Before discussing iteration and clearing, you can focus on these core operations. Retrieving a random key in both an object and a Map shows negligible differences, as performance depends on the key's position. However, in all other operations, Maps are significantly faster – sometimes up to *28 times faster than objects*.

For instance, when adding, checking, and deleting keys, *Maps consistently perform under 0.004 ms*, whereas objects are at least an order of magnitude slower. Deletion is the exception: the two structures are roughly on par.

When it comes to *iteration*, however, the story changes. Because objects are not natively iterable, the only way to loop through their values is by first extracting their keys and then iterating over them. There are two common ways to do this:

1. Using `Object.keys()`

```
for (const key of Object.keys(myObject)) {
    myObject[key];
}
```

2. Using `Object.entries()`

```
for (const [key, value] of Object.entries(myObject)) {
    value;
}
```

As you can imagine, these approaches are slower because they require *two iterations* – one to collect the keys and another to process them. The benchmark confirms this, showing a significant performance difference:

```
iterate object using for of started at 124.6129ms and took 64.4848ms
iterate object using entries started at 189.1074ms and took 229.2351ms
iterate map started at 418.3732ms and took 5.3252ms
```

In this case, because Maps are directly iterable using `forEach` or `for...of`, their execution time is *almost 43 times faster* than iterating over an object.

The same holds true for *clearing an object*. The only way to remove all entries without losing its reference is by iterating through all the keys and deleting them one by one:

```
for (const key of Object.keys(myObject)) {
     delete myObject[key]
 }
```

While you can do `myMap.clear()` in the case of a map. As expected, one is much faster than the other:

```
clear object started at 423.8397ms and took 96.3512ms
clear map started at 520.2020ms and took 0.0080ms
```

So, changing your code from before, you could replace your lookup object for a map:

```
const lookup = new Map([
  [1,infoLogFn],
  [2,warnLogFn],
  [3,errorLogFn],
  [4,fatalLogFn]
 ]);

function getLogFunction(level) {
  return lookup.get(level) ?? debugLogFn;
 }
```

In this particular case, replacing an object with a Map won't make a significant difference because you are dealing with a small, fixed set of keys. Both structures would perform similarly. However, for larger lookups and larger datasets, Maps offer a significant performance advantage.

There are also other key differences between Maps and objects. Some are minor, for example, to count the number of keys in an object, you need to iterate over them using `Object.keys(obj).length`, whereas Maps provide a built-in size property for this purpose.

However, there's a more serious issue with objects that go beyond performance.

Objects and prototype pollution

Another issue with objects is **prototype pollution**. Since objects are part of the JavaScript prototype inheritance chain (they're essentially why it exists), adding properties to them can unintentionally change their behavior:

```
const anObject = {}
```

Although this object may seem empty, it inherits all properties from `Object.prototype`. Now, if you do something like this:

```
const anObject = {
    hasOwnProperty: (prop) => console.log('nope')
}
```

Congratulations, you have just overridden the default `Object.prototype.hasOwnProperty` method. Now, every time someone uses this object, because they think it's a normal object and try to call `anObject.hasOwnProperty()`, they'll get `nope` in the console.

This issue, combined with that objects can't store keys that are anything other than strings or symbols, makes *objects a poor choice for hashmaps*.

However, you can mitigate prototype pollution attacks with defensive programming by borrowing the `hasOwnProperty` method from `Object.prototype`:

```
if (Object.prototype.hasOwnProperty.call(anObject, 'key')) {}
```

There is also the static `hasOwn` property added to the object which basically does this as well:

```
if (Object.hasOwn(anObject, 'key')) {}
```

Or a better way is to create an Object that does not inherit from the core prototype with `Object.create` which allows you to specify the prototype of an object:

```
const anObject = Object.create(null, {})
```

However, you will always need to use other methods like the ones outlined previously because the created object will not have any properties.

However, objects are not the only ones that can be optimized! Let's talk about arrays!

Arrays and Sets

Arrays, along with objects, are probably the most used structures in JavaScript. Therefore, they are also the most prone to performance issues.

Arrays, in particular, are susceptible to this because they are easy to create but can be difficult to work with in certain cases. For example, when representing a bidimensional array, you often nest one array inside another. However, iterating through these requires a nested loop, resulting in an $O(n^2)$ complexity.

It's possible to reduce this to O(n) by using a grid data structure, which is just a single array of contiguous bytes.

Another simple but common example is deduplication. Several times when looking at the code, I stumbled upon code that's either using written code for deduplication of items in the array, or worse, using external libraries, such as Lodash's `_.uniq` function just to do this. And this is how you can steal compute time from your program and make it super slow.

To counter that, you have **Sets**, the twin brother of Maps, but they work with arrays. Sets accept any type of value, but the catch is that this value can occur *only once in the whole set*. If the value appears again, the set will simply not add it. Which means you already have deduplication out-of-the-box. You can compare these two.

In this benchmark, you will create a normal array, and a function called `pushUnique`, which will only push the data to the array if it's not already there:

```
function pushUnique(arr, item) {
    if (arr.includes(item)) return
    return arr.push(item)
}

const arr = []
const set = new Set()

performance.mark('add-arr-start')
pushUnique(arr, 1)
pushUnique(arr, 1)
pushUnique(arr, 2)
performance.mark('add-arr-end')

performance.mark('add-set-start')
set.add(1)
set.add(1)
set.add(2)
performance.mark('add-set-end')

performance.measure('add key to array', 'add-arr-start', 'add-arr-end')
performance.measure('add key to set', 'add-set-start', 'add-set-end')
```

Note

Before the code, I also set up the performance observers just as we did in the Maps example.

The result you will get will look something like this:

```
add key to array started at 15.1592ms and took 0.0693ms
add key to set started at 15.2310ms and took 0.0023ms
```

Just in this single execution, you can see that arrays are about 30 times slower than sets when adding items. If you directly use `arr.push` instead of the function, it's even slower:

```
add key to array started at 15.1592ms and took 0.0777ms
add key to set started at 15.2310ms and took 0.0023ms
```

One interesting take from these benchmarks is that, if you repeatedly run this test, the time that Sets take to add another item fluctuates between 0.0020 ms to 0.0028 ms. This suggests that, if you ignore local factors, such as CPU time or system load, adding an item to a set is also done in constant time (`O(1)`) on average.

Sets also have a `Set.has` method that is the same as `Array.prototype.includes`. Because arrays need to sweep the entire structure to find something in it, if the item is at the beginning of the list, then it will take less time, and the opposite is true if the item is at the end of the list. However, due to the way sets are created, they can return data in constant time (just like in additions). This means that, for larger datasets, sets work way better, while smaller datasets might favor arrays. How large? You can measure it!

Working with large datasets, with Sets

To start, you will clear both structures with:

```
arr.length = 0
set.clear()
```

Then, you will populate the initial items in the first position of each structure with a very small value; then generate a million items using a small `randomBetween` helper to get a number from 1 to 100; and finally add a very large number at the end of both. This will cover the following cases:

- Checking for a known number in the beginning and end of the array
- Checking for a random number that may or may not already exist in the middle
- Checking for a nonexistent number in both structures

You can do this with this code:

```
// add initial item
set.add(0)
arr[0] = 0
// add middle items
const randomBetween = (min, max) => Math.floor(Math.random() * (max - min + 1)) +
min;
```

```
const TOTAL = 1_000_000
for (let i = 0; i < TOTAL; i++) {
  arr.push(randomBetween(1, 100))
  set.add(randomBetween(1, 100))
}
// add final item
pushUnique(arr, TOTAL)
set.add(TOTAL)
```

Then, you can write your performance assertions:

```
performance.mark('includes-end-arr-start')
arr.includes(TOTAL)
performance.mark('includes-end-arr-end')

performance.mark('has-end-set-start')
set.has(TOTAL)
performance.mark('has-end-set-end')

performance.measure('check array existence in the end', 'includes-end-arr-start',
'includes-end-arr-end')
performance.measure('check set existence in the end', 'has-end-set-start', 'has-
end-set-end')

performance.mark('includes-beginning-arr-start')
arr.includes(0)
performance.mark('includes-beginning-arr-end')

performance.mark('has-beginning-set-start')
set.has(0)
performance.mark('has-beginning-set-end')

performance.measure('check array existence in the beginning', 'includes-beginning-
arr-start', 'includes-beginning-arr-end')
performance.measure('check set existence in the beginning', 'has-beginning-set-
start', 'has-beginning-set-end')

performance.mark('includes-middle-arr-start')
arr.includes(86)
performance.mark('includes-middle-arr-end')
```

```
performance.mark('has-middle-set-start')
set.has(86)
performance.mark('has-middle-set-end')

performance.measure('check array existence in the middle', 'includes-middle-arr-
start', 'includes-middle-arr-end')
performance.measure('check set existence in the middle', 'has-middle-set-start',
'has-middle-set-end')

performance.mark('includes-non-existent-arr-start')
arr.includes(1000)
performance.mark('includes-non-existent-arr-end')

performance.mark('has-non-existent-set-start')
set.has(1000)
performance.mark('has-non-existent-set-end')

performance.measure('check array existence for non-existent', 'includes-non-
existent-arr-start', 'includes-non-existent-arr-end')
performance.measure('check set existence for non-existent', 'has-non-existent-set-
start', 'has-non-existent-set-end')
```

When you run it, you get something close to:

```
check array existence in the end took 0.5457ms
check set existence in the end took 0.0858ms
check array existence in the beginning took 0.0040ms
check set existence in the beginning took 0.0062ms
check array existence in the middle took 0.0025ms
check set existence in the middle took 0.0008ms
check array existence for non-existent took 0.2054ms
check set existence for non-existent took 0.0008ms
```

You can see that, in almost all cases, sets were much faster (the only exception is the item at the beginning of the array). The interesting part is that, when the item you are looking for doesn't exist, this is the worst possible case since the array needs to go through all the items and return nothing, while a set, being a hash map, can calculate that the index of the item is outside its bounds with a single computation, therefore it's 260x faster than the array.

You can also repeat this test for a smaller dataset by changing the TOTAL variable, let's try 10,000 items:

```
check array existence in the end took 0.0110ms
check set existence in the end took 0.0555ms
check array existence in the beginning took 0.0037ms
check set existence in the beginning took 0.0012ms
check array existence in the middle took 0.0022ms
check set existence in the middle took 0.0008ms
check array existence for non-existent took 0.0031ms
check set existence for non-existent took 0.0008ms
```

In this run, checking for items at the end is 5 times faster in the array than in the set – probably due to engine optimizations (we will talk about them in the upcoming chapters) – but all the others are still slower. You can see how it behaves for just 100 items:

```
check array existence in the end took 0.0107ms
check set existence in the end 0.0641ms
check array existence in the beginning took 0.0519ms
check set existence in the beginning took 0.0013ms
check array existence in the middle took 0.0037ms
check set existence in the middle took 0.0010ms
check array existence for non-existent took 0.0014ms
check set existence for non-existent took 0.0009ms
```

The pattern repeats itself. This result means that, when you reduce the number of items in the collection, arrays tend to perform better in searches but that is basically it; all other operations are still slower than sets.

Also, arrays are very *crude* structures. The array gives you a barebones interface for the most common operations, but it won't give you more advanced methods like intersection or difference. Sets, on the other hand, already come with some very interesting methods, for example, assuming A and B are sets:

- `A.difference(B)`: Returns all elements of **A** not in **B**
- `A.intersection(B)`: Returns elements present in both sets
- `A.symmetricDifference(B)`: Returns elements that are different in both sets, for example, if A is `[1, 2, 3]` and B is `[3, 4, 5]` the symmetric difference of A and B is `[1,2,4,5]`
- `A.union(B)`: Joins both sets (like Array.prototype.concat, but without duplicates)

- `A.isDisjointFrom(B)`: if no element of A is in B then this is true
- `A.isSubsetOf(B)`: if all elements of A are in B
- `A.isSupersetOf(B)`: if all elements of B are in A

As an example, to implement both difference and intersection in arrays you would need to do the following:

```
function difference(arr1, arr2) {
    return arr1.filter(i => !arr2.includes(i))
}

function intersection(arr1, arr2) {
    return arr1.filter(i => arr2.includes(i))
}
```

There is no point in even trying to measure those because it's clear that the operations in a hash map structure, such as sets will be constant, while the operations on arrays will require two loops to iterate in both arrays.

However, we work with data, so you can create a very simple test initializing arrays and sets:

```
const arr1 = [1, 2, 3]
const arr2 = [2, 3, 4]
const set1 = new Set(arr1)
const set2 = new Set(arr2)
```

Then, you will use the simple `console.time` function to measure the time for them just for ergonomics:

```
console.time('difference')
difference(arr1, arr2)
console.timeEnd('difference')

console.time('intersection')
intersection(arr1, arr2)
console.timeEnd('intersection')

console.time('setintersection')
set1.intersection(set2)
console.timeEnd('setintersection')
```

```
console.time('setdifference')
set1.difference(set2)
console.timeEnd('setdifference')
```

The difference is clear:

```
difference: 0.017ms
intersection: 0.013ms
setintersection: 0.004ms
setdifference: 0.004ms
```

In a small set of data like this, they are not far apart, but as the dataset grows, the difference will be staggering.

On the other hand, sets do not come with some handy methods, such as sort, which makes them less suitable for sorting unsorted data as you would need to transform the set into an array and then sorting it:

```
const unsorted = new Set([1,55,23,2,45,99])
const sorted = new Set(Array.from(unsorted).sort())
```

In addition to that, Sets also don't have other methods, such as `slice, substring, splice, reverse` and so on. They're purely made to be used as computation structures.

Which might lead you to think: *When then, should I use each one?* Fear not! This is covered, too!

Choosing the tool for the job

Having powerful tools is only helpful if you know how to use them effectively. So far in this section, you have explored some of the key data structures available in JavaScript. While these aren't the only ones you will cover in this book, they provide a solid foundation. That said, like most things in tech, no single tool or structure is suitable for every situation.

Objects versus Maps

The first comparison you need to make is whether to use a Map or an Object.
Your consciousness should say: *If Maps are faster, why use objects at all?* This is a valid question, but mostly because Maps take much longer to initialize and can consume more memory depending on the situation, especially smaller datasets. Think of it as using a box the size of a car, just to fit a penny inside.

Use objects when:

- You have a defined (and static) set of keys, meaning they will likely not change (or change little) over time, for example, environment configurations or method options
- You need simple key-value storage where you won't be iterating through keys
- You need to pass immutable entities around (like classes, and so on)
- You don't need to delete a lot of keys

Use Maps when:

- You need ordered data in the entries (maps keep the order of insertions)
- Keys are not strings nor Symbols, for example, numbers or object keys
- You have large datasets
- You will need to iterate or search for keys and values in the map
- You will be changing the data many times, either inserting, updating, or deleting entries

One last piece of advice is to know what the destination of your map or object will be, for example, if you are serving a web request, then Maps are worse because they are not directly *JSONifiable* then you will eventually need to convert the map into entries, and those entries into an object, which will eventually make it slower

For these cases, it is better to do the heavy computation on the Map, then store the results in objects

Sets versus Arrays

The same questions apply to Sets versus Arrays: *Why don't you just use sets?* Simply because sets behave differently from arrays, then it's a bit easier to know when to use one and the other.

Use Arrays when:

- Your data can be repeated
- You won't need to perform frequent searches and checks
- You have a static set of data, for example, arrays are perfect for the case of storing binary data because the entire array represents a single entity
- You need to sort unsorted data
- You need to perform operations in a subset of the data, since sets do not have methods like `substring`

Use sets when:

- You need to deduplicate the data

- You need to perform frequent checks for existence of an element
- You need to constantly add or remove items from the set
- You have large datasets
- You need to perform operations such as intersections and differences

The same advice that you had for Maps also goes with Sets. They are not stringifiable, so if you need to return a Set to a user in a request, for example, you will need to convert that set into an Array first

> **Note**
>
> It is important to mention that, while Maps and Sets still don't have some of the methods that Arrays do, the Iterator Helpers proposal, part of ES2025 and available in Node.js 22 and later, adds methods such as map, filter, and take to all iterators: `https://lsantos.dev/nodebook-tc39-proposal`.

Obviously, the use of Maps and sets depends on the project you are designing and the actual goal of your application; do not take these as hard rules, as there are no such things in programming.

Summary

This chapter focused on two key topics: blocking I/O and data structures. Though you covered a lot, there's still much more to explore on both of them.

You first tackled I/O blocking, learning how to delegate execution to other CPU cores. You started with Node's `fs` module, then moved to `crypto` for creating and validating signatures. You extended this to web services, benchmarking the impact using Autocannon.

Next, you explored JavaScript data structures, mainly Maps and Sets. Maps outperform objects in cases requiring frequent key-value manipulation, offering better performance and memory efficiency. Unlike Objects, Maps allow any value as a key and maintain insertion order. Objects, however, work better for small, static datasets like configs.

You also compared Sets and Arrays. Sets prevent duplicates and are faster for lookups but lack methods, such as `filter` and `map`. They excel in operations like intersections and differences, while Arrays are better for ordered and sequential processing.

With data structures covered, it's time to dive into one of JavaScript's most debated topics in the next chapter: Promises and Callbacks.

5

Understanding Promises and Callbacks

You have now moved from the realm of regular code into the scary realm of asynchronous code. This part of the book focuses entirely on how you manage those pesky promises, callbacks, and events. In this chapter, we will cover the beginnings of that.

First, you will learn the differences between callbacks and promises and how they work on the surface and under the hood, but you will not (yet) go through the deep technical stuff that is underlying the V8 JavaScript engine. Next up you will understand some common callback anti-patterns followed by the most common promise anti-patterns and how you can not only spot but also fix, all of them.

This chapter is a starting point. However, remember that, as for all the things, the tips and tricks you see here might not apply to your specific use case. Not all the techniques here are anti-patterns, but rather a different way to write your code, so take it all with a grain of salt and do not forget to apply your structures and logic to it as well.

By the end of this chapter, you will understand the difference between callbacks and promises in practical and technical detail. You will also understand the main setbacks each can cause and the anti-patterns they can expose.

Therefore, we will cover the following main topics in the chapter:

- Taming the asynchronous code
- Understanding the callback anti-patterns
- Understanding the promise anti-patterns

Let's dive in!

Technical requirements

For this chapter, it's recommended that you already have some experience with JavaScript promises and asynchronous code execution as this chapter does not cover exactly what those are from the start. If you don't, you should check the Node.js and JavaScript documentation before going further, otherwise you might not be able to get the most out of the chapter.

Also, it is important to notice that asynchronous code is not only related to Node.js, but to JavaScript itself. While this chapter discusses Node.js-only libraries, such as I/O file reading, some of the underlying examples relate to JavaScript.

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Taming the asynchronous code

Talking about asynchronous code in JavaScript is almost, such as rediscovering how to code every time. It is arguably the most complicated concept to understand in the language, and it has multiple quirks that span not only high-level code but also go deep into the internals of Node and V8 itself. This chapter is not about teaching you how to do asynchronous code, but rather how **not** to do it. To do this, you need a baseline understanding of what the main actors in this ballet are.

This chapter does not cover every detail of how asynchronous processes work, as *Chapter 8* discusses that in detail. The aim of this part is to differentiate what is and is not good practice with promises, while still getting a base understanding of how these two constructs interact with Node.js internals. So, if any terms confuse you, you will probably find better explanations for them later.

A tale of callbacks and promises

Let's start with the question I've heard the most in my career: **Are not callbacks a special case of promises?**

Short answer, no. Internally they behave differently. The long answer is that a callback used to be the only way of dealing with asynchronous operations in Node.js until the addition of promises during ES6. The term **callback hell** that you encountered in the very first chapter was originally used to describe callbacks.

Internally, a callback triggers a signal to Node's libuv (the library responsible for handling asynchronous tasks), which signals the event loop to enqueue the callback to execute on the main thread in a future iteration of the loop. This scheduling ensures non-blocking behavior,

but as with all non-blocking code, the code executes in another call stack, which is not directly accessible to you.

Summarizing it all, a callback is nothing more than a scheduled function call that enters a queue to be executed at a later date. The callback has its own call stack and runs independently of the main thread until the time it is executed.

A promise, on the other hand, is not *just* a pointer to a function call; it is actually a JavaScript object that represents a value that's not yet known. That's why it is called a promise. Like promises in real life, a promise in JavaScript holds either the success (fulfillment) of an operation, or the error (rejection) of it in the future.

Unlike callbacks, a promise does not need a function to be passed in as a callback so, instead of expecting a callback as an argument, a function can *return a promise,* and the caller will attach callbacks to that promise object via the `.then()` and `.catch()` methods.

A promise will have three states. Initially it will be in the **pending** state, which is the initial state. If the operation succeeds, the promise will jump into **fulfilled**, otherwise it will be **rejected**. A promise can *never* transition *back* to *pending* if it's already *fulfilled* or *rejected*. The states of *fulfilled* and *rejected* are called **settled states**, when a promise is in either of them, *the promise is said to be settled*.

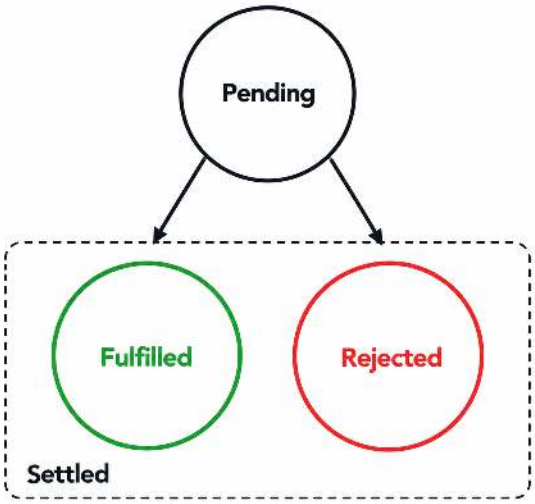


Figure 5.1: Settled promise diagram

Internally, a promise also relies on the event loop as callbacks do, but it uses a different mechanism called a **microtask queue** (which you will explore in detail in *Chapter 8*), which runs after the current stack is completed.

Callbacks or promises?

One of the main questions you get when working with asynchronous programming is: *If there are promises and callbacks, which one is correct? The* answer is as esoteric as the question. *The choice depends on the situation*. To understand it you need to first understand the differences between one and the other.

Callbacks are the simplest type of asynchronous operation you can have in JavaScript; it is literally just a function call that is executed at the end of some event. For example, a callback is nothing more than an event emitter handle:

```
function readFileWithCallback(filePath, callback) {
  let content = "";

  const stream = fs.createReadStream(filePath, { encoding: "utf-8" });

  // yielded control back to the main thread
  stream.on("data", (chunk) => {
    content += chunk; // accumulate the chunks
  });

  stream.on("end", () => {
    // call the callback with null error and the content
    callback(null, content);
  });

  stream.on("error", (err) => {
    // call the callback with the error
    callback(err, null);
  });
}
```

In this example, you can read a file by opening a stream, the stream returns an `EventEmitter` instance where you can attach handlers; these handlers take in a callback. When you call the read stream, and during the entire operation that happens within the `data` event, the control is yielded back to the main thread, which means the code will continue to run while the file is read. When one of the `end` or `error` events is called, the function pointer to the callback is activated, and the function will be called.

So, callbacks are as lightweight as a function call, which gives them a significant performance advantage in operations that are performance-sensitive, especially because the execution of a

callback does not pass through any other logic layers, it is what C++ code directly executes in Node.js' core. In some benchmarks, a callback can achieve up to 1.9x the performance of promises for sequential operations and up to 4.9x in parallel ones *[1]*.

> **Note**
>
> A fun fact of this is that callbacks are **still** the recommended form for dealing with filesystem operations using the `node:fs` package due to this performance gain

Promises, on the other hand, add an *amazing* convenience wrapper around the complexity of asynchronous operations; however, this convenience is not free. Each promise has an inherent overhead that is tied to that it is a JavaScript object, so each promise needs to be allocated in memory and, later, garbage collected. Each `.then` or `.catch` handler creates yet another closure and can itself start other promises that will repeat the entire process. With `async/await` this gap increases even more.

> **Note**
>
> It is important to note that, while promises are technically slower than callbacks, this performance gap is negligible in most of the common use cases of Node. The difference will matter when you are executing hundreds of thousands or millions of operations per second.
>
> Another reason why promises are *slower* than callbacks is because they work with what is called a microtask queue, a slightly more complex scheduling mechanism that you will explore in the next chapters.
>
> For the vast majority of Node applications that will be using HTTP handlers, database calls, I/O handlers, and so on. This overhead is dwarfed by I/O latency, which is orders of magnitude bigger than the promise latency. So, default to promises/async-await for readability and maintainability; only use raw callbacks after profiling shows the promise overhead is actually a bottleneck, which, in almost all cases, is not.

Now that you understand what promises and callbacks are all about and, more importantly, the differences between them, you can examine the main anti-patterns that arise when you deal with those structures.

Understanding the callback anti-patterns

Now that you are up to speed on what callbacks and promises are, you can explore some of the problems they pose, so this chapter discusses only the most common anti-patterns in callbacks. Let's start with the most common one.

The pyramid of doom

Probably the most well-known callback problem is the **callback-hell**, which people also call it **the pyramid of doom**. As you saw earlier, callbacks are asynchronous operations, however, sometimes, you need to wait for one of these operations to finish to start another asynchronous operation. This will lead to an increasingly longer nesting of the same callbacks as they wait for one another. For example, reading user data and setting permissions for them:

```
// This could be something like a database
import { readFile } from "node:fs";

function getUser(callback) {
  readFile("./user.json", "utf8", callback);
}

function getSettings(userId, callback) {
  readFile(`./settings-${userId}.json`, "utf8", callback);
}

function getPermissions(groupId, callback) {
  readFile(`./permissions-${groupId}.json`, "utf8", callback);
}

getUser((err, userData) => {
  if (err) return console.error("User read error:", err);
  const user = JSON.parse(userData)
  getSettings(user.id, (err, settingsData) => {
    if (err) return console.error("Settings error:", err);
    const settings = JSON.parse(settingsData)
    getPermissions(settings.user.groupId, (err, permissionsData) => {
      if (err) return console.error("Permissions error:", err);
      console.log("Loaded:", { userData, settingsData, permissionsData });
    });
  });
});
```

In this example, you need to fetch the permissions for a user, however, these permissions exist within a settings object that you can access only if you know the user ID, which means you have a chain of calls that you need to make before being able to fetch the permission.

The pyramid of doom is a bad pattern for several reasons:

- It is difficult to read, especially if there are more levels
- Adding steps to this structure is error-prone and complicated
- To propagate errors, you need to add a handler to every level
- It is hard to extract the logic of individual parts to be reused in other parts
- To test this, you would need to pass the context to all the functions, since they require the context of the previous execution

Unfortunately, the solution to this problem is not super simple. When using callbacks, what you can do, while continuing to use callbacks, is rename and restructure them so that each one lives in its own function, thereby breaking the nesting:

```
function handlePermissions(err, permissionsData) {
  if (err) return console.error("Permissions error:", err);
  console.log("All done!");
}

function handleSettings(err, settingsData) {
  if (err) return console.error("Settings error:", err);
  getPermissions(JSON.parse(settingsData).user.groupId, handlePermissions);
}

function handleUser(err, userData) {
  if (err) return console.error("User error:", err);
   getSettings(JSON.parse(userData).id, handleSettings);
}

// Single call
getUser(handleUser);
```

This approach reduces the nesting level to zero, because you need only to make a single call to trigger all other calls, plus it makes the functions reusable in case you need to fetch user data for other purposes, for example. However, it is still a problem if you want to make it simpler to

read. Then, the only option you have is to move away from callbacks altogether and start using promise chains:

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

readFile(./user.json, "utf8")
  .then((raw) => readFile(`./settings-${JSON.parse(raw).id}.json`, "utf8"))
  .then((raw) =>
    readFile(`./permissions-${JSON.parse(raw).user.groupId}.json`, "utf8"),
  )
  .then((permissionsData) => {
    console.log("Loaded everything:", permissionsData);
  })
  .catch((err) => {
    console.error("Something went wrong:", err);
  });
```

Which makes the code way easier to read, but still you would have the problem of passing the value to the global scope, for this you could use `async/await`, which the language originally introduced for this purpose:

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

async function loadAppData() {
  try {
    const user = await readFile(`./user.json`, "utf8");
    const settings = await readFile(`./settings-${JSON.parse(user).id}.json`,
"utf8");
    const permissions = await readFile(
      `./permissions-${JSON.parse(settings).user.groupId}.json`,
      "utf8",
    );

    console.log({ user, settings, permissions });
  } catch (err) {
    console.error("Failed to load:", err);
  }
}

loadAppData().then(console.log).catch(console.error);
```

To avoid callback hells as much as possible, you need to make a conscious effort to either avoid dependencies between multiple callbacks, or to avoid using callbacks at all if this is possible in your situation.

Taming Zalgo

If you have been around Node.js for a long time, you are probably familiar with the concept of **Releasing Zalgo**, an analogy proposed by *Isaac Schlueter* in his blog post titled *Designing APIs for Asynchrony [2]*. The original article is from 2013, so there are a lot of improvements that have been made between then and now, of course, but the main idea is still valid.

In this article, he describes a bad practice named after *Zalgo*, a text effect often applied to horror stories to make them seem glitchy in a way, as shown in the following figure:

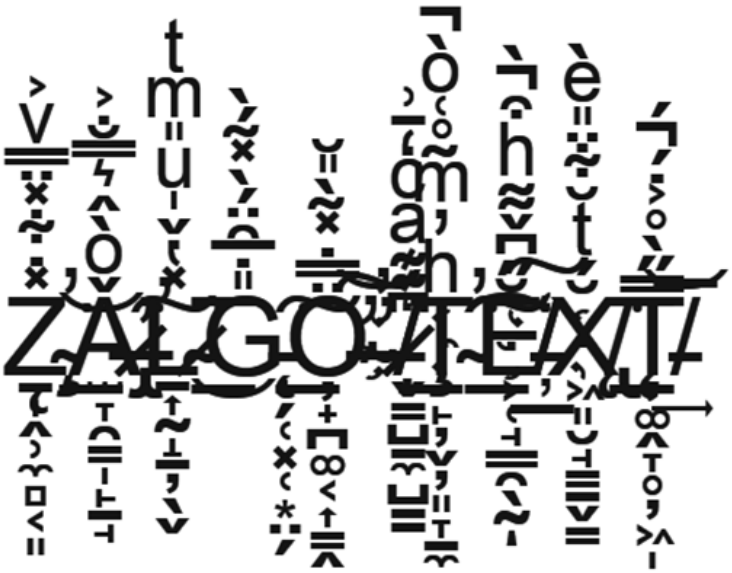


Figure 5.2: Zalgo text

> **Note**
>
> Zalgo is often applied to horror stories, depicting some creature that is terrible beyond comprehension, in this case, the analogy makes the point that this pattern is very bad in general and can cause massive problems to your application.

However, what is Zalgo? The main point of that article is avoiding a single anti-pattern: *Callbacks that are unpredictably sync or async*. Even though this section is included under

callbacks, this anti-pattern can be applied to *all* the asynchronous structures JavaScript has. The main advice both from the article and from my experience is: *choose a pattern and stick to it throughout the entire application*.

Ideally, if you have any API that takes a callback and you cannot say for sure if that callback will be called synchronously or asynchronously or, even worse, *sometimes* it is indeed called synchronously, and some other times it is called asynchronously, then the code this API relies on will be impossible to reason about. This behavior is what *releasing Zalgo* means.

Note

The Zalgo anti-pattern pairs up nicely with another bad practice discussed later, when you have multiple executions of the same callback. So, keep it in mind!

Let's make this simpler using an example. The `Array.prototype.forEach` is a *Zalgo-safe* function because it will `always` call the callback immediately before finishing itself:

```
const someArray = [1, 2, 3];
someArray.forEach((i) => console.log(i * 2));
```

In this example, you can be *completely sure* that the callback you are providing will only be called during the execution of the function, only once per item and then no more. This is a *Zalgo-safe sync function*. An example of an asynchronous function that is also Zalgo-safe is `setTimeout` (and the likes of it, such as `setInterval` and `setImmediate`):

```
setTimeout(() => console.log("this will run some time in the future"), 1000);

// we continue to do our work here
```

`The setTimeout` function *always* runs the callback you pass to it *after* the function has ended. You first call `setTimeout;` the function registers the callback, ends, then it runs the callback you passed to it. This is an example of a *Zalgo-safe async function*. So, the problem is not about having multiple functions, some of them that return after the call ends, or some of them that return before the call ends. It's about having *a single function that behaves like both*.

Below is an example of a *Zalgo-unsafe* function:

```
function fetchData(condition, callback) {
  if (condition === "cache") {
    return callback(null, "Called synchronously");
  }
```

```
  setTimeout(() => {
    callback(null, "Called asynchronously");
  }, 1500);
}
```

Can you see why? In this case, the function behaves differently depending on the input parameters. If you set the condition to cache, then the function will return directly and you will get the response as soon as the function ends, but if not, this function will return and then will execute your callback. This behavior can be complex to reason about because you cannot, in any condition, understand whether the code will behave like a synchronous function or not, especially under real-world conditions.

Think of it in real-world terms: imagine fetchUser reads from an in-memory cache when the value is present, but falls back to a database query otherwise:

```
function fetchUser(id, callback) {
  if (cache.has(id)) {
return callback(null, cache.get(id)); //sync
  }
  db.query(id, callback); //async
}
```

From the consumer's point of view, there is no way to tell whether the result will be synchronous or not without looking into the source code. Also, sometimes the result is available immediately, sometimes it is not. If you move this example to a production-ready application where the condition can vary depending, let's say, on a database column, it will be nearly impossible to understand what the behavior is unless one debugs the code for a very long time.

The solution to this? *A choice*. You should choose one of the methods and stick to it until the end. You can take the fetch-data example and make it completely asynchronous, so you will *always* return immediately after the function is called. In this case, your callback will *always* be scheduled to run after the function call using `process.nextTick`:

```
function fetchData(condition, callback) {
  process.nextTick(() => {
    if (condition === "cache") {
      return callback(null, "Called synchronously");
    }
```

```
    setTimeout(() => {
      callback(null, "Called asynchronously");
    }, 1500);
  });
}
```

Now if you call it the same way you did before:

```
let result = null;

fetchData("cache", (err, data) => {
  result = data;
});
console.log(result); // logs null

fetchData(null, (err, data) => {
  result = data;
});
console.log(result); // logs null
```

The error will not occur because `process.nextTick` will schedule the execution of the function body to be executed on the next tick of the event loop, no matter what the condition is. You do not need to wrap the entire function like that either, just the sync branch of the function:

```
function fetchData(condition, callback) {
  if (condition === "cache") {
    return process.nextTick(() => callback(null, "Called synchronously"));
  }

  setTimeout(() => {
    callback(null, "Called asynchronously");
  }, 1500);
}
```

Obviously, the choice does not have to be asynchronous all the time; you should spot this pattern and realize whether it makes sense for the returns to be either one or the other based on the type of function you are executing. In this particular case you are running a data fetch function where one of the branches will always be sync and another will always be async, but you don't have control over the asynchronous branch like you have over the synchronous one,

so it's easier to transform the synchronous branch into an asynchronous one than the other way around.

Let's check another anti-pattern that is related to consistency.

Multiple execution

The main bad practice of any callback-based code is to break the so-called **single-call-contract**. In other words, a callback-based API can execute its callback only once.

A very simple example of this is when you have multiple async operations that rely on the same callback to execute. For example, you can create a function that has a timeout operation:

```
function withTimeout(callback) {
  const timeout = setTimeout(() => {
    callback(new Error("Operation timed out"));
  }, 1000);

  // This is simulating any other asynchronous operation
  setTimeout(
    () => {
      callback(null, "some result");
    },
    Math.random() > 0.5 ? 1500 : 500,  );
}
```

If you call it like this:

```
withTimeout((err, result) => {
  if (err) return console.error(err);
  console.log({ result });
});
```

You cannot guarantee that the callback will be called only once, actually, you can guarantee that the callback will be called at least twice. Both the timers are triggering the callback. To solve this, you need to make sure that the callback is never called twice, with promises you could leverage `Promise.race`, or even `Promise.all, but` with callbacks, this is not possible. You need to implement a semaphore:

```
function withTimeout(callback) {
  let called = false;

  const safeExecution = (...args) => {
```

```
    if (called) return;
    called = true;
    callback(...args);
  };

  // timeout timer
  const timeout = setTimeout(() => {
    safeExecution(new Error("Operation timed out"));
  }, 1000);

  // Some async operation
  setTimeout(
    () => {
      clearTimeout(timeout);
      safeExecution(null, "result");
    },
    Math.random() > 0.5 ? 1500 : 500,
  );
}
```

In this context, the callback will only ever be called once because it is impossible for it to be called twice. If you time out, the operation will return, even if the operation is completed later. The `safeExecution` wrapper will not call the callback because the `called` semaphore will be set to `true`.

Promise combinators

When you need to coordinate several promises instead of writing a manual semaphore, you can reach for the built-ins:

```
// All or nothing, rejects as soon as the first rejects
const [user, settings] = await Promise.all([getUser(), getSettings()]);

// never rejects, returns an object with all the statuses
const results = await Promise.allSettled([getUser(), getSettings()]);

// returns the same result as the first promise
const first = await Promise.race([fetchData(), timeout(1000)]);

// resolve the first fulfillment
const firstSuccess = await Promise.any([mirror1(), mirror2()]);
```

Hanging operation

When you never call a callback, it's the opposite problem from the previous anti-pattern. The function call will leave the caller hanging forever, which can lead to memory leaks because the function call is never removed from the scheduler. If this process gets repeated multiple times, then the same callback is just added but never taken away.

This usually happens when you have flags in place to activate or deactivate a code path, for example:

```
const pending = new Set();

function operation(someFlag, callback) {
  pending.add(callback);
  if (someFlag) {
    pending.delete(callback);
    return callback(null, "Operation succeeded");
  }
}

operation(false, (err, result) => {
  if (err) return console.error(err);
  console.log(result); // this is never called
});
```

In this case, the callback will never be invoked because the flag is always false. Which means that the operation will build a continuous stack of possible function returns that will never be removed from the scheduler. You can see this in action when we execute this operation multiple times:

```
for (let i = 0; i <= 100_000_000; i++) {
  operation(false, (err, result) => {
    if (err) return console.error(err);
    console.log(result); // never called
  });
if (i % 100_000 === 0) await new Promise(setImmediate);
}
```

You can then monitor the usage of memory either by using a memory monitor, such as `htop` or `btop` (or the common `top`), but you can get more information if you use Node itself to introspect the code, you can add this to the top of the file:

```
setInterval(() => {
  const memoryUsage = process.memoryUsage();
  console.clear();
  process.stdout.write("Memory Usage (in MB):\n");
  process.stdout.write(`RSS : ${memoryUsage.rss / 1024 / 1024} MB\n`);
  process.stdout.write(
    `Heap Total  : ${memoryUsage.heapTotal / 1024 / 1024} MB\n`,
  );
  process.stdout.write(
    `Heap Used   : ${memoryUsage.heapUsed / 1024 / 1024} MB\n`,
  );
  process.stdout.write(
    `External    : ${memoryUsage.external / 1024 / 1024} MB\n`,
  );
  process.stdout.write("\n\n");
}, 300);
```

Now, run the code and keep watching the "Heap Used" line, you will see that the value keeps increasing:

```
Memory Usage (in MB):
RSS         : 59.6484375 MB
Heap Total  : 5.24609375 MB
Heap Used   : 3.7477951049804688 MB
External    : 1.4014978408813477 MB
```

The solution? Simply make sure that a callback will be *always* called, no matter what the operation does:

```
function safeOperation(flag, callback) {
  if (flag) {
    return callback(null, "Operation succeeded");
  }
  return callback(new Error("Operation failed"));
}
```

A small application provides one example of how it can have a huge impact on the application performance and execution if it does not stop running. Even if this takes months, eventually the resources will be depleted.

The event loop explains why it is important to know how the event loop behaves internally and how to avoid long-running memory leaks as much as possible.

Also, you can use the native AbortController for operations that should be abandoned after a timeout, instead of rolling out your own flag:

```
const controller = new AbortController();
const timeout = setTimeout(() => controller.abort(), 1000);

fetch(url, { signal: controller.signal })
  .then(res => res.json())
  .catch(err => {
    if (err.name === "AbortError") console.error("timed out");
  })
  .finally(() => clearTimeout(timeout));
```

Most Node APIs (and JavaScript too) accept a signal option as a parameter, replacing a manual timeout/callback.

Loss of context

Moving from executions to contexts, you encounter one of the most common problems in JavaScript in general, but it's slightly more annoying inside callbacks than in other places: **loss of context**.

The context object (more commonly known as `this`, in JavaScript), is very important in general, but it is even more important when you are dealing with callbacks and classes because when the application executes the callback, it will rely on the context that is passed to it, and if this context is not there, you will have missing information and, most likely, errors.

Consider this simple example, what do you think the result will be:

```
class Greeter {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello ${this.name}`);
```

```
  }
}

const greeter = new Greeter("John");
setTimeout(greeter.greet, 1000);
```

The `setTimeout` call passes greeter.greet without its context and uses a 1000-millisecond delay. If you run this code, you will get `Hello undefined`. Why? Because the context object is not passed on callbacks, every callback is running in its own realm with its own set of contexts and variables. If you just reference a function pointer in the callback of a `setTimeout` function, the implicit context it will use is that of `function () {}` that is not a *lexical context*, but a *local context*, local contexts will *always* get its `this` value from the context that's directly above it:

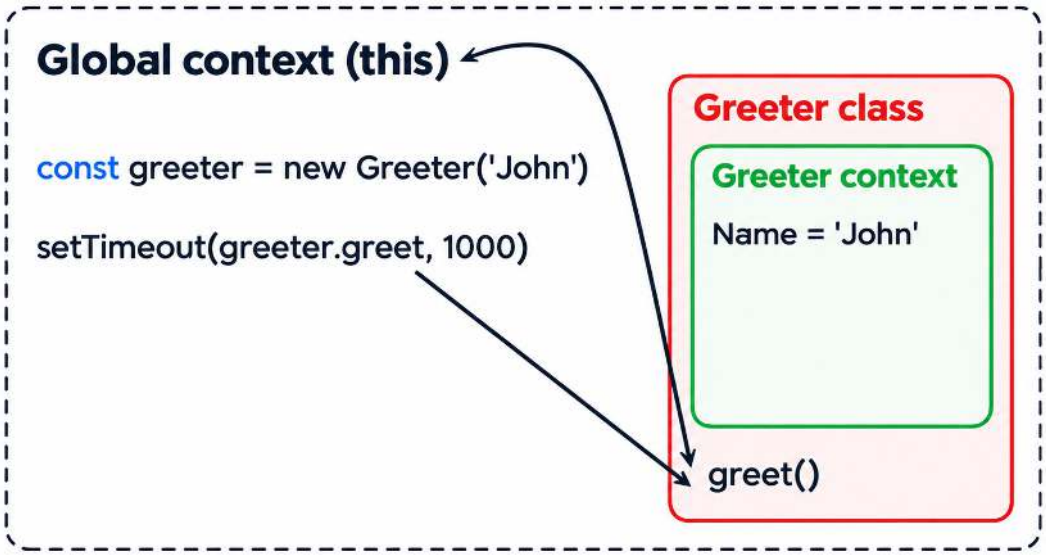


Figure 5.3: Loss of context with global contexts

Essentially, what you are doing when you run the preceding code is this:

```
class Greeter {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello ${this.name}`);
```

```
  }
}

const greeter = new Greeter("John");
setTimeout(greeter.greet.bind(this), 1000);
```

However, how can you pass other objects from outside the realm that the callback is running? You can, of course, reference other variables from outside the realm, for example, you could pass a variable to the `greet` function instead:

```
class Greeter {
  greet(name) {
    console.log(`Hello ${name}`);
  }
}

const person = "Lucas";
const greeter = new Greeter();
setTimeout(() => greeter.greet(person), 1000);
```

This will work as you expect, printing `Hello Lucas` because, when you run the callback, the reference to the `person` variable is passed to the callback context from the outside, and it's not inferred by the local context, but you are actively passing the person object to the function, which is a good example of dependency injection, but you need to instantiate the person before and send it to the class, which breaks the idea that the greeter holds all the state for that action.

For it to work as you originally intended, with all the state internalized in the Greeter class, one option is to manually pass the `this` context from the `greeter` instance, essentially forcing the `this` parameter to be the greeter instance:

```
class Greeter {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello ${this.name}`);
  }
}
```

```
const greeter = new Greeter("John");
setTimeout(greeter.greet.bind(greeter), 1000);
```

Or you can just wrap the call in a function, this way the compiler will pass the correct context to the callback:

```
class Greeter {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello ${this.name}`);
  }
}

const greeter = new Greeter("John");
setTimeout(() => greeter.greet(), 1000);
```

The wrapper function will also work, because the method call will ask for the context that the wrapper then provides lexically by the wrapped function.

Contexts can be tricky to handle, especially when they are executed within async blocks because they often do not produce clear error messages. In fact, the message is often something like:

```
TypeError: Cannot read properties of undefined (reading 'x')
```

Which is why it's super important to understand how to deal with contexts in callbacks. In short, if you're executing a function call that relies on the context, it's a good practice to just identify the context where the call should get its values from, either with a `bind` call or with a function wrapper.

Let's now move from callbacks to promises.

Understanding the promise anti-patterns

Callbacks were a well-defined pattern in the past, but they have since given way for a more modern, easy-to-write promise API. Even though promises are considered to be the best option for writing asynchronous operations as of today, they still have their own quirks.

Let's examine them now.

The pyramid of doom (again)

Even with promises, the pyramid of doom can still appear, just in a slightly different outfit. Instead of nesting callback functions, developers might fall into the trap of chaining too many `.then()` blocks in a deeply nested structure, especially when error handling is done at a chain level.

Take the following example:

```
readFile("user.json", "utf8")
  .then((user) => {
     return readFile(`settings-${JSON.parse(user).id}.json`, "utf8").then(
      (settings) => {
      return readFile(
        `permissions-${JSON.parse(settings).user.groupId}.json`,
        "utf8",
      ).then((permissions) => {
        console.log({ user, settings, permissions });
      });
    });
  })
  .catch((err) => {
    console.error("An error occurred:", err);
  });
```

This is slightly better than callbacks, but it still suffers nested chains and makes the code harder to read, test, and debug. The solution is the same as before: use `async`/`await` to flatten the structure:

```
async function loadAll() {
  try {
    const user = await readFile("user.json", "utf8");
    const settings = await readFile(
      `settings-${JSON.parse(user).id}.json`,
      "utf8",
    );
    const permissions = await readFile(
      `permissions-${JSON.parse(settings).user.groupId}.json`,
      "utf8",
    );
```

```
    console.log({ user, settings, permissions });
  } catch (err) {
    console.error("Failed to load data:", err);
  }
}
```

The takeaway here is: even when using promises, do not fall back into callback-style thinking. Always look for ways to flatten and simplify your control flow.

Dangling Promises

One of the easiest patterns to spot (even easier than the pyramid of doom) is the **dangling promise pattern**. A dangling promise is a promise that's executed without registering either an error or a resolution handler, in other words, it is a promise executed without a `.then` or `.catch` handler. The problems of dangling promises are many, but this section will focus on the most common ones.

Most of this can be caught automatically before code review by using linters, such as ESLint with rules like no-floating-promises, and so on.

Uncaught errors

The main problem you have with dangling promises is uncaught errors. Take this code, for example:

```
async function risky() {
  throw new Error('Oops');
}
risky(); // No await, no catch
```

Since Node.js 15, an unhandled rejection like this prints the error and crashes the process. In older versions, or when a framework swallows the rejection, nothing is shown at all, which makes debugging a nightmare.

To mitigate dangling errors, you can, of course, put a .catch in the risky() call, but what if you have a huge application that cannot be checked fast enough, you should create a handler for the `unhandledRejection` event, which is fired every time Node cannot find a proper catch clause for a promise. So, this piece of code should be in your main file in every app:

```
process.on('unhandledRejection', console.error)
```

With that, you are sure to, at least, log all the errors to the console. However, you should throw the exception or treat it properly according to your own application logic.

Memory leaks

More than simple error messages being swallowed, dangling promises are objects that are still being referenced, even though they might not be in use anymore. This causes a major memory leak. For example, you can imagine you have a database and you open your connection like this:

```
function getConnection() {
  return db.connect(); // Returns a promise
}

function handler() {
  getConnection(); // Forget to await or close
}
```

You will never close the connection, and every time you call the handler function, a new connection will be opened without the previous one being closed. The build-up of these objects can take a toll on the computer's memory over time. Fortunately, the solution is simple. You can either chain a `.finally` call to your promise:

```
function handler() {
  getConnection()
   .then(() => ...)
   .finally(async () => await db.close())
}
```

Or, you can wrap the whole method in a `try/finally` directive:

```
async function handler() {
  const conn = await getConnection();
  try {
    // do stuff
  } finally {
    conn.close();
  }
}
```

If you have a transpiler (such as such as Babel or TypeScript) you can also take advantage of JavaScript's explicit resource management feature, which, as of the writing of this book, is not widely available yet.

This feature allows you to take full control of the resource disposal of any object in JavaScript if you have either a `Symbol.dispose` or `Symbol.asyncDispose` method in your structure. In this case, since the function is async, you can implement the `asyncDispose`:

```
async function getConnection() {
  const dbConn = await db.connect()
  return {
    connection: dbConn,
    [Symbol.asyncDispose]: async () => {
      await dbConn.close(); // Your disconnection logic here
    }
  }
}
```

Now, you can use it like this:

```
async function handler() {
await using conn = await getConnection();
  // logic
}
```

What the preceding code does is to automatically call the `asyncDispose` method once the `handler` function reaches its end, so you do not need to do the disposal yourself.

Understanding Async/Await

The **async/await syntax** first appeared in *ECMAScript 2017* as a way to write asynchronous code that looks synchronous. The syntax provides cleaner syntax and is easier to reason about than chaining multiple `.then()` and `.catch()` blocks.

There are a few ways you can use async/await, one of them is to combine `return` with `await`, which makes the promise in question be resolved within the function block you are executing it on, forcing catch blocks to handle errors that are thrown by those promises. In other words, `return await` makes the result of the promise be delegated to the function that is executing it, the callee, and not the caller. However, most of these use cases are not fully understood and `return await` is widely used where it should not.

For example, here:

```
async function loadData() {
  return await readFile("user.json", "utf8");
}
```

Even though it works, `await` here will delegate the result of the function to the `loadData` function, which has no catch blocks, so this error will be propagated up, which is the same behavior as if you were calling it without the `return await` and just with `return`.

Also, be careful when using `await` inside loops. A very common mistake is trying to use `await` inside a `.forEach()` loop, which won't behave the way you expect. That's because `forEach` doesn't handle asynchronous functions properly and will not wait for the `await` to finish. Consider this code:

```
const files = ["a.txt", "b.txt", "c.txt"];

files.forEach(async (file) => {
  const data = await readFile(file, "utf8");
  console.log(data);
});
```

Despite the use of `await`, the loop does not actually wait for the operations to finish before continuing, which might lead to confusing behavior or missed outputs.

Instead, always use a `for...of` loop when you need to `await` inside:

```
for (const file of files) {
  const data = await readFile(file, "utf8");
  console.log(data);
}
```

This pattern ensures that each asynchronous operation completes before the next one begins, preserving the expected sequence and timing of your logic.

Wrong promisification

After the advent of promises, it became increasingly common to see code like this:

```
import { readFile } from "node:fs";

function readFileAsync(path) {
```

```
  return new Promise((resolve, reject) => {
    readFile(path, "utf8", (err, data) => {
      if (err) return reject(err);
      resolve(data);
    });
  });
}

readFileAsync("myFile.txt")
  .then((contents) => console.log(`File contents: ${contents}`))
  .catch((err) => console.error(`Error reading file: ${err}`));
```

This is called **promisification**, which is a technique that aims to convert a callback-based API into a promise-based one. The main idea is to wrap the callback-based function into a new promise only allowing for parameters to be passed inside while controlling the resulting callback invocation. When the callback is invoked, then the error is either passed to `reject` or the successful result is passed to `resolve`.

> **Note**
>
> Promisification can be considered a design pattern because it solves a very common problem faced by most people when using APIs that span a large time gap, such as older callback-based APIs alongside newer promise-based APIs.

This example uses the `fs` library for simplicity, but `fs` already has its own promise-based API under `fs/promises`. This type of solution is so common that Node.js has included a shorthand called `promisify` in the native library under `util`:

```
import { promisify } from "node:util";
import { readFile } from "node:fs";
const readFileAsync = promisify(readFile);
```

Nevertheless, there are many possible problems that come with wrong promisifications. The following sections examine some of them.

The never-caught expression

Consider this promisification of the `fs` module:

```
function readFileAsync(path) {
  return new Promise((resolve, reject) => {
```

```
    fs.readFile(path, "utf8", (err, data) => {
      if (err) throw err;
      resolve(data);
    });
  });
}
```

You may think there is nothing wrong with this, but the throw happens inside the fs.readFile callback, which runs later, outside the promise constructor. The promise never rejects, and the error becomes an uncaught exception that bypasses catch completely. (A synchronous throw directly inside the executor would be turned into a rejection.)

To fix this, *never* throw inside a promise constructor, use `reject` instead:

```
function readFileAsync(path) {
  return new Promise((resolve, reject) => {
    fs.readFile(path, "utf8", (err, data) => {
      if (err) return reject(err);
      resolve(data);
    });
  });
}
```

Promisification of events

When dealing with event-based code, you might feel compelled to do this:

```
function onDataEvent(emitter) {
  return new Promise((resolve, reject) => {
    emitter.on("data", (data) => resolve(data));
    emitter.on("error", reject);
  });
}
```

However, `data` events may resolve multiple times, so the callback will be called over and over again. In a promise, a `resolve` can only be called once; the first resolve will remain while the others may cause side effects. To mitigate this problem, use the `once` helper from Node's `events` library:

```
import { once } from "node:events";
await once(emitter, "data");
```

Repromise

Another common mistake is wrapping a function that already returns a promise in a new promise. This practice, called **repromising,** is unnecessary and harmful and can take two forms.

The first is when you add a promise constructor on top of an already promisified function, such as `readFile`:

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

function repromiseFile(path) {
  return new Promise((resolve, reject) => {
    readFile(path, "utf8").then(resolve).catch(reject);
  });
}
```

Or if you use `async`, which **already returns a promise**, and then add the promise constructor to the already promisified function, this gives you two promise objects with a third implied promise layer (from the async function):

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

async function repromiseFile(path) {
  return new Promise((resolve, reject) => {
    readFile(path, "utf8").then(resolve).catch(reject);
  });
}
```

Remember that a promise, above all else, is still a JavaScript object, so wrapping a promise in another promise creates extra promise objects that have to be resolved one after the other, which introduces additional microtasks and unwrapping logic just to flatten them out to the final promise, which costs memory, CPU cycles, and introduces scheduling overhead in the event loop.

Resolving this is simple: *do not double (or triple) promise*. If you see an `async` function, there is no need to return yet another promise, just return the original promise:

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

async function repromiseFile(path) {
```

```
  return readFile(path, "utf8");
}
```

More than that, repromising includes severe performance issues. If you have a repetitive operation that keeps repromising, you will eventually end up with a huge resource (and performance) problem that can be measured. Take this code:

```
import { readFile } from "node:fs/promises";
import { performance } from "node:perf_hooks";
import { setTimeout as delay } from "node:timers/promises";

function repromise(path) {
  return new Promise(function innerPromise(resolve, reject) {
    readFile(path, "utf8").then(resolve).catch(reject);
  });
}

async function benchmark(label, fn) {
  const start = performance.now();
  for (let i = 0; i < 100_000; i++) {
    await fn("./sample.txt");
  }
  const end = performance.now();
  console.log(`${label}: ${(end - start).toFixed(2)}ms`);
}

benchmark("Repromised", repromise);
benchmark("Direct", readFile);
```

Next, write this to `repromise.js` and then create a sample file containing only `Hello World` in the same directory and run the code above using the `clinic.js` performance tool:

```
npx clinic flame -- node repromise.js
```

The command generates a profile summary in HTML. Open this file in your browser and compare the time spent in garbage collection function calls and event loop overhead between the two calls. You should see things such as:

- Extra frames due to promise constructor overhead

- Redundant microtask queue queries
- Longer event loop blocks causing by this double wrapping logic

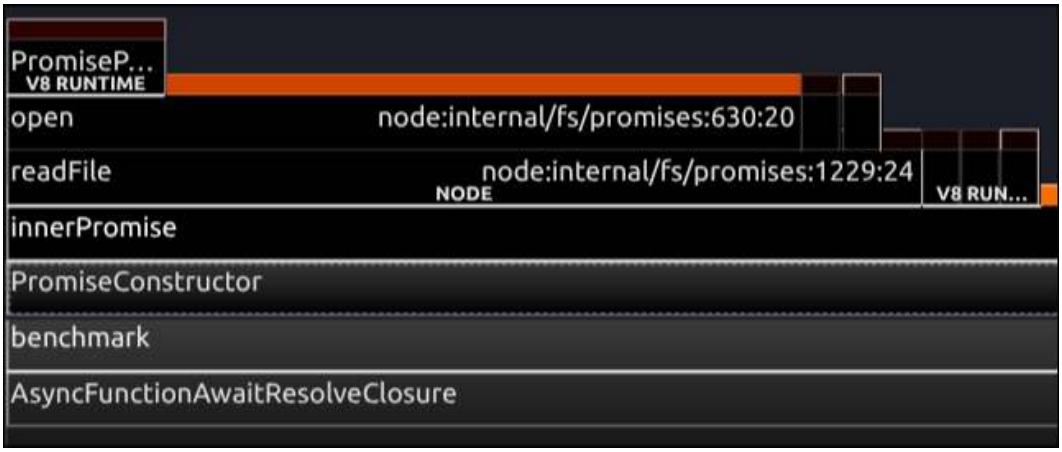


Figure 5.4: Repromised call benchmark

See that you have an `innerPromise` constructor on top of the Promise constructor, and just then you have the `readFile` call and the internal open call. All of that is topped by the `PromisePrototypeFinally` (that's cut off in the picture) that is always added. So, you are adding one extra function call on top of the already existing promise call.

Now, you can uncomment the `direct` execution and comment out the `repromised` one and execute again. When you compare both graphs, the direct version will have more consistent (and compact) flame patterns, which indicates a smaller resource footprint. Check it out:

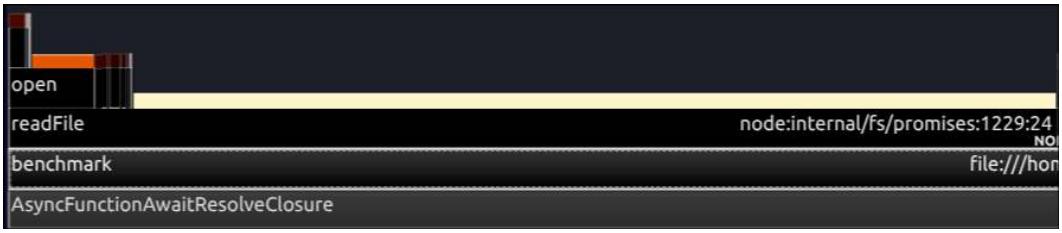


Figure 5.5: Direct call benchmark

Without the extra promise wrapping up the function, the call goes straight to the `readFile` native function. Removing two calls off the stack per run of your function. Now, if you run this function 100,000 times, for example, this means you will have at least 200.000 fewer stack allocations, which makes the code faster and less resource intensive. All this just by removing the extra new Promise wrapper.

Under a small load like this, the performance seems negligible, but on a larger scale, this performance overhead becomes a problem.

Summary

Asynchronous programming is central to Node.js development. It is powerful, flexible, and essential for building performant systems, but it's also filled with pitfalls. In this chapter, you've seen how callbacks work and the common mistakes when using them, such as callback hell, multiple execution, and loss of context. You have also explored how Zalgo – the unpredictability of sync versus async execution – can haunt your code if you're not careful. Promises, while more elegant, are not bulletproof; nested .then() chains, unnecessary awaits, and bad promisification introduce their own set of anti-patterns. You also looked at how dangling promises can become a huge problem and how repromising often creates more harm than good.

In the end, writing asynchronous code well is about clarity, predictability, and discipline. Do not try to be clever, try to be consistent. Favor `async/await` when possible, avoid mixing patterns, and always aim for code that reads like a story. If you can read it out loud and it makes sense, you are doing it right.

Here is an async anti-pattern checklist so you know you're good to go:

- Is this callback guaranteed to run exactly once?
- Can this function behave both sync and async?
- Are all promise rejections handled (.catch or try/catch)?
- Are you using async/await instead of nested .then() chains?
- Are you using await inside .forEach() by mistake?
- Are you wrapping an already-promise-returning function in new Promise()?
- Are resources (connections, handles) closed in finally?

Now that you know what makes up the asynchronous high-level structures of Node.js, it is time to dive in deep on how they work, you will now enter the incredible realm of the event loop.

Let's keep taming the async beast.

References

- [1] `https://hackernoon.com/nodejs-48x-faster-if-you-go-back-to-callbacks`
- [2] `https://blog.izs.me/2013/08/designing-apis-for-asynchrony/`

6

Logging Done Right

Logs are the first thing you reach for when something breaks in production. They are also, somehow, the last thing most developers think carefully about when building a system. The result is predictable: when the 3 a.m. alert fires and you need to figure out why the payment service is returning 500s, you open your logging dashboard and find either a wall of undifferentiated noise or, worse, almost nothing useful at all.

This chapter is about the gap between those two outcomes. We will not be going through library documentation because you can read that yourself. The goal here is for you to understand how logging works at the Node.js level, why the tools people reach for first (and most often) are the wrong ones for production, and what the patterns look like that turn your logs from noise into real information.

In this chapter, we will cover:

- How `console.log` works internally, and why its blocking behavior can hurt you in unexpected ways
- What the `debug` package is for, and the anti-pattern of using it as a production logger
- The major production logging libraries, with a focus on Pino and why its architecture makes a difference at scale
- Log levels: what they mean, and the most common ways developers misuse them.
- Structured logging and why plain text logs are an anti-pattern in any serious deployment
- What to log, what to never log, and the real-world consequences of getting that second category wrong
- How to calibrate log volume so you are neither blind nor drowning
- Where logs should go according to the Twelve-Factor App methodology

- How to preserve the full error context when things go wrong
- What observability means for a Node.js application and how logging fits into it

Technical requirements

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

console.log is not a logger

Every Node.js developer knows `console.log`. It requires no installation, it is globally available, and it gives you immediate feedback in your terminal. For debugging a value or tracing through a function during development, it is exactly the right tool.

You have seen `console.log` sprinkled throughout this book already, the same way it gets sprinkled throughout real codebases. That is fine while you are writing and debugging. The problem is when it stays there.

Understanding why requires a look at what actually happens when you call it.

What console.log is really doing

When you call `console.log('hello')`, three things happen in sequence. First, `util.format()` processes the arguments, handling `%s`, `%d`, and `%j` format specifiers and calling `util.inspect()` on any non-string values. Second, a newline is appended to the result. Third, `process.stdout.write()` is called with the formatted string.

`console.error()` and `console.warn()` do the same thing but write to `process.stderr` instead of `process.stdout`.

That is basically it. There is no timestamp attached, no severity level recorded, no structured format, no routing to different destinations, and no redaction of sensitive values. You get a string, written to a file descriptor.

The thing that makes this more complicated than it looks is the behavior of `process.stdout` itself. Node.js documentation includes a table that most developers have never seen:

Destination	Windows	POSIX (Linux/macOS)
Files	Synchronous	Synchronous
TTYs (terminals)	Asynchronous	Synchronous
Pipes and sockets	Synchronous	Asynchronous

On a POSIX system, when stdout is connected to a terminal (which is the case when you run `node app.js` directly in your shell), every `console.log` call blocks the event loop until the write completes. Node explicitly sets the file descriptor to blocking mode in the source code in `lib/tty.js` via `this._handle.setBlocking(true)` for exactly this reason. The comment in the source says it prevents interleaved or dropped output on terminals when you have multiple calls writing to the same place.

When you are running locally on your laptop with a fast terminal, this is imperceptible. Under load in production, or on a slower or more constrained machine, it is a different story.

The pipe problem

The more dangerous behavior shows up when stdout is piped, for example, in a container where logs are captured, or when you run `node app.js | jq` to inspect JSON output. On POSIX systems, piped writes are asynchronous. The data goes into a kernel pipe buffer and the event loop continues without waiting for it to drain.

That buffer has a hard limit. On Linux, the pipe capacity is 65,536 bytes (16 pages × 4,096 bytes). If your process calls `process.exit()` before the buffer has been flushed, any data in that buffer is silently lost.

You can check this for yourself with this small code piece. Let's write this to a file named `script.js` anywhere on your machine:

```
process.stdout.write('@'.repeat(128 * 1024));
process.exit(0);
```

Run the following code with `node script.js | wc-c` on any POSIX system, such as a Linux distribution or even macOS.

```
$ node script.js | wc -c
65536
```

You wrote 131,072 bytes and only 65,536 arrived. Half the data vanishes without any error. This behavior is documented in the Node.js GitHub repository, in issues #7743, #6456, and #11568, but most people will only discover it when they notice truncated logs in their production containers (which was my case).

You might think this only matters for CLIs that pipe their output into other programs. It does not: a container runtime captures your stdout through a pipe too, so any production service that logs with console.log and exits abruptly is exposed to the same truncation. And this is not the whole problem with console.

Mind the performance

It is widely known that the `console.log` statement (and its variants) is orders of magnitude slower than logging libraries that are made specifically for logging.

Numbers depend on hardware and payload, but benchmarks with Fastify under load (10,000 requests, 1,000 concurrent) show that adding a single `console.log` call per request reduces throughput by around 51%. Two calls per request brings that to roughly 88%. The overhead comes from multiple layers: `util.format()` serialization, `util.inspect()` for objects, synchronous I/O blocking on TTYs, and that every call is a separate `write()` syscall with no batching.

This does not mean `console.log` is useless. It means that it is a debugging tool that was never designed to be a production logging solution, and that treating it as one creates measurable problems.

My point is that what `console.log` costs you is not worth what it gives you:

- No timestamps on log entries
- No severity levels to filter by
- No structured, machine-readable output
- No way to route messages to different destinations
- No log rotation or file management
- No redaction of sensitive fields before writing

That last point will come up again later in this chapter, with consequences that are harder to measure in milliseconds.

Of course, you can mitigate all these points yourself: log an object instead of a string, add a timestamp and a level property, and strip sensitive fields before writing. But at that point you are rebuilding a logger. Isn't it better to just use a solution that was made for that?

What about `debug`? One of the most famous JavaScript packages in npm? That should do the trick, right?

The debug package is not a production logger

Before we talk about production logging libraries, we need to clear up a common source of confusion. The `debug` package pulls around 480 million weekly downloads and is a dependency of over 56,000 npm packages. If you have ever used Express, Socket.IO, or most other major Node.js frameworks, `debug` is somewhere in your dependency tree. And, somehow, I've seen it being used as a production logging solution more than once.

To make it very clear, `debug` is **not** a logging library in any production sense of the word. You can literally see it in the name: `debug`.

It was created for library authors. Its entire purpose is to give libraries a way to emit diagnostic output that is completely silent by default and can be turned on by the consumer without modifying any library code. You enable it with an environment variable:

```
# Enable all Express internals
DEBUG=express:* node app.js

# Enable specific namespaces
DEBUG=app:server,app:db node app.js

# Enable everything (don't do this in production)
DEBUG=* node app.js
```

Each namespace gets its own output, colored by a hash of the namespace name, with millisecond-precision timing diffs between calls. Output goes to `process.stderr` by default, not stdout. In a TTY it shows relative timing like `+3ms`. In a non-TTY it uses ISO 8601 timestamps.

Express creates debug instances for `express:application`, `express:router`, `express:router:route`, `express:router:layer`, and `express:view`. Running `DEBUG=express:*` reveals the full request lifecycle without touching a single line of Express' code. That is exactly what the package was designed to do.

Using `debug` in your own library code makes sense:

```
import debug from 'debug';
```

```
// Library authors: create namespaced debug instances
const log = debug('mylib:core');
const logHttp = debug('mylib:http');

export function fetchData(url) {
 log('fetching %s', url);
 // ...
  logHttp('response received in %dms', elapsed);
}
```

Consumers of your library who need to diagnose issues can enable `mylib:*` without you needing to add any logging infrastructure. This is the use case `debug` was built for. But the moment you try to use `debug` as your application logger, its design becomes a serious limitation.

There are no log levels. You cannot distinguish a catastrophic error from a routine info message. There is no way to say "show me only warnings and above" because the concept does not exist.

The output is unstructured. A debug message looks like `app:ordersOrder 12345` placed by user `u789 +3ms`. Production systems need `{"level":"info","time":"2026-04-03T10:00:00Z","orderId":"12345","userId":"u789","msg":"Order placed"}`. The first form is impossible to query or aggregate programmatically without writing a custom parser.

There is no transport mechanism. Debug writes to stderr. It has no concept of file output, rotation, shipping to an external service, or anything else you need in a real deployment.

Think of it like: "`debug` is for the library code you publish so that *other* developers can diagnose their usage of your library". Your application code needs something else.

Picking a production logger

Okay, now we know what is **not** a production logger, but what *is* a production logger?

Once you accept that `console.log` is a debugging tool and `debug` is for library authors, the next question is which logging library to use. The Node.js ecosystem has a few options with very different histories and design philosophies.

For most of Node.js history, two logging libraries dominated application code, and they deserve their own part in this book for that.

Winston has been the most downloaded Node.js logging library for years, sitting around 15 million weekly downloads. Its main selling point is flexibility: you can define multiple transports (destinations for your log output), stack custom formatters, define your own log levels, and find a community plugin for almost any logging backend you might need. That flexibility has a cost, though. Winston's architecture runs every log entry through a transform pipeline, which means per-entry overhead that stacks under heavy load. In Pino's own benchmark suite (100,000 log calls), Winston completes in around 270 ms.

Bunyan was one of the first Node.js libraries to push structured JSON logging as a first-class output format, which made it genuinely influential. Pino is, in many ways, its direct spiritual successor. Bunyan still works, but active development has slowed significantly. If you are starting a new project today, Pino does everything Bunyan does, faster and with a more active community. If you are maintaining a codebase that already uses Bunyan, it is worth evaluating a migration, but it is not urgent.

Winston and Bunyan matter in historical context because they shaped the vocabulary and expectations of Node.js logging. Log levels, structured output, named transports. All these patterns came largely from Winston and Bunyan. But for new work, neither is the right choice: a new production service gains more from Pino's speed and JSON-first defaults than from Winston's flexibility, while an existing Winston or Bunyan deployment that works does not need an urgent migration. They were great tools for another time.

Enters Pino

Pino was created by Matteo Collina and David Mark Clements in 2016, announced at Node Summit. Matteo is a Node.js TSC member and co-creator of Fastify, and Pino's design reflects a single constraint: **logging should never become the bottleneck of your application**. It ships as the default logger in Fastify and is the current best answer for production Node.js logging at the time of writing.

In the same 100,000 log call benchmark where Winston takes ~270 ms, Pino completes in ~114 ms, so it is roughly 2.4 times faster. That gap comes mostly from architecture:

```
import pino from 'pino';

const logger = pino();

logger.info({ userId: 'alice', action: 'login' }, 'User authenticated');
logger.error({ err, requestId: req.id }, 'Database query failed');
```

The output is newline-delimited JSON (NDJSON) by default:

```
{"level":30,"time":1775350440627,"pid":
12345,"hostname":"api-1","userId":"alice","action":"login","msg":"User
authenticated"}
{"level":50,"time":1775350440628,"pid":12345,"hostname":"api-1","err":
{...},"requestId":"...","msg":"Database query failed"}
```

`level` 30 is info, 50 is error. Those numbers follow the same RFC 5424-inspired scale used by Bunyan and the proposed `node:logger` core module. But why is Pino so much faster?

There are six specific things Pino does differently that account for its performance.

First, Pino never calls `JSON.stringify` on the full log object. Known fields like `level`, `time`, `pid`, and `hostname` are pre-computed as string fragments at logger creation and cached. Each log call concatenates pre-built strings with dynamic parts rather than serializing a fresh object.

Instead of Node's core `Writable` stream, Pino uses `sonic-boom`, a utf8-only stream implementation that provides roughly 2x faster writes by handling EAGAIN errors gracefully and batching writes where possible.

For string formatting, Pino uses `quick-format-unescaped` instead of `util.format`. This is around 23% faster for basic cases and 52% faster when objects are appended at the end of the message.

When you set `level: 'info'`, Pino replaces the `debug()` and `trace()` methods with actual no-op functions, which is different than using functions that check a flag and return early. The check happens once at level-assignment time. The cost of calling a disabled log level is as close to zero as JavaScript allows.

With the default stdout destination, all processing happens synchronously within the calling tick. There are no promises, no microtask scheduling, no deferred work.

For complex objects that cannot use the fast path, Pino uses `safe-stable-stringify` which handles circular references without throwing. That covers speed. The other thing that makes Pino shine is how it carries context around, and that is where child loggers come in.

The child logger pattern is one of Pino's most useful features. When you call `logger.child({ requestId: 'abc-123' })`, Pino serializes the bindings to a JSON string

fragment at construction time and stores it internally. Every subsequent log call from that child just concatenates this pre-built string.

```
// In your request middleware
app.use((req, res, next) => {
 req.log = logger.child({ requestId: req.id ?? crypto.randomUUID() });
 next();
});

// In any route handler
req.log.info({ orderId: order.id }, 'Processing order');
```

The `requestId` field appears automatically on every entry from that child:

```
{"level":30,"time":1775350444801,"pid":
12345,"hostname":"api-1","requestId":"fc0b87fe-7d7f-4a1d-994a-
fc9bea4197e2","orderId":"order-99","msg":"Processing order"}
{"level":50,"time":1775350444802,"pid":
12345,"hostname":"api-1","requestId":"fc0b87fe-7d7f-4a1d-994a-
fc9bea4197e2","orderId":"order-99","msg":"Order failed"}
```

Notice the same `requestId` on both entries? It was serialized once at child creation and concatenated on every subsequent call without re-serialization.

Pino v7 (released October 2021) introduced `pino.transport()`, which moves log processing to a Worker Thread via the `thread-stream` library. In practice, that means formatting and delivery work (pretty-printing, writing to a file, shipping to a remote service) runs in a worker thread instead of your application's main thread.

```
const logger = pino({
 transport: {
 target: 'pino-pretty',
 options: { colorize: true }
 }
});
```

You can also route to multiple targets with per-target level filtering:

```
const logger = pino({
 transport: {
 targets: [
```

```
  {
  target: 'pino/file',
  level: 'info',
  options: { destination: '/var/log/app.log' }
  },
  {
  target: '@logtail/pino',
  level: 'warn',
  options: { sourceToken: process.env.LOGTAIL_TOKEN }
  }
  ]
  }
});
```

A word on that file target: it is there to show the mechanics. If your infrastructure captures stdout (containers, most PaaS platforms), keep writing to stdout and let it do the routing, as we will discuss in Where Logs Should Go. Writing files from the app only makes sense when nothing else is there to collect them, such as a single VM with no log shipper.

pino-pretty is a development tool

pino-pretty is a development tool: it transforms Pino's ndjson into human-readable colorized output for your terminal. Install it as a dev dependency and keep it out of production, which is also what the official documentation recommends.

In production, raw JSON goes to log aggregators (ELK stack, Datadog, Grafana Loki) that parse and index it natively. Using `pino-pretty` in production adds parsing overhead, defeats the performance advantages, and produces output that log aggregators cannot index. So, enable pino-pretty based on the environment the app is running in:

```
const logger = pino({
 transport: process.env.NODE_ENV !== 'production'
 ? { target: 'pino-pretty', options: { colorize: true } }
 : undefined
});
```

We will come back to environment-specific configuration in more detail in Chapter 7.

Adjust your log levels

Every production logging library implements severity levels. They exist for a simple reason: not all log messages are equally important, and you need a way to filter the noise without losing the important parts.

The standard set, from lowest to highest severity, is:

- `trace` (10): extremely fine-grained diagnostic information. Query parameters, internal state dumps, function entry/exit in hot paths. Almost never appropriate in production because it generates too much noise.
- `debug` (20): information that is useful during development, such as variable values, intermediate computation results, and branching decisions. This is where your temporary console.log('here') calls belong.
- `info` (30): normal operational events: server started, user authenticated, background job completed. These are useful in production because they show the system is working, but use them sparingly.
- `warn` (40): something unexpected happened that is not an error yet but might become one. For example: a deprecated API was called, or a request had to be retried.
- `error` (50): an operation failed. The application is still running but something went wrong that needs attention.
- `fatal` (60): the application is about to go down due to an unrecoverable condition.

In production you typically run at `info` or `warn`, which means `trace` and `debug` messages are completely disabled. In development you run at `debug` or `trace` so you can see everything. When diagnosing a production issue, you temporarily lower the level to get more detail, then raise it again.

The simplest way to make that possible is to read the level from the environment, with a sane default:

```
const logger = pino({ level: process.env.LOG_LEVEL ?? 'info' });
```

With that, LOG_LEVEL=debug node app.js gets you the detail you need without a code change or a redeploy. Combined with the NODE_ENV check you used for pino-pretty, this is all the environment-specific configuration most applications need: pretty output and debug level locally, plain JSON at info in production, and one variable to turn the detail up when something breaks.

Getting levels wrong

The most common mistake is collapsing everything to `info`. I have seen codebases where every single log call goes through `logger.info()`. If everything is info, severity filtering is useless. You cannot write an alert that fires on errors if errors are indistinguishable from basically anything else.

The second mistake is downgrading errors because they were handled:

```
try {
 const user = await db.findUser(id);
 return user;
} catch (err) {
  logger.info('Could not find user, returning null');
 return null;
}
```

The reasoning is usually "well, I handled it, so it's not really an error." But it is an error. Your fallback worked, which is good. But you still need to know that errors are happening; queries are failing. Handled failures still need to be counted and investigated, and an info line will never show up in your error rate.

```
try {
 const user = await db.findUser(id);
 return user;
} catch (err) {
  logger.error({ err, userId: id }, 'Failed to query user from database');
 return null;
}
```

The third mistake is using `warn` as a softer version of `error`. Warn is for things that are not failures but are worth noting, like a configuration value that fell back to its default, or a call to a deprecated method. If the operation actually failed, that is an error regardless of what your fallback strategy was.

A useful rule of thumb:

If you would want to be paged at 3am because of it, it is `error` or `fatal`. If you would want it in a weekly summary but would not wake up for it, it is `warn`. If it is routine operation, it is `info`. If it is only useful when debugging a specific problem, it is `debug` or `trace`.

Structured Logging

If your log output looks like this:

```
Server started on port 3000
User alice logged in from 192.168.1.1
Error processing order 12345: timeout
```

That is perfectly readable when you are tailing output in your terminal during development. The moment you have a centralized logging system, those strings need to be parsed to extract anything useful. Regex-based parsing of free-form log text is fragile, slow to write, and a maintenance burden every time a message format changes (and running those regexes on every line costs CPU in your log pipeline too).

Structured logging means each log entry is a data object:

```
{
 "level": "info",
 "time": "2025-06-15T14:23:04.436Z",
 "pid": 28565,
 "hostname": "api-server-3",
 "msg": "User logged in",
 "userId": "alice",
 "ip": "192.168.1.1",
 "requestId": "req-abc-123"
}
```

Now you can ask it questions: "show me all errors from the last hour", "show me every log entry for request `req-abc-123`". None of that requires a custom parser.

Pino outputs structured JSON by default. The first argument is the data object, the second is the message string:

```
logger.info({ userId: 'alice', ip: req.ip }, 'User logged in');
```

The anti-pattern is building your own ad-hoc structured format using string concatenation or template literals when a structured logger is already available:

```
// Unstructured, unparseable without custom regex
console.log(
  `[${new Date().toISOString()}] [INFO] User ${userId} logged in from ${ip}`
```

```
);

// Structured, immediately queryable
logger.info({ userId, ip }, 'User logged in');
```

What to Log and what not to log

Getting the content of your logs right is harder than the infrastructure around them. Too little and you are blind when things break. Too much and you are burning money on storage while drowning in noise. And logging the wrong things can have consequences that dwarf any performance or cost concern.

What to Always Log

As a baseline, every production application *should* be logging:

- Application startup and shutdown (including configuration values, excluding secrets)
- Incoming HTTP requests: method, path, status code, response time.
- Authentication events: login attempts, successes, failures.
- Authorization failures: when a user attempts something they are not allowed to do.
- Database connection lifecycle: connected, disconnected, pool exhausted.
- External service calls: requests to third-party APIs with response times and status codes.
- Unhandled errors and exceptions with full stack traces
- Business-critical operations: payments processed, orders created, records deleted.

Every log entry should carry enough context to be useful in isolation. A message that says "Database error" is basically useless. The same message with the query, the error, the stack trace, and the request ID that triggered it is actionable:

```
// Almost useless
logger.error('Database error');

// Useful
logger.error({
 err,
 query: 'SELECT * FROM orders WHERE user_id = $1',
 params: [userId],
  requestId: req.id,
```

```
  duration: Date.now() - queryStart
}, 'Order query failed');
```

One caveat on params: query parameters carry whatever the caller passed in, which can include emails, tokens, or other personal data. Only log them when you know what they contain. We will come back to this in What to Never Log.

Correlation IDs

In a system handling concurrent requests, without a way to tie log entries to the request that generated them, your logs are essentially a pile of unordered events with no causal chain. Assign a unique ID to each incoming request and include it in every log entry that request generates. Pino's child logger pattern makes this quite easy to do:

```
import { randomUUID } from 'node:crypto';

app.use((req, res, next) => {
 req.log = logger.child({
    requestId: req.headers['x-request-id'] ?? randomUUID()
 });
 next();
});
```

Then you can use this child logger in every request:

```
// In any route handler, just use req.log
app.get('/orders/:id', async (req, res) => {
  req.log.info({ orderId: req.params.id }, 'Fetching order');

 try {
 const order = await db.getOrder(req.params.id);
    req.log.info({ orderId: order.id, status: order.status }, 'Order
retrieved');
    res.json(order);
 } catch (err) {
    req.log.error({ err, orderId: req.params.id }, 'Failed to fetch
order');
    res.status(500).json({ error: 'Internal server error' });
 }
});
```

Every log entry from that request shares the same `requestId`. When something goes wrong, you can filter your logging system to that single ID and see the complete picture of what happened.

What to never log

This section matters more than the previous one. Logging the wrong data can result in regulatory fines, legal liability, and the kind of headlines nobody wants.

This should be on everyone's mind already, but it cannot be said often enough: **Never log sensitive data.** Passwords, API keys, access tokens, credit card numbers, and so on. None of it should ever appear in a log entry. This is obvious in principle and violated constantly in practice.

The typical failure is an accident rather than a decision. A developer adds `logger.debug(req.body)` during a debugging session and forgets to remove it before committing. Now every login request dumps plaintext passwords into your log aggregation system, where they may be retained for 90 days, replicated across multiple storage backends, and accessible to anyone with read access to your logs.

You might be thinking: "Well, no one would be THIS careless, right?" Let me bring some data. Keep in mind that none of these incidents were deliberate; they came from lack of attention or from rushing a delivery. In general, devs tend to be very careful with this sort of data:

- **GitHub, May 1, 2018**: A bug in the password reset flow wrote passwords to internal logs before hashing was applied. GitHub discovered it during a routine audit and notified affected users. Many recipients initially thought the email was a phishing attempt because the scenario seemed too... Careless... To be real.
- **Twitter, May 3, 2018** (two days after GitHub): Passwords were being written to an internal log before the bcrypt hashing process completed.
- **Facebook, disclosed March 21, 2019**: This was way worse. Hundreds of millions of user passwords had been stored in plaintext on internal servers, searchable by employees, with the data going back to 2012. In September 2024, Ireland's Data Protection Commission fined Meta €91 million under GDPR for this breach.

That is the cost of logging sensitive data. So, how do we prevent it?

First, never log entire request or response objects. Log the specific fields you actually need. Second, implement redaction at the logger configuration level so you are not relying on every developer to remember to exclude sensitive fields at every call site. Pino has built-in redaction:

```
const logger = pino({
 redact: {
```

```
  paths: [
  'req.headers.authorization',
  'req.headers.cookie',
  'req.body.password',
  'req.body.token',
  'req.body.creditCard',
  'req.body.ssn'
  ],
  censor: '[REDACTED]'
  }
});
```

With that configuration, logging a login request that contains a password and an authorization header produces:

```
{"level":30,"time":1775350450228,"pid":12345,"hostname":"api-1","req":{"body":
{"username":"alice","password":"[REDACTED]","creditCard":"[REDACTED]"},"headers":
{"authorization":"[REDACTED]","cookie":"[REDACTED]"}},"msg":"Login attempt"}
```

`username` passes through because it is not in the redact list. password, creditCard, authorization, and cookie were all replaced with [REDACTED] before the line was written.

Pino's redaction uses the `fast-redact` library which compiles the paths to a function at startup, so the runtime cost per log entry is minimal.

Third, treat `logger.debug(req.body)` in code review the same way you would treat a SQL injection vulnerability.

Too much, too little

There is a temptation, especially after reading about observability, to log everything. After all, the more data, the better, right?

Not really. Over-logging creates three kinds of costs that are easy to underestimate until they become problems.

The first is financial. Log storage at scale is not cheap. A busy application generating 10 million events per day can easily run into thousands of dollars per month in log costs alone, depending on vendor and retention period. Enabling debug-level logging in production can increase volume by 5–10x.

The second is performance. Every log call involves serialization. Even though Pino does "pre-serialize" your message, it is not completely free. Every log call still has serialization work to do, and at volume that adds up.

The third is signal-to-noise ratio. If your `info` level produces 10,000 entries per minute and five of them are relevant to the problem you are debugging, you are filtering through an enormous pile to find a small signal. Over-logging gives you a false sense of observability: lots of data, little diagnostic value.

Under-logging is the other failure mode. An error that says "something went wrong" with no context is pretty useless. The cost of under-logging is invisible until an incident happens, and then it is very visible.

Some practical guidelines:

- Use `debug` and `trace` for development-time diagnostics. Disable them in production unless you are actively diagnosing something.
- Log at natural boundaries: incoming requests, outgoing responses, significant state transitions, errors.
- Do not log inside tight loops
- Audit your log volume periodically. If one endpoint is generating 80% of your log output, either the endpoint has a problem or your logging strategy for it does.

Where logs should go

The Twelve-Factor App methodology has a clear position on this. Factor XI, "Logs", states:

A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage log files. Instead, each running process writes its event stream, unbuffered, to stdout.

Your application's job is to produce logs. The infrastructure's job is to route them somewhere useful. In a containerized deployment, the container runtime (Docker, Kubernetes) captures stdout and stderr automatically. A log shipper (Fluentd, Filebeat, Vector) picks them up, enriches them with metadata such as the hostname, container ID, and Kubernetes pod name, and forwards them to an aggregation backend. From there, a dashboard such as Kibana or Grafana lets you search, filter, and alert.

```
Node.js app (Pino) writes JSON to stdout
  -> Container runtime captures it
  -> Log shipper picks it up
  -> Log aggregator stores it
  -> Dashboard lets you query and visualize
```

The anti-pattern is application code that manages log files directly:

```
const logger = pino(pino.destination('/var/log/myapp/combined.log'));
```

What happens when the disk fills up? What about log rotation? What if the app runs in a container that gets destroyed and recreated? Do the logs survive? If you are running 50 instances across 20 servers, are you going to SSH into each one to grep through files?

For HTTP request logging specifically, `pino-http` integrates directly with your Pino logger instance, outputs JSON, and automatically attaches the request ID so access logs and application logs share the same correlation identifier:

```
import pinoHttp from 'pino-http';

app.use(pinoHttp({ logger }));
// req.log is now available in all handlers with request context baked in
```

This is preferable to Morgan for JSON-first deployments because Morgan outputs plain text by default, which means your access logs end up in a different format than your application logs and you lose the correlation between them.

Always Log the full error

When something throws, log the full error object, not just the message string.

```
try {
 await processPayment(order);
} catch (err) {
  logger.error(`Payment failed: ${err.message}`);
 // Output: "Payment failed: connect ECONNREFUSED 10.0.0.5:5432"
 // Which file? Which line? Which function? No idea.
}
```

Above, you lose the file, the line, the function, and the rest of the stack that led to the error. But if you log the full error, then it becomes apparent:

```
try {
 await processPayment(order);
} catch (err) {
  logger.error({ err, orderId: order.id }, 'Payment processing failed');
```

```
 // Pino recognizes the `err` key and extracts message, stack, and type
}
```

Pino treats the `err` key specially and automatically serializes the error's `message`, `stack`, and `type` fields into the log entry.

A related anti-pattern is re-throwing errors without preserving the cause:

```
try {
 await db.query(sql, params);
} catch (err) {
 throw new Error('Database query failed');
}
```

The Error.cause property was added to JavaScript for this precise reason:

```
async function outer() {
     try {
       throw new Error('Inner DB error');
     } catch (err) {
       throw new Error('Outer operation failed', { cause: err });
     }
   }
```

Node.js has supported the `Error.cause` property since v16.9. You can verify the chain is intact at the point of logging:

```
outer().catch(err => {
  console.log('message:', err.message);
  console.log('cause message:', err.cause?.message);
  console.log('cause stack exists:', !!err.cause?.stack);
});
```

```
message: Outer operation failed
cause message: Inner DB error
cause stack exists: true
```

Both the outer message and the original low-level error are accessible, each with its own stack trace. Without `Error.cause`, you see "Database query failed" and have to guess at the actual problem.

Observability

Logging is one piece of a larger practice called observability. The term comes from control theory and refers to how well you can understand the internal state of a system from its external outputs. In practice, observability in software is built on three pillars:

- Logs
- Metrics
- Traces

Logs tell what happened, when, and with what context. Metrics are numbers aggregated over time, such as request rate, error rate, or p99 latency. They are compact to store and are what you build dashboards and alerts on. Traces are the path of how a single request moved through your system across services, with timing for each step. A mature production system uses all three. Most teams start with logs because they require the least infrastructure. But nothing forces you to start with logs.

For a Node.js application, the observability stack typically looks like this:

- Logger writes structured JSON to stdout
- A log shipper picks it up from the container runtime and forwards it to the log aggregator
- A metrics library (anything that is OpenTelemetry compatible) exposes a `/metrics` endpoint for scraping
- Distributed tracing (OpenTelemetry with a Jaeger or Tempo backend) instruments your service calls
- A dashboard provides a unified view

OpenTelemetry and the future of Node.js logging

OpenTelemetry (OTel) is worth knowing about because it is becoming the standard for instrumenting applications across all three observability pillars. The Node.js SDK (`@opentelemetry/sdk-node`) provides automatic instrumentation for HTTP servers, database clients, and many popular frameworks, injecting trace context into your logs automatically.

When OTel injects trace context, your Pino logs gain `traceId` and `spanId` fields that correlate them directly to distributed traces:

```
import { trace } from '@opentelemetry/api';

// OTel-aware log helper that injects current trace context
```

```
function getTraceContext() {
 const span = trace.getActiveSpan();
 if (!span) return {};
 const ctx = span.spanContext();
 return { traceId: ctx.traceId, spanId: ctx.spanId };
}

req.log.info({ ...getTraceContext(), orderId }, 'Processing order');
```

This means you can click a slow trace in your Jaeger UI and jump directly to the log entries from that exact request. That correlation between traces and logs is what lets you reconstruct what happened: the trace gives you the path a request took, and the log entries are the breadcrumbs it left along the way.

It is also worth noting that the Node.js core is moving in this direction. The proposed `node:logger` module (PR #60468) uses `diagnostics_channel` as its dispatch mechanism, which decouples log production from consumption at the platform level. The proposal is still under TSC review as of early 2026, but it signals where the platform is heading. Deno, for example, already ships a logging module with levels and handlers in its standard library (@std/log).

The anti-pattern checklist

Let's recap! We covered a lot of things in this chapter. So, to help you out, let's create a list of all the 20 anti-patterns we have discussed:

1. Using console.log in production instead of a structured logging library.
2. Using the debug package as an application logger (it is for library authors).
3. Relying on console.log in a piped process and calling process.exit()
4. Not choosing a logging library at all, leaving each developer to invent their own format.
5. Using the wrong log level.
6. Collapsing all log calls to a single level, making severity filtering useless.
7. Unstructured plain-text logs in production.
8. Logging entire request or response objects without filtering fields.
9. Logging sensitive data.
10. No request or correlation IDs.
11. Over-logging
12. Under-logging

13. Logging only err.message instead of the full error object.
14. Re-throwing errors without preserving the cause chain (Error.cause).
15. Managing log files from application code instead of writing to stdout.
16. Identical logging configuration in all environments.
17. No centralized log aggregation in multi-instance deployments.
18. Swallowing errors silently (catching, returning a fallback, and logging nothing).
19. Using pino-pretty in production.
20. Not setting a LOG_LEVEL environment variable override for production diagnostics.

Individually, none of these issues is hard to fix. Legacy applications are a different story: they usually have several of these problems baked in, and the hardest part is finding where each anti-pattern lives. In those systems, start small and fix logging issues opportunistically, correcting each one as you touch the code around it, rather than going after everything at once. The exception is anything critical, such as sensitive data being written to logs or errors being swallowed: fix those immediately.

Summary

We started at the bottom, with what console.log really does: a formatted string written to a file descriptor whose behavior depends on whether it points at a terminal, a file, or a pipe, blocking the event loop in one case and silently truncating output in another. From there, we ruled out debug as an application logger, saw how Winston and Bunyan shaped the vocabulary, and settled on Pino as the recommendation for new work because of its speed and JSON-first output. We then covered the practices that make logs useful: picking the right level, logging structured data instead of strings, carrying a request ID through child loggers, logging the full error with its cause, and never writing sensitive fields, with redaction configured once at the logger rather than at every call site. Finally, we put the environment in charge: pretty output and debug level locally, plain JSON at info in production with a LOG_LEVEL override, and stdout as the only destination the application needs to know about.

That concludes the discussion of logging in Node.js. On to the next chapter.

7

Taming Your Environment

Every Node.js application has configuration that changes between environments. Database URLs, API keys, port numbers, and feature flags. The mechanism the ecosystem settled on decades ago is environment variables, and the way most developers interact with them is `process.env`.

That object is a bag of untyped, unvalidated, mutable strings that any module in your process can read or write at any time. The process.env object has no schema, no defaults, no type coercion, and no way to tell you at boot time that you forgot to set `STRIPE_SECRET_KEY`. You find out when the first customer tries to pay, and the request fails.

This chapter is about closing that gap. You will look at how environment variables work in Node.js, why `process.env` scattered across your codebase creates real operational failures, and what the current best practices look like for parsing, validating, and securing your configuration. Node.js ships native support for `.env` files, which changes some of the tooling choices but none of the underlying principles.

In this chapter, you are going to cover the following topics:

- How Node processes environment variables by default
- Node's native file parsing for environment files
- Anti-patterns of process.env
- Packages, such as dotenv
- Composing env files for different environments

Technical requirements

You can download the example project and code for this book by following the instructions in the *Download the example code files* section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

How process.env Works

The `process.env` object in Node.js is a thin wrapper around your operating system's environment. When the Node.js process starts, it reads the environment block from the OS and populates `process.env` as a plain JavaScript object. The TypeScript type definition tells you everything you need to know about its limitations:

```
interface ProcessEnv {
  [key: string]: string | undefined;
}
```

Every value is either a string or undefined. If you think of a database schema, `process.env` is a table where every column is `VARCHAR NULL`. The important point is that every value is a string in practice because it leads to several errors:

```
const maxRetries = process.env.MAX_RETRIES;
console.log(maxRetries > 3);
console.log(maxRetries > "10");
```

If `MAX_RETRIES` is set to `"5"`, the first comparison (`"5" > 3`) returns `true` because JavaScript coerces the string to a number when one operand is a number. The coercion happens to produce the right answer. The second comparison (`"5" > "10"`) also returns `true`, but for the wrong reason. In this case, both operands are strings, so JavaScript compares them lexicographically, and the character `"5"` comes after `"1"` in Unicode order. In the end, both comparisons return true. The same problem shows up with booleans:

```
const debug = process.env.DEBUG;

if (debug) {
  console.log('debug mode on');
}
```

This prints "debug mode on" when `DEBUG` is set to `"true"`, which is what you expect. It also prints it when `DEBUG` is set to `"false"`, because `"false"` is a non-empty string, and all non-empty strings are truthy in JavaScript. The only way to get a falsy result is if `DEBUG` is `undefined` (unset) or `""` (empty string).

Here is another example. A cache time to live configured via environment variable:

```
const cacheTTL = process.env.CACHE_TTL_SECONDS || 60;
```

When `CACHE_TTL_SECONDS` is `"300"`, `cacheTTL` is the string `"300"`, not the number `300`. The `||` operator sees a truthy string and returns it. Later, when you do `Date.now() + cacheTTL * 1000`, JavaScript coerces the string to a number for the multiplication, and it happens to work. However, if you do `cacheTTL + 10` expecting `310`, you get `"30010"` because string concatenation takes over. The code "works" in your tests, breaks in a way that is invisible until someone notices cache entries expiring after 30,010 seconds (about eight hours) instead of 310 (about five minutes).

Beyond the type problem, `process.env` is fully mutable at runtime:

```
process.env.NODE_ENV = 'production';
process.env.PORT = '9999';
delete process.env.DATABASE_URL;
```

Any module, anywhere in your dependency tree, can modify environment variables. A third-party library could overwrite your `NODE_ENV` and you would have no stack trace pointing to when or why it happened. It is like having a global mutable dictionary that every piece of code in your process shares, with no access control and no audit log.

This becomes especially important in automated testing. When mocking or running fake code, sometimes you will think it's "easier" to just use `process.env`, but if everyone has access to this object, a CI environment can easily change its values during the test runtime, and you will never know why.

The five failure modes

The properties of process.env combine into five concrete ways that raw `process.env` access causes problems in production:

1. Stringly-typed values. Everything is a string
2. Runtime mutability, because the object is not frozen
3. Late failures, because you can only figure out it's broken in runtime

4. Scattered access if you do not centralize your environment variables
5. No IDE support, because it's not typed

The first is stringly-typed values. Every configuration value arrives as a string. Parsing is left to each call site, which means the same conversion logic ends up scattered across multiple files:

```
const port = parseInt(process.env.PORT, 10);
const timeout = Number(process.env.TIMEOUT_MS);
const maxConn = parseInt(process.env.MAX_CONNECTIONS);
```

If you miss one parse or do not document it, then a subtle `NaN` can propagate silently through your math until something inexplicable happens downstream. Notice that the third line is missing the radix argument (`10`). In older JavaScript engines, a leading zero triggered octal parsing, and `parseInt("08")` would return `0` because `8` is not a valid octal digit. Modern engines default to base-10 and handle it correctly, but you still see the explicit radix carried forward in codebases as a habit.

The second is **runtime mutability**. Any code can write to `process.env`. There is no way to make it read-only without breaking native addons and Node internals that depend on writing to it (such as `TZ` for timezone changes). If you want to see this in action:

```
console.log(process.env.NODE_ENV);
require('some-third-party-lib');
console.log(process.env.NODE_ENV);
```

Those two log calls can print different values if your library changes them. You would have no way to know without reading the source of every dependency.

The third is **late failure**. Any missing required envs are not discovered until the first time a request hits the code path that needs it, and following murphy's law, this will be at 11 p.m. on a Saturday.

Consider a payment service. You deploy on Monday morning. The service starts, health checks pass, and traffic begins flowing. Everything looks normal because the first few hundred requests hit the product catalog, user profiles, and search. Then, at 2 p.m., someone tries to check out, and the payment handler gets to read `process.env.STRIPE_SECRET_KEY` the first time:

```
async function processPayment(order) {
  const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
  return stripe.charges.create({
```

```
    amount: order.total,
    currency: 'usd',
  });
}
```

`STRIPE_SECRET_KEY` is undefined. The customer gets 500. The variable has been missing since deployment. You can argue that test coverage should catch this, and it should, but if your test environment has the variable set (which it usually does), the tests pass fine. Failure is environment-specific and timing-dependent, in other words, it is not deterministic. Computing needs to be deterministic, **always**.

The fourth is **scattered access**. Without discipline, `process.env.SOME_KEY` appears in several files. Renaming a variable, removing one, or auditing that variables your service actually needs becomes an adventure through the codebase. You can apply the previous failure to it as well. If you forget to rename one of these variables, the value is just undefined at runtime.

The fifth is **no IDE support**. Without type augmentation, your editor cannot autocomplete environment variable names or warn you that something contains a misspelling. That typo compiles, passes tests (if your tests use a different variable name), and fails only in the specific environment where you expect the correct one to exist. You will stare at the error log for twenty minutes, read the word "DATABSE" six times without noticing the missing A, and finally find it when you copy-paste the variable name into a diff tool.

These are the main problems you face with the default env handling, but there are better ways to do it. Start by not reading `process.env` directly and setting the values in environment files.

Node's Native –env-file Flag

One task is to read the loaded environment variables. The other is to actually load them. Node.js introduced the `--env-file` CLI flag in v20.6.0. The entire API, CLI flag and programmatic APIs reached stable status in v22.21.0 and carry forward into v24+.

The behavior is what you would expect from a `.env` loader: it reads `KEY=VALUE` pairs from a file and puts them in `process.env`. The important rule is that real environment variables take precedence over the file. If your OS environment already has `PORT=8080` and your `.env` file says `PORT=3000`, the process will use `8080`.

This is the correct behavior for production containers where the orchestrator sets the real values, and it was not the original behavior. Here is a typical `.env` file:

```
PORT=3000
DATABASE_URL=postgres://localhost:5432/myapp
```

```
LOG_LEVEL=debug
STRIPE_SECRET_KEY=sk_test_abc123
```

The ways you can load it:

```
node --env-file=.env src/index.ts
```

You can load multiple files. Later files override earlier ones for keys that appear in both:

```
node --env-file=.env --env-file=.env.local src/index.ts
```

This setup is useful for a team setup where `.env` has shared defaults and `.env.local` has developer-specific overrides. If `.env` sets `PORT=3000` and then you load `.env.local` that sets `PORT=4000`, the process will only get the later `4000`.

For containers or CI environments where the file might not exist, there is a non-fatal variant:

```
node --env-file-if-exists=.env.local src/index.ts
```

The softer variant hides a missing file; the strict `--env-file` is the one that tells you, but in a Docker container where local overrides are optional, you need the softer version.

What --env-file Does Not Do

There are a few hard limits you will run into. First, there is no variable expansion. If your `.env` file contains this:

```
DB_USER=admin
DB_HOST=localhost
DB_URL=postgres://${DB_USER}@${DB_HOST}/mydb
```

The value of `DB_URL` will be the literal string `postgres://${DB_USER}@${DB_HOST}/mydb`. If you need `${VAR}` expansion, you need `dotenvx` on top. This defeats the purpose of using the flag altogether.

Multi-line values must be inside double quotes. The following format works:

```
RSA_KEY="-----BEGIN RSA PRIVATE KEY-----MIIEowIBAAKCAQEA...-----END RSA PRIVATE
KEY-----"
```

But (in the time of writing this book) there is a known issue where large multi-line values can be silently truncated. If you are embedding RSA keys or large certificates, test that the full value

comes through, and you **cannot** put `NODE_OPTIONS="--env-file=.env"` inside the `.env` file itself. Node.js blocks this to prevent an infinite loading loop.

Programmatic Loading

For cases where you cannot control the CLI flags, such as those in test runners, scripts, dynamically loaded bootstrap modules, there are two APIs.

The first loads a file directly into `process.env`, exactly like the CLI flag:

```
process.loadEnvFile('.env.test');
```

The second parses env file content into a plain object without touching `process.env`:

```
import { parseEnv } from 'node:util';

const vars = parseEnv('PORT=3000\nHOST=localhost');
```

The `vars` object is `{ PORT: '3000', HOST: 'localhost' }`. This is useful when you want to read a config file without side effects, for example to validate it or merge it with other sources before deciding what will be added into the environment.

You can also use `parseEnv` to parse env-like files, for example, app configurations and other things. You can also combine `parseEnv` with file reading for more control:

```
import { readFileSync } from 'node:fs';
import { parseEnv } from 'node:util';

const content = readFileSync('.env.production', 'utf8');
const productionVars = parseEnv(content);

for (const [key, value] of Object.entries(productionVars)) {
  if (process.env[key] === undefined) {
    process.env[key] = value;
  }
}
```

This snippet reads a production env file but only sets variables that are not already present in the environment, giving you the same behavior as the CLI flag but with explicit control over which file and which keys the code applies to Both APIs use the same parser as `--env-file`, so the output is consistent.

You have seen the options, but what about the anti-patterns?

process.env Everywhere

With the mechanics out of the way, here is the core argument of this chapter: **reading process.env directly throughout your application code is an anti-pattern.**

A `process.env` scattered across a codebase looks like a house with a hundred breaker boxes. Every single module, class or function inside your app will read the envs it needs. You might think this is "scoped," but this chapter already showed that having multiple calls to this object is flaky.

Now, imagine a new developer joins your team and asks: "What environment variables does this service need?" The answer requires grepping the entire codebase because there is no single place to look. You might find 15 variables in the code and 12 in the `.env.example` file, with three that exist in the code but were never added to the template. So, this can be a real problem.

Each of those reads is a potential failure point that only triggers when that specific code path executes. Each returns `string | undefined` and forces the consumer to handle the missing case. More than that, you cannot get static analysis, nor can you get formatting or checks. Besides all that, you also cannot be sure when you need to rename one of those envs, because you will need to do a full search in the entire codebase. You will also see this pattern, which is even worse:

```
function getConfig() {
  return {
    dbUrl: process.env.DATABASE_URL ||     'postgres://localhost:5432/myapp',
    port: parseInt(process.env.PORT || '3000', 10),
    apiKey: process.env.API_KEY || '',
  };
}
```

The application calls this function every time someone needs config, and each call reads `process.env` again. In a tight loop or a frequently-called middleware, you are doing string lookups and `parseInt` calls on every request. More importantly, the fallback values are hidden inside the function.

The fix reads `process.env` **exactly once**, at application startup, through a validation layer. Everything that passes validation gets exported as a typed, frozen object. Everything that fails validation kills the process immediately with a clear error message. The rest of your

application imports the validated object and never touches `process.env` again for the entire duration of the process.

It will suck for a while when you get used to it because it comes with tradeoffs, especially the fact that you can't just replace the values of the variables on the fly, and you also can't add new envs directly and reload them in runtime. So, every new environment will need a clear restart of the application, and this is how it should be. However, how do you do it, so it sucks less when you start?

The validated singleton pattern

The validated singleton pattern is the pattern the Node.js community has converged on, regardless of which validation library you prefer. The idea is:

> **Tip**
>
> Define a schema that describes every environment variable your service needs, parse process.env through it exactly once at startup.

Then, export those as a typed object. Every other module imports that object instead of using `process.env`.

The examples here use Zod because it is the most common choice in TypeScript projects today, it already appears in multiple places throughout the book, and many codebases already depend on it for API validation or form parsing. However, the pattern works with any schema library. These examples are using Zod 4.

Start with the schema in a file you can call `config.ts`. Each environment variable gets a Zod type that describes what it should be, not what `process.env` gives you (which is always a string):

```
import { z } from 'zod';

const envSchema = z.object({
  NODE_ENV: z.enum(['development', 'test',  'production']).default('development'),
  PORT: z.coerce.number().int().positive().default(3000),
  DATABASE_URL: z.url(),
  STRIPE_SECRET_KEY: z.string().min(1),
  LOG_LEVEL: z.enum(['debug', 'info', 'warn', 'error']).default('info'),
});
```

A method named `z.coerce.number()` is important in this context, even though it is usually best avoided. `PORT` will come from the environment as the string `"3000"`, but `z.coerce.number()` converts it to the actual number `3000`. The `.int().positive()` constraints reject `PORT=3.5` or `PORT=-1`, which are technically valid strings but nonsensical as port numbers. If someone sets `PORT=banana`, the coercion produces `NaN`, and `.int()` rejects it with a clear error.

Next, parse `process.env` through the schema. Use `safeParse` so that invalid input gives you a structured error object instead of throwing, and **do not** use async parsing, you want this to happen synchronously on app launch:

```
const parsed = envSchema.safeParse(process.env);
```

If validation fails, print every error and kill the process. The `.flattenError()` will give you a clean mapping of field name to error messages, so if three variables are missing or invalid, the person will see all three at once instead of fixing them one at a time:

```
if (!parsed.success) {
  console.error('Invalid environment variables:');
  console.error(z.flattenError(parsed.error).fieldErrors);
  process.exit(1);
}
```

The importance of this 'report all errors at once' behavior cannot be stressed enough. Without it, you will see one error at a time, which means you will deploy, see "`DATABASE_URL` is missing," fix it, redeploy, see "`STRIPE_SECRET_KEY` is missing," fix it, redeploy again. With `flattenError()`, the first deploy attempt already tells you everything you need to know, for example:

```
{
  "DATABASE_URL": ["Invalid input: expected string, received undefined"],
  "PORT": ["Invalid input: expected number, received NaN"],
  "STRIPE_SECRET_KEY": ["Too small: expected string to have >=1 characters"]
}
```

Finally, freeze the parsed data and export it:

```
export const env = Object.freeze(parsed.data);
export type Env = z.infer<typeof envSchema>;
```

`Object.freeze` prevents any module from writing onto the object. You can do that one property at a time as well with `Object.defineProperty` if you want more control or avoid listing it when you console out, but it is just more work.

In the end, the Env type comes directly from the schema, so it stays in sync automatically. This way `env.PORT` is `number` and `env.NODE_ENV` is `'development' | 'test' | 'production'` so you get a beautiful nice autocomplete from the IDE. Now, compare the before and after across your codebase. Before:

```
import { Pool } from 'pg';

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  max: parseInt(process.env.DB_POOL_SIZE || '10', 10)
});
```

After:

```
import { Pool } from 'pg';
import { env } from './env.js';
const pool = new Pool({
  connectionString: env.DATABASE_URL,
  max: env.DB_POOL_SIZE,
});
```

There are no strings or nulls either because the process would not have started otherwise. If you mistype, TypeScript catches it at compile time. To make this rule permanent across a team, add an ESLint rule that flags direct `process.env` access anywhere outside the validation module:

```
{
"no-restricted-syntax": ["error", {
    "selector": "MemberExpression[object.name='process'][property.name='env']",
    "message": "Import { env } from '@/env' instead."
    }]
}
```

With this rule, a developer who writes `process.env.PORT` will receive an immediate lint error pointing them to the correct import. This "singleton" becomes the only way to access configuration, enforced by tooling rather than convention. The only place you use `process.env`

is the Zod file where you describe the schema, because it needs `safeParse(process.env)`. But, it doesn't come free of charge, as always.

Gotchas with environment variables and Zod

There are two Zod behaviors that are pretty annoying.

The first is that `z.coerce.boolean()` calls JavaScript's `Boolean()` constructor internally, and `Boolean("false")` returns `true` because `"false"` is a non-empty string. If you have an env var like `ENABLE_FEATURE=false`, this happens:

```
const schema = z.object({
  ENABLE_FEATURE: z.coerce.boolean(),
});

const result = schema.parse({ ENABLE_FEATURE: 'false' });
console.log(result.ENABLE_FEATURE);
```

That prints `true`. Your feature flag meant to disable the feature, and the schema turned it on. The behavior is the same truthy-string problem you saw earlier with raw `process.env`, but now it is hiding behind a validation layer that should fix that same thing.

The fix is to treat the value as a string enum and transform it explicitly:

```
const schema = z.object({
  ENABLE_FEATURE: z.enum(['true', 'false']).transform(v => v === 'true'),
});
```

Now `ENABLE_FEATURE=false` correctly produces `false`, and `ENABLE_FEATURE=yes` or `ENABLE_FEATURE=1` will fail validation with a clear error, that forces the operator to use the expected values.

Tip

You can also use the znv library, which wraps Zod schemas in a string-aware preprocessor that handles Boolean env vars correctly out of the box.

The second gotcha is that defaults skip re-validation. If you write:

```
const schema = z.object({
  CALLBACK_URL: z.url().default('not-a-url'),
});
```

The string `"not-a-url"` is not a valid URL, but it passes through because Zod applies the default when the value is missing and does not re-validate against the `.url()` constraint. Your application will go on normally, but you will get the same problem as the previous errors. A safe version is:

```
const schema = z.object({
  CALLBACK_URL: z.url().default('https://localhost:3000/callback'),
});
```

One thing to be careful with the Zod singleton is that some deployment platforms (notably some PaaS providers) set variables to empty strings rather than leaving them unset. If you use `z.string().min(1)` for a required key, an empty string passes the `z.string()` check but fails the `.min(1)`.

The validation behavior is correct and catches the problem at startup. However, if you use just `z.string()` without `.min(1)`, an empty variable will also pass validation and you get a runtime failure later when the code uses the key. For required string variables, always add `.min(1)` to catch this edge case.

TypeScript declare global

You will find tutorials suggesting you augment the `ProcessEnv` interface with your known variables:

```
declare global {
  namespace NodeJS {
    interface ProcessEnv {
      DATABASE_URL: string;
      NODE_ENV: 'development' | 'test' | 'production';
      PORT: string;
    }
  }
}
export {};
```

Do not do that... If you put this in a `.d.ts` file included in your `tsconfig.json`, and, suddenly, `process.env.DATABASE_URL` has autocomplete and no `| undefined`. Your IDE lights up with suggestions. It feels like the problem is solved. It is not. Watch what happens:

```
const url: string = process.env.DATABASE_URL;
```

TypeScript is happy, right? However, at runtime, if `DATABASE_URL` does not exist in the environment, `url` is `undefined`. The type system says the value is a `string` but the runtime does not have it. Everything will work locally, but break in production.

There is also no coercion. Even with the augmentation, `PORT`, for example, is still a string:

```
const port = process.env.PORT;
app.listen(port + 1);
```

You might expect this to listen on port 3001 if `PORT` is `"3000"`. It actually tries to listen on `"30001"` because string concatenation will take precedence over numeric parsing.

If you use this pattern, use it **in addition** to a validated singleton, never **instead** of one. The `declare global` augmentation is useful inside the validation module itself, so you get autocomplete while writing the Zod schema, which then becomes useless if you come to think about it later on.

What about packages?

Let's talk about dotenv

`dotenv` has roughly 126 million weekly npm downloads. It does exactly what this chapter has been doing: reading `.env` files and populating `process.env`. For years it was the only reasonable way to load environment files in Node.js, and every tutorial from 2015 to 2023 will probably recommend it.

With Node.js's native `--env-file`, you should not assume that is true anymore. For new projects, you can use the built-in flag for local development and skip `dotenv`. For projects that need to support Node.js versions older than v20.6, dotenv remains useful (with dotenv-expand if you need ${VAR} expansion).

However, there is an important distinction you need to understand. The distinction is that `dotenv` is a **local development tool**. Its job is to simulate the environment that your production orchestrator provides. The tool should never run in production.

Not that it will bring you massive problems, but in production code, the fewer dependencies you have, the better.

Let's talk about dotenvx

The `dotenv` maintainer has moved active development to `dotenvx`, a standalone CLI that handles the features `dotenv` never had natively, like variable expansion, encryption, and multi-environment management.

If your `.env` file needs `${VAR}` interpolation, `dotenvx` is the current recommendation. Go back to the earlier env file example:

```
DB_USER=admin
DB_PASS=secret
DB_HOST=localhost
DATABASE_URL=postgres://${DB_USER}:${DB_PASS}@${DB_HOST}/mydb
```

Running through `dotenvx`:

```
dotenvx run -- node src/index.ts
```

The `dotenvx` command reads the file, expands `${DB_USER}`, `${DB_PASS}`, and `${DB_HOST}` into their values, and injects the fully interpolated `DATABASE_URL` into the process environment. So, all the things will work as you expect.

The encryption feature is where `dotenvx` diverges from the old `dotenv` model. Instead of keeping `.env` files out of version control, you can encrypt them in place:

```
dotenvx encrypt
```

This replaces the plaintext values in your `.env` file with `encrypted:` prefixed ciphertext and stores the private decryption key in a separate `.env.keys` file. The encrypted `.env` is safe to commit. The `.env.keys` file is not.

At runtime, you pass the private key through the `DOTENV_PRIVATE_KEY` environment variable, and `dotenvx` decrypts the values before injecting them into the process. In general, this is nice but not so useful if you have multiple people in multiple places because you still need the "password" for your dotfiles. If you will still keep one file off version control, it might as well be the envs directly.

Tip

When should I use each one?

Generally, speaking, it depends. There is no one-size-fits-all solution, but default to the engine's native implementation whenever you can. Not because it's "better implemented," mostly out of security. It is much harder for an attacker to infiltrate Node.js than to infiltrate dotenvx for example. However, in multiple cases you will need to look for convenience, especially in big teams or big projects with multiple envs to maintain. In these cases, it could be beneficial to use dotenvx to remove the step of thinking about loading the envs onto the environment from the app itself and move it to a container initialization step that is default for everyone.

Cascading .env Files

In practice, a team needs more than one `.env` file more often than not. There are values that every developer shares (the default port, the local database URL, the log level for development). There are also values that individual developers override like a different port because they run multiple services, or a remote database for testing against staging data. There are values that change per environment, like production credentials, and test-specific feature flags.

The cascading model handles this with layered files that stack on top of each other, where later layers override earlier ones for keys that appear in both. With Node's native `--env-file` flag, you control the stack explicitly:

```
node --env-file=.env --env-file-if-exists=.env.local src/index.ts
```

The following list shows the resolution order for any given variable, from lowest to highest priority:

- .env: contains the team-wide defaults. Every developer gets the same base values.
- .env.local: contains per-developer overrides. This file is optional (hence `--env-file-if-exists`) and never committed. A developer who needs PORT=4000 because port 3000 takes precedence by another service puts the override here.
- The real OS environment will always overwrite everything. If your shell has PORT=8080 exported, that value takes precedence over both files.

Here is a concrete example, take your `.env`, which stays out of version control, with a versioned copy named `.env.example` that holds placeholder data

```
PORT=3000
DATABASE_URL=postgres://localhost:5432/myapp
LOG_LEVEL=debug
REDIS_URL=redis://localhost:6379
```

A developer's `.env.local`:

```
PORT=4000
DATABASE_URL=postgres://staging-db.internal:5432/myapp
```

The process will get:

- PORT=4000,
- DATABASE_URL=postgres://staging-db.internal:5432/myapp,
- `LOG_LEVEL`=debug
- `REDIS_URL`=Redis://localhost:6379

If the same developer also has `PORT=9000` exported in their shell profile, the process will get `PORT=9000`.

For environment-specific files like `.env.test`, you load them alongside the base:

```
node --env-file=.env --env-file=.env.test src/test-runner.ts
```

`A file named .env.test` might set `LOG_LEVEL=error` and `DATABASE_URL=postgres://localhost:5432/myapp_test`. Keys not redefined in `.env.test` fall through to `.env`.

You can also build this stack programmatically using `process.loadEnvFile()` and `util.parseEnv()` if you need more control over which files load in which order, or if your test runner does not let you control the CLI flags:

```
import { readFileSync, existsSync } from 'node:fs';
import { parseEnv } from 'node:util';

const files = ['.env', '.env.local', `.env.${process.env.NODE_ENV}`];
const merged = {};

for (const file of files) {
```

```
  if (!existsSync(file)) continue;
  const vars = parseEnv(readFileSync(file, 'utf8'));
  Object.assign(merged, vars);
}

for (const [key, value] of Object.entries(merged)) {
    if (process.env[key] === undefined) {
      process.env[key] = value;
    }
}
```

This precedence follows the same precedence as the CLI flag. The first loop merges all files into a single object where `.env.local` values overwrite `.env` values. The second loop writes the merged result into `process.env`, but only for keys that are not already set in the real environment. However, this is rarely the best decision.

Like all the things in Node.js, there is also a package for that!

Enter config

If you find yourself writing that manual file-loading loop and adding more layers to it, there is a package built exactly for this: the `config` package (`npm install config`). It handles the cascading automatically by convention.

You create a `config/` directory at the project root with JSON files named after environments:

```
config/
  default.json
  development.json
  production.json
  test.json
  custom-environment-variables.json
```

A file named `default.json` has your base values:

```
{
  "server": {
    "port": 3000,
    "host": "localhost"
  },
  "db": {
    "url": "postgres://localhost:5432/myapp",
```

```
    "pool": { "min": 2, "max": 10 }
  },
  "logLevel": "debug"
}
```

A file named production.json overrides only what changes in production:

```
{
  "server": { "host": "0.0.0.0" },
  "db": { "pool": { "min": 5, "max": 50 } },
  "logLevel": "info"
}
```

And `custom-environment-variables.json` maps environment variables into the configuration tree:

```
{
  "server": { "port": "PORT" },
  "db": { "url": "DATABASE_URL" }
}
```

The package merges these automatically based on `NODE_ENV`. In production, the merge order is `default.json`, then `production.json`, then environment variables mapped through `custom-environment-variables.json`.

Later files override earlier ones, same as the `--env-file` flag. In your application code:

```
import config from 'config';

const port = config.get<number>('server.port');
```

Note that when a value comes from an environment variable (for example, PORT through custom-environment-variables), config returns it as a string, so the `<number>` type is a promise TypeScript cannot check. Use __format: 'number' in the mapping or validate the values with Zod.

```
const dbUrl = config.get<string>('db.url');
const poolMax = config.get<number>('db.pool.max');
```

The main difference from the manual approach is that `config` defines the convention, and you merely follow it. Every developer on the team needs to know that `config/default.json` is where base values live, that `config/production.json` is where production overrides go, and that `custom-environment-variables.json` is the bridge to the OS environment. The library defines the file-loading order and precedence, so there is no ambiguity. This is useful for larger teams when team members have different preferences.

The trade-off is that `config` does not validate at startup. For that reason, you might combine it with a Zod schema that validates the merged config object after assembling the configuration, getting the best of both worlds.

For most services where configuration is flat key-value pairs, the native `--env-file` flag plus a Zod singleton is simpler and safer. `config` is better when you have a deeply nested configuration with many environment-specific overrides, where managing it all through flat env vars would mean dozens of `SECTION_SUBSECTION_KEY` variable names. This usually appears only in large applications.

However, there is a moment where env file stacking becomes its own problem. The moment you have `.env`, `.env.local`, `.env.development`, `.env.staging`, `.env.production`, and `.env.test`, you have a configuration management problem that `.env` files were not designed to solve. Each file will eventually differ from each other because developers forget to add new variables to all the files.

A reasonable threshold to this is that once you have more than three `.env` files, or more than two developers share credentials, move to a secret manager (AWS Secrets Manager, HashiCorp Vault, 1Password), then you can load secrets in different namespaces (or whatever it is called in the manager) and then load that specific namespace.

Putting It Together

For a new Node.js service, the full configuration stack works across two environments.

In local development, start the application with the native `--env-file` flag:

```
node --env-file=.env --env-file-if-exists=.env.local src/index.ts
```

The `.env` file has shared non-secret defaults. The `.env.local` file (optional, per-developer) has individual overrides. The validation module parses everything at startup and kills the process if anything is missing or invalid.

In production, secrets come from your secrets manager (AWS Secrets Manager, Vault, Doppler, Kubernetes Secrets). The orchestrator injects them at container startup by the orchestrator.

No `.env` file exists anywhere in the production image. The validation module runs at boot, and the service refuses to start if configuration is incomplete.

> **Tip**
>
> You could use the secret manager's solution in development. However, everyone would need a namespace containing all the development keys and would need to keep it updated. The drawback is that you lose the coupling between the value and the key, forcing deleted envs to be deleted from both places.

The validation module is the single point of truth, and it must exist in all places. It is the only file in your codebase that reads `process.env`. Every other module just imports the validated `env` object. If a new developer joins the team, they just need to look at `src/env.ts`, and will immediately know every configuration variable the service needs, what type each one is, which ones have defaults, and what the valid values are. That validation behavior is the goal.

The Anti-pattern Checklist

Here are all the anti-patterns this chapter has discussed, collected in one place:

- Reading `process.env` directly in application code instead of through a validated singleton
- No startup validation, discovering missing variables at request time
- String comparisons on values that should be numbers or booleans
- Relying on `declare global` type augmentation without runtime validation
- More than three or four `.env` file variants without moving to a secrets manager
- Using `dotenv` in production instead of platform-native secret injection
- Mutable configuration objects that any module can overwrite
- No ESLint rule preventing direct `process.env` access
- `z.coerce.boolean()` on string env vars
- Not using `--env-file-if-exists` in containers where the file is optional

Again, none of these are difficult to fix individually. The hard part is catching them before they cause an incident. The validated singleton pattern eliminates most of these failure modes by making the correct behavior the default and the dangerous behavior something you must actively opt into.

Summary

This chapter looked into something that's often overlooked in developer security and application best practices: The environment variables.

The bottom line of this chapter is that you should never trust the human sense what can be done automatically. Because of that, you will protect your environment variables and make sure that the environment is controllable and deterministic in all the cases you have.

You do that by implementing a simple pattern of single access with initial validation. A single object that will protect you from multiple problems that will bite you in the future.

8

What is Synchronous and What is Asynchronous

If you ever start talking about Node.js, you will quickly hear the term **event loop** mentioned. It is a fundamental concept that underpins how Node operates, especially when it comes to handling asynchronous operations such as timers, input and output, and callbacks.

The Node.js event loop is the core process of every Node application. It is what allows Node to handle multiple tasks simultaneously, despite running on a single main thread. Understanding how the event loop works internally is critical for writing efficient Node code and avoiding performance-killing anti-patterns.

This chapter will demystify the event loop's mechanisms and highlight common misunderstandings that lead to bad practices. You will understand exactly what synchronous and asynchronous code are and what they look like.

A previous chapter lightly touched on the event loop, but this time, you will dive deep into it.

Technical requirements

You can download the example project and code for this book by following the instructions in the **Download the example code files** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Understanding the Node.js event loop

In Node.js, the **event loop** is the mechanism that orchestrates the execution of asynchronous operations on a single thread. Even though JavaScript code in Node runs in a single main thread, Node can handle many operations concurrently thanks to the event loop's design.

The event loop works by offloading operations to the operating system or background threads (via the `libuv` library, a core part of the runtime) and then scheduling JavaScript callbacks to run when those operations complete. In short, Node's event loop lets the runtime perform non-blocking I/O and multitask efficiently *without* spawning a new OS thread for every request, which is what happens in traditional multithreaded runtimes. This is one of the reasons why Node.js scales so well for I/O-heavy workloads, but it also means that if you misuse the event loop, you can easily kill all processing and halt the entire app.

This is so true and so important that it is worth repeating: **Do not block the event loop,** no matter what you do.

A useful rule of thumb is to keep each unit of work on the event loop small and quick. When used properly, this single-threaded model is highly efficient; but when abused, it becomes a single point of failure that slows everything down.

Blocking vs. non-blocking

Before going deeper into the event loop phases and how it works under the hood, let's take a moment to understand what *blocking code* and *non-blocking code are.*

Blocking **code** is code in which the execution of any JavaScript code in the Node process needs to wait until another non-JavaScript operation completes. This is a limitation of the event loop: it cannot run any JavaScript code while any other synchronous non-JavaScript operation is running. This is considered a *blocking operation*.

This means that, contrary to what most people believe, JavaScript code with a heavy CPU footprint that prevents the system from running efficiently is not **technically** considered blocking code, because JavaScript is still executing on the main thread. But practically, any synchronous code, be it JS or non-JS, will block the event loop because it only runs on a single thread.

An example of blocking code is synchronous I/O, represented mostly by the `node:fs` library. All the methods within this library have synchronous and asynchronous versions. Let's look at this example of a fictional doWork function:

```
import { readFileSync } from "node:fs";

const blockingData = readFileSync("./some-path.js"); // blocking
console.log("This is gonna run after the read");
doWork(); // After all the things
```

In this example, all the code is blocked from running until the `blockingData` variable is filled with the contents of the file. This will essentially halt the Node.js process until the I/O call to the file responds. Let's look at the asynchronous version of this example:

```
import { readFile } from "node:fs";

readFile("./some-path.js", (err, data) => {
  if (err) throw err;
  console.log("This is gonna run after the read");
  console.log(data);
}); // non-blocking

doWork(); // This will run before the console
```

In this new example, the file reading will start and be delegated to the OS with a handler. Once it completes, the code will continue to run, and the process will not exit until the file has been read. So, in this case, you would first see the result of the `doWork` call, and then you would see the logs inside the callback. This is another example of blocking code:

```
console.log("This runs");
while (true) {
  let a = 1 + 1;
}
console.log("This will never be run");
```

To summarize this section in one phrase, we could say: *Blocking code is the code that prevents the event loop from performing the work defined in its phases.*

But... What phases?

Event loop phases

So how does the event loop manage tasks internally? At a high level, the Node event loop is often described as having multiple **phases**, each with a specific purpose. In each phase, callbacks in a **first-in, first-out (FIFO)** queue are executed.

When one phase of the queue is exhausted or the maximum number of callbacks is run, the loop moves to the next phase. In a way, the loop will never be stuck by itself because it will always unblock its own phases, but it can be blocked externally, as mentioned before, or if a callback contains blocking code.

As in this example:

```
console.log(`Will print`)
// infinite loop
setTimeout(() => {
  while (true) {
    // this blocks the loop from within
    let a = 1 + 1
  }
}, 1000)
// this prints once
setTimeout(() => console.log(2), 500)
// this prints once
console.log(1)
```

The output of this code will be:

```
Will print
1
2
```

Because 1 is synchronous, it will be executed directly. 2, however, is asynchronous and runs in half the time of the blocking loop, so it will be executed before 1. Then the loop starts, and it will continue to run infinitely, essentially blocking the loop from within.

Let's check out what these phases are:

- **Timers phase:** Executes callbacks scheduled by `setTimeout()` and `setInterval()`. These callbacks run once their scheduled time threshold has passed, but note that the exact timing can be affected by OS scheduling or other busy tasks. So, a *2-second* timer may not fire exactly at 2,000 ms if the loop is busy (this is why you saw timers running with slight delays before).
- **Pending callbacks phase:** Executes system-level I/O callbacks deferred to the next loop iteration.
- **Idle, prepare phase:** Used internally by Node and libuv for housekeeping.

- **Poll phase:** The most important phase of the loop, and the busiest. Here, the loop:
  - Retrieves new I/O events and executes I/O related callbacks (such as callbacks for completed network or filesystem operations). This is a *busy* phase since Node will wait here for new I/O events if it has nothing else to do, up to a certain limit.
  - Executes most of the asynchronous callbacks, except for the timers and `setImmediate`
- **Check phase:** Executes callbacks scheduled by `setImmediate()`.
- **Close callbacks phase:** Handles some cleanup callbacks for closing resources, such as a socket's `'close'` event.

Between these phases, Node will also perform checks and decide whether it should continue running or, if there are no tasks in the event loop, shut down the process.

To visualize this, let's look at this diagram:

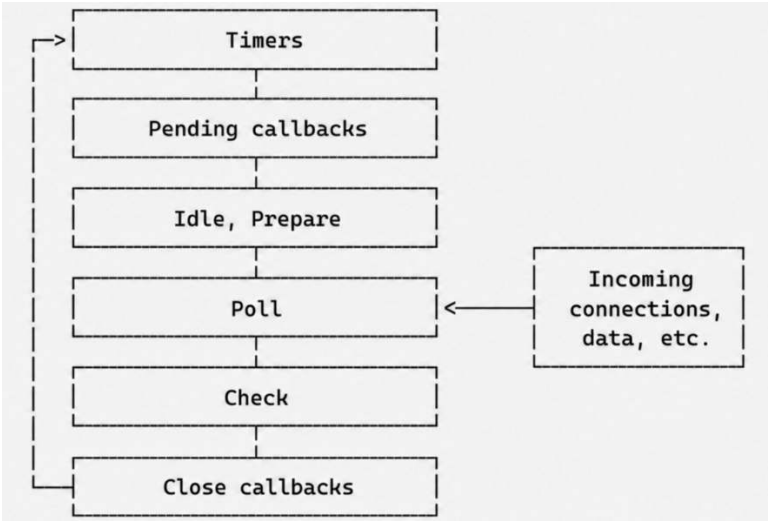


Figure 8.1: The Node.js event loop stages

Each of these boxes has its own queue of callbacks. Node will run all callbacks in one phase (or up to a system-defined limit) before moving on. If new events get added to an earlier phase while the loop is in a later phase, for example, a timer callback added when the loop is in the **check** phase will not execute until the loop cycles back around to the **timer** phase in the next iteration.

This is why a long-running callback can delay other timed events. For example, a timer might expire while the event loop is still in the **poll** phase, reading a large file. In this scenario, the

timer's callback will not execute until the loop finishes the current phase and returns to the **timer** phase on the next cycle.

In other cases, it might never return at all, like you saw before with the infinite loop inside a callback.

To make this easier to understand, let's use an example from the Node.js documentation:

```
import { readFile } from "node:fs";

function someAsyncOperation(callback) {
  // This takes 95ms to complete
  readFile("/path/to/file", callback);
}

const timeoutScheduled = Date.now();

setTimeout(() => {
  const delay = Date.now() - timeoutScheduled;

  console.log(`${delay}ms have passed since I was scheduled`);
}, 100);

// do someAsyncOperation which takes 95 ms to complete
someAsyncOperation(() => {
  const startCallback = Date.now();

  // this takes 10ms
  while (Date.now() - startCallback < 10) {
    // do nothing
  }
});
```

Let's break it down:

1. When Node starts, the event loop is initiated, and it starts to loop.
2. When it enters the poll phase, there is nothing for the loop to execute because the callback from `readFile` has not yet completed; the loop will then wait for a set amount of time before moving on.
3. During the wait, the `95ms` pass and the `readFile` function finishes reading the file; it then adds the callback to the poll queue.

4. Since the poll phase still contains work, it executes the callback from the `someAsyncOperation` function, which takes `10ms`
5. When the callback finishes, there are no more callbacks in the poll queue, so the loop checks whether the soonest timer is due and moves on through the other phases until it gets to the timers phase again.
6. In the timers phase, it will run the callback from `setTimeout` but not at `100ms`, but at `105ms` (95 + 10 ms), so the timer will have printed `105ms`.

It is important to note that the event loop does execute phases in a fixed order as described earlier, but not every phase will have work every time. Also, Node may have platform-specific differences in implementation that are not usually visible to you as a developer.

For example, Node 20 and above changed when the timers phase runs relative to poll, which can slightly affect ordering between `setTimeout()` and `setImmediate()` in some specific cases when you are creating native addons or embedding the runtime.

But, for now, let's start exploring what tasks are.

Exploring microtasks and macrotasks

Before you learn about the task queues, there is one important thing to mention: the role of the *JavaScript call stack*.

The explanation of what a call stack is beyond the scope of this book, but think of it as a stack (like the name says) that keeps a record of all the called functions. Once a function ends and returns, it is taken out of the stack.

In Node (and in any JS runtime), when your code is executing a function, it runs to completion before any other scheduled work can happen. The call stack is not part of the event loop itself, but it is where the currently executing code lives. This is a common part of almost any programming language.

The important concept to understand is that **the call stack must be empty before the event loop will pick up the next callback from a queue**. In other words, Node will not stop an ongoing JavaScript function to start running a different callback, which is what happens in multithreaded programs that are able to run multiple tasks in parallel.

In Node, each JavaScript function (or callback) runs to completion (or until it awaits or yields) before other events get a turn. This means that if you have a computationally intensive function, it can hog the call stack for a while, blocking the event loop from progressing to other tasks. This is the easiest way to understand why the event loop is blocked.

Aside from the call stack, Node also has what are known as task queues. They are divided into **microtasks** and **macrotasks.** Once the call stack is completely cleared, which means no function is running, Node will look for microtasks to run, and only when these microtasks are completed, the loop will move to the next callback from the macrotask queue. This is important because it determines what is going to run first.

But what are microtasks and macrotasks? They are two terms used to describe two categories of scheduled operations in the JS runtime. Each of them has a different set of rules:

Macrotasks (also just called "tasks" or "events"): These are tasks that go through the event loop phases discussed earlier (timers, poll, check, and so on). Each callback is scheduled via timers such as `setTimeout`, `setInterval`, `setImmediate`, or I/O events (such as an incoming HTTP request). Macrotasks are handled one at a time per loop iteration, and each macrotask can generate more macrotasks or microtasks.

Microtasks: These are smaller, fast tasks that always execute before the next macrotask is taken from the event loop. In Node, the typical microtasks are Promise callbacks and `process.nextTick()` callbacks. ES6 introduced the concept of a microtask queue for Promises, and Node.js additionally uses a separate queue for `process.nextTick()` callbacks. Microtasks are intended to run immediately after the current operation is completed, to finish minor work before allowing the event loop to continue.

It is important to highlight that, while microtasks and macrotasks are part of the execution model, the event loop still exists. Task priorities are simply a way to organize the work within the loop.

The Node.js execution model says that, after executing any JavaScript callback (be it a top-level script, a timer, or an I/O callback), Node will process all pending microtasks before moving on. More than that, Node gives `process.nextTick()` even higher priority. Any calls to it will be processed immediately after the current function ends, even before the Promise microtask queue is drained. This leads to some interesting examples you will see later.

After that, the regular event loop, with the macrotasks, will continue. In summary, Node's order of execution is:

1. Current call stack
2. `process.nextTick` queue
3. Microtask queue (Promises etc.)
4. Macrotask (event loop) queue.

The order of events is not guaranteed per the documentation and depends on the environment that is running the code, whether it is being run in a CommonJS context or ESM. However, the

chances of you hitting any of those small differences are so thin that you can treat the order as guaranteed.

Microtasks: process.nextTick() and Promises

As discussed before, a microtask is a small task that should be run as soon as possible in the same iteration of the event loop, after the current code execution, and not at some later time. The main example of a microtask is a Promise. Every function passed as the callback of a `.then()` or `.catch()` handler will be put into the Promise microtask queue and run soon after the current call stack execution.

Node, however, has a mechanism that no other runtime has: `process.nextTick`. Think about it as a very fast Promise, one that has priority over all the other things. Any callbacks passed to it will be placed on a special queue that will be run immediately after the current operation finishes, before any other Promises.

Let's see an example of that in action. Can you guess what will be the execution order of this code?

```
console.log("A");
process.nextTick(() => {
  console.log("B");
});
Promise.resolve().then(() => {
  console.log("C");
});
console.log("D");
```

If you run this as a CommonJS script, the output is A D B C. (Under ESM, the order of B and C is reversed, as you will see at the end of this chapter)! Let's break it down step by step:

- The script starts executing top-to-bottom. It logs `"A"` immediately.
- You call `process.nextTick(() => console.log("B"))`. This queues the callback to run after the current function (script) finishes, in the special `nextTick` queue.
- You call `Promise.resolve().then(() => console.log("C"))`. This queues the `.then` callback into the general microtask queue (for Promises). Since the promise is already resolved (immediately), the callback will be scheduled to run as a microtask.
- You log `"D"` immediately because it is a call stack function, and those run immediately
- At this point, the current script (top-level code) is done executing, and the call stack is about to empty. Now Node will process microtasks before moving on. First, it checks

the `process.nextTick` queue. The callback that prints `"B"` is there, so it executes, and `"B"` is printed before anything else in this phase. Then that queue is empty.

- Next, Node processes the promises microtask queue, where the callback printing `"C"` is waiting. It executes, printing `"C"`.
- Now all microtasks are done. The event loop would then proceed to any macrotasks, so if there were any pending timers or I/O, they would run next. In this example, there are no macrotasks scheduled, so the program will exit because the call stack is empty as well.

Note

There is yet another way to schedule a microtask in Node.js which is through `queueMicrotask()`. You may think that the microtasks queued by it would behave like a `process.nextTick`, but they do not. They are treated like normal, promise-like microtasks. This means that `process.nextTick` will always run before any `queueMicrotask` callback and essentially before any given task in the event loop.

Macrotasks: timers and I/O callbacks

On the opposite side of the task line are the macrotasks (or just tasks). These are the *normal* event loop tasks and are composed of timers (like `setTimeout`, `setInterval`), any I/O callbacks such as network or file-reading and file-writing events, `setImmediate` callbacks, and the `close` callbacks mentioned back in the *Event loop phases* section.

Let's run an example:

```
console.log("start");

setTimeout(() => {
  console.log("timeout 0");
}, 0);


setTimeout(() => {
  console.log("timeout 50");
}, 50);

setImmediate(() => {
  console.log("immediate");
```

```
});

console.log("scheduled tasks");
```

Can you figure out what will be the output of this program? Let's look at it:

```
start
scheduled tasks
immediate
timeout 0
timeout 50
```

Doesn't something seem strange? Why does the immediate callback run before the timeout callback with `0` milliseconds, even though you defined it later on? Let's break it down step-by-step:

- The console with `start` is synchronous, so it will always run first. There are no other callbacks registered in the loop, so it will tick.
- Your first `setTimeout` is called with a 0 milliseconds delay, but the event loop is already after the timers phase, so it will just be picked on the next iteration; however, the callback is registered.
- The next timeout of `50` milliseconds is scheduled as normal.
- An immediate callback is scheduled; this callback runs on the *check* phase of the loop in the next tick, which comes before the timers phase
- The final `console.log` function result is also synchronous, so it is put on the call stack and run.
- When the call stack is empty, the event loop starts picking up tasks. Whether the 0-millisecond timer or the immediate callback runs first depends on how quickly the loop gets going, so Node.js does not guarantee the order here. In this run, the loop reached the check phase first, so `setImmediate` ran first and printed "immediate" to the screen.
- The event loop cycles back again because there are still callbacks waiting in the queue. This time, during the timers phase, it picks up the 0-millisecond callback. The 50-millisecond timer is not due yet, so it runs on a later iteration of the loop, once 50 milliseconds have passed.

Run this a few times and you may see timeout 0 printed before immediate. When both are scheduled from the main module, Node.js does not guarantee their order; it depends on how

quickly the event loop starts. The order is only guaranteed when both are scheduled from inside an I/O callback, where `setImmediate` always runs first.

Mixed interactions

It is going to be rare (and even unusual) to have *microtask-only* or *macrotask-only* code; the most common case is that you will have them all mixed. What would happen then? For example, what would be the execution order of this code:

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout callback");
}, 0);

Promise.resolve().then(() => {
  console.log("Promise callback");
});

console.log("End");
```

The final execution will be like this:

1. `Start`
2. `End`
3. `Promise callback`
4. `Timeout callback`

This is a simpler flow to understand:

- All the synchronous calls to console happen first because they are not on any queues.
- All the promises happen after that because they are in the microtask queue.
- All the callbacks happen later because they are macrotasks.

Let's take a more complex example. You do not need to read it all or understand it in one go; we will dissect it step by step:

```
console.log("start");

Promise.resolve().then(() => {
  setTimeout(() => {
    console.log("timeout 10ms");
```

```
    setImmediate(() => {
      console.log("immediate");
      Promise.resolve().then(() => console.log("promise in immediate"));
    });
  }, 10);
  console.log("first promise");
});

setTimeout(() => {
  console.log("0ms timeout");
  setImmediate(() => {
    console.log("immediate inside 0ms timeout");
  });
}, 0);

setImmediate(() => {
  console.log("outer immediate");
  process.nextTick(() => console.log("nextTick"));
});

console.log("end");
```

This will output the following order:

```
start
end
first promise
outer immediate
nextTick
0ms timeout
immediate inside 0ms timeout
timeout 10ms
immediate
promise in immediate
```

This is an example in which there are a lot of states at the same time; it can be called the stress test of queueing in the event loop.

Remember that the code is run line by line, from top to bottom. And here things get interesting, starting from line 1 down:

1. The first console is synchronous, so it will print directly.

2. You set a promise that resolves immediately and schedule its callback to run on the microtask queue (but it has not run yet), so it will run on the next tick.
3. The `setTimeout` call with 0 ms is scheduled with its callback, but it will not run until the next tick.
4. The outer `setImmediate` call is scheduled, and now there is nothing inside the call stack, so the loop will start running the queues.
5. The first thing will be the microtasks, so it starts with the promise that schedules a timeout of 10 ms.
6. Since the outer `setImmediate` call has a higher priority than the others, it will run first, scheduling a `process.nextTick` callback in the loop.
7. Then `process.nextTick` runs in the same loop iteration, and there are no other callbacks to be run in the same tick, so it will run, printing `nextTick`
8. Now that the threshold for 0 ms has passed, the loop is in its next iteration and will run the timers in order. So, the 0 ms callback will be run first (before the 10 ms one from the promise) scheduling a callback that has a `setImmediate` call in it, which will be put before the next `setTimeout` because the 10 ms timeout is not due yet.
9. The `setImmediate` call inside the 0-ms `setTimeout` handler is run because the check phase comes right after the timers phase (so the 10-ms callback is still waiting).
10. There are still tasks to be run in the queue, namely, the 10-ms callback, which is run next, scheduling another `setImmediate` handler.
11. The `setImmediate` call inside the promise is run because it is the only thing left in the loop, scheduling another promise
12. Now the loop only has another promise to run, which is run in the next tick, and now there are no other tasks or items in the call stack.

So, the order of the tasks is:

1. Anything synchronous.
2. process.nextTick
3. Microtasks (promises)
4. Timers and I/O callbacks, then setImmediate callbacks, following the order of the event loop phases.

Now that you understand the order of things, let's see how this can be a problem for us.

`setImmediate` runs before a timer only when both are scheduled from inside an I/O callback.

Understanding why you should not block the event loop

The discussion covered how blocking the event loop was a very bad thing and explained somewhat why or how this can be done, but let's get into the details. In essence, blocking the event loop means performing long, synchronous operations on the main thread. Since Node has only one event loop thread per process, when this thread is busy, it cannot serve any other tasks.

The main impacts of it are divided into three categories:

- **Performance**: The throughput will drop significantly because one task will delay the others, and so on.
- **Security**: Anything that can perform long-running event loop tasks will cause a denial-of-service attack, denying any other operations of compute power. Essentially, things consuming your service will starve it.
- **UX**: Stemming from performance, any experience of using the application will be terrible due to the *appearance of being frozen.*

Let's quickly demonstrate this:

```
const start = Date.now();
console.log("Start");

setTimeout(() => {
  const delay = Date.now() - start;
  console.log(`Timeout executed after ${delay}ms`);
}, 2000);

// while is synchronous and blocks for 3s
while (Date.now() - start < 3000) {
  // busy loop
}

console.log("End");
```

How do you know if something is blocked? In this case, your `setTimeout` call will execute only *after* the `while` loop is done, which means it will be run just after the 3s mark, firing later than the originally scheduled time:

```
Start
End
Timeout executed after 3005ms
```

This 2-s timer did fire 3005 ms late because the `while` loop was blocking the event loop thread. If this were a web server, you would not be serving any requests until the loop was done. This is called an **event loop lag**.

The main offender of a block is usually I/O tasks (not always), but those are easily solvable by using only the async versions of I/O blocking operations, for example, `fs.readFile` instead of `fs.readFileSync`. This was discussed at the beginning of the chapter.

Unfortunately, there are only a handful of things you can do to avoid blocking the loop. You can either partition the tasks, which is called **cooperative scheduling**, or you can delegate the tasks to external workers through the `worker_threads` module.

Let's take a look at those solutions.

Cooperative Scheduling (chunking)

Cooperative scheduling basically means that you break large tasks into smaller chunks and yield the tasks to the loop in small parts through asynchronous code such as `setTimeout`. Let's see an example.

First let's import all the packages and create the helper functions:

```
import { setTimeout, setImmediate, setInterval } from "node:timers";

const hugeArray = new Array(100_000).fill(0);
const CHUNK_SIZE = 20_000;

function busy(ms) {
  const start = Date.now();
  while (Date.now() - start < ms) {
    // simulates heavy work
  }
}

// Simulates some interleaved work
```

```
const interWork = setInterval(
  () => process.stdout.write(`\nWork between phases`),
  100,
);
```

Here, you are creating a very big array with a hundred thousand positions that are all zeroes. You also want to split this array into chunks of 20 thousand, which means you will have five chunks. To simulate some work happening between the phases, you created the `interWork` timer, which adds an interval handler to the event loop with a delay of 100 milliseconds, so it will print some work between the loop phases.

Note

The variable assigned to `interWork` is needed so that you can stop the interval later with `clearInterval(interWork)`.

Now let's create the function that will deal with the scheduling and interleaving of the work:

```
// Deals with scheduling all the work one at a time
function scheduleNext(index, useImmediate = true) {
  const schedule = useImmediate ? setImmediate : (fn) => setTimeout(fn, 0);
  const phase = useImmediate ? "immediate" : "timeout";

  schedule(() => {
    // we could also use slice or splice here
    const start = index;
    const end = Math.min(index + CHUNK_SIZE, hugeArray.length);

    process.stdout.write(
      `\n[${phase}] About to process ${start}..${end - 1}`
    );

    for (let pointer = start; pointer < end; pointer++) {
      // simulate array walking
      hugeArray[pointer];
    }

    // cpu work
    busy(120);
    process.stdout.write(`\n[${phase}] Processed ${start}..${end - 1}`);
```

```
    // work is done, clear all out
    if (end >= hugeArray.length) {
      clearInterval(interWork);
      console.log("\nAll done");
      return;
    }

    // Invert the phase so we use multiple loop phases
    scheduleNext(end, !useImmediate);
  });
}

console.log("\nStarting heavy computation in chunks...");
scheduleNext(0);
console.log("Event loop is free to do other work in between chunks.\n");
```

In this example, you are scheduling a huge array of 100k items to be walked. If you did this manually, you would block the event loop for a long time, so one of the options is to split it into chunks. You then process a small part of these chunks synchronously, but at interleaved phases of the event loop.

In the code, you are setting two functions: `busy`, which simulates some CPU task and will block the main thread for a set amount of time, and `scheduleNext` which will take the items in the predefined array and call itself recursively until the array is finished. At each iteration of the loop, you will schedule an alternating timeout and immediate handler containing one of the chunks to be processed. This will allow you to use different phases of the loop and avoid blocking it.

Note

There are multiple ways you can code this function; one of them is to pass the array directly and use `splice` or `slice` to return a subset of it to the next iteration, but this method is easier to understand.

You also started an interval that will print something simulating interleaved work, such as a request, a file read, or any other thing. If you run this code, you will get something close to this:

```
Starting heavy computation in chunks...
Event loop is free to do other work in between chunks.
```

```
[immediate] About to process 0..19999
[immediate] Processed 0..19999
Work between phases
[timeout] About to process 20000..39999
[timeout] Processed 20000..39999
[immediate] About to process 40000..59999
[immediate] Processed 40000..59999
Work between phases
[timeout] About to process 60000..79999
[timeout] Processed 60000..79999
[immediate] About to process 80000..99999
[immediate] Processed 80000..99999
All done
```

This shows that, between the phases, you can still perform some work, even if the event loop already has other work in it.

Now that you know how to avoid blocking the event loop, let's learn how to block it! In the following example, you will work with blocking code.

Let's start by doing the same, creating your variables and importing the packages:

```
import { setInterval } from "node:timers";

const hugeArray = new Array(100_000).fill(0);
const CHUNK_SIZE = 20_000;

function busy(ms) {
  const t0 = Date.now();
  while (Date.now() - t0 < ms) {}
}

const interWork = setInterval(
  () => process.stdout.write("\n[heartbeat] (should be starved now)"),
  100,
);

console.log("\nStarting synchronous, fully blocking work...");
```

Nothing changed except for the strings inside the log calls. Now let's process the entire array in one pass inside a for loop, which is synchronous:

```
for (let index = 0; index < hugeArray.length; index += CHUNK_SIZE) {
  const start = index;
  const end = Math.min(start + CHUNK_SIZE, hugeArray.length);

  process.stdout.write(`\n[blocking] About to process ${start}..${end - 1}`);

  for (let p = start; p < end; p++) {
    // simulates some work
    hugeArray[p];
  }

  // heavy CPU
  busy(120);

  process.stdout.write(`\n[blocking] Processed ${start}..${end - 1}`);
}
```

When you finish the work on the array, you will just write a final line and clear the interval you created originally.

```
process.stdout.write("\nAll done (interval finally resumes after this).\n");
// cleaning the interval so the process ends
setTimeout(() => clearInterval(interWork), 350);
```

The previous code will output something like this:

```
Starting synchronous, fully blocking work...
[blocking] About to process 0..19999
[blocking] Processed 0..19999
[blocking] About to process 20000..39999
[blocking] Processed 20000..39999
[blocking] About to process 40000..59999
[blocking] Processed 40000..59999
[blocking] About to process 60000..79999
[blocking] Processed 60000..79999
[blocking] About to process 80000..99999
[blocking] Processed 80000..99999
All done (interval finally resumes after this).
```

```
[heartbeat] (should be starved now)
[heartbeat] (should be starved now)
[heartbeat] (should be starved now)
[heartbeat] (should be starved now)
```

Do you see that the interval will only start *after* the execution of all the work in the array? This means that any pending I/O task would have been stuck. This is just one of the ways to block the loop, but the other solution is also just one of the ways to **unblock** it. Let's look at other ways.

Work delegation (Worker Threads)

The other option is for you to actually send the work somewhere else. This way you will not block the event loop, and even if you do, it will be on a different thread that has its own main iteration and will not block the main process.

You can do this natively through the `worker_threads` module. You first import the packages and define what would happen inside the main thread, which is the code that will spawn the other threads:

```
import { Worker, isMainThread, parentPort } from "node:worker_threads";
import { setTimeout } from "node:timers/promises";

// We need to check if this is the current main thread we don't wanna block
if (isMainThread) {
  // We start workers only here
  const worker = new Worker(import.meta.filename);

  worker.on("message", (result) => {
    console.log(`Worker result: ${result}`);
  });
  worker.on("error", (err) => {
    console.error(`Worker error: ${err}`);
  });
  worker.on("exit", (code) => {
    if (code !== 0) console.error(`Worker stopped with exit code ${code}`);
  });

  console.log("Main thread: Worker started. Main is free to do other work...");

  let count = 0;
  const interval = setInterval(() => {
```

```
    count += 1;
    // this is synchronous
    console.log(`Main thread is still responsive... (${count})`);
    // this is also synchronous
    if (count === 5) {
      clearInterval(interval);
    }
  }, 100);
}
```

Now, let's add the worker code. For this, you will need to add an else block:

```
else {
  // here we are in the worker context
  // very big number
  const N = 1e8;
  let sum = 0;
  // long computation
  for (let i = 0; i < N; i++) {
    sum += i;
  }
  // some delay
  await setTimeout(2000);
  // send back the result
  parentPort.postMessage(sum);
}
```

This will give you a concurrent output:

```
Main thread: Worker started. Main is free to do other work...
Main thread is still responsive... (1)
Main thread is still responsive... (2)
Main thread is still responsive... (3)
Main thread is still responsive... (4)
Main thread is still responsive... (5)
Worker result: 4999999950000000
```

The downside of this approach is that you will be working with event-based code, which is completely asynchronous and distributed. This adds a burden of communication complexity and handling that you would not have in a single process.

Picture yourself essentially communicating between different threads in the same process, which means that if the main process dies or crashes, all the workers will crash with it since they all share the same parent.

If you have some work that *cannot* be interrupted, this might not be the best option unless you want to deal with race conditions and corrupted states (spoiler: not really fun).

But after all this explanation, what are the best practices you need to keep in mind?

Learning not to block the event loop

Let's wrap up this chapter with a list of good practices and common anti-patterns that you can avoid in the event loop. First, let's talk about how you can work with `process.nextTick` and `setImmediate`:

- Prefer `setImmediate` for yielding to the next tick (so I/O and timers can run in between chunks)
- Avoid using `process.nextTick` to slice work because it runs before the event loop continues and can starve I/O if overused. Reserve it for rare, specific reordering needs (for example: to ensure a callback fire after construction), not for general chunking.
- Avoid the use of heavy work in `process.nextTick` or `queueMicrotask` because those will almost always starve the event loop
- Be mindful of the environment you are in. Remember that ESM is different from CommonJS. In CommonJS, your main script runs synchronously before the event loop starts. When the synchronous code finishes, the `nextTick` queue is drained, followed by the microtask queue. So, the order is held. But in ESM, this order is the opposite: the script is already wrapped in an async function and will execute in the microtask queue, which means that any promises and `queueMicrotask` callbacks will run before the `process.nextTick` ones since they are being scheduled from inside another microtask.

Now, for `setTimeout` and `setInterval`:

- Avoid using `setTimeout(fn, 0)` to "defer" work, expecting it to run immediately. A zero-delay timeout does not mean zero delay; it means "run on the next timers phase," which may come after `setImmediate` and any pending I/O callbacks. If you need to yield to the loop quickly, `setImmediate` is the better choice. This is well-documented in the Node.js docs, but it is still a common misconception.
- Be careful with `setInterval` for recurring tasks. If the callback takes longer than the interval period, callbacks will pile up, and you may end up with overlapping executions or a permanently lagging schedule. Prefer a pattern where you schedule the next

iteration only after the current one completes (such as calling `setTimeout` at the end of the callback) instead of relying on `setInterval` for long-running or variable-duration work.

- Always clear your timers (using `clearTimeout`, `clearInterval`) when they are no longer needed. Forgotten intervals are a classic source of memory leaks and can keep a Node process alive indefinitely because the event loop still sees pending work.
- Never put CPU-heavy synchronous work inside a timer callback. The timer fires on the main thread just like any other callback, so a blocking operation inside `setTimeout` or `setInterval` will starve the rest of the event loop the same way it would anywhere else.

And finally, for Promises and async/await:

- Remember that the `.then()` and `await` handlers are *microtasks*. They run before the event loop moves to the next phase, which means a long chain of resolved promises (or a recursive async function that never truly yields) can starve timers and I/O just like an overused `process.nextTick`.
- Avoid creating promises inside loops without awaiting them or controlling concurrency. Spawning thousands of uncontrolled promises at once can overwhelm the microtask queue, spike memory usage, and cause unpredictable ordering.
- Always handle promise rejections. An unhandled rejection in Node will, since Node.js 15, crash the process. Use a `.catch()` handler or `try/catch` around every `await` and consider adding a global `process.on('unhandledRejection')` handler as a global safety, but not as a replacement for proper error handling.
- Do not mix callback-based APIs and promises carelessly. If you await a function that also takes a callback, you may end up with double execution or silent errors. Pick one style and stick with it or use `util.promisify` to convert callback APIs into promise-based ones.

Summary

In this chapter, you uncovered the beating heart of Node.js, the event loop, and learned how it manages to make a single-threaded runtime feel asynchronous and fast. You explored how **microtasks** and **macrotasks** orchestrate the rhythm of execution, how Node prioritizes them, and why understanding their interaction is crucial for writing predictable asynchronous code. You also saw that **blocking the event loop** is one of the most damaging anti-patterns you can introduce: a single long-running operation can bring an entire system to a halt. To prevent that, you practiced using asynchronous I/O, chunking CPU-heavy work, and offloading tasks to worker threads.

In short, the event loop is the main part of every Node.js application, even if unseen. Keep it flowing freely, avoid unnecessary bottlenecks, and your code will stay fast, scalable, and maintainable.

In the next chapter, you will take this understanding one step further and explore the issues you can have with memory leaks: how to identify them, how to fix them, and how to avoid them in the first place. See you there!

9

Memory Leaks and Where They Come From

There is a particular kind of bug that does not show up in your tests, rarely shows up in code reviews, and only appears a few hours after the deployment. It is hard to debug because it leaves very few traces behind. Your service starts clean and handles everything perfectly. Memory creeps up for a while, then the process gets killed, a new one spins up, and the cycle starts again. By the time someone notices, there is very little left to look at, because the logs went down with the processes that wrote them.

This is the anatomy of a memory leak in Node.js, and it confuses people because Node has a garbage collector. The whole point of a garbage collector is that you do not manage memory by hand, so a memory leak feels like a contradiction. If the runtime cleans up after you, how can you leak anything?

The answer is the core idea of this chapter. In a garbage-collected language, a leak is not something forgotten by the collector; it is rather memory that is still reachable, still referenced by something that you are simply never going to use again. The collector is not broken. It is doing exactly what you told it to do, which is to keep alive everything that can still be reached.

To find and fix these, you need to understand how Node decides what to keep and what to throw away. So, you will start with how memory actually works in V8 and define what a leak is. Then you will walk through the handful of patterns behind almost all of them, track one down with the tools that ship with the runtime, and finish with how to write code that does not leak in the first place.

Technical requirements

To follow the examples in this chapter, you need Node.js 22 or later, and Chrome or any Chromium-based browser for the DevTools Memory tab. The examples use ES module syntax, so save them with the `.mjs` extension or set `"type": "module"` in your `package.json` file. The load-testing examples use `autocannon`, which you can run through `npx` without installing it globally.

How Node.js manages memory

Node runs your JavaScript on V8, the same engine that powers Chrome, and V8 manages a region of memory called **the heap**. Almost every object, array, string, and closure you create lives on the heap. V8 owns that heap and decides when to collect, expand, or shrink that space.

The first thing to know is that the heap is not one single pool. V8 splits it into generations, because most objects will not ever go past a young age. If a request comes in, you allocate a pile of objects to handle it; you send the response, and all of those objects become garbage and are collected. But a small number of objects will live for the entire life of the process. Global objects such as your configuration object or database pools are, in general, things that you want to keep.

V8's design is built on that observation that most objects die young, and only a small set survives the life of the process. New objects are allocated into a small area called the young generation (also called new space). It is deliberately small, and it gets collected often and cheaply. The collector for the young generation is a copying collector. It walks the objects that are still alive, copies them to a fresh area, and then declares the entire old area empty in one move. Because most young objects are already dead by the time it runs, it only has to copy a few survivors, which makes it fast.

An object that survives a couple of these young-generation collections is assumed to be one of the long-lived ones, so it gets promoted into the old generation (called "the old space"). The old generation is much larger and is collected far less often. It uses a heavier mark-and-sweep collector that walks the whole graph of live objects, marks everything it can reach, and frees the rest. This collection is more expensive, which is exactly why V8 tries to avoid running it by keeping short-lived garbage out of the old generation entirely.

Historically, these collections would stop your program completely while they ran, which showed up as latency spikes. Modern V8 does most of this work in parallel and concurrently, on background threads, while your JavaScript keeps running.

You mostly do not need to think about the details, but it is worth knowing. Also, when an app is leaking, you often see the collector running more and more often while reclaiming less and less.

All you need is reachability

The mark-and-sweep collector decides what is alive using one rule: reachability. It starts from a set of roots, things that are always considered alive, like the currently executing call stack and the global object, and it follows every reference from there. Anything it can reach by following references is kept. Anything it cannot reach is garbage and gets freed.

If there is anything worth keeping from this chapter, it is this: *an object is kept because something still points at it*. If a giant object is referenced by a small object, which is referenced by a module-level variable, then the giant object is alive, no matter how done with it you think you are. The chain back to a root is called the **retaining path**, and tracking down a leak almost always comes down to finding that path and figuring out why it still exists.

Reading memory usage

Node gives you a direct window into all of this through `process.memoryUsage()`. You will use it constantly, so it is worth understanding each number it returns.

```
const usage = process.memoryUsage();
console.log(usage);
// {
//   rss: 38_900_000,
//   heapTotal: 12_000_000,
//   heapUsed: 9_400_000,
//   external: 1_600_000,
//   arrayBuffers: 120_000
// }
```

`heapTotal` and `heapUsed` are the V8 heap. The first is how much space V8 has reserved for JavaScript objects, and the second is how much of that space is holding live objects right now. When people talk about a JavaScript memory leak, this is usually the number that grows.

The `rss`, the *resident set size*, is the total memory the operating system has given the process. It includes the V8 heap, but also the stack, the code itself, and a lot of native memory that has nothing to do with your objects. It is always larger than the heap.

`external` and `arrayBuffers` are the important ones that are often overlooked. They measure memory that lives outside the V8 heap but is still owned by your JavaScript: the backing stores of `Buffer`s, `ArrayBuffer`s, and other native-backed objects. If you allocate a 50 MB buffer,

`heapUsed` barely moves, because the buffer's data is not on the V8 heap. It shows up in `external` and `arrayBuffers` instead. If you only ever watch `heapUsed`, an off-heap leak is invisible to you.

RSS can keep growing even when `heapUsed` is flat, and that is usually not a JavaScript leak at all. It is memory fragmentation in the underlying allocator, which hands memory back to the OS reluctantly. So a rising RSS with a stable heap points at a different problem than a rising heap.

V8 will not grow the old generation forever. There is a limit, and when V8 cannot free enough memory to stay under it, the process dies with a message you will eventually meet in production:

```
FATAL ERROR: Reached heap limit Allocation failed - JavaScript heap out of memory
```

In older versions of Node, that limit was a fixed value regardless of the machine. Modern Node sizes it based on the memory available to the process, and you can override it explicitly with a flag when you start the process (replace the path with your own entry file):

```
node --max-old-space-size=4096 src/index.js
```

That sets the old-generation ceiling to 4096 MB. When a service keeps running out of memory, it is tempting to raise this number and move on. It will look solved for a while, but you must resist that. A leak grows without bound by definition, so a bigger ceiling only means the crash happens later, after more memory and more time have been wasted.

The legitimate use of this flag is giving an app that really does need a lot of memory, such as a parser or an indexer, more headroom so it stops hitting the limit. It is never a fix for a leak.

What a memory leak actually is

Now you can define the thing precisely. A memory leak is memory that stays reachable from a root through a reference you did not mean to keep, so the collector can never reclaim it, and that grows without bound over time.

Two parts of that definition do the work. *"Reachable through a reference you did not mean to keep"* is the cause, and *"Grows without bound"* is the effect. That second part is what separates a leak from an app that simply uses a lot of memory.

Because high memory usage and a memory leak look similar at a glance, this second part becomes even more important. The way to tell them apart is to watch memory across garbage collections, not at a single moment. A healthy Node process traces a sawtooth: memory climbs

as you allocate, a collection runs and drops it back down to a baseline, then it climbs again. The key is the baseline at the bottom of each drop. In a healthy app, that baseline is flat. The app returns to roughly the same resting level after every collection.

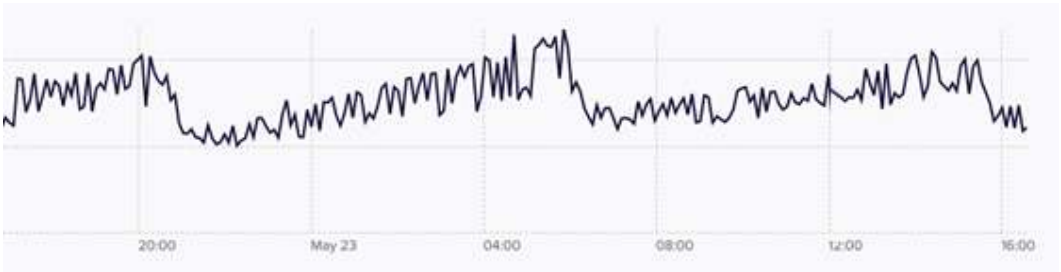


Figure 9.1: Sawtooth memory usage from a parser

Also, notice that the sawtooth curve is **not** linear. It will grow and shrink during the process, but it should be close to the baseline every time. The chart shows memory usage over almost a full day. Look at the low points of the line: after every drop, memory comes back to about the same gridline instead of creeping higher.

In a leaking app, the baseline grows upward after each cycle. Each collection frees less than was allocated, so the resting level after every collection is a little higher than the last. Over hours, the sawtooth marches steadily up and to the right until it hits the ceiling. That upward trend in the post-collection baseline is the signature of a leak, and it is the single most reliable thing to look for. A high reading on its own means nothing, because the collector may simply not have run yet.

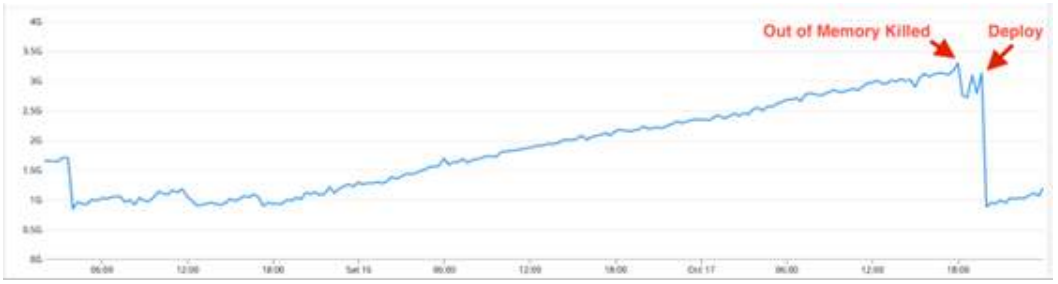


Figure 9.2: A memory leak (source: railsatscale.com)

This chart shows the memory of a single process over about two days. The small dips are the collections. Early on, they bring memory back to around 1 GB, but each one lands a little higher than the last until the process is killed for running out of memory, and the next deploy starts it near the bottom again. That climbing line of low points is what gives the leak away, not any single peak.

Where memory leaks come from

Almost every leak you will meet in Node is one of a small number of patterns. They all come from the same root cause: something is still reachable that you meant to let go of. It is still worth seeing each one concretely, because once you recognize the shapes, you start spotting them in code review before they ever ship.

For the examples that follow, a tiny helper to print memory will be useful:

```
function logMemory(label) {
  const mb = (n) => (n / 1024 / 1024).toFixed(1);
  const { heapUsed, rss } = process.memoryUsage();
  console.log(`${label} heapUsed=${mb(heapUsed)}MB rss=${mb(rss)}MB`);
}
```

The next example uses this helper, and you can reuse it in your own experiments. You can paste it at the top of each example file, or save it once in a file such as `log-memory.js`, add `export` in front of the function, and import it with `import { logMemory} from './log-memory.js'`. Calling `logMemory('start')` prints a line such as `start heapUsed=4.1MB rss=42.3MB`. The other snippets in this chapter are short excerpts focused on the leak itself, so names such as `loadUserFromDatabase`, `externalSystem`, or `databaseCursor` stand in for your own code.

Out-of-control module-level states

This is the most common leak and the easiest to write by accident. Anything you declare at the top level of a module lives for the entire life of the process, because the module itself does. If one of those things is a collection that you only ever add to, it grows forever.

Here is a request-logging middleware that looks harmless:

```
const requestHistory = [];

export function trackRequests(req, res, next) {
  requestHistory.push({ req, at: Date.now() });
  next();
}
```

Every request pushes an entry onto `requestHistory`, and nothing ever removes one. Worse, each entry holds a reference to the full `Request` object, which in turn references the socket, the headers, and any body that was parsed. So this is not a simple log leak; you are essentially

keeping track of the entire Request object forever. Under steady traffic, this climbs in a straight line until the process dies.

The fix is not complicated. Keep only the small piece of data you actually need instead of the whole `Request`, and put a hard cap on the collection so it cannot grow without bound:

```
const MAX_HISTORY = 1000;
const requestHistory = [];

export function trackRequests(req, res, next) {
  requestHistory.push({ method: req.method, url: req.url, at: Date.now() });
  if (requestHistory.length > MAX_HISTORY) requestHistory.shift();
  next();
}
```

Now the array holds at most a thousand small records and never pins a `Request`. Proper bounded structures such as LRU caches appear later, but the shape is what I want you to pay attention to: a module-level collection plus an operation that only ever adds to it equals a leak.

Closures and variable leaks

A closure keeps alive every variable it references for as long as the closure itself is alive. That much is intuitive. The trap is that a closure captures whole variables, not the individual properties you read from them. If you reach for one small field of a large object inside a closure, the closure holds the entire object, and if that closure is stored somewhere long-lived, so is everything the object also references.

Here is the shape. A task queue keeps a list of callbacks to run later:

```
const tasks = [];

function addTask(record) {
  // We only use record.id, but the closure references `record`,
  // so it captures the whole object, including its large payload.
  tasks.push(() => `task queued for ${record.id}`);
}
```

Each call pushes a tiny-looking arrow function. It only uses `record.id`, so it is tempting to assume it only holds onto that id. But it needs `record`, the variable, which means the entire `record` stays reachable through the closure, including whatever large payload came attached to it. Because `tasks` is a long-lived array, every `record` you ever passed in is pinned in memory, payloads and all.

The fix is to capture only what you need. Pull the field out into its own variable before you build the closure, so the closure uses the small value instead of the big object:

```
function addTask(record) {
  const { id } = record;
  tasks.push(() => `task queued for ${id}`);
}
```

Now the closure holds a single number. The rest of `record`, including the payload, becomes collectable as soon as the caller is done with it. The general rule: when a long-lived closure only needs one field from a large object, copy that field into a local variable before you create the closure.

Caches with no bound

A cache is supposed to hold things, so a growing cache does not look like a bug. It becomes one the moment it has no upper bound and no expiry, because then it is just a leak with a friendly name.

```
const userCache = new Map();

async function getUser(id) {
  const cached = userCache.get(id);
  if (cached) return cached;

  const user = await loadUserFromDatabase(id);
  userCache.set(id, user);
  return user;
}
```

If your application has a bounded set of users that get requested over and over, this might be fine in practice. But if the key space is large or unbounded, every unique id adds a permanent entry. A cache keyed by request id, session id, or anything else that is effectively infinite will grow until the process dies. A time-to-live alone does not always save you either. If expired entries are only checked when someone reads them, an entry that is never read again is never noticed and never evicted.

Size is not the only way a cache leaks, because a cache can also keep its keys alive. A regular `Map` holds a strong reference to its keys because it has to be able to find any of them later. So, if you

key a cache by an object, the cache keeps that object alive even after the rest of your program has let go of it:

```
const metadata = new Map();

function track(session) {
  metadata.set(session, { lastSeen: Date.now() });
}
// Even after `session` is dropped everywhere else, `metadata` still points
// at it, so neither the session nor its entry can ever be collected.
```

The answer is a `WeakMap`, which holds its keys weakly. The entry lives only as long as something else still references the key object and becomes collectable once the key is gone. The garbage collector decides when that actually happens, so do not count on it being immediate:

```
const metadata = new WeakMap();

function track(session) {
  metadata.set(session, { lastSeen: Date.now() });
}
// When `session` is no longer referenced anywhere else, its entry here is
// collected automatically.
```

`WeakMap` and its cousins appear in the prevention section, but the rule is simple: if you must key data by an object whose lifetime you do not control, use a `WeakMap`, not a `Map`.

Event listeners need to stop listening at some point

Node's `EventEmitter` keeps a reference to every listener you register until you explicitly remove it. That is correct behavior, but it becomes a leak when you register listeners on a long-lived emitter from short-lived code and never clean them up.

```
import { eventBus } from './event-bus.js'; // a long-lived singleton emitter

app.get('/updates', (req, res) => {
  function onUpdate(data) {
    res.write(`data: ${JSON.stringify(data)}\n\n`);
  }

  eventBus.on('update', onUpdate);
```

```
  // Nothing ever calls eventBus.off('update', onUpdate).
});
```

Every request adds another `onUpdate` listener to the shared `eventBus`, and each one closes over its `res` object, which keeps the response and its socket reachable from the long-lived bus long after the request has finished. The more requests come in, the more listeners will be created.

Node actually tries to warn you about this. By default, once a single emitter has more than ten listeners for one event, it prints a warning to stderr:

```
(node:12345) MaxListenersExceededWarning: Possible EventEmitter memory leak
detected. 11 update listeners added to [EventBus]. MaxListeners is 10. Use
emitter.setMaxListeners() to increase limit
```

It is important to read that warning correctly. It is a leak detector, not a configuration problem. The wrong response is to call `setMaxListeners` and raise the limit, which only silences the messenger while the listeners keep accumulating. The right response is to find out why listeners are being added without being removed. The fix is to tie the listener to the request and take it off when the response closes:

```
app.get('/updates', (req, res) => {
  function onUpdate(data) {
    res.write(`data: ${JSON.stringify(data)}\n\n`);
  }

  eventBus.on('update', onUpdate);
  res.on('close', () => eventBus.off('update', onUpdate));
});
```

Every listener that goes on the bus now comes back off when its request ends, so the number of listeners on the bus can never be more than the number of open requests.

Timers that are never cleared

A timer created with `setInterval` keeps running until you clear it, and while it runs, it holds onto its callback and everything that callback captures.

```
io.on('connection', (socket) => {
  const buffer = Buffer.alloc(5 * 1024 * 1024); // 5MB per connection

  const timer = setInterval(() => {
    socket.emit('heartbeat', buffer.length);
```

```
  }, 1000);
});
```

Each connection allocates a 5 MB buffer and starts an interval whose callback captures both the buffer and the socket. When the connection drops, nothing clears the interval, so the timer keeps running, the callback stays alive, and the buffer it captured can never be collected. Every dropped connection leaves another orphaned timer holding another five megabytes.

The fix is to pair the `setInterval` with a `clearInterval` on the matching lifecycle event:

```
io.on('connection', (socket) => {
  const buffer = Buffer.alloc(5 * 1024 * 1024);
  const timer = setInterval(() => socket.emit('heartbeat', buffer.length), 1000);

  socket.on('disconnect', () => clearInterval(timer));
});
```

A quick aside: calling `unref()` on a timer does not fix this. `unref()` only tells Node that this timer should not, by itself, keep the process alive. The timer still runs and still retains everything its callback captures. It is a lifecycle control, not a cleanup.

Promises and streams

Asynchronous patterns add a few leak shapes of their own. Usually they are hard to track because they do not happen exactly at the time the code runs, but rather later on.

Promises that never settle

The first is promises that never settle. When you `await` a promise, the async function pauses, and V8 keeps the entire paused function alive until the promise resolves or rejects. If the promise never settles, that frame is pinned forever.

```
const pending = [];

for (const job of incomingJobs) {
  const promise = new Promise((resolve) => {
    externalSystem.onComplete(job.id, resolve);
  });
  pending.push(promise);
}

await Promise.all(pending);
```

Each promise here holds its `resolve` function, which the external system holds onto, and the executor closes over `job`. As long as those promises stay unresolved in the array, everything they capture stays in memory, and the `Promise.all` never returns.

The fix is to never wait without a bound. Do a race for each job against a timeout; this way, a missing callback will reject, and use `Promise.allSettled` so one stalled job does not throw for them all:

```
import { setTimeout as delay } from 'node:timers/promises';

const TIMEOUT_MS = 30_000;
const bounded = pending.map((p) =>
  Promise.race([p, delay(TIMEOUT_MS).then(() => { throw new Error('job timed
out'); })]),
);
await Promise.allSettled(bounded);
```

A job that never calls back now settles as a rejection after thirty seconds, and the code awaiting the batch moves on. The timeout does not cancel anything, though. If the external system still holds the `resolve` callback for that job, the original promise and the job it closes over stay reachable through it. To fully release them, also deregister the callback when the timeout fires, if the external system gives you a way to do that.

Unbounded fan-out

The second shape is unbounded fan-out. `Promise.all` over a large array starts everything at once and keeps all of it in memory simultaneously. If the array is too big, this can exhaust the entire heap in one go.

```
const results = await Promise.all(rows.map((row) => enrich(row)));
```

The peak memory here is proportional to the size of the input, not to how much work is going on at a sensible level of concurrency. With a large enough input it will exhaust memory before it finishes.

The fix is to cap concurrency so only a bounded number of operations and their data exist at once. Processing in fixed-size batches is the simplest way:

```
const BATCH_SIZE = 50;
const results = [];

for (let i = 0; i < rows.length; i += BATCH_SIZE) {
```

```
  const batch = rows.slice(i, i + BATCH_SIZE);
  results.push(...await Promise.all(batch.map((row) => enrich(row))));
  // each batch's objects can be collected before the next batch starts
}
```

Streams without backpressure

The third shape is streams without backpressure. When you write to a stream faster than it can drain, Node buffers the unwritten data in memory. `writable.write()` returns `false` when its internal buffer is full, and that return value is a signal to stop and wait. If you ignore it, the buffer grows without limit.

```
for await (const row of databaseCursor) {
  response.write(formatAsCsv(row)); // returns false when full, and we ignore it
}
```

If the consumer (here, the client receiving the response) is slower than the database cursor producing rows, the unwritten rows pile up in the stream's buffer and memory climbs until the process is killed.

The fix is to let the stream tooling handle backpressure for you. `pipeline` connects the source to the destination and pauses the source whenever the destination's buffer is full, then destroys every stream if any stage fails:

```
import { pipeline } from 'node:stream/promises';
import { Readable } from 'node:stream';

await pipeline(
  Readable.from(databaseCursor),
  async function* (rows) {
    for await (const row of rows) yield formatAsCsv(row);
  },
  response,
);
```

Now the cursor only produces as fast as the client can consume, so the buffer stays bounded.

Finding a leak

Suspecting a leak and proving where it is are different things. Before changing any code, follow a short workflow: confirm that the growth really is a leak, then capture the heap to find what is holding on to the memory.

Step one: confirm it is actually a leak

Before you go hunting, confirm the growth is a real and unbounded memory leak rather than warm-up or bounded growth that levels off. The easiest way is to sample `process.memoryUsage()` on an interval under load and watch the trend. In an optimal setting, you would use instrumentation such as OpenTelemetry and other tools to do it, but if you do not have it then you can use an interval (store the timer and clear it with `clearInterval` when you are done; `unref()` only lets the process exit):

```
setInterval(() => logMemory('tick'), 5000).unref();
```

Next, drive sustained traffic at the suspect endpoint while this runs. A load generator such as `autocannon` is perfect for this because a leak needs many iterations to separate itself from one-time startup costs:

```
npx autocannon -c 100 -d 300 http://localhost:3000/some/endpoint
```

Watch the `heapUsed` numbers across the run. If they climb and then settle at a stable baseline, you do not have a leak; you have a warm-up. If they climb and keep climbing across the whole run, you do.

You can get a more direct view of the collector with the `--trace-gc` flag:

```
node --trace-gc src/index.js
```

This prints a line on every collection showing how much memory was in use before and after. In a healthy app, the major collections bring usage down by a lot each time. In a leaking app, you see frequent major collections that reclaim less every time, the before and after numbers are nearly equal, and the heap will stay close to the limit.

Step two: capture the heap and compare

Once you know there is a leak, you need to see what is piling up. The tool for this is a heap snapshot, a complete dump of every object on the heap and every reference between them. The most approachable way to read one is with Chrome DevTools, which you can attach to a running Node process:

```
node --inspect src/index.js
```

Then open `chrome://inspect` in Chrome, connect to the process, and go to the Memory tab. Use `--inspect-brk` instead if you need to break before your code runs, for example to catch a leak that happens during startup.

The single most effective technique here is comparison, sometimes called the three-snapshot technique. Take a snapshot after the app has warmed up. Exercise the suspect code path many times, for example by running your load generator. Take a second snapshot. Then switch DevTools to the Comparison view, which shows the difference between the two snapshots grouped by constructor, and sort by the delta column. A large positive delta means objects of that type were created between the snapshots and never freed. Those constructors are your leak candidates. A third snapshot after more load confirms whether the delta keeps growing.



Constructor	# New	# Deleted	# Delta ▾	Alloc. Size	Freed Size	Size Delta
▸ Array	49 246	0	+49 246	3.932 MB	0 B	+3.932 MB
▸ LeakedRecord	49 115	0	+49 115	2.0 MB	0 B	+2.0 MB
▸ Function	243	19	+224	18.5 kB	1.9 kB	+16.6 kB
▸ (object shape)	246	121	+125	50.6 kB	24.7 kB	+25.9 kB
▸ system / Context	80	3	+77	3.2 kB	240 B	+3.0 kB

Figure 9.3: The Chrome DevTools heap-snapshot comparison view, sorted by the delta column, with the leaking constructor at the top

When you select a leaking object, you should look for two numbers. Shallow size is how big the object is by itself. Retained size is how much memory would be freed if that object went away, including everything that only it keeps alive. A leak is usually a small object with a huge retained size, because it is the thing pinning a large graph. DevTools also shows the retainers, the chain of references holding the object alive, all the way back to a root. That chain is the retaining path from earlier.

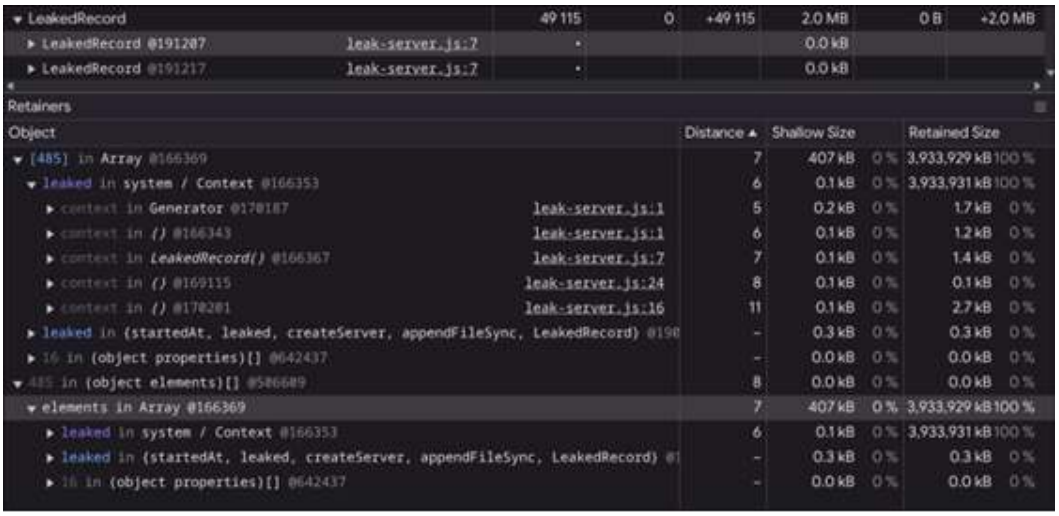


Figure 9.4: The Retainers panel showing the retaining path from a leaked object back to the module-level array that holds it alive.

Capturing snapshots without a browser

You cannot always attach a browser, especially to a process that is misbehaving in production. Node can write a snapshot to a file itself:

```
import { writeHeapSnapshot } from 'node:v8';

const file = writeHeapSnapshot();
console.log(`heap snapshot written to ${file}`);
```

You then load that `.heapsnapshot` file into the DevTools Memory tab and analyze it the same way. Be careful with this in production, because writing a snapshot briefly pauses the process and temporarily needs roughly twice the current heap in memory, which can itself trigger an out-of-memory kill. Take snapshots on an instance you have drained of live traffic where you can.

For the worst case, where a process is dying and you want to see the heap just before it goes, Node can capture a snapshot automatically as memory approaches the limit:

```
node --heapsnapshot-near-heap-limit=2 src/index.js
```

This writes up to the given number of snapshots as the heap nears its ceiling, so you get a picture of what filled it up right before the crash.

Other tooling

Beyond the built-in flags, a few packages help. `clinic.js` wraps these diagnostics in a friendlier interface, with a `doctor` command that flags health problems and a heap profiler for memory specifically, and `0x` produces flame graphs. Both are useful, though it is worth knowing that `clinic.js` is not very actively maintained anymore, so check that it still works with your Node version before relying on it. Older packages like `heapdump` and `memwatch` exist in tutorials but have largely been superseded, and the built-in `v8.writeHeapSnapshot` and the snapshot flags cover what they did.

The workflow is the same regardless of tooling: reproduce the leak under load, confirm the trend, capture and compare heap state, follow the retaining path to whatever is holding on, fix it, and run the same load again to confirm the baseline is now flat.

How to not leak in the first place

You have already seen the fix for each leak as it came up. They all apply one principle: give everything that holds memory an explicit, bounded lifetime. This section covers a few general tools that turn those one-off fixes into defaults, so you stop relying on yourself to remember them case by case.

Bound everything that grows

Any structure whose size grows with traffic needs an upper bound. For caches, reach for a library built for this, such as `lru-cache` or `lru.min`, which evicts old entries automatically once it is full.

```
import { LRUCache } from 'lru-cache';

const userCache = new LRUCache({
  max: 1000, // never hold more than 1000 entries
  ttl: 1000 * 60 * 5, // and expire entries after five minutes
});

async function getUser(id) {
  const cached = userCache.get(id);
  if (cached) return cached;

  const user = await loadUserFromDatabase(id);
  userCache.set(id, user);
```

```
  return user;
}
```

`lru-cache` requires you to set at least one bound, a maximum count, a maximum size, or a time-to-live, precisely so you cannot accidentally create the unbounded cache from earlier.

WeakRef and FinalizationRegistry, with care

You already saw `WeakMap`, the right tool for keying data by an object whose lifetime you do not control. It has two cousins that people reach for when they need cleanup hooks, and both deserve caution. `WeakRef` lets you hold a reference to an object without keeping it alive, and `FinalizationRegistry` lets you ask for a callback when an object is finally collected. They sound perfect for releasing resources, but the specification makes no promise about when, or even whether, those callbacks run.

A reasonable use is a cache that should not keep its values alive. The `WeakRef` holds each value loosely, and a `FinalizationRegistry` sweeps the dead slot once the value is collected:

```
const cache = new Map(); // key -> WeakRef holding the value

const registry = new FinalizationRegistry((key) => {
  // best-effort: drop the dead slot once the value is collected
  if (cache.get(key)?.deref() === undefined) cache.delete(key);
});

function getReport(id, build) {
  const cached = cache.get(id)?.deref();
  if (cached) return cached;

  const report = build();
  cache.set(id, new WeakRef(report));
  registry.register(report, id);
  return report;
}
```

`deref()` returns the value, or `undefined` once the garbage collector has reclaimed it, and the registry callback then removes the stale `Map` entry. The catch is in the words **"once the value is collected"** because you do not control when that happens, so a value can linger long after you expected, and the cleanup callback may run much later or not at all. That is why this pattern only fits caches of non-critical data, where a stale slot hanging around a little longer costs nothing.

They are fine as a best-effort optimization and a bad choice for anything that is critical or should be consistent. Never use a `FinalizationRegistry` to close a database connection or release a file handle. Do that kind of cleanup explicitly.

Keep scopes... Scoped

The main idea for this is the same for everything: **keep state alive only as long as the work that needs it**. Prefer creating state per request and letting it fall out of scope when the request ends rather than storing it in module-level variables that live forever.

When you really need long-lived state, such as a cache, bound it (back to the LRU). Pair every new attachment with a release. That means pairing every `on` with an `off`, every `setInterval` with a `clearInterval`, and every stream with a `pipeline` or a `destroy`.

When you would otherwise track several of these teardowns by hand, an `AbortController` groups them into one main switch. You hang each teardown off its `signal` (through `events.addAbortListener`, or the `{ signal }` option on APIs that accept one), then call `controller.abort()` once when the work finishes:

```
import { addAbortListener } from 'node:events';

function openStream(req, res) {
  const controller = new AbortController();
  const { signal } = controller;

const onTick = (data) => res.write(`data: ${JSON.stringify(data)}\n\n`);
  eventBus.on('tick', onTick);
  addAbortListener(signal, () => eventBus.off('tick', onTick));

const keepAlive = setInterval(() => res.write(':keepalive\n\n'), 15_000);
  addAbortListener(signal, () => clearInterval(keepAlive));

  res.on('close', () => controller.abort());
}
```

The listener and the timer are both registered against the same `signal`. When the response closes, a single `controller.abort()` runs every teardown for that signal, so the listener comes off the bus and the interval is cleared. To add another resource later you register one more teardown on the signal, so there is never a separate cleanup to forget.

The anti-pattern checklist

Here are the patterns from this chapter in one place:

1. Module-level arrays, maps, or objects that you only ever add to.
2. Caches with no maximum size and no expiry.
3. Using a regular `Map` to key data by object when a `WeakMap` would let it go.
4. Registering event listeners on a long-lived emitter without ever removing them.
5. Responding to `MaxListenersExceededWarning` by raising the limit instead of finding the leak.
6. `setInterval` or `setTimeout` without a matching `clearInterval` or `clearTimeout`
7. Believing `unref()` cleans up a timer when it only changes whether the timer keeps the process alive.
8. Promises that can never settle, with nothing awaiting them defensively.
9. `Promise.all` over an unbounded or very large input with no concurrency limit.
10. Writing to a stream without respecting backpressure or using `.pipe()` where `pipeline` belongs.
11. Long-lived closures that capture a whole large object just to read one field off it.
12. Raising `--max-old-space-size` to make an out-of-memory crash go away.

Summary

Memory leaks in Node are rarely clever bugs. They are ordinary references that outlive their usefulness, hidden inside ordinary code, and they only show up once the application has been running for a while.

In this chapter, you learned that V8 splits the heap into generations and keeps anything still reachable from a root, which is why a leak is memory you meant to let go of rather than memory the collector forgot.

You learned to read the numbers Node reports and to watch the baseline across collections instead of a single reading. You then went through the shapes behind almost every leak: module-level collections, closures that capture too much, unbounded caches, listeners and timers that are never removed, promises that never settle, unbounded fan-out, and streams that ignore backpressure.

To find one, you confirm the trend under load, compare heap snapshots, and follow the retaining path. To prevent them, you give everything that grows a bound and pair every attachment with a release.

This is also the last chapter of the book. Memory leaks are a good place to end, because they share something with most of the anti-patterns covered: the code looks correct, passes its tests, and only shows its cost after it has been running in production for a while.

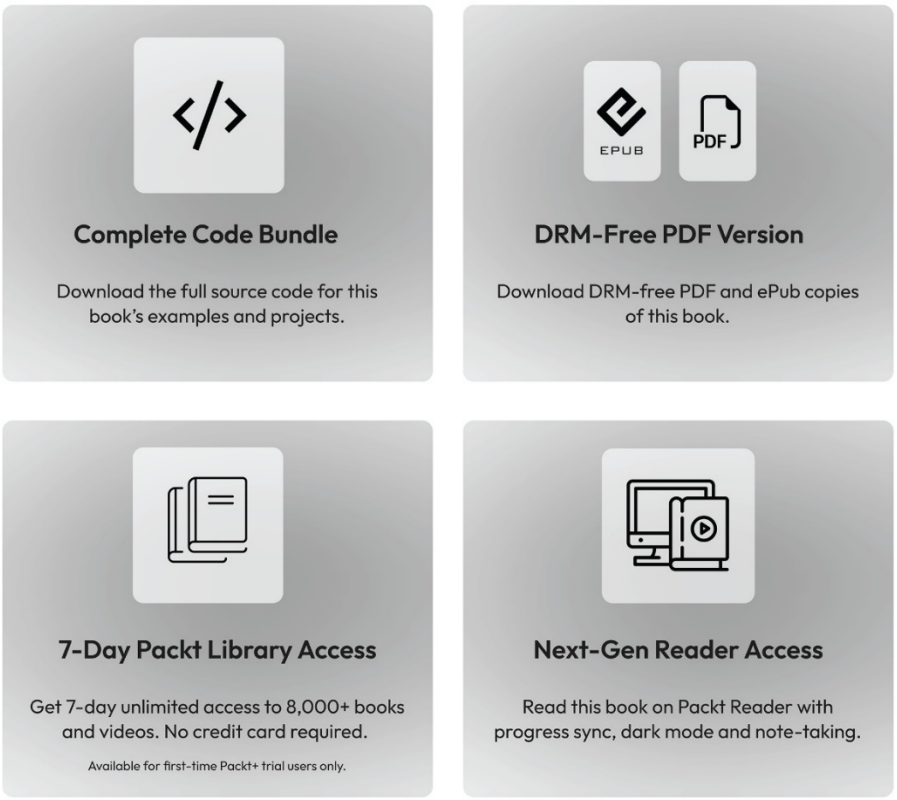
Most of them go away once you know how the runtime actually behaves and write code that works with it instead of against it. Take that habit into your code reviews, measure before you change things, and when something looks wrong in production, start from how Node works rather than from the symptom.

Thank you for reading.

10

Unlock the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

Have your purchase invoice ready for *step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/unlock-benefits/help`.

> **Note**
>
> **Note**: If you bought this book directly from the Packt website, no invoice is required. After *step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 10.1* on desktop), search for this book by name and select the correct edition.

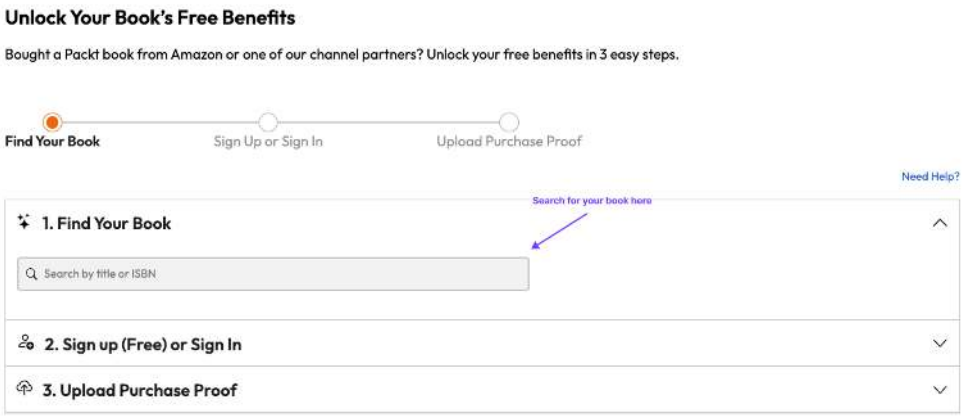


Figure 10.1: Packt unlock landing page on desktop

Step 3

After selecting the book, sign in to your Packt account or create one for free. Then, upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoice and more. This QR code will take you to the help page.

Note

Note: If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Building Production AI Agents for the Web

Christoffer Noring

ISBN: 978-1-80610-333-1

- Integrate LLMs into web apps and extend their capabilities with tool calling and external systems
- Build RAG pipelines that ground AI applications in relevant business and application data
- Use MCP to connect AI applications with tools, resources, and capabilities
- Build intelligent AI agents using ReAct, advanced agent architectures, and multi-agent patterns
- Test and evaluate LLMs and agents to improve reliability and accuracy
- Build an agent harness with the context, tools, guardrails, and infrastructure needed to take AI applications into production



Claude Visual Bible

Mark J. Price

ISBN: 978-1-80865-839-6

- Set up and secure your Claude account
- Write well-structured prompts and improve weak responses
- Search the web, assess sources, and analyze uploaded files
- Produce professional documents, spreadsheets, and presentations
- Plan and supervise multi-step assignments with Cowork
- Create reliable automated workflows with Skills and plugins
- Develop your skills in your current role and learn new skills
- Design and sell services and products

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *How Not to Node.js*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



https://packt.link/r/1836648499

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

.env files **178**
- cascading 174 – 176
- config package 176 – 178

A

anti-pattern checklist **226**

anti-patterns **2, 9, 18**
- callback-hell 11, 12
- checklist 179
- examples 10, 11
- global scoping 12 – 15
- resolving, strategies 43 – 45

application design
- pattern, impact 9

arrays **93 – 95**
- need for 101
- versus sets 101

async/await syntax **126, 127**

asynchronous code **104**
- callback 104 – 106
- promise 105 – 107

B

Bitwise Cyclic Tag **56**

bad code **24**

blocking I/O **70, 71**
- operations 71
- overview 72 – 75
- password encryption 75 – 77

blocking code **182**
- versus non-blocking code 182, 183

bound **223**

C

cache **214, 215**

callback anti patterns **108**
- callback-hell 108 – 111
- loss of context 119 – 122
- multiple execution 115, 116
- operation, hanging 117, 118
- Zalgo, releasing 111 – 114

callback-hell **10, 11, 108 – 111**

callbacks phase **184**

check phase **185**

circuit breaker **10**

close callbacks phase **185**

closure **213, 214**

code
- reading 24 – 27

code black hole **25**

code smells **18, 48 – 50**

console.log **136, 137**
- performance 138
- pipe problem 137

constants object **40**

cooperative scheduling **196 – 201**

cryptographic functions **71**

D

DRY violations
- patterns 28
- preprocessing 28
- sanitizations 28
- validations 28

Don't Repeat Yourself (DRY) **27 – 33**

dangling promise **124**
- memory leaks 125
- uncaught errors 124

data structures **85**
- job tool, selecting 100
- maps and objects 89
- need for 85 – 88

debug package **139, 140**

design patterns **2 – 8**

advantage 9, 10
dotenv 172
dotenvx 172

E

European computer manufacturers association (ECMA)-262 62
environment variables (envs) 41
event listener 215, 216
event loop 71, 181
best practices 203, 204
blocking 195, 196
cooperative scheduling 196 – 200
lag 196
phases 183 – 187
event loop, phases
callbacks phase 184
check phase 185
close callbacks phase 185
poll phase 185
prepare phase 184
timers phase 184

F

FinalizationRegistry 224
first-in, first-out (FIFO) 183
full error object log 153, 154

G

God Object 21 – 24
generateKeyPairSync 71
good code 24

H

heap 208
capturing 221
comparing 221
high coupling
between modules 18 – 20
hoisting 13

I

implicit type coercion 52
Boolean 56 – 59
numbers 52
numbers, using 53 – 56
object 56 – 59
strings 52

J

JavaScript code smells 50, 51
implicit type coercion 52
Prototype Pollution 60

L

late failure 162
leak
confirmation 220
finding 219
log 148, 149
correlation IDs 149
sensitive data log, avoiding 150, 151
storage 151 – 153
log levels 145
common mistake 146
loss of context 119 – 122

M

macrotasks 188
exploring 187
timers and I/O callbacks 190, 191
magic values 39 – 42
maps 89 – 91
versus objects 100, 101
memory leak 210 – 212
memory usage
reading 209, 210
microtasks 188
exploring 187
process.nextTick() and Promises 189, 190
mixed interactions 192 – 194
modules
high coupling 18 – 20
mutex 5

N

Node's Native 163, 164

programmatic loading 165
Node.js
blocking I/O 70, 71
memory management 208
requests, optimizing 77 – 84
working with 70
Node.js log
future 156
no IDE support **163**
non-blocking code **182**
versus blocking code 182, 183

O

OpenTelemetry (OTel) **155**
object lookups **86**
objects **90 – 92**
versus maps 100, 101
observability **155**
out-of-control module-level states **212**

P

Password-Based Key Derivation Function version 2 (pbkdf2) **76**
Pyramid of Doom **11**
parameters **34 – 36**
pattern **2**
pino-pretty tool **144**
poll phase **185**
prepare phase **184**
process.env **166, 167, 179**
failure modes 161 – 163
TypeScript 171, 172
validated singleton pattern 167 – 169
working 160, 161
production logger **140**
Pino 141 – 143
promise anti patterns **122**
.then() block 123
dangling promise 124
promises **217, 218**
promisification **127, 128**
never-caught expression 128
promisification of events 129
repromising 130 – 132
prototype pollution **92, 93**

R

race condition **4**
radix **54**
reachability **209**
repromising **130**
retaining path **209**
runtime mutability **162**

S

SOLID principles **20**
Singleton **8**
scattered access **163**
scope **225**
sets **93 – 95**
large datasets, working with 95 – 100
need for 101
versus arrays 101
single-call-contract **115**
smart functions **36 – 39**
snapshots
capturing, without browser 222
specification
tracking 62, 63
streams **217**
without backpressure 219
stringly-typed values **162**
structured logging **147**

T

Technical Committee 39 (TC39) **62**
TypeScript **64**
types, runtime 65
timer **216**
timers phase **184**
tooling **223**

U

unbounded fan-out **218**

V

validated singleton pattern **167 – 169**
with environment variables and Zod 170, 171
variable leaks **213, 214**

W

WeakRef **224**
work delegation **201, 202**
worker_threads module **201**

Z

Zalgo
releasing 111 – 114